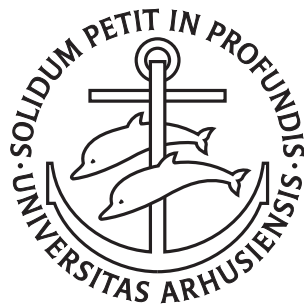

A Journey through Persistence, External Memory, Trees, and other Unrelated Problems

Casper Moldrup Rysgaard

PhD Dissertation



Department of Computer Science
Aarhus University
Denmark

A Journey through Persistence, External Memory, Trees, and other Unrelated Problems

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Casper Moldrup Rysgaard
July 31, 2025

Revised April 2026

Abstract

During my Ph.D. I have been fortunate to work on many projects, which have various amounts of overlap. The largest overlap is the use of *Persistence* and *External Memory*.

In the functional model, all data is immutable and therefore persistent. However, the immutability force copying parts of the structure upon updates. In this work we construct offline-partially-persistent trees, which removes this overhead on space.

We improve in this work the amortized runtime of both fully- and partial-persistent search trees in the external-memory model, to match the ephemeral B-trees of Bayer and McCreight and the B^E -trees of Brodal and Fagerberg, respectively. In the future work, we cover how to extend the partial-persistent trees to support new operations.

Search trees maintain the order of all elements, where priority-queues in the other extrema only maintain order on the minimum element. We manage to interpolate between these structures by creating external-memory Lazy B-trees, building on the internal-memory work by Sandlund and Wild. At the extrema, the Lazy B-tree yield a new efficient external-memory priority-queue, which supports decrease-key operations. In the future work, we simplify the construction to obtain this priority-queue.

Manipulating large formulae can be done efficiently using binary decision diagrams, which naturally grows exponentially large, prompting the use of external-memory solutions. We show in this work how to provide a speed-up in external-memory computation for narrow diagrams by introducing random access.

The art gallery problem asks for a given polygon how many guards are needed to cover the interior points or the boundary of the polygon, which both are $\exists\mathbb{R}$ -complete. In our work we show a variant where each guard cover a contiguous part of the boundary, and provide a polynomial time algorithm with bounded bit complexity.

Two classical textbook examples on dynamic programming is finding the Longest Common Subsequence (LCS) of two input strings and the Longest Palindromic Subsequence (LPS) of a single input string. In this work we prove the folklore claim the LPS can be computed by the LCS of the string and its reverse, for the classical Wagner and Fischer LCS algorithm. Building on the tools of the initial work, we show in the future work, for six additional LCS algorithms if they compute the LPS.

Countless finger search trees have been constructed in the literature, by adding additional information to the nodes of search trees. We construct finger search trees without any additional information in the nodes. In the future work, we extend the work to create simpler and faster search algorithms, and compare their runtime experimentally. In the future work we show simplified adaptive sorting using inversions.

Resumé

Under min ph.d. har jeg været heldig at arbejde på flere projekter, som har haft varierende grader af overlap, med det største værende *Persistens* og *Ekstern Hukommelse*.

I den funktionelle model er alt data uforanderligt og derfor persistent. Uforanderligheden tvinger dog til at kopiere en del af strukturen ved opdateringer. I dette arbejde konstruerer vi offline-delvist persistente træer, som fjerner denne ekstra plads.

Vi forbedrer i dette arbejde den amortiserede kørselstid for både fuldt og delvist persistente søgetræer i den eksterne hukommelsesmodel, for at matche henholdsvis de ikke persistente B-træer fra Bayer og McCreight og B^E -træer fra Brodal og Fagerberg. Yderligere dækker vi understøttelsen af nye operationer til de delvist persistente træer.

Søgetræer opretholder rækkefølgen af alle elementer, hvor den anden ekstrem er prioritetskøer, som kun opretholder nok rækkefølge til at opdatere minimumselementet. Vi skaber Lazy B-træer i ekstern hukommelse, som kan interpolere mellem disse strukturer, ved at udvide arbejde af Sandlund og Wild. I et ekstrem giver Lazy B-træet en ny effektiv prioritetskø til ekstern hukommelse, som understøtter at formindske elementer. Som udvidelse forenkler vi konstruktionen for at opnå denne prioritetskø.

Manipulation af store formler kan udføres effektivt ved hjælp af binære beslutningsdiagrammer, som naturligt vokser sig eksponentielt store, hvilket fører til ekstern hukommelses løsninger. I dette arbejde viser vi, hvordan man kan opnå en hastighedsforøgelse i beregningen på smalle diagrammer, ved at introducere tilfældig adgang.

Kunstgalleriproblemet spørger for en polygon, hvor mange vagter der er skal bruges for at se alle indre punkter eller polygonens grænse, som begge er $\exists\mathbb{R}$ -komplette. Vi viser en variant, hvor hver vagt dækker en sammenhængende del af grænsen, samt en algoritme med polynomial kørselstid og begrænset bitkompleksitet.

To lærebogseksempler på dynamisk programmering er at finde den *længste fælles delsekvens* (LCS) for to inputstrengene og den *længste palindromiske delsekvens* (LPS) for en enkelt inputstreng. I dette arbejde beviser vi den folkelige påstand, at LPS kan beregnes ved hjælp af LCS for strengen og den omvendte streng, for den klassiske Wagner og Fischer LCS-algoritme. Byggende på disse værktøjerne viser vi for seks yderligere LCS-algoritmer, om de beregner en LPS eller ej.

Utallige fingersøgetræer er blevet konstrueret i litteraturen ved at tilføje yderligere information til knuderne i søgetræerne. Vi konstruerer fingersøgetræer uden yderligere information i knuderne. I det fremtidige arbejde udvider vi arbejdet til at skabe enklere og hurtigere algoritmer til søgning og sammenligner deres kørselstid eksperimentelt. Derudover viser vi en forenklet adaptiv sortering ved hjælp af inversioner.

Acknowledgments

The four years of my Ph.D. studies have gone by fast. Throughout this journey, I have been fortunate to work and interact with many people, who have all helped in shaping me into the researcher I have become, and who all deserve a huge thanks.

First, I want to extend a thanks to my advisor Gerth Stølting Brodal. Over the years, his endless support, guidance and patience has been a calm in the storm that sometimes is the life of a Ph.D. student. I want to thank him for providing a safe environment, where I could grow as a researcher, allowing me to work on the problems at my own pace. In my world, I could not have hoped for a better advisor on this journey.

Secondly, I want to thank the late Lars Arge for providing me with the DFF grant, giving me the opportunity to even have this amazing job. From all the people who surrounded him, I have come to know of what a great loss it is to have never gotten the opportunity to meet this great man.

I want to thank my fellow Ph.D. students Jens Kristian Refsgaard Schou and Rolf Svenning, for the many great days we have had. I firmly believe that the ability we have had as peers to bounce ideas of each other, has strengthened our academic journey. Further I want to extend a special thanks to Jens Kristian for being the best office mate I could have ever hoped for. Even though we may not match on paper, we managed to use each other's strengths and weaknesses to both grow as people. Further, he has forever altered my taste in the "music" he keeps playing. Further, I want to extend a special thanks to Rolf who always managed to end up at the same conferences as me. Hanging out with Rolf always ensured that a fun time was afoot, where I could stay close to my comfort zone. With Rolf, no task was ever too difficult or scary, and he pushed me gently to try new and exiting things. Both of you will forever be my academic brothers.

Traveling have been the most uncertain part of my journey. I want to thank Sebastian Wild for both welcoming me to the University of Liverpool and again later at the Philips University of Marburg. His kindness and openness have steered me safely onwards. Further I want to thank, John Sylvester, Benjamin Smith, Konstantinos Tsakalidis, and the rest of the research group in Liverpool for making my first academic stay a great one. Additionally, I want to thank Riko Jacob, Thore Husfeld, and the other great people of the IT University of Copenhagen for welcoming me into their research group.

Academic breakthroughs are seldom made alone. On my journey I believe that I

might have just come close to one such breakthrough, and I want to thank the rest of my coauthors Steffan Christ Sølvsten, Jaco van de Pol, Rolf Fagerberg, and Magnus Christian Ring Merrild for the amazing work we have completed together.

I have traveled not only for research stays, but also for many conferences out and about. I want to thank all the great people I have met at these conferences, who made the trip worth making. Thanks for the countless research discussions and the many late evenings.

No great work is ever completed without first drinking a cup of coffee, as well as enjoying said coffee with good people. I want to thank the people of the ☕-ping chat, for many a well-deserved break, and for listening in on my new and crazy ideas.

Whenever work was not to be completed, I spend time with the many great friends I have gained from the hub of Aarhus University. I want to thank the people of the student kitchen *Regnecentralen* for making every lunch break and odd evening a time to enjoy, and for providing me with a home away from home throughout my studies. Thanks to Lasse Letager Hansen, Frank Skorstengaard, Andreas H. H. Hansen, Simon and Louise B. Dohn, Mathias Ravn Tversted, Alexander Munch-Hansen, Anna Hallenberg, Anne Kirstine D. Overgaard, Ulrik B. Djurtoft, Magnus B. Wind, Martin M. Jensen, Magdalena Kalin, and far many others.

I want to thank Fredagscaféen for always being a safe and cider-carrying harbor at the end of every week. When (mis)adventures was afoot, *TÅGEKAMMERET* never failed to provide great company, laughter, and just an ounce of stupidity, as should be for any relaxing researcher. I want to thank the many great people I have met at this wonderful location, and further thank the grand trust bestowed onto me for when I was chairman UU. A special thanks go to my fellow board, who helped make the work into a wonderful year, and which helped to shape me as a person.

On my travels I found safe harbors across the country. I want to thank Christian Påbøl Jacobsen and Julie S. Kofod-Jensen for welcoming me into their home in Copenhagen and providing me a place to live. I want to thank Caféen? at Copenhagen University for making my stay a fun one. Thanks to August Laursen, Jo Liisborg, Anthon Dirckinck-Holmfeld, Mikkel Stolborg, Stefan K. Rasmussen, and many others, for bringing me along on your (mis)adventures.

Lastly I want to thank my parents and sister for their endless support and love throughout this journey. I want to give a special thanks to my soulmate My Bang Petersen. I am eternally grateful for all the hugs and kisses you have provided, as well as putting up with my endless rants on algorithms and data structures. This one is for you.

*Casper Moldrup Rysgaard,
Aarhus, July 31, 2025.*

Contents

Abstract	i
Resumé	iii
Acknowledgments	v
Contents	vii
I Overview	1
1 Introduction	3
1.1 The I/O Model	4
1.2 Persistence	6
2 Space-Efficient Functional Offline-Partially-Persistent Trees with Applications to Planar Point Location	7
2.1 Constructing Functional Offline-Partial-Persistent Search Trees . . .	8
2.2 Generalizing the Construction	9
2.3 Implementation	10
3 External Memory Fully Persistent Search Trees	13
3.1 Interface of a Fully Persistent Search Tree	14
3.2 Geometric Interpretation	15
3.3 Maintaining the Partition	16
3.4 Amortized Case Analysis	19
3.5 Corrections to the Paper	23
4 Buffered Partially-Persistent External-Memory Search Trees	27
4.1 Interface of a Partially-Persistent Search Tree	28
4.2 Geometric Interpretation	28
4.3 Partitioning the Plane	29
4.4 Worst-Case Bounds	32
4.5 Correction of the Stated Worst-Case Improvement	34

5	Lazy B-Trees	35
5.1	Outline of Lazy Search Trees	37
5.2	External-Memory Biased Search Tree	37
5.3	Corrections to the Paper	40
6	Random Access on Narrow Decision Diagrams in External Memory	43
6.1	Binary Decision Diagrams	44
6.2	The Apply Operation	45
6.3	Introducing Random Access	47
6.4	Omitted Experiment	49
7	The Contiguous Art Gallery Problem is Solvable in Polynomial Time	51
7.1	The GREEDYINTERVAL algorithm	52
7.2	The REPEATEDGREEDY Algorithm	54
7.3	The Analysis of the REPEATEDGREEDY Algorithm	55
7.4	Vertex Restricted Variants	56
7.5	Bounding the Bit Complexity	58
7.6	Concurrent Work	61
7.7	Open Problems	62
8	On Finding Longest Palindromic Subsequences Using Longest Common Subsequences	65
8.1	The Wagner and Fischer Algorithm	66
8.2	Relationship between LCS and LPS	67
8.3	Properties of LCS algorithms	68
8.4	Computing an LPS using an LCS algorithm	70
9	Pure Binary Finger Search Trees	73
9.1	Overview of the Paper	74
9.2	Encoding extra Information in Metanodes	75
9.3	Folding a Balanced Binary Tree	77
9.4	Supporting Finger Searching	79
9.5	Worst-case Lower Bounds and a New Tree Structure	80
II	Publications	81
10	Space-Efficient Functional Offline-Partially-Persistent Trees with Applications to Planar Point Location	83
10.1	Introduction	83
10.2	Initial Analysis of Binary Search Trees	88
10.3	Freezer and Query Structure Construction	89
10.4	Proof of Theorem 1	93
10.5	Further Applications	94
10.6	Implementation and Experiments	95

10.7	References	97
11	External Memory Fully Persistent Search Trees	99
11.1	Introduction	99
11.2	Static Data Structure	101
11.3	Updates	103
11.4	Colored Predecessor Queries	109
11.5	Point Location	110
11.6	Partitioning the Version Tree	110
11.7	Lazy Clones	111
11.8	Conclusion and Future Work	112
11.9	References	112
12	Buffered Partially-Persistent External-Memory Search Trees	113
12.1	Introduction	113
12.2	The Buffered Persistent Data Structure	118
12.3	Worst-Case Bounds	124
12.4	Discussion and Open Problems	128
12.5	References	129
12.6	Appendix	131
13	Lazy B-Trees	133
13.1	Introduction	133
13.2	The Gap Structure: A New Restricted External Biased Search Tree	140
13.3	The Interval Structure	145
13.4	Lazy B-Trees	146
13.5	Open Problems	148
13.6	References	149
13.7	Appendix	152
14	Random Access on Narrow Decision Diagrams in External Memory	163
14.1	Introduction	163
14.2	Preliminaries	164
14.3	Using Random Access for Narrow Decision Diagrams	166
14.4	Experimental Evaluation	167
14.5	Conclusion	169
14.6	References	170
15	The Contiguous Art Gallery Problem is Solvable in Polynomial Time	173
15.1	Introduction	173
15.2	The Repeated Greedy Algorithm	178
15.3	The GreedyInterval algorithm	179
15.4	Algorithm 1, a combinatorial viewpoint	186
15.5	Algorithm 1, a geometric viewpoint	193

15.6 Bit Complexity of Algorithm 1	205
15.7 Open problems	208
15.8 References	208
15.9 Appendix	211
16 On Finding Longest Palindromic Subsequences Using Longest Common Subsequences	221
16.1 Introduction	221
16.2 Relationship between LCS and LPS	225
16.3 Properties of LCS algorithms	227
16.4 Computing an LPS using an LCS algorithm	231
16.5 References	234
16.6 Appendix	235
17 Pure Binary Finger Search Trees	237
17.1 Introduction	237
17.2 Red-black trees	241
17.3 Finger searches using predecessor and successor links	243
17.4 Folded red-black trees	244
17.5 Metanodes	252
17.6 Representation of pure binary search trees	254
17.7 Pure binary finger search trees	254
17.8 Worst-case lower bound for pure binary search trees	255
17.9 Linked list metanodes	256
17.10 Conclusion and open problems	257
17.11 References	258
III Extensions and Future Work	261
18 Supporting Additional Operations for Buffered Partially-Persistent External-Memory Search Trees	263
18.1 Partial RANGE Queries	263
18.2 Supporting Partially Ordered Sets	264
19 Extending Finding Longest Palindromic Subsequences Using Longest Common Subsequences to other Algorithms	267
19.1 Hirschberg	268
19.2 Hunt-Szymanski	268
19.3 Kuo and Cross	275
19.4 Myers	276
19.5 The use of LCS algorithms in Real World Applications	281
20 Extending Pure Binary Finger Search Trees with New Algorithms	283
20.1 Simplified Pure Binary Finger Search Trees	283

20.2 Overview of Finger Searching	284
20.3 Limitations of Finger Searching	285
20.4 A Simple Optimization to Finger Searching with Predecessor and Successor Links	286
20.5 A New Algorithm for Finger Searching with Predecessor and Succes- sor Links	287
20.6 Finger Searching in Folded Trees	290
20.7 Implementation and Experiments	293
21 Using Lazy B-Trees for External-Memory Priority-Queues	325
21.1 Simplified External-Memory Priority-Queue	326
21.2 Cache-Oblivious External-Memory Priority-Queue	327
21.3 Corrections to the Result	328
22 Simplified Adaptive Sorting using Inversions	331
22.1 Simplified Upper Bound	332
22.2 Simplified Lower Bound	333
Bibliography	335

Part I

Overview

Chapter 1

Introduction

This dissertation covers the work I have done during my PhD, which concluded in the following eight published papers.

- Space-Efficient Functional Offline-Partially-Persistent Trees with Applications to Planar Point Location (WADS'23 [42])
- External Memory Fully Persistent Search Trees (STOC'23 [43])
- Buffered Partially-Persistent External-Memory Search Trees (ESA'25 [44])
- Lazy B-Trees (MFCS'25 [110])
- Random Access on Narrow Decision Diagrams in External Memory (SPIN'24 [125])
- The Contiguous Art Gallery Problem is Solvable in Polynomial Time (SoCG'25 [22])
- On Finding Longest Palindromic Subsequences Using Longest Common Subsequences (ESA'24 [35])
- Pure Binary Finger Search Trees (SOSA'25 [40])

The structure of the dissertation is split into three parts. First, an overview of the results produced in the different projects is covered in Part I. This overview includes a short introduction to a model and an overall theme I have used for some of the projects. Then a chapter for each project follows, which describes some required preliminaries, the results of the paper, potential challenges the project had along the way, and an overview of the construction. Secondly, the unedited publications follow in Part II. Due to constraints imposed by the conferences, some included papers will not be the conference version, but the full version published at arXiv. Which type is included will be stated for the related project in Part I. Finally, unpublished extensions of the projects, as well as future work is stated in Part III.

In the different projects I have worked on, there is not a large common overlap. The two main common themes in the projects are the use of the *I/O model* and the notion of *persistence*. In the following sections, I will give a brief overview of these two areas.

1.1 The I/O Model

In classical algorithms and data structure introduction courses, the model of computation used to analyze the runtime of constructions is the *Random Access Memory Model* (RAM model), also denoted as the *internal-memory model*. This model assumes that access to any part of memory is of unit cost, and the model therefore counts the number of *operations* performed, be that memory access or computations on the fetched data, as a way to measure the runtime. A real computer is, however, more complicated than the RAM model suggests. Specifically, the memory layout of the computer is a *hierarchy* of multiple layers. Closest to the Central Processing Unit (CPU) is multiple levels of *cache*, followed by a layer of *Random Access Memory*, and lastly a Hard Disk Drive (HDD) or Solid State Drive (SSD) layer. Taking this one layer further, one may even consider cloud storage as a fourth layer of memory. This hierarchy is build such that access to the layers closest to the CPU, the cache, is fastest. However, the speed comes with the cost that the size of the memory is limited. The access time then grows outwards in the memory layers from the CPU, but as the access time grows, so does the memory size. As the access time to outer layers of memory is slower, the computer tries to optimize the fetch, by instead of just fetching the single requested element fetching a *block* of *contiguous* memory.

To put numbers on the access time, we shall use the CPU cycles as the unit cost. Accessing data in the cache levels takes $\sim 4 - 50$ clock cycles [81, Table 2-7, 2-14], accessing data in RAM uses ~ 200 cycles [81, Section 3.6.10], and lastly accessing data on an HDD uses $\sim 20.000.000$ cycles [107, 64]¹. The access time to the HDD is thereby ~ 100.000 times as slow as accessing data on RAM. Replacing the HDD with an SSD improves the access time, however, a large factor in speed still separates the RAM and SSD memory fetching time.

When the input to an algorithm no longer fits in the RAM of the computer, the construction must use the disk. An efficient RAM algorithm may be performing many random accesses, which for the large inputs may require reading often from the disk, rather than from the portion of the data stored in RAM. We therefore wish to use a model which better represents that accessing data on disk is a lot slower than data on RAM.

1.1.1 Specification of the I/O Model

Aggarwal and Vitter introduced the *I/O model* or *external-memory model* in 1988 [6]. The model simplifies the memory hierarchy to contain just two levels, the *internal* and *external* memory. The size of the internal memory is bounded to contain at most M data records, where the external memory is unbounded. Computations on any of the internal data records is considered *free* in the I/O model, instead the runtime is measured in how many data transfers, or I/Os, are performed between the internal and external memory. The model allows for reading (Input) or writing (Output) a block of

¹The number of cycles for fetching data on HDD stated in [107, 64] have been extrapolated to match the specification of the intel manual [81].

B consecutive data records between the memory layers using 1 I/O. The parameter B is not only used to capture the size of a block, but also in general the latency of transferring data between the two layers of memory. The model assumes that $M \geq 2B$, as otherwise moving data between blocks are infeasible, and similarly that $B \geq 2$. Initially, the N data records input to any algorithm is stored in the external memory.

To *scan* over the input, the N elements in external memory are fetched in blocks of B into the internal memory one at a time, spending in total $\Theta(\text{scan}(N))$ I/Os, where $\text{scan}(N) = \frac{N}{B}$. In the RAM model, the similar operation has a runtime of $\Theta(N)$. Sorting can similarly be improved in the I/O model. A lower bound on sorting is $\Omega(\text{sort}(N))$ I/Os [6], where $\text{sort}(N) = \frac{N}{B} \log_{M/B} \frac{N}{B}$. A matching upper bound of $\mathcal{O}(\text{sort}(N))$ I/Os can be achieved by an M/B -way merge-sort algorithm [6].

In constructing I/O efficient algorithms and data structures, we have access to the parameters M and B . If access to these parameters are not allowed, but the runtime still is based on the number of I/Os performed, the *cache oblivious model* is obtained, which was first described by Frigo, Leiserson, Prokop, and Ramachandran in 1999 [66]. The advantages of the cache oblivious model over the standard I/O model, is that since the values of M and B are unknown, the algorithm is efficient across the multiple different layers of memory present in a real computer.

1.1.2 Basic I/O Data Structures

In general, pointer based constructions are not I/O efficient, as each pointer chase generally may require an I/O. Instead of using Binary Search Trees, an I/O variant was constructed by Bayer and McCreight known as the *B-tree* [15]. In the B-tree, each node contains $\Theta(B)$ children instead of 2. Such a node may be read using $\mathcal{O}(1)$ I/Os, as all separator keys and child pointers fits inside $\mathcal{O}(1)$ blocks of memory. As the degree of nodes are higher, the height of the tree is $\Theta(\log_B N)$, when it contains N keys. B-trees allows for queries, insertions and deletions using $\mathcal{O}(\log_B N)$ I/Os, and range queries using $\mathcal{O}(\log_B N + K/B)$ I/Os, when K keys are returned from the tree.

As access to consecutive memory is efficient in the I/O model, arrays are the go-to structure to store data. In the RAM model, another widely used list type is the *linked-list*. However, as the linked-list is a pointer based structure, they are not efficient in the I/O model. A variant of the linked-list exists, which we in our papers have denotes as a *blocked-linked-list* [56, Section 5.4]. In this structure, each node is a block, each containing $\Theta(B)$ elements, and a pointer to the next and previous node in the list. As each block contains $\Omega(B)$ elements, a list of N elements may be scanned efficiently using $\mathcal{O}(N/B)$ I/Os. Inserting or deleting an element anywhere in the list can be supported in amortized $\mathcal{O}(1/B)$ I/Os, when the block containing the location is in the internal memory. When a block contains too few or too many elements, it is either merged with a neighbor or split into two new blocks, using $\mathcal{O}(1)$ I/O. If choosing the bounds carefully, we can ensure that $\Omega(B)$ updates to a block must happen before it merges or splits.

If random access into the list is needed, this must be provided by pointers into specific elements, by pointing to the given block. When a block is split or merged,

these incoming pointers must then be updated. If we allow each block to contain $\mathcal{O}(B)$ incoming pointers, we can support this operation using $\mathcal{O}(B)$ I/Os per block update. By increasing the amortized update time to $\mathcal{O}(1)$ I/Os, this cost is covered.

1.2 Persistence

Classically, any update to a data structure alters the structure, and the previous version is thereby lost. We denote this as the structure being *ephemeral*. If instead the updates to the structure preserves the version of the structure before the update, we denote the structure as *persistent*. If only access and not updates to older versions are allowed, we denote the structure as *partial persistent*, and if updates to older versions are allowed, creating new versions, we denote the structure as *fully persistent*. The difference between these two versions of persistence can be seen in the structure of the versions, where partial persistence creates a *list* over the versions, whereas fully persistence creates a *tree* over the versions.

In the internal-memory model, efficient partial persistence for search trees was solved by Driscoll, Sarnak, Sleator, and Tarjan [61]. Their solution in essence allows each node of the tree to contain multiple pointers, each of which is associated with a *liveness timestamp*, i.e., the range of versions this pointer is relevant to. When the node then contains too many pointers, the live pointers are copied to a fresh node, which is inserted into the tree, updating the incoming pointers of the old node, to point to the new node from that time forward. This solution was then in essence ported to the external-memory model by Arge, Danner, and Teh [13].

To create efficient fully persistent search trees in the internal-memory model, the partial persistence solution of Driscoll, Sarnak, Sleator, and Tarjan [61] does not immediately apply, as the version no longer form a natural order, as they no longer form a list, but instead form a tree. Some order may be created by a *preorder traversal* of the version tree, which is sufficient to apply the partial persistent solution. The versions are then no longer compared by value, as the preorder traversal outputs the versions in some arbitrary order independent on the version numbers, and new versions may be inserted anywhere in this order. By applying the dynamic order maintenance structure of Dietz and Sleator [57], the versions may be compared w.r.t. their locations in the preorder traversal. This leads to the fully persistent search tree, presented in the same paper as the partial persistent tree by Driscoll, Sarnak, Sleator, and Tarjan [61].

Chapter 2

Space-Efficient Functional Offline-Partially-Persistent Trees with Applications to Planar Point Location

This chapter discusses the paper “Space-Efficient Functional Offline-Partially-Persistent Trees with Applications to Planar Point Location”, which was published at WADS’23 [42]. This paper was created in collaboration with Gerth Stølting Brodal, Jens Kristian Refsgaard Schou, and Rolf Svenning from Aarhus University. The included paper is the conference version, which fit all details. The paper is included in Chapter 10.

As mentioned in the introduction, persistence in data structures have been studied in details in the imperative model, most notably by Driscoll, Sarnak, Sleator, and Tarjan, who showed how to convert any ephemeral data structure with nodes of constant in-degree into a persistent version, without introducing any asymptotic time or space overhead [61]. On the contrary, persistence have been studied very little in the functional model. This may be due to the nature of the functional model, in which only functions and immutable data is allowed, to which Okasaki states that “*A distinctive property of functional data structures is that they are always persistent*” [104].

However, the implicit persistence of the functional data structures comes with an overhead in space. Assume for simplicity, that the stored structure is a binary tree. Upon updating a node, then by the immutability of the functional model, a new copy of the node must be created, which contains the updated values. The parent of the node must then be updated to point to the updated node, and therefore must also be copied. This results in the entire path down to the initially updated node, being copied. This paradigm matches exactly what Sarnak and Tarjan named as the *path copying* technique for making imperative data structures persistent [115]. If persistence is not needed, then copying the path does not impose an overhead for the functional model, as the old nodes may be garbage collected. However, for persistence, the old path

must be remembered, which therefore impose the space overhead.

The goal of this paper was to port the efficient technique of Driscoll, Sarnak, Sleator, and Tarjan [61] to the functional model, to gain partial persistence, with no overhead in time and space, for updates and queries. The issue is that the technique in essence produces a structure which forms a *directed acyclic graph* (DAG), which cannot be maintained efficiently in the functional model, as updating any node would require updating any node which transitively points to the updated node. A direct approach would therefore be efficient in space, but highly inefficient in time.

We therefore opted for *offline partial persistence*, where first a series of updates is applied yielding a series of versions of the structure, then a *build phase* is applied, where the query DAG is build, such that finally, queries are allowed to all previous versions of the structure. However, the update and query operations may no longer be intermixed, as in *online* partial persistence.

2.1 Constructing Functional Offline-Partial-Persistent Search Trees

The inspiration for the result of the paper came when we noticed that the *planar point location* problem was not solved efficiently in the functional model. We let n denote the number of segments partitioning the plane. Planar point location is a problem classically solved by partially persistent search trees. The previously best known result to the problem in the functional model matched the space and query bounds of the imperative solution, but came with a $\mathcal{O}(\log n)$ overhead in space. We then studied the imperative solution of Sarnak and Tarjan [115], which uses partially persistent search trees, to construct a structure for the planer point location queries. In studying what limitations their solution had in the functional model, we were able to improve the space usage, by instead of maintaining a partial persistent structure in the preprocessing phase, instead only maintaining the current tree, and a list of old edges. We then add a $\mathcal{O}(n \log n)$ build step, to build the final DAG of the partially persistent search tree, which does not asymptotically increase the preprocessing time. A table of the previous solution compared to our new result is shown in [42, Table 1], and restated in Table 2.1.

The result of the paper builds on the following simple idea. We cannot maintain a DAG. We can, however, maintain the current tree, and upon an update, we can detect which edges this update will add to the DAG, i.e., we can detect the edges the update removes from the tree, which we will timestamp and add to the DAG. When only considering the updates, we do not need to maintain the DAG, but only the current tree. When updates alters the tree, we shall then detect the edges which no longer are part of the current tree, but are part of the DAG, and place the detected edges into a list we denote as the *freezer*. After all updates have been applied, we can then use the edges of the final tree, along with the edges of the freezer, to build the query DAG. The DAG must be build bottom up, as the functional model only allows to point to already build substructures. We therefore apply Kahn's algorithm, which performs

Table 2.1: ([42, Table 1]) Previous and new results for planar point location, where † are expected bounds and * are results based on persistent data structures.

Reference	Construction	Query	Space	Model
David Kirkpatrick [85]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative
Seidel [117]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative †
Dobkin and Munro [60]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n \log n)$	Imperative *
Richard Cole [54]	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative *
Sarnak and Tarjan [115]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative *
Sarnak and Tarjan [115]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$	Functional *
<i>Our Paper</i> [42]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Functional *

a topological sort of the nodes, to build the DAG. The paper therefore combines the technique of Driscoll, Sarnak, Sleator, and Tarjan [61] to get low space complexity, with the idea of maintaining only a list of the DAG edges, to then finally build the DAG using Kahn’s algorithm [83].

Note that the structure of Driscoll, Sarnak, Sleator, and Tarjan [61] is not guaranteed to produce a DAG, but may produce cycles in the structure. This structure cannot be created explicitly in the functional model, and we therefore had to adjust the construction slightly. Specifically, the cycles appear when *rotations* are applied to keep the tree balanced, as some node is moved above its ancestor. Similarly, this issue arises upon deleting elements of the tree, as a leaf is lifted in the tree, to replace the deleted location. To fix this issue, we introduce new nodes for nodes that are moved up in the tree, which allows for the final structure to be a DAG. See [42, Figure 1] and [42, Figure 3] for an illustration of the extra introduced nodes. These extra introduced nodes does not impose an overhead in space, if at most $\mathcal{O}(1)$ rotations are performed upon any update.

2.2 Generalizing the Construction

After solving the planar point location problem, we noticed that the construction could be applied to other structures and not just binary search trees. This lead to the TUNA conditions stated in [42, Definition 1], which are restated below.

T: The data structure forms a rooted tree of constant degree d .

U: Updates create $\mathcal{O}(1)$ new edges and nodes.

N: No cycles are created by updates when considering the edges that have been created across all versions.

A: Attribute values of nodes are static, i.e., the information stored in a node is not changed after its insertion. This does not include fields pointing to the children.

The TUNA conditions are what is required to apply the construction. At first glance, the TUNA conditions may seem too restrictive, but are applicable to many structures of the functional model.

Immediately, the TUNA conditions and the construction yields an offline partially persistent version of, among other, binary search trees (BST), linked lists, and the *random access arrays* of Okasaki [103]. It can then be shown that condition U may be relaxed to each update creating *amortized* $\mathcal{O}(1)$ new edges and nodes. This then allows for the balanced BSTs of red-black trees [74] to be made partially persistent. Similarly, if *expected* $\mathcal{O}(1)$ new edges and nodes are created, then the time and space use of the build step becomes expected $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$, respectively. This allows for Treaps [119] to satisfy the TUNA conditions.

By a small modification, the construction allows relaxing condition A, s.t. nodes may contain extra information, which may be arbitrarily updated, as long as the extra information is only used for updates and not queries. In essence, this extra information is only stored in the tree, and not in the DAG. This similarly allows for red-black trees, but also AVL-trees [5], to be made offline partially persistent.

2.3 Implementation

The TUNA conditions allow any complying structure to be made offline partially persistent. This may also be seen as the conditions leading to a general *framework* for offline partial persistence. To demonstrate this, I implemented the construction as a decorator like framework in Haskell, which is available at [41].

When decorating a data structure complying with the TUNA conditions, the framework separates the logic of the offline partial persistence from the logic of the data structure. The separation is simple for the logic of building the query DAG from the freezer, and for supporting queries into the DAG, as the logic of the data structure is not needed in these steps. To separate the logic during the queries, the framework uses decorating lambdas and an explicit state monad to update the freezer and timestamps upon updates.

Before this final separation was supported, the logic of the offline partial persistence and the logic of the data structure during update operations was intertwined, and the maintenance of the freezer edges and timestamps on edges needed to be explicitly handled. In this initial version of the framework, I implemented as an example regular unbalanced binary search trees which supported insertion and deletion updates. Within these updates, the logic of the offline partial persistent was mixed with the logic of the update, which lead to the implementation using 387 lines of Haskell code. For comparison, after the final version of the framework was created, which allows for separating the logic, the same data structure operations required only 47 lines of code. The separated logic of the offline partial persistence during queries takes up 239 lines of code. The separation therefore lead to saving ~ 100 lines of code.

This simplification in supporting queries lead to the ability of implementing offline partially persistent red-black trees, by decorating the implementation from the

Haskell standard library. The decorated version increases the number of lines of code from 150 to just 165. Similarly, a decorated version of the random access arrays of Okasaki [103] was constructed in the Haskell framework.

Using the framework, we then tested the construction of the paper, and concluded that the construction lead to no asymptotic overhead in update and query time, and that the space usage was linear in the number of updates, as expected. The runtime of the DAG building was shown to theoretically be $\mathcal{O}(n \log n)$, however, this does not match with the measured runtime. As a sanity experiment, we ran the experiment of constructing a perfectly balanced search tree over n nodes, and then measured the runtime of querying all nodes in the tree from the root, which has the theoretical runtime of $\mathcal{O}(n \log n)$. The experiment showed a similar mismatch as to the DAG building, leading us to conjecture that the overhead in runtime stems from the Haskell, or general computer, environment.

Chapter 3

External Memory Fully Persistent Search Trees

This chapter discusses the paper “External Memory Fully Persistent Search Trees”, which was published at STOC’23 [43]. This paper was created in collaboration with Gerth Stølting Brodal and Rolf Svenning from Aarhus University. The paper is included in Chapter 11.

In our work, we create a *fully persistent* search tree in external-memory, matching the I/O costs of the ephemeral B-trees of Bayer and McCreight [15]. The best previous result of Brodal, Sioutas, Tsakalidis, and Tsihlias [46] had an additive term of $\mathcal{O}(\log_2 B)$ over the update time of the B-tree, which our work rids off. See Table 3.1 for an overview of the related work.

In the internal-memory solution to fully persistent search trees, the dynamic order maintenance structure of Dietz and Sleator [57] was applied to allow for comparisons between versions, by their location in the preorder traversal of the version tree. Using this structure allows for comparisons between the order of versions in the preorder traversal list in $\mathcal{O}(1)$ internal time, but porting this to the external-memory model then adds a cost of $\mathcal{O}(1)$ I/O to each comparison. This is in essence the source of the additive $\mathcal{O}(\log_2 B)$ overhead in the construction of Brodal, Sioutas, Tsakalidis, and Tsihlias [46]. The result of this paper is obtained by embracing that comparisons between versions are not directly allowed, and instead applying several other tricks to allow for indirect comparisons, sufficient to support the stated I/O costs.

The result of the paper is obtained through three layers of construction. For the first layer, some subtree of the version tree is maintained, which allows for at most $\bar{N}$ further clones and updates. Secondly, the bound of maximum $\bar{N}$ updates is removed, by cutting the subtree into smaller trees after $\bar{N}$ updates have been performed, using global rebuilding to rebuild the first layer of the structure for each newly obtained tree. By carefully choosing $\bar{N}$, we can ensure that for all versions v contained in the subtree that $N_v = \Theta(\bar{N})$. Finally, clone operations are made *lazy*, to allow them to only affect the structure when an update in the cloned version occur, lowering their

Table 3.1: I/O bounds for previous and new results for partial and fully persistent search trees in external memory. Here, N denotes the number of updates, N_v denotes the number of elements contained in the set at version v , and K denotes the number of reported elements. All structures use linear space in N . This table corrects the wrongly stated I/O costs depicted in [43, Table 1] for the results of [16, 132, 13].

	Reporting	Update
Ephemeral		
Bayer and McCreight [15]	$\mathcal{O}(\log_B N + K/B)$	$\mathcal{O}(\log_B N)$
Partial persistence		
Lomet and Salzberg [94]	$\mathcal{O}(N_v/B)$	$\mathcal{O}(\log_B N_v \log_B N)$
Becker, Gschwind, Ohler, Seeger, and Widmayer [16]	$\mathcal{O}(\log_B N_v + K/B)$	$\mathcal{O}(\log_B N_v)$
Varman and Verma [132]		
Arge, Danner, and Teh [13]		
Full persistence		
Lanka and Mays [89]	$\mathcal{O}((\log_B N_v + K/B) \log_B N)$	$\mathcal{O}(\log_B^2 N_v)$ am.
Brodal, Sioutas, Tsakalidis, and Tsihlias [46]	$\mathcal{O}(\log_B N_v + K/B)$	$\mathcal{O}(\log_B N_v + \log_2 B)$ am.
<i>Our Result</i> [43]	$\mathcal{O}(\log_B N_v + K/B)$	$\mathcal{O}(\log_B N_v)$ am.

I/O cost to $\mathcal{O}(1)$. In this overview I will focus on a part of the first structure.

3.1 Interface of a Fully Persistent Search Tree

The data structure of the paper supports the operations of a fully persistent search tree, which are stated in [43, Section 1.1], and restated below. We let $\mathcal{S}_v$ denotes the set of elements contained in version v . The versions form a *version tree*, where we allow updates (insertions and deletions) only to the leafs, but allow queries and clones to any version. We denote the sets that allow updates as *unlocked*, and those that do not allow updates as *locked*. Further, we denote N_v as the size of the set at version v , i.e., $N_v = |\mathcal{S}_v|$. The supported operations are as stated below.

CLONE(v) Creates a new version w , where $\mathcal{S}_w = \mathcal{S}_v$. Returns the new version identifier w . Version w becomes a child of v in the version tree T , i.e., after the operation version v is locked and version w is unlocked.

INSERT(v, x) Adds x to $\mathcal{S}_v$. Requires version v is unlocked.

DELETE(v, x) Removes x from $\mathcal{S}_v$. Requires version v is unlocked. If x is not contained in $\mathcal{S}_v$, nothing is changed.

RANGE(v, x, y) Returns all values in $\mathcal{S}_v \cap [x, y]$ in increasing order.

$\text{SEARCH}(v, x)$ Returns the predecessor of x in $\mathcal{S}_v$.

Note that a regular B-tree support all but the CLONE operation using $\mathcal{O}(\log_B N)$ I/Os, when it contains N values, with RANGE using additional $\mathcal{O}(K/B)$ I/Os when reporting K elements. The goal is therefore to add the CLONE operation to the set of supported operations.

3.2 Geometric Interpretation

The structure of the paper build on a geometric interpretation of fully persistent search trees, which is illustrated in [43, Figure 1] and restated in Figure 3.1. The geometric interpretation is a plane, where the x -axis is the value axis, and the y -axis is the version axis. The version order from bottom to top is obtained from a *preorder traversal* of the version tree. Each value contained in the structure is then represented as a *segment* in the plane $\{x\} \times [v, u[$, which covers a single point x on the value axis, but may span multiple versions on the version axis from v to but not including u . This corresponds to all versions from v to but not including u containing the value x . When performing a CLONE operation of some version v to obtain version w , this corresponds to inserting w immediately after v in the ordering. This then extends all segments spanning over v , which allows version w to contain all values of the old version, i.e., $\mathcal{S}_w = \mathcal{S}_v$. A SEARCH query for a value in a version then correspond to a horizontal ray from the corresponding point in the plane, finding the first segment crossing the ray, and a RANGE query corresponds to reporting all values of segments crossing a horizontal line of the plane.

To efficiently support queries in the plane, the plane is partitioned into axis-aligned *rectangles*. This partition can be seen in [43, Figure 2], which is restated in Figure 3.2. Central to the partition is the parameter

$$R = B \log_B \bar{N}.$$

The intuition of R is that a list of $\mathcal{O}(R)$ elements may be scanned using $\mathcal{O}(R/B) = \mathcal{O}(\log_B \bar{N})$ I/Os. We therefore impose the invariant that each rectangle contains at most $2R$ segments. To support RANGE queries, we further impose the invariant that each rectangle contains at least $R/4$ *spanning segments*, i.e., segments which spans the whole height of the rectangle. This ensures that a constant fraction of all segments stored in the rectangle are contained in all versions. Any RANGE query which spans the whole width of the rectangle therefore reports asymptotically the same number of values as the number of segments scanned. Another consequence of the spanning segments is that if a SEARCH query does not locate a return value in the first rectangle scanned, it must find some value in the neighboring rectangle, and therefore constantly many rectangles partake in the query.

Locating a rectangle containing a given query point (x, v) is non-trivial, as the y -axis does not allow for comparisons. We solve this by the structure presented in [43, Section 5], which allows for point location queries. Inside each rectangle, the

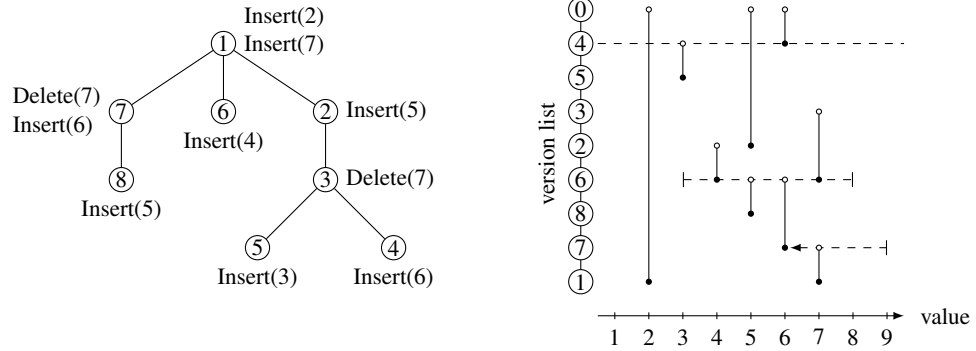


Figure 3.1: (Reordering of [43, Figure 1]) (Left) Version tree illustrating 8 versions of a set after $N = 10$ updates. Versions 1, 2, 3, and 4 contain the sets $\{2, 7\}$, $\{2, 5, 7\}$, $\{2, 5\}$, $\{2, 5, 6\}$, respectively. Updates to a version are shown next to the node of the version tree. (Right) The version list, consisting of a left-to-right preorder traversal of the version tree terminated by 0, and the half-open liveness intervals of versions containing a value. Note that the value 7 is only contained in versions 1, 6, and 2 of the set. The topmost dashed line shows that version 4 of the set contains $\{2, 5, 6\}$, the dashed line segment at version 6 shows that the range reporting query $\text{RANGE}(6, 3, 8)$ has result $\{4, 7\}$, and the bottom most dashed arrow that the query $\text{SEARCH}(7, 9)$ has result 6.

segments must be checked for containing a given version. This is solved by storing with each version in the rectangle, a local ordered value, such that comparisons are possible between versions contained in the rectangle. To then locate where the queried version falls in this local order, a structure presented in [43, Section 4] is constructed. For both structures, the query operations are supported using amortized $\mathcal{O}(\log_B \bar{N})$ I/Os, with insertions of new rectangles being supported using $\mathcal{O}(R \log_B \bar{N})$ I/Os. In this dissertation I will not go into further details with these structures.

3.3 Maintaining the Partition

When inserting or deleting values from a version, this corresponds to adding or removing a segment of unit length from the plane. By locating the rectangle covering this unit segment, the update may be applied. This may then break the invariants of the rectangle, as it then may contain too many segments or not enough spanning segments. The partition may therefore need to be locally updated. The rectangles are allowed to be updated in three primitive ways, which are summarized in [43, Figure 3], and restated in Figure 3.3. The rectangle may split or join with a neighbor. Notice that splits are allowed both horizontally and vertically, but joins are not allowed horizontally, as this may otherwise convert spanning segments to non-spanning, which is undesired. As the two rectangle must perfectly align to perform a vertical join, which is the operation allowing a rectangle to receive more spanning segments, a

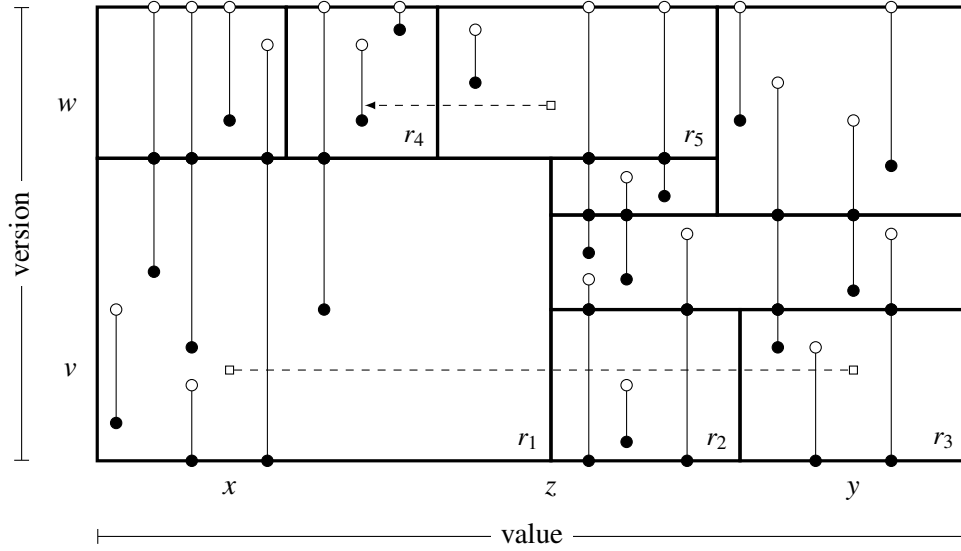


Figure 3.2: ([43, Figure 2]) The partitioning of the plane into rectangles, a query RANGE (v, x, y) represented by the dashed line between two square endpoints spanning rectangles r_1 , r_2 , and r_3 , and a query SEARCH (w, z) represented by the dashed arrow originating from a square endpoint, spanning rectangles r_5 and r_4 . Note that black dots on vertical segments correspond to the upper endpoint of the segment in the rectangle below and the lower endpoint of the segment in the rectangle above.

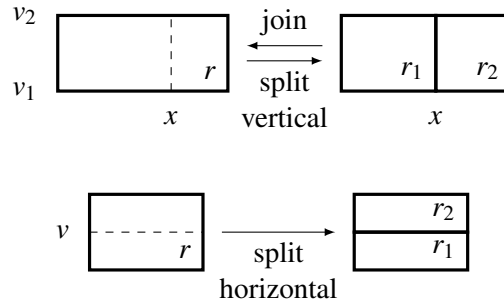


Figure 3.3: ([43, Figure 3]) Primitive rectangle transformations: Vertical split and join at value x , and horizontal split at version v .

merge rebalancing operation is constructed from the primitives, which is shown in [43, Figure 4], and restated in Figure 3.4. To perform efficient merges, a rectangle must have constantly many neighbors, which is introduced as another invariant. We let t_N denote the constant bound on the number of neighbors to one side a rectangle may have, where the value of t_N will be decided later.

When a rectangle begins to contain too many segments, we denote this as an *overflow potential*. The too many segments may either be many spanning segments, and otherwise there must be many endpoints of segments inside the rectangle. For

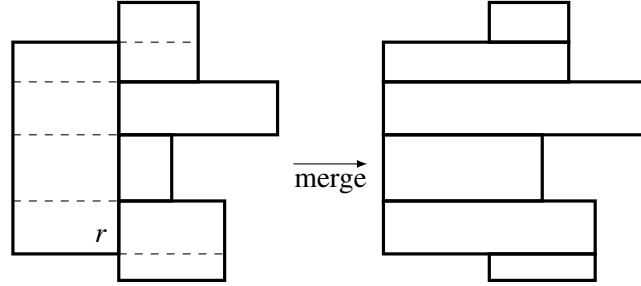


Figure 3.4: ([43, Figure 4]) Merging of a rectangle r with its right neighboring rectangles. The dashed lines represent the location that the rectangles are horizontally split, before they are vertically joined.

both types, the cutoff from when a rectangle has overflow potential is beyond the first $R/2$ spanning segments or endpoints. Symmetrically, when a rectangle begins to contain too few spanning segments, we denote this as an *underflow potential*, which is when a rectangle contains less than $R/2$ spanning segments. The potentials do not mean that the invariants of the rectangle are broken, but only that the rectangle is on the way to break them.

When some invariant is broken, then the rebalancing operations, when applied, will fix the broken invariant of the current rectangle. However, it may then break another invariant of the rectangle or one of the neighboring rectangles. In order to show that the rebalancing operations does not cascade forever, an involved amortized analysis was performed.

There are four types of rebalancing, triggered by four different cases. There may be more than $2R$ segments, in which case the rectangle splits into two new rectangles. If the rectangle contains at least R spanning segments, then it splits vertically, allowing the two new rectangles to both have at least $R/2$ spanning segments. Otherwise, the rectangle splits horizontally, distributing the internal endpoints of the segments equally between the two rectangles. There may be less than $R/4$ spanning segments, in which case the rectangle is merged with the neighbors. And finally there may be more than t_N neighbors, in which case the rectangle splits horizontally to distribute the neighbors over two new rectangles.

Upon performing a merge on a rectangle r , the top and bottom of the right neighbors split before being merged with r . If any of these rectangles contain an underflow, this underflow is briefly duplicated, but after joining one of the new rectangles into some part of r , this duplication of underflow disappear. If however, there only is a single right neighbor r' , then r' must be cut twice to align with r . This triples the potential underflow, and only removes it once when merging a part of r' with r , leaving a duplication in underflow. To remove this underflow duplication, one of the remaining parts of r' is merged to the left, where there by definition cannot be a single neighbor which is cut twice, as r cut r' twice from the left, thereby ensuring that the merge operation may terminate. This merge is symmetrical to the merge right. From this extra variant of the merge operation, it must be further imposed that the

number of neighbors on both sides must be bounded by t_N .

Note that we do not actually need the left and rightmost rectangles of the plane to contain at least $R/4$ spanning segments, as they are never covered entirely by a RANGE query. Due to the merge operation primarily merging with rectangles towards the right, we can therefore safely remove the invariant on spanning segments for rightmost rectangles of the plane, as any right neighbor of r either has enough spanning segments, or is a rightmost rectangle which after merging lets the rectangles of r be rightmost rectangles. By similar argument, we can remove the invariant on spanning segments on the leftmost rectangles of the plane, and in the construction simply not perform merge operations on the leftmost rectangles when there is less than $R/4$ spanning segments. However, removing the spanning invariant on the leftmost rectangles complicates the analysis needlessly, so we let the invariant remain.

3.4 Amortized Case Analysis

The amortized analysis stated in [43, Section 3.3-3.6] contains a more involved set of potential functions as well as four additional rebalancing operations. Initially, the potential functions were more simple, and only the previously stated four rebalancing operations was used. In this section I will present the initial analysis, and the flaws it contained, which prompted us to update the analysis, to the one stated in the paper.

We shall assign for each of the four possible rebalancing operations a potential function, which grows when the rectangle is on its way to break the invariant which triggers the operation to be applied. These four potential functions for a rectangle r was initially as follows.

$$\begin{aligned}\Phi_O^r &= \max \left\{ 0, \frac{S_r}{R} - \frac{1}{2} \right\} \\ \Phi_I^r &= \max \left\{ 0, \frac{I_r}{R} - \frac{1}{2} \right\} \\ \Phi_U^r &= \begin{cases} 0 & \text{if } y^r = +\infty \\ \max \left\{ 0, \frac{1}{2} - \frac{S_r}{R} \right\} & \text{o/w} \end{cases} \\ \Phi_N^r &= \max \left\{ 0, N_r^L - \frac{1}{2}t_N \right\} + \max \left\{ 0, N_r^R - \frac{1}{2}t_N \right\}\end{aligned}$$

Here, S_r denotes the number of spanning segments in r , I_r denotes the number of internal endpoints of segments, and N_r^L and N_r^R denotes the number of left and right neighbors of r , respectively. Note that the underflow potential Φ_U^r of any rightmost rectangle is 0, as there is no spanning requirement on these rectangles. The use of max ensures that the functions are never negative. In general the potential function can be interpreted as *placing potential* onto elements of the structure, e.g., for Φ_O^r the function can be viewed as placing $1/R$ potential on all, but the first $R/2$, spanning segments of the rectangle. We then create for each of the four potentials of a rectangle a *subpotential function*, which is the sum of the given potential over all rectangles, i.e., define $\Phi_O = \sum_r \Phi_O^r$, $\Phi_U = \sum_r \Phi_U^r$, $\Phi_I = \sum_r \Phi_I^r$, and $\Phi_N = \sum_r \Phi_N^r$. The master

Table 3.2: Initially analyzed difference in the subpotential functions for the different update operations and rebalancing operations. For the horizontal split and neighbor split the operation is analyzed both for rectangles without and with underflow potential.

Operation	$\Delta\Phi_N$	$\Delta\Phi_O$	$\Delta\Phi_U$	$\Delta\Phi_I$
Insert	0	$\leq 1/R$	≤ 0	$\leq 2/R$
Delete	0	≤ 0	$\leq 1/R$	$\leq 2/R$
Vertical Split	0	$-1/2$	0	≤ 0
Horizontal Split	≤ 4	≤ 4	≤ 0	$-1/2$
	≤ 4	0	$\leq 1/4$	$-1/2$
Neighbor Split	$\leq -t_N/2 + 2$	$\leq 1 + 1/2$	≤ 0	≤ 0
	$\leq -t_N/2 + 2$	0	$\leq 1/4$	≤ 0
Merge	$\leq t_N/2 + 2$	$\leq 2t_N$	$\leq -1/4$	$\leq 1/2$

potential function is then

$$\Phi = (c_O \cdot \Phi_O + c_U \cdot \Phi_U + c_I \cdot \Phi_I + c_N \cdot \Phi_N) \cdot R \cdot \log_B \bar{N}.$$

When rebalancing, the rectangles must be updated in the secondary structures, which as previously stated uses amortized $\mathcal{O}(R \log_B \bar{N})$ I/Os. A rebalancing operation must therefore make the parenthesized term of Φ decrease by 1 to free potential to pay for the operation. Similarly, upon an INSERT or DELETE operation, without considering the possible rebalancing, the term may increase by at most $\mathcal{O}(1/R)$, to preserve that the updates initially use $\mathcal{O}(\log_B \bar{N})$ I/Os. The goal is therefore to compute a set of constants t_N, c_O, c_U, c_I , and c_N , s.t. this increase and decrease of the term occur.

To compute what the constants should be, notice that the parenthesized term may be viewed as a linear constraint on the subpotential functions. By then analyzing for each of the update and rebalancing operations, how each of the four subpotential functions change, a set of linear constraints is constructed. To then find the set of constants, the constraints may be solved as a linear program. The analysis is therefore reduced to a case analysis, where the computed values are summarized in Table 3.2.

3.4.1 Infeasibility of the Amortized Analysis

After reaching this point of the analysis, we discovered two flaws, which results in the set of linear constraints being infeasible to solve. The first flaw can be seen, by isolating part of Table 3.2, as below.

Operation	$\Delta\Phi_U$	$\Delta\Phi_I$
Horizontal Split	$\leq 1/4$	$-1/2$
Merge	$\leq -1/4$	$\leq 1/2$

Notice, that no matter what constants are chosen for c_U and c_I , both of these two constraints can never both be bounded by -1 . In essence a *cycle* is created between

the two constraints. Therefore, a change is required in the analysis. The issue is that a horizontal split may duplicate underflow potential, but that merging rectangles may increase the potential of internal points. However, upon a merge, the internal points are moved from a rectangle with few spanning segments to a rectangle with many spanning segments. As there are many spanning segments, the rectangle may be split vertically later to solve overflow issues. Or viewed differently: having many internal endpoints in a rectangle is only a problem if there are few spanning segments. We shall therefore update the potential function for the internal points, such that it scales by the number of spanning segments. The four potential functions are replaced by the following. Note that only Φ_I^r is updated to include the scale value for all non-rightmost rectangles of the plane.

$$\begin{aligned}\Phi_O^r &= \max \left\{ 0, \frac{S_r}{R} - \frac{1}{2} \right\} \\ \Phi_I^r &= \begin{cases} \max \left\{ 0, \frac{I_r}{R} - \frac{1}{2} \right\} & \text{if } y^r = +\infty \\ \max \left\{ 0, \left(\frac{I_r}{R} - \frac{1}{2} \right) (9 - 16 \cdot \min \left\{ \frac{S_r}{R}, \frac{1}{2} \right\}) \right\} & \text{otherwise} \end{cases} \\ \Phi_U^r &= \begin{cases} 0 & \text{if } y^r = +\infty \\ \max \left\{ 0, \frac{1}{2} - \frac{S_r}{R} \right\} & \text{otherwise} \end{cases} \\ \Phi_N^r &= \max \left\{ 0, N_r^L - \frac{t_N}{2} \right\} + \max \left\{ 0, N_r^R - \frac{t_N}{2} \right\}\end{aligned}$$

Note that the scale value $9 - 16 \cdot \min \left\{ \frac{S_r}{R}, \frac{1}{2} \right\}$ of Φ_I^r is bounded in the interval $[1, 5]$, which gives just enough slack to remove the observed cycle. However, Table 3.2 contains yet another cycle, which is isolated below.

Operation	$\Delta\Phi_N$	$\Delta\Phi_U$
Neighbor Split	$\leq -t_N/2 + 2$	$\leq 1/4$
Merge	$\leq t_N/2 + 2$	$\leq -1/4$

The issue is that performing a neighbor split on a rectangle to decrease the number of neighbors duplicates the underflow potential of the rectangle. When merging a rectangle, all the neighbors on the right becomes neighbors of the rectangles on the left side of the merged rectangle, which may increase the neighbors by a lot. This cannot be fixed by altering the potential functions, as it relates to the structure of the partition and not to the inner segments of the rectangles. Therefore, four new operations for merging was constructed, which ensures that the neighbor increase was minimized, to rid of the cycle. These are the merging operations stated in [43, Figure 5], which is restated in Figure 3.5.

The intuition behind the extended merging operations are as follows. Let r denote the rectangle which is to be merged. Note that r may not be a rightmost rectangle, as these do not need to be merged. If r is a leftmost rectangle, as there are no rectangles on the left, then the right neighbors of r is not moved to another rectangle, and the regular merge operation in Figure 3.4 is applied.

Otherwise, r contains neighbors on both sides. If some side of r has two neighbors, then applying the regular merging operation will not cause some neighboring

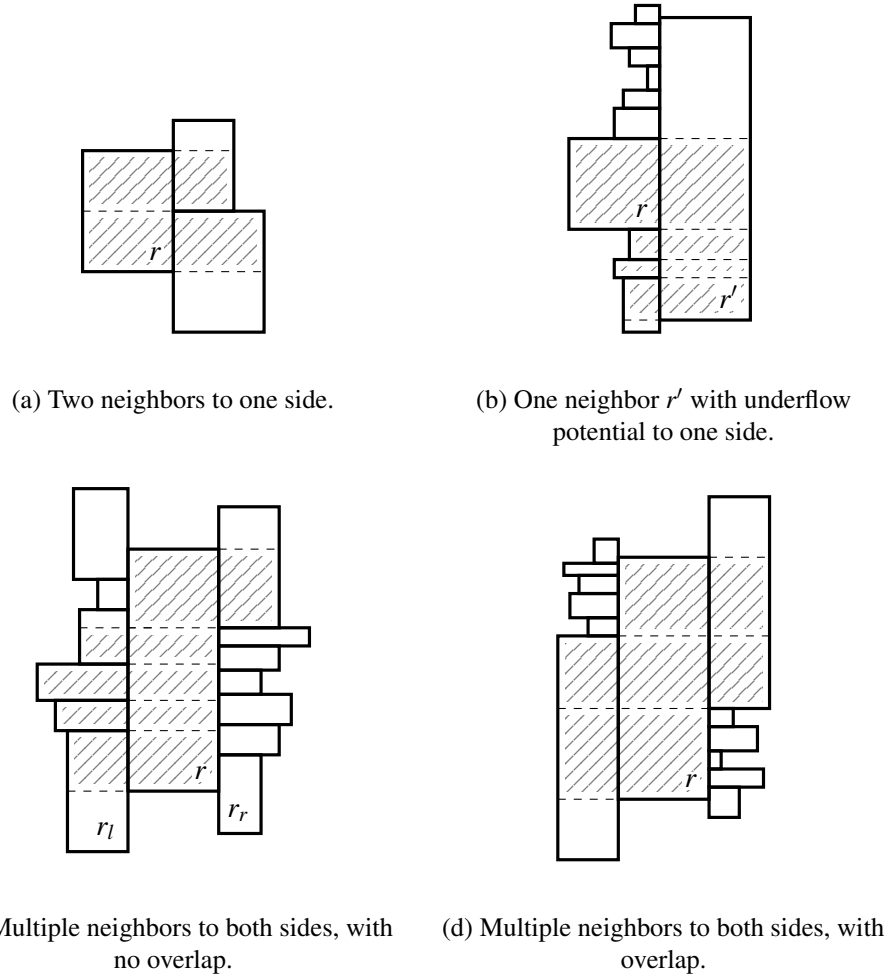


Figure 3.5: ([43, Figure 5]) The different merge cases, with the underflowing rectangle being r , apart from when r is a leftmost rectangle. The dashed lines represent the location where the rectangles are horizontally split. The grayed areas indicate the newly joined rectangles.

underflow potential to be duplicated, and many neighbors cannot be moved. This is the merging operation in Figure 3.5a. If instead both sides have many neighbors, then the top and bottom neighbor on both sides fence off the intermediate neighbors. Each of these intermediate neighbors only has r as a neighbor, and as the merge will remove r , then joining up the two sides of r will not increase the overall number of neighbors. This is the merging operation in Figure 3.5c. The argument relies on the initial top split of r being on the extreme of two top neighbors, and symmetrically for the bottom split. If these extreme neighbors overlap, the merge operation is instead stated in Figure 3.5d. Finally, it must hold that some side of the rectangle has a single neighbor. This operation follows closely the variant of the regular merge operation,

Table 3.3: ([43, Table 3]) Final analyzed difference in the subpotential functions for the different update operations and rebalancing operations. Note that parts of the first column have been omitted to fit the table.

Operation	$\Delta\Phi_N$	$\Delta\Phi_O$	$\Delta\Phi_U$	$\Delta\Phi_I$
Insert	0	$\leq 1/R$	≤ 0	$\leq 10/R$
Delete	0	≤ 0	$\leq 1/R$	$\leq 58/R$
Vertical split	0	$-1/2$	0	≤ 0
Horizontal split				
$(\Phi_U^r = 0)$	≤ 2	≤ 6	0	$-1/2$
$(\Phi_U^r > 0)$	≤ 2	≤ 6	$\leq 1/2 - S_r/R$	$\leq -1/2 \cdot (9 - 16S_r/R)$
Neighbor split	$\leq -t_N/2 + 1$	$\leq 3/2$	$\leq 1/4$	≤ 0
Merge				
(Figure 3.5a)	≤ 3	$\leq 5 + 1/4$	$\leq -1/4$	$\leq 1/2$
(Figure 3.5b)	$\leq t_N/4 + 2$	$\leq t_N + 3 + 1/4$	$\leq -1/4$	$\leq 1/2$
(Figure 3.5c)	≤ 3	$\leq 2t_N + 3 + 3/4$	$\leq -1/4$	$\leq 1/2$
(Figure 3.5d)	≤ 4	$\leq 10 + 3/4$	$\leq -1/4$	$\leq 1/2$
(Figure 3.4)	≤ 2	$\leq 2t_N + 1 + 1/4$	$\leq -1/4$	$\leq 1/2$

which performs two merges. The difference is that the part of the neighbor r' chosen for the extra merge, is the part containing the fewest neighbors towards the side containing r . Of all the merge operations, this is the operations which may move the most neighbors. By careful analysis, it can be shown that the operation only allows increasing the neighbor potential by at most roughly $t_N/4$. Note that if r is a leftmost rectangle with only a single right neighbor, this case must also be applied.

3.4.2 Final Amortized Analysis

With the updated potential functions and merging operations, the case analysis on the change in potential was remade, leading to the computed values of [43, Table 3], which is restated in Table 3.3.

By trying increasing values for the constant t_N , a linear program can be used to solve for the remaining constants. A set of constants, that allows each update to increase the term of the potential by $\mathcal{O}(1/R)$ and for each rebalancing operation to decrease the term by at least 1 is

$$t_N = 22 \quad c_N = 573 \quad c_O = 2 \quad c_U = 22904 \quad c_I = 2754.$$

3.5 Corrections to the Paper

After the paper [43] was published, we found a few smaller errors. As previously stated, some I/O costs stated in [43, Table 1] of the related work on partially persistent

Table 3.4: Updated difference in the subpotential functions for the different update operations and rebalancing operations.

Operation	$\Delta\Phi_N$	$\Delta\Phi_O$	$\Delta\Phi_U$	$\Delta\Phi_I$
Insert	0	$\leq 1/R$	≤ 0	$\leq 10/R$
Delete	0	≤ 0	$\leq 1/R$	$\leq 58/R$
Vertical split	0	$-1/2$	0	≤ 0
Horizontal split				
$(\Phi_U^r = 0)$	≤ 2	$\leq 5 + 1/2 + 1/R$	0	$-1/2$
$(\Phi_U^r > 0)$	≤ 2	$\leq 5 + 1/2 + 1/R$	$\leq 1/2 - s_r/R$	$\leq -1/2 \cdot (9 - 16s_r/R)$
Neighbor split	$\leq -t_N/2 + 1$	$\leq 3/2$	$\leq 1/4$	≤ 0
Merge				
(Figure 3.5a)	≤ 3	$\leq 5 + 1/4$	$\leq -1/4$	$\leq 1/2 + 8/R$
(Figure 3.5b)	$\leq t_N/4 + 2$	$\leq t_N + 3 + 1/4$	$\leq -1/4$	$\leq 1/2 + 8/R$
(Figure 3.5c)	≤ 3	$\leq 2t_N + 3 + 3/4$	$\leq -1/4$	$\leq 1/2 + 8/R$
(Figure 3.5d)	≤ 4	$\leq 10 + 3/4$	$\leq -1/4$	$\leq 1/2 + 8/R$
(Figure 3.4)	≤ 2	$\leq 2t_N + 1 + 1/4$	$\leq -1/4$	$\leq 1/2 + 8/R$

were wrong. This was discovered when we later worked on “Buffered Partially-Persistent External-Memory Search Trees” [44]. In Table 3.1 the wrongfully stated I/O costs are corrected, to match [44, Table 1].

A few weeks after submitting the final version of this paper [43], we found a small off-by-one error in the case analysis of the amortized argument, regarding the potential of the internal endpoints during the merge operations. Here, we assumed that the newly made rectangles would have $R/2$ spanning segments, when in fact they may have as few as $R/2 - 1$ spanning segments. The scale value placed on the internal points is therefore not 1, but may be as large as $1 + 16/R$. Therefore, the increase in the potential $\Delta\Phi_I$ is not bounded by $1/2$, but by $1/2 + 8/R$. Improving the analysis for the horizontal splits, to also include a $\Theta(1/R)$ term, the updated analysis leads to the computed values of Table 3.4. As R is not a constant, the linear program may not immediately be used to solve the system. Instead, it must be assumed that $R \geq c$, for some constant c . If $R < c$, then it holds that $\bar{N} = \mathcal{O}(1)$, and the structure may be replaced by a list of all updates, and each operation is supported in $\mathcal{O}(1)$ I/Os.

Choosing $R \geq 2$ for the R used in the constraints created by horizontal splits, and $R \geq 32$ for the merge constraints lead to the following constants.

$$t_N = 34 \quad c_N = 822 \quad c_O = 2 \quad c_U = 52592 \quad c_I = 5922.$$

Note that this results in for constructions where $R < 32$, the structure becomes a simple list of updates. The updated constants only have an impact on the construction for t_N , as the other constants are not parts of the construction but only the analysis.

Choosing different values of R for the different constraints in solving the linear program was only done to obtain integer constants. If choosing $R \geq 32$ for both

constraints, the constants are instead

$$t_N = 34 \quad c_N = 819.1875 \quad c_O = 2.0 \quad c_U = 52412.0 \quad c_I = 5901.375 ,$$

which are only marginally less worst-case.

Chapter 4

Buffered Partially-Persistent External-Memory Search Trees

This chapter discusses the paper “Buffered Partially-Persistent External-Memory Search Trees”, which was published at ESA’25 [44]. This paper was created in collaboration with Gerth Stølting Brodal and Rolf Svenning from Aarhus University. Due to page constraints, the conference version of the paper omits the appendix, which includes a refined proof of the *subtraction game* [58] and *zeroing games* [59, Theorem 5]. The proof improves the previous upper bounds by a small constant, which improves our analysis. The included version in this dissertation is therefore the arXiv version, which is identical to the conference version, but includes the appendix [45]. The paper is included in Chapter 12. Extensions to the paper is included in Chapter 18.

In the world of I/O efficient data structures, the B-tree of Bayer and McCreight [15] is a simple structure, used to maintain an ordered set, supporting both updates and queries to the set using $\mathcal{O}(\log_B N)$ I/Os for a set of N keys. The I/O cost for a single query is efficient, following an argument similar to the optimality of binary search [56, Theorem 7]. However, the cost of updates may be improved. Upon inserting a single key into the tree, $\mathcal{O}(\log_B N)$ blocks are traversed to insert the new key into some leaf, preserving the order. At each layer of the tree, an entire I/O is used to move only a single key, effectively only using a $1/B$ fraction of the power of a block move. If multiple keys are to be moved in the same direction, then by waiting, this single I/O may be used by multiple updates, thereby spreading the I/O cost over multiple insertions.

This is exactly the idea of the *buffer tree* by Arge [8, 11], where each node contains a buffer of up to M delayed updates. The buffer tree leads to an I/O efficient sorting algorithm, where inserting all N keys of a list and then traversing the tree allows sorting with the optimal $\mathcal{O}\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/O cost. In contrast, the regular B-tree allows sorting using $\mathcal{O}(N \log_B N)$ I/Os, which is not optimal. The buffer tree comes with a cost, which is that single queries becomes batched, and are therefore not guaranteed to be answered immediately.

By using smaller buffers, a tradeoff between the query and update time can be obtained, where queries are answered immediately. This structure is the B^ε -tree by Brodal and Fagerberg [32], which for a tradeoff parameter $0 < \varepsilon < 1$, creates a tree where each node contains a buffer of size B , and the degree of each node is $\Theta(B^\varepsilon)$. When a buffer is full, then by the pigeonhole principle, at least one child must receive at least $B/B^\varepsilon = B^{1-\varepsilon}$ of the updates in the buffer, which can then be flushed to the child. This leads to a structure, where updates use amortized $\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N)$ I/Os, and queries use amortized $\mathcal{O}(\frac{1}{\varepsilon} \log_B N)$ I/Os.

Partial persistence allows for updates only to the current version, but queries to all previous versions. On the contrary, the buffering techniques inherently delay updates, s.t. updates are applied to the previous versions, but the queries are only valid on the current version. We manage to combine these conflicting behaviors, to achieve I/O costs matching the B^ε -tree, while supporting partial persistence, in both the amortized and worst-case setting. A table of related work and the new results of the paper is shown in [44, Table 1], and restated in Table 4.1.

4.1 Interface of a Partially-Persistent Search Tree

As the versions of partial persistence are simple, compared to full persistence, we do not introduce an explicit CLONE operation to create new versions, and instead let any update performed yield a new version of the maintained set. Each version is denoted by a unique version identifier, which are incrementing integer values, allowing for order comparisons. At version v , the set of keys is denoted as $\mathcal{S}_v$. We let v_c denote the most recent version, which is the version allowing updates, i.e., updates are only allowed to the set $\mathcal{S}_{v_c}$. The supported operations of a partial persistent search tree are stated in [44, Section 1.2], and restated below.

- INSERT(x) Creates $\mathcal{S}_{v_c+1} = \mathcal{S}_{v_c} \cup \{x\}$, increments v_c , and returns v_c .
- DELETE(x) Creates $\mathcal{S}_{v_c+1} = \mathcal{S}_{v_c} \setminus \{x\}$, increments v_c , and returns v_c .
- RANGE(v, x, y) Reports all keys in $\mathcal{S}_v \cap [x, y]$ in increasing order.
- SEARCH(v, x) Returns the successor of x in $\mathcal{S}_v$, i.e., $\min\{y \in \mathcal{S}_v \mid x \leq y\}$.

4.2 Geometric Interpretation

Similarly to our result on fully persistent search trees [43], this results builds on a geometric interpretation of persistence. We similarly let the x -axis be the key axis and the y -axis be the version access. This geometric interpretation is illustrated on ([44, Figure 1(left)]), and restated in Figure 4.1. The versions of a partial persistence form a list and not a tree, and therefore comparisons on the order of versions are easy, simply using the version number. This, contrary to the fully persistent result [43], allows for comparisons on the y -axis, which simplifies the problem. Additionally, new versions

Table 4.1: ([44, Table 1]) Overview of results on the I/O complexity of ephemeral and partially-persistent search trees. Results marked by “am.” hold amortized, and results marked by “rand.” are randomized and hold with high probability. All other bounds are worst case. The parameter ε must satisfy $0 < \varepsilon < 1$. All results assume $M = \Omega(B)$, further $\dagger$ assumes $B = \Omega(\log N)$ and $M = \Omega(\max\{B \log^{\Theta(1)} N, B^2\})$; $\ddagger$ assumes $M = \Omega(B \log_B N)$; and $*$ assumes $M = \Omega(B^{1-\varepsilon} \log_2(\max_v N_v))$. For both queries and updates in [17, 55], we include the multiplicative dependency on $\frac{1}{\varepsilon}$ (that can be omitted when treating ε as a constant), allowing, for example, setting $\varepsilon = \frac{1}{\log_2 B}$. All ephemeral results use space linear in N and all partial persistence results use space linear in the total number of updates.

	Range Query	Update
Ephemeral		
Bayer and McCreight [15]	$\mathcal{O}(\log_B N + K/B)$	$\mathcal{O}(\log_B N)$
Brodal and Fagerberg [32]	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N + K/B)$ am.	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N)$ am.
Bender, Das, Farach-Colton, Johnson, and Kuszmaul [†] [17]	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N + K/B)$	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N)$ rand.
Das, Iacono, and Nekrich [‡] [55]	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N + K/B)$	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N)$
Partial Persistent		
Becker, Gschwind, Ohler, Seeger, and Widmayer [16] Varman and Verma [132] Arge, Danner, and Teh [13]	$\mathcal{O}(\log_B N_v + K/B)$	$\mathcal{O}(\log_B N_v)$
Our Result [44, Theorem 1]		
Our Result [44, Theorem 2] [*]		
Our Result [44, Theorem 3]		
	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N_v + K/B)$ am.	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v)$ am.
	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N_v + K/B)$	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v)$
	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N_v + \gamma + K/B)$	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} (\frac{1}{\varepsilon} \log_B N_v + \gamma))$
	$\gamma = \text{sort}(B^{1-\varepsilon} \log_2 N_v) = \frac{\log_2 N_v}{B^\varepsilon} \log_{M/B} \frac{\log_2 N_v}{B^\varepsilon}$	

are only introduced at the top of the plane, which results on all segments below the top of the plane being static.

In the paper, we show how to use this geometric interpretation, to successfully buffer partial persistence, to match the amortized I/O bounds of B^ε -trees. Additionally, we show how to deamortize the structure, to achieve worst-case I/O bounds for a bounded and unbounded internal memory size M .

4.3 Partitioning the Plane

Following the geometric interpretation, we shall similarly to the fully persistent result [43], partition the plane into a set of axis aligned rectangles. The partition of

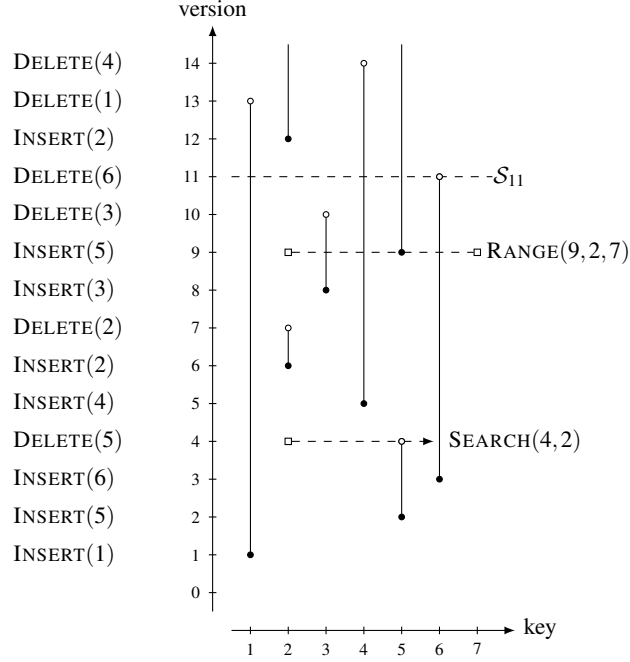


Figure 4.1: ([44, Figure 1(left)]) A list of updates performed on an initially empty set and the geometric interpretation of the updates. Vertical lines illustrate the half-open intervals of versions containing a key. Note that the key 3 is contained in versions $[8, 10[$, i.e., versions 8 and 9, whereas the key 2 is contained in version 6 and versions $[12, \infty[$. The topmost dashed line shows that version 11 of the set is $\mathcal{S}_{11} = \{1, 4, 5\}$, the dashed line segment at version 9 shows that the result of the query $\text{RANGE}(9, 2, 7)$ is $\{3, 4, 5, 6\}$, and the bottom most dashed arrow shows that the result of the successor search $\text{SEARCH}(4, 2)$ is 6.

the plane is illustrated in [44, Figure 1(right)], which is restated in Figure 4.2. As new versions are only created at the top of the plane, then maintaining the rectangles is easier, as only the top most rectangles can receive updates. We therefore denote the rectangles at the top of the plane as *open*, with the remaining rectangles being *closed*.

We maintain a B^ε -tree T over the top rectangles, and impose the invariant that all buffered updates in T must belong in the top open rectangles. We allow the first version at the bottom of the plane to be non-empty. Let $\bar{N}$ denote the size of the set in the first version in the structure. By allowing at most $\alpha \cdot \bar{N}$ updates for some parameter $0 < \alpha < 1$, then the size of each version v contained in the structure is $N_v = \Theta(\bar{N})$. To allow for more updates, we employ global rebuilding after $\alpha \cdot \bar{N}$ updates.

To avoid rounding errors, we define four central parameters, which are stated in [44, Section 2.2], and restated below.

$$\Delta = \lceil B^\varepsilon \rceil \quad H = 1 + \lceil \log_\Delta \bar{N} \rceil \quad F = \lceil B^{1-\varepsilon} \rceil \quad R = H \cdot 2\Delta \cdot F$$

We impose that the degree of the nodes of the B^ε -tree T is $< 2\Delta$. The buffers

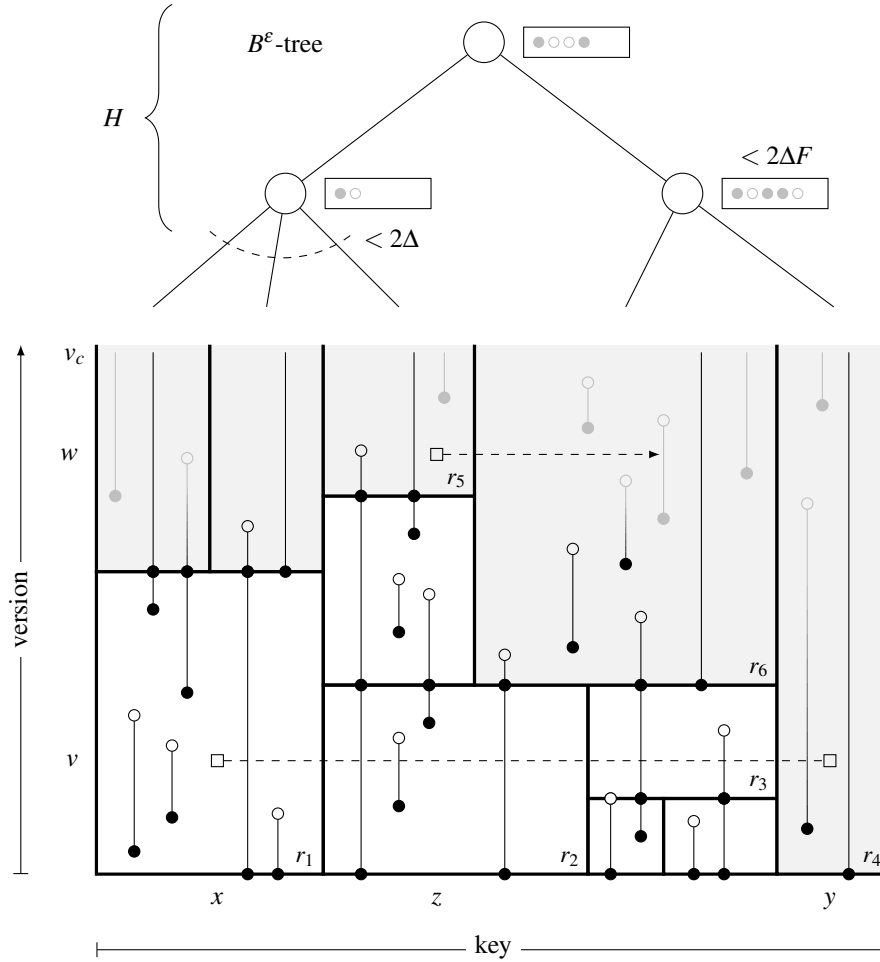


Figure 4.2: ([44, Figure 1(right)]) A B^ϵ -tree of the open rectangles and below the geometric interpretation of the updates, split into multiple rectangles, where gray rectangles are open rectangles. The black endpoints represent updates present in the rectangles and the gray endpoints represent buffered updates present in the buffers at the internal nodes of the B^ϵ -tree. A query $\text{RANGE}(v, x, y)$ is represented as the dashed line between two square endpoints, spanning rectangles r_1 , r_2 , r_3 , and r_4 , and a successor query $\text{SEARCH}(w, z)$ is represented as the dashed arrow from a square endpoint, spanning two rectangles r_5 and r_6 . Black dots on vertical segments correspond to the upper endpoint of the segment in the rectangle below and the lower endpoint of the segment in the rectangle above.

of each node contains $< 2\Delta F$ updates. Note that the buffer sizes are $\mathcal{O}(B)$. Upon a buffer overflow, we can then flush exactly F updates to some child node. In [44, Section 2.5], we show that the height of T is bounded by H .

Similarly to the fully persistent tree, we let each rectangle hold $\Theta(R)$ segments, with $\Omega(R)$ being spanning. Note that R is a bound on the number of buffered updates T may contain to a single open rectangle. By choosing the constants correctly, it must then hold that the buffered updates cannot delete everything inside a rectangle.

Note that $R/B = \Theta(H) = \Theta(\frac{1}{\epsilon} \log_B \bar{N})$, which accounts for the I/O cost of traversing down a path of the tree T , as well as scanning all segments of a rectangle. Upon a query, if the query touch a closed rectangle, then by invariant, all relevant segments must be contained in the rectangle, and the query may be answered by scanning the segments using $\mathcal{O}(R/B)$ I/Os. If, however, the query falls in an open rectangle, then some relevant updates may be buffered inside T . We therefore need to *actualize* the rectangle, bringing all updates into the rectangle. We show that this operation is supported in amortized $\mathcal{O}(H)$ I/Os. The query may then be answered regularly. If too many updates are moved to an open rectangle, we close the rectangle to ensure that the rectangle contains $\Omega(R)$ spanning segments. We denote this as *finalizing* the rectangle, and show that the operation is supported using amortized $\mathcal{O}(H)$ I/Os. New rectangles must then be opened to cover the range the now closed rectangle covered. By requiring that the newly opened rectangle contains $\Omega(R)$ initial keys at the bottom version, we can ensure the $\Theta(R)$ segments and $\Omega(R)$ spanning segments requirement. This may then require the range of the closed rectangle to be split in two new rectangles, or a merge with the neighboring rectangle, if too many or too few initial keys are present, respectively.

To support point location in the plane, we maintain for each version v a regular B-tree P_v over the rectangles present at version v . As there are at most constantly many changes to the rectangles between consecutive versions, we maintain the trees using partially persistence. As each rectangle contains $\Omega(R)$ unique updates, there are at most $\mathcal{O}(\bar{N}/R) = \mathcal{O}(\frac{\bar{N}}{BH})$ rectangles. Therefore, the simple *path-copying* technique is sufficient to maintain P_v , as each copy corresponds to a rectangle closing, and copies $\mathcal{O}(H)$ blocks, leading to a total space usage of $\mathcal{O}(\bar{N}/B)$ blocks.

4.4 Worst-Case Bounds

B^ϵ -trees only provide amortized bounds for the I/O cost. Notably, the cascading operations is when nodes contain too few children, in which case they are merged. This also requires merging the buffers, which may then overflow, leading to a cascading behavior. In contrast, splitting nodes, requires splitting the buffer, but this does not cascade. Deamortizing B^ϵ -trees have been studied both in the randomized and worst-case setting [17, 55]. These results manage to deamortize B^ϵ -trees, however, they both require a minimum bound on the memory size M as a function of the number of keys N , thereby bounding the value of N .

As we employ global rebuilding after $\alpha \cdot \bar{N}$ updates, our construction does not need to merge nodes. This therefore immediately leads to some worst-case behavior of the operations. In the paper we study these in [44, Section 3], where we show that worst-case I/O costs of B^ε -trees may be obtained with a weaker requirement on the memory bound. Additionally, we show that when no bound is given on M (other than $M \geq 2B$), we can obtain worst-case I/O costs of B^ε -trees with a small additive overhead. For both constructions, the structure presented is altered slightly.

First, we note that when the root buffer is flushed, F updates are removed from the buffer, and flushed down the path. The buffer may therefore first overflow after F new buffered updates are added to the root. We therefore spread the I/O cost of flushing down a root-to-leaf path out over the next F updates. As we have the I/Os to flush all the way to a leaf, we always flush at each level, potentially moving less than F updates at a time. We then apply the *subtraction game* of Dietz and Raman [58] on the buffers of the nodes, using the refinement of the bound from [45, Appendix], to show that at most $\mathcal{O}(F \log_2 \Delta)$ updates of a buffer may go to the same child, leading to not up to R buffered updates going to a specific leaf, but at most $\mathcal{O}(HF \log_2 \Delta) = \mathcal{O}(B^{1-\varepsilon} \log_2 \bar{N})$. Upon finalizing a rectangle, these updates may be found using $\mathcal{O}(H)$ I/Os. Assuming that $M = \Omega(B^{1-\varepsilon} \log_2 \bar{N})$, the updates may then be sorted in internal memory at no I/O cost, and then merged into the rectangle using $\mathcal{O}(R/B) = \mathcal{O}(H)$ I/Os. In total, the flushing down a path and finalizing uses $\mathcal{O}(H)$ I/Os. The F updates then perform $\mathcal{O}(H/F) = \mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B \bar{N})$ worst-case I/Os each. If a query happens during this partial flushing, the flushing is completed, and the query proceeds as normal, using that moving the buffered updates to the given rectangle use the internal memory sorting. In total a SEARCH query uses $\mathcal{O}(\frac{1}{\varepsilon} \log_B \bar{N})$ I/Os and a RANGE query uses $\mathcal{O}(\frac{1}{\varepsilon} \log_B \bar{N} + K/B)$ I/Os. Notably, the previous state-of-the-art worst-case B^ε -trees of Das, Iacono, and Nekrich [55], required that $M = \Omega(B \log_B N)$. As stated in [44, Section 3.2], our bound on M is therefore an improvement of a factor $\Theta(B^\varepsilon / \log_2 B)$.

To remove the assumption on the memory size, we first analyze how the I/O cost is affected by using external-memory sorting on the buffered updates to an open rectangle. For all operations, this introduces a small additive term. However, for RANGE queries, this term is multiplied by the number of reported keys K . To remove this multiplication, we convert from treating independently each rectangle a RANGE query touch, to instead treat all rectangles at once, and merge the updates down layer by layer. We then need to ensure that nodes of degree 1 has no buffered updates, to ensure a bound on the number of buffered updates at each layer. To do so, we impose the invariant that the buffer size of nodes scales with the degree of the node. Flushing a root-to-leaf path may then delete some child of some node. This lowers the buffer size, which may then need flushing. This then allows for the flushing to cascade, which is undesired for the worst-case result. We therefore change the construction, s.t. not the rectangle that receives the flush may be finalized, but instead the rectangle which has received the most updates is finalized. For the partial flushing, we finalize a rectangle after concluding the first flush, hereafter at most 4 more flushes are required to restore the invariant on the buffer size. By employing the *zeroing game* by Dietz and Sleator [59, Theorem 5], using the refinement of the bound from [45, Appendix],

we can show that each rectangle is still finalized after receiving $\Theta(R)$ updates.

4.5 Correction of the Stated Worst-Case Improvement

For the worst-case result, our bound on the memory size is $M = \Omega(B^{1-\varepsilon} \log_2 N_v)$. As stated in [44, Section 3.2], this yields an improvement factor of $\Theta(B^\varepsilon / \log_2 B)$, over the previous best bound $M = \Omega(B \log_B N)$ of Das, Iacono, and Nekrich [55]. This factor is, however, not guaranteed to be an improvement for all $0 < \varepsilon < 1$ and $B \geq 2$. For both our construction and the previous construction, the bound on M is imposed to allow sorting all buffered updates to some leaf in internal memory. However, the height of the tree is $\mathcal{O}(\frac{1}{\varepsilon} \log_B N)$ for both constructions. By including this missing term in the previous bound of Das, Iacono, and Nekrich, yielding the bound $M = \Omega(\frac{1}{\varepsilon} B \log_B N)$, our improvement factor then changes to $\Theta(\frac{B^\varepsilon}{\varepsilon \log_2 B})$. This is indeed an improvement, as it holds that $\frac{B^\varepsilon}{\varepsilon \log_2 B} \geq 1$ for all $0 < \varepsilon < 1$ and $B \geq 2$. This can be shown by noting that

$$\frac{B^\varepsilon}{\varepsilon \log_2 B} = \frac{2^{\varepsilon \log_2 B}}{\varepsilon \log_2 B}.$$

By then defining $x = \varepsilon \log_2 B$, the problem condenses to showing that $2^x/x \geq 1$ for all $x > 0$. The left-hand term is minimized at $x = 1/\ln 2$, where $2^x/x = e \ln 2 \approx 1.88 \geq 1$. This therefore concludes that our new bound is an improvement.

Chapter 5

Lazy B-Trees

This chapter discusses the paper “Lazy B-Trees”, which was published at MFCS’25 [110]. This paper was created in collaboration with Sebastian Wild from the University of Marburg. Due to page constraints, the conference version of the paper omits the appendix, which contains in detail our analysis of the *interval structure* in external-memory. The included version in this dissertation is therefore the arXiv version, which is identical to the conference version, but includes the appendix [111]. The paper is included in Chapter 13. The construction of the paper allows for a simplified external-memory priority-queue without altering the structure but only the update and query pattern, as stated in [110, Corollary 2]. If the structure should only be used as a priority-queue, the construction may be greatly simplified. This simplified construction is shown as an extension in Chapter 21.

In internal memory, the canonical solution to storing ordered elements, is using a *balanced binary search tree* (BBST), which allows for updating and querying a collection of n elements in $\mathcal{O}(\log n)$ time per operation [5, 74]. This collection maintains the elements in total order, and therefore maintains the rank of every element. If only the smallest (or symmetrically largest) element is needed to be known, i.e., the element of the smallest (largest) rank, instead a *priority queue* (PQ) may be used, which instead maintains only just enough information on the order, to allow for updates and queries in $\mathcal{O}(\log n)$ time [134]. Over time, the PQs have been improved to allow updates in $\mathcal{O}(1)$ time, and queries in $\mathcal{O}(\log n)$ time [65, 28, 39], demonstrating that maintaining less order leads to better runtimes. These structures are the extrema of maintaining a sorted order, which allows for accessing elements of any rank (BBST) or just the element of smallest (or largest) rank (PQ).

To create a tradeoff between these two extrema, Sandlund and Wild introduced *Lazy Search Trees* (LST), which allows for accessing any elements of any rank, but which only maintain the ordering w.r.t. the previously queried elements [112, 113]. The elements are partitioned around the queried elements into *gaps* Δ_i , s.t. the elements are ordered between the gaps and in relation to the queried elements, but the elements inside each of the gaps are unordered. When inserting an element x , which

Table 5.1: ([110, Table 1]) Overview of results on Search Trees and Priority Queues for both the internal- and external-memory model. For the internal model the displayed time is the number of RAM operations performed, while for the external model the displayed time is the number of I/Os performed. Amortized times are marked by “am.” and results doing batched updates and queries are marked by “batched”. For priority queues, query is only for the minimum. All results use linear space.

	Insert	Query
Internal-Memory		
Balanced BST [5, 74]	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Priority Queue [28, 39]	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$
Lazy Search Tree [112]	$\mathcal{O}(\log \frac{N}{ \Delta_i } + \log \log N)$	$\mathcal{O}(\log N + x \log c)$ am.
Optimal LST [114]	$\mathcal{O}(\log \frac{N}{ \Delta_i })$	$\mathcal{O}(\log N + x \log c)$ am.
External-Memory		
B-tree [15]	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\log_B N)$
B^ε -tree [32]	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N)$ am.	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N)$ am.
Buffer Tree [8, 11]	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ batched	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ batched
I/O-eff. heap [87]	$\mathcal{O}(\frac{1}{B} \log_2 \frac{N}{B})$ am.	$\mathcal{O}(\frac{1}{B} \log_2 \frac{N}{B})$ am.
x -treap heap [80]	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ am.	$\mathcal{O}(\frac{M^\varepsilon}{B} \log_{M/B}^2 \frac{N}{B})$ am.
<i>Our paper, LST</i> (Theorem 1)	$\mathcal{O}(\log_B \frac{N}{ \Delta_i } + \log_B \log_B \Delta_i)$	$\mathcal{O}(\log_B \min\{N, q\} + \frac{1}{B} \log_2 \Delta_i + \log_B \log_B \Delta_i + \frac{1}{B} x \log_2 c)$ am.
<i>Our paper, PQ</i> (Corollary 2)	$\mathcal{O}(\log_B \log_B N)$	$\mathcal{O}(\frac{1}{B} \log_2 N + \log_B \log_B N)$ am.

ends in gap Δ_i , LSTs support this operation in time $\mathcal{O}(\log \frac{n}{|\Delta_i|} + \log \log n)$. When a query is performed in some gap, either by element or rank, s.t. the gap is split into two new gaps of size x and cx , for $c \geq 1$, then LSTs support this operation in amortized $\mathcal{O}(\log n + x \log c)$ time. This initial LST structure had an additive overhead of $\mathcal{O}(\log \log n)$ time for insertions, w.r.t a static lower bound [112]. The construction was later improved by Sandlund and Zhang, which removed the additive overhead [114].

In external-memory, both balanced trees and priority queues have been tailored to be I/O efficient. However, a tradeoff construction between these, as given by the Lazy Search Trees, have not been studied. The result of this paper is an External-Memory Lazy Search Tree, yielding a construction with this tradeoff. The construction is based on the initial LSTs, and therefore contains a $\mathcal{O}(\log_B \log_B N)$ overhead. As a corollary, our LST construction allows for external PQs which support decrease key operations using $\mathcal{O}(\log_B \log_B N)$ I/Os. An overview of the related work and new results of the paper is presented in [110, Table 1], which is restated in Table 5.1.

5.1 Outline of Lazy Search Trees

To create External-Memory Lazy Search Trees, we opted for basing it on the initial construction of Sandlund and Wild [112, 113], which has the additive $\mathcal{O}(\log \log n)$ overhead. This is due to this construction being simpler in nature, and therefore easier to initially convert, than the later improved construction of Sandlund and Zhang [114].

Both constructions are based on a structure with two layers. The first layer is the *gap structure*, which stores the gaps, such that they may be efficiently queried and updated, based on the size of the gap. In essence, the larger the gap is, the faster it needs to be accessed. This is achieved by a *biased search tree*.

The second layer is the *interval structure*, which differs between the two versions, and it is the part of the structure of Sandlund and Wild [112] which adds the additive $\mathcal{O}(\log \log n)$ overhead. The interval structure is placed on each gap, and holds the elements of the gap. Crucially, to match the lower bound on the query time presented by Sandlund and Wild [112], the structure must support that queries that split a gap close to the edge is faster than a query which splits closer to the middle of the gap. Specifically, if a query splits the gap into two new gaps of sizes x and cx for some value $c \geq 1$, the query needs to be supported in time $\mathcal{O}(x \log c)$.

The interval structure used by Sandlund and Wild contains simple trees and lists, where all operations only touch $\mathcal{O}(1)$ parts of a single list, or scans the whole list. By replacing the lists by blocked-linked-lists and the tree by a B-tree, the construction easily carries to the external-memory model. The analysis of the construction is found in [111, Appendix], where the new parts of the analysis is a cleaner case analysis, as well as performing the amortized analysis using a well-defined potential function. The novelty of our paper [110] therefore lies in the construction of external biased search trees, used for the gap structure.

5.2 External-Memory Biased Search Tree

The first hurdle is creating an external-memory version of the gap structure. The internal solution use a *biased search tree* for this structure, which was first described by Bent, Sleator, and Tarjan [20]. Later, Bagchi, Buchsbaum, and Goodrich designed *Biased Skip Lists* [14], which have roughly the same time bounds.

In a biased search tree, let w_i denote the weight of element x_i stored in the tree, and $W = \sum_i w_i$ denote the total weight of all elements. The query for element x_i is then supported in $\mathcal{O}\left(\log \frac{W}{w_i}\right)$ time, i.e., the larger the weight the faster the element is accessed. Further the tree allows for updating the weight of elements in, as well as inserting and deleting elements in time proportional to the weight, and the weight of the neighboring elements in sorted order. An overview of the time bounds of biased search trees and skip lists are shown in Table 5.2.

Initially, we only wish to create external biased search trees, matching the internal runtimes with the base of the logarithm being switched to B , similar to the advantage B-

trees gains over internal BSTs. For both internal biased structures, it is not immediately possible to convert either to external-memory.

For the tree solution [20], the degree of each node is bounded to $[2, b]$ for some parameter b . The parameter can be chosen as the block size B , to ensure that each node can be read using $\mathcal{O}(1)$ I/Os. By packing all nodes into a blocked-linked-list, the space usage can be limited to $\mathcal{O}(N/B)$ blocks. However, as the height of a tree is bounded by the logarithm of the smallest degree of the nodes, the tree may grow to height $\Omega(\log N)$. If $\mathcal{O}(1)$ I/Os is used to access any node, regardless of the degree, then this leads to some operations requiring $\Omega(\log N)$ I/Os.

For the skip list solution [14], the degree of the inner branching is bounded to be in the interval $[a, b]$, for parameters a and b . By choosing $a = B$ and $b = 2B$, this therefore leads to bounding the height of the structure by $\Omega(\log_B N)$. However, packing the structure into blocks, s.t. access to each branching area uses $\mathcal{O}(1)$ I/Os and the structure use overall $\mathcal{O}(N/B)$ blocks of space is non-trivial. The internal structure is build using a two-dimensional doubly-linked-list, to ensure efficient traversal. Each layer corresponds to values of decreasing weight from the top down, and each column or *tower* corresponds to a single element contained in the structure. This structure must then be packed into blocks for the external version. From the behavior of the query operation, we can conclude that packing the lists layer by layer into blocked-linked-lists leads to efficient I/O costs of the query operation. However, the internal structure uses more than linear space. One such worst-case example for n elements is when the elements have altering small ($w_i = 1$) and huge ($w_i = n$) weight, where the structure would need to maintain $\Theta(n \log n)$ space. To gain linear space in n , the internal structure therefore cannot instantiate the entire structure, but only maintain a *weak skyline* of the structure. One other issue with the space usage of the structure, is that when inserting an element of large weight, little time is allowed, but the large weight requires a tall tower to be build in space, which is not allowed in the little time. The space must therefore be build *lazily*, which further supports that only a weak skyline of the two-dimensional linked-list can be maintained. This can be done in internal-memory, as the random access allows to easily create the needed uncreated parts of the structure on demand, however, it is unclear how to efficiently detect and create the missing parts on demand in external-memory.

Therefore, instead of basing our external biased search tree on either of the internal solutions, we opted for a different approach. Our approach does not create a general external biased search tree, as it is only I/O efficient enough to support the operations of the gap structure. In the overview of the runtimes of the internal solution in Table 5.2, the final obtained I/O cost of our structure is stated, which also highlights the limitations of our structure.

The first attempt at creating external biased search trees, used the idea of maintaining layers of blocked-linked-lists from the internal skip list solution. An element x_i with weight w_i is then placed into layer $\lfloor \log_B w_i \rfloor$, with the layers numbered from 0 to ℓ bottom-up. The layers are then sorted by value. By then introducing *bridges* between the layers, by copying elements into neighboring layers, as used in fractional cascading [50], s.t. each block of each layer contains an upwards and downwards

Table 5.2: Runtime of previous internal biased constructions, and the new external construction, for different operations. The stated runtime for the external solution is measured in I/Os. W denotes the total weight before the operation and W' denote the total weight after the operation. Symmetrically w_i denote the weight of the manipulated element x_i before the operation, and for the Reweight operation w'_i denotes the new weight of the element. w_{i-} denotes the weight of the predecessor element to x_i by value, and w_{i+} denotes the weight of the successor. W_L and W_R denote for the Join operation the total weight of the joined two trees L and R , and $w_{L_{\max}}$ is the weight of the largest element in L by value, with $w_{R_{\min}}$ denoting the weight of the smallest element in R by value. For the FingerSearch operation $V(i, j)$ denote the total weight of all elements from the starting element x_i to the end element x_j . Cells marked with 'am.' denote amortized results, and cells marked with 'N/A' denote that the construction does not support the operation.

Operation	Internal [20] Tree based	Internal [14] Skip list based	External [110] <i>New</i>
Search	$\mathcal{O}\left(\log \frac{W}{w_i}\right)$ am.	$\mathcal{O}\left(\log \frac{W}{w_i}\right)$	$\mathcal{O}\left(\log_B \frac{W}{w_i}\right)$
Insertion	$\mathcal{O}\left(\log \frac{W+w_i}{\min(w_{i-}, w_{i+}, w_i)}\right)$ am.	$\mathcal{O}\left(\log \frac{W+w_i}{\min(w_{i-}, w_{i+}, w_i)}\right)$	$\mathcal{O}(\log_B N)$
Deletion	$\mathcal{O}\left(\log \frac{W}{w_i}\right)$ am.	$\mathcal{O}\left(\log \frac{W+w_i}{\min(w_{i-}, w_{i+}, w_i)}\right)$	$\mathcal{O}(\log_B N)$
Reweight	$\mathcal{O}\left(\log \frac{\max(W, W')}{\min(w_i, w'_i)}\right)$ am.	$\mathcal{O}\left(\log \frac{W'}{\min(w_i, w'_i)}\right)$	$\mathcal{O}\left(\log_B \frac{\max(W, W')}{\min(w_i, w'_i)}\right)$
Join	$\mathcal{O}\left(\log \frac{W_L + W_R}{w_{L_{\max}} + w_{R_{\min}}}\right)$	$\mathcal{O}\left(\log \frac{W_L}{w_{L_{\max}}} + \log \frac{W_R}{w_{R_{\min}}}\right)$	N/A
Split	$\mathcal{O}\left(\log \frac{W}{w_{i-} + w_{i+}}\right)$ am.	$\mathcal{O}\left(\log \frac{W}{\min(w_{i-}, w_{i+})}\right)$	N/A
Finger- Search	N/A	$\mathcal{O}\left(\log \frac{V(i, j)}{\min(w_i, w_j)}\right)$	N/A

bridge, then given a location for a value x in layer k , the location for the value can be found in layer $k - 1$ and $k + 1$ using $\mathcal{O}(1)$ I/Os. By an amortized argument the introduction of bridges allows insertions and deletions of elements in a layer to use amortized $\mathcal{O}(1/B)$ I/Os. Similarly, the space can be bounded by $\mathcal{O}(N/B)$ blocks. The bridges continue upwards until the top layer ℓ contains $\mathcal{O}(B)$ elements, which therefore fit in $\mathcal{O}(1)$ blocks. Upon a query, the location of the query value x_i can be found in the top layer using $\mathcal{O}(1)$ I/Os, and in each subsequent layer using $\mathcal{O}(1)$ I/O pr layer. As x_i must be reached in layer $\lfloor \log_B w_i \rfloor$, the total I/O cost is $\mathcal{O}(\ell - \log_B w_i)$. If it can be shown that $\ell \leq \log_B W$, this leads to the query time being the desired $\mathcal{O}\left(\log_B \frac{W}{w_i}\right)$ I/Os. If the elements was only copied upwards, then this holds trivially, as any copied element to layer $k + 1$ represents $\Theta(B)$ elements at layer k , and the weights of the elements increase by a factor $\sim B$ upwards. The total weight of any layer is therefore W , with each layer k containing elements of effective $\sim B^k$ weight. The maximal layer is therefore $\log_B W$. However, to ensure that navigation uses $\mathcal{O}(1)$ I/Os pr layer, the downwards copying is needed, which breaks this argument.

The final version of external biased search trees opted for a simpler approach, which is described in detail in [110, Section 2]. By separating the elements into buckets of doubly exponential size by weight, a query may search in a prefix of the buckets, and locate element x_i using $\mathcal{O}\left(\log_B \frac{W}{w_i}\right)$ I/Os. The downside of the doubly exponential sizes is that classical amortized arguments for allowing varying sizes of the buckets, where the buckets are rebuilt and rebalanced after linear updates in the bucket, does not work. I.e., for inserting elements, when a bucket becomes too large, the size of the next bucket a *spill-over* would touch is of a far greater size, prompting the amortized time to grow. We therefore opted for letting the bucket sizes be *exactly* doubly exponential, and updates are handled by swapping single elements between consecutive buckets. This leads to efficient query and reweight operations, however, insertions and deletions must touch all buckets, leading to $\mathcal{O}(\log_B N)$ I/Os used, when the structure stores N elements. This I/O cost is sufficient to use the construction as the *gap structure* of Lazy B-trees, as an update to the gap structure correspond to a query in the tree, which are allowed the higher I/O cost.

5.3 Corrections to the Paper

After the paper [110] was published, and this dissertation was completed, we found an error in the stated I/O costs.

To understand this subtle error, we must dive into the interval structure. In the interval structure the elements of gap Δ_i is stored in $\mathcal{O}(\log |\Delta_i|)$ lists, where the sizes of the lists roughly grows exponential from the sides of the gap towards the middle, to allow for queries near the edges to be faster. This property is maintained lazily and is only enforced when queries are executed. The lists are modified in two different ways to ensure the size property. They may either be partitioned around a given key or rank, or they may be concatenated. The partition must touch all elements, and is therefore a linear time operation, where the concatenation crucially must be performed in $\mathcal{O}(1)$ time, to obtain the needed time bounds. In the internal structure, the lists are linked lists. In the external model, we therefore opted for using blocked-linked-lists, as these have the same properties of linked lists, but with the added factor B speed-up. Here, concatenations of two blocked-linked-lists uses $\mathcal{O}(1)$ I/Os, and partitions may be performed, including deterministic selection [25] to find a value of a given rank, using $\mathcal{O}(K/B)$ I/Os for a list of size K . The space usage is linear. For a full overview of the external interval structure, see [111, Appendix].

For delete and change-key operations, we crucially need a pointer to the desired element in the interval structure, as scanning the unordered lists are inefficient. We must therefore maintain these pointers into the structure whenever elements are moved in the blocked-linked-lists. From the introduction of this dissertation in Chapter 1, it is known that incoming pointers can be handled amortized for blocked-linked-lists during insertions and deletions, as there must be $\Omega(B)$ updates to a block before $\mathcal{O}(B)$ elements are moved, therefore each update contribute $\mathcal{O}(1)$ amortized I/Os. However, the list are also updated when a partition is performed,

where up to K elements may move, prompting K incoming pointers to be updated, thereby increasing the cost from $\mathcal{O}(K/B)$ to $\mathcal{O}(K)$ I/Os. This results in removing a factor $\mathcal{O}(1/B)$ from the I/O cost of the query operation, on the $x \log c$ term, as shown in [110, Theorem 1]. The updated query time is therefore amortized $\mathcal{O}(\log_B \min\{N, q\} + \frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + x \log_2 c)$ I/Os.

Note that the I/O costs stated in the paper still holds, if delete and change-key operations are not to be supported, as the pointers into the blocked-linked-lists are then no longer needed. The problem of the pointers cannot be directly removed. However, we do not need actual outside pointers directly into the blocked-linked-lists, but only the functionality of finding a given element using $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os. To achieve this, a secondary data structure, which uses some other pointer representation to find the elements faster, must be constructed.

This issue is also present in the simplified construction leading to an external-memory priority queue, which is shown as an extension in Chapter 21. The error was caught, when trying to simplify this structure to be cache oblivious. This has the implication, that the main selling point of the structure, the fast decrease-key operations, does not hold. Instead, we pivoted the result to the internal model, where the pointer problem disappears, as the linked list nodes may be reused in a stable manner, to create simple heaps, with $\mathcal{O}(\log \log n)$ insertion and decrease-key operations. This result was published at LATIN'26 [37], with the full version on arXiv [38].

Chapter 6

Random Access on Narrow Decision Diagrams in External Memory

This chapter discusses the paper “Random Access on Narrow Decision Diagrams in External Memory”, which was published at SPIN’24 [125]. This paper was created in collaboration with Steffan Christ Sølvesten and Jaco van de Pol from Aarhus University. The included paper is the conference version, which fit all details. The paper is included in Chapter 14.

The Binary Decision Diagram (BDD) was first introduced by Bryant [49], as a way to state a boolean formula as a Directed Acyclic Graph (DAG). The most common way to manipulate BDDs is by recursive depth-first algorithms, which are made to be efficient by using a unique node table supported by a hash table [26, 101]. When the number of nodes exceeds the size of the internal memory, this table begins to be inefficient, and the cost of the I/O model begins to take over. The size of manipulable BDDs are thereby bounded by the size of the internal memory.

To overcome this, I/O efficient BDDs and manipulation algorithms needed to be designed. This was initially created by Arge [9, 10], which uses *time-forward processing* [51, 99] to achieve I/O efficient BDDs. Based on this idea, Sølvesten et al. [123] build the tool *Adiar*, which simplifies the construction of Arge, as well as extending the construction with additional functionality. As *Adiar* builds on the I/O model principles, it manages to handle BDDs far larger than the internal memory of the machine.

Adiar was build to be an efficient tool for model checking. Therefore, not only the theoretical I/O complexity is of interest, but so is the actual runtime spend to perform computations. This project touches one such optimization, and shows that by replacing efficient I/O time-forward processing by random access in some cases of the construction, a speedup in performance can be achieved.

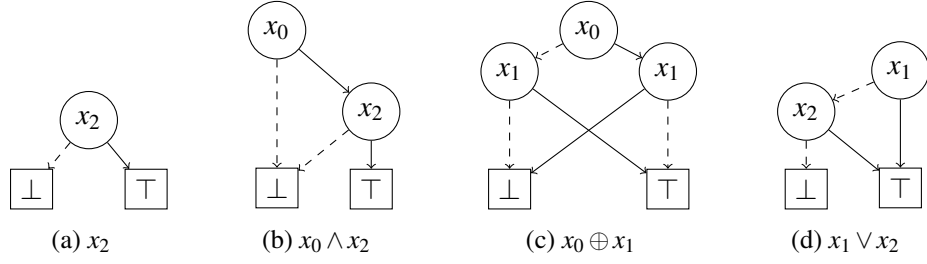


Figure 6.1: ([123, Figure 1]) Examples of Reduced Ordered Binary Decision Diagrams. Leaves are drawn as boxes with the boolean value and internal nodes as circles with the decision variable. *Low* edges are drawn dashed while *high* edges are solid.

6.1 Binary Decision Diagrams

A Binary Decision Diagram (BDD), is a rooted directed acyclic graph (DAG), that represents a boolean function $f : \mathbb{B}^n \mapsto \mathbb{B}$ on n variables $x_0, x_1, \dots, x_{n-1}$ [49]. The DAG contains a *root* node r and up to two *terminals* corresponding to the boolean values $\mathbb{B} = \{\perp, \top\}$. Each internal node in the DAG, referred to as a *BDD node*, contains the *label* i of some input variable x_i , as well as two outgoing arcs, one for when $x_i = \perp$ and one for $x_i = \top$, referred to as the *low* and *high* arc, respectively. To distinguish nodes with equal labels, each internal node further contains an *identifier* (id), which are unique within nodes of equal labels. Some BDDs and their corresponding boolean expression function is depicted in [123, Figure 1], which is restated in Figure 6.1. To evaluate the function on some given input, the process starts in the root, and then follows the arc in each node according to the value of the corresponding variable of the node, until a terminal is reached, which is the result of the function.

One version of BDDs is the *Ordered Binary Decision Diagram* (OBDD), where the nodes of the DAG are separated in *layers* of equal labels, s.t. arcs only go from higher levels to lower levels. In this layered representation, we assume that the terminals are at the infinite level below all nodes of the OBDD. One other way to observe OBDDs, is that they follow a permutation π of the input variables, where all root-to-leaf paths visits nodes with labels consistent to the ordering of the variables in π . Specifically, it must hold that for all arcs $v_1 \rightarrow v_2$ in the BDD, then $label(v_1)$ is before $label(v_2)$ in π . For the remaining text, we shall assume that all considered BDDs are OBDDs following the identity permutation.

Some nodes of the OBDD may have no effect on the computed output, or may be otherwise removable. These nodes are (1) nodes where the low and high arc both go to the same node, and (2) where another node with the same label contains the same set of outgoing arcs. We denote the nodes of type (1) as *don't care* nodes, and the nodes of type (2) as *duplicates*. If the OBDD does not contain any nodes of either type, we denote it as a *Reduced Ordered Binary Decision Diagram* (ROBDD). ROBDDs come with two advantages over OBDDs. Firstly, they are potentially smaller, and may therefore be manipulated faster. Secondly, the ROBDD of a given boolean formula f , assuming that the permutation π is fixed, is unique and therefore uniquely represents

the function f [49].

6.2 The Apply Operation

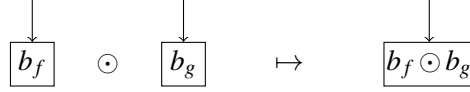
The advantage of BDDs is that formulae may be manipulated indirectly by manipulating their BDD representation. One such important manipulation is the *apply* operation, which given two BDDs representing formulae f and g , and a binary operator $\odot : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$, produce the BDD representing $f \odot g$. This operation is performed as the product construction of the two input BDDs. The construction is recursive, and starts at the product of the two root nodes (r_f, r_g) . The rules for recursively producing the output node v_o from the product (v_f, v_g) is shown in Figure 6.2. When a product node containing two terminals (b_f, b_g) is reached, the product is replaced by the terminal $b_f \odot b_g$. This case is shown in Figure 6.2a. Otherwise, the product node contains two nodes (v_f, v_g) of the two input BDDs, possibly with one of them being a terminal. If the two nodes contains the same label i , i.e., they are both branching on the same variable x_i , the label of the new node v_o is i . The outgoing arcs from v_o are to the product node, containing for the low arc the pair of the low endpoints of the nodes v_f and v_g , and symmetrically for the high arc. These nodes are then constructed recursively. This case is shown in Figure 6.2b. Otherwise, to preserve the ordering on the variables, the label of v_o is $l = \min(\text{label}(v_f), \text{label}(v_g))$. The outgoing arcs then only takes a step for the node matching label l , to preserve the semantics that node v_o represents a branching on only the variable x_l . These outgoing nodes are then constructed recursively. This case is shown in Figure 6.2c. This concludes the recursive procedure.

This simple recursive procedure support the BDD apply operation. However, the recursive procedure may visit the same node pair multiple times, leading to an increased runtime. As mentioned earlier, one common technique to circumvent this, is by using a unique node table supported by a hashing, to avoid processing the same node pair more than once [26, 101]. This has the added effect, that the output BDD can be guaranteed to be *reduced*. This table is, however, not I/O efficient, as it requires random access.

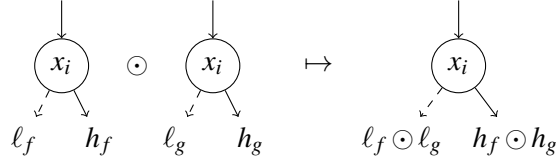
6.2.1 I/O Efficient Apply Operation

To build an I/O efficient product construction, Arge [9, 10] replaced the recursive calls by *time-forward processing* [51, 99]. Sølvesten et al. later simplified the construction of Arge, to build the I/O efficient BDD tool *Adiar* [123]. The operation is split into two parts. First a forwards *apply* operation is made, which constructs the initial output BDD. The technique allows for only scanning the input once and only process each product node once, however, it cannot be guaranteed that the output BDD is reduced. The apply operation is therefore followed by a backwards *reduce* operation, which ensures that the intermediate BDD is reduced to the final output BDD.

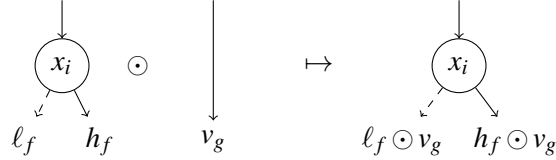
Time-forward processing is the technique, where, instead of processing a recursive call immediately, the call is placed as a request into a priority queue, ordered by a



(a) Both v_f and v_g are terminals of value b_f and b_g respectively. The output node v_o is the terminal with value $b_f \odot b_g$.



(b) Both v_f and v_g have the same label i . The output node v_o has the label i , with the low arc going to the product of the nodes ℓ_f and ℓ_g , which are the low arcs of v_f and v_g , respectively, and symmetrically for the high arc.



(c) The labels of v_f and v_g are not equal. For symmetry, let $label(v_f) = i$ and let v_g be in some later layer, i.e., $label(v_f) < label(v_g)$. The output node v_o has the label i , with the low arc going to the product of the low arc ℓ_f of v_f and v_g , and symmetrically for the high arc.

Figure 6.2: (Based on slides by Sølvesten [121]) Rules of the product construction between two BDD nodes v_f and v_g on the left, which produces the output node v_o on the right. Internal nodes are drawn as circles with terminals drawn as squares. Low arcs are shown by a dashed arrow, and high arcs by a solid arrow.

time metric. The time metric is build such that a request reaches the top of the queue at the point in time when the input to the request is easily available, e.g. by matching the time of the request to a linear scan of the input. The time metric therefore requires building an ordering, which allows for I/O efficient computation.

Time-forward processing is applied in the BDD context, by observing that the layers of the OBDD imposes a natural ordering on the BDD nodes. We shall place the BDD nodes ordered by layer and identifier, such that the BDD may be scanned I/O efficiently from the top-down by layer, and within each layer from the left to right. Specifically, a node a is before node b in the ordering iff

$$label(a) < label(b) \vee (label(a) = label(b) \wedge id(a) < id(b)) .$$

The apply operation then starts at the product of the roots of the two input BDDs, similar to the recursive procedure. By the ordering, then the roots must be the smallest nodes of both input BDDs, and they are therefore available as the first elements of the BDD in a linear scan. The ordering also ensures that any produced request

from a product node, must be to a pair of nodes, which both must be later in the linear scan. Assume that some request is made to process the product node (v_f, v_g) . If $label(v_f) < label(v_g)$, which corresponds to the recursive case depicted in Figure 6.2c, then to process this request, only information for the node v_f is needed. The request is therefore forwarded in time to where the scanning of the input BDDs reach node v_f , at which point the request can be processed. The case is symmetrical for when $label(v_f) > label(v_g)$.

If $label(v_f) = label(v_g)$, which corresponds to the recursive case depicted in Figure 6.2b, then, as the product nodes both branch on the same variable, information is needed on both v_f and v_g to process the request. The ordering the nodes are scanned within a layer is determined by the identifier of the nodes. If the identifier of v_f is close to the identifier of v_g , then the nodes are reached almost simultaneously in the scan, and may be processed without an I/O cost overhead. However, there is no guarantee on the closeness of the identifiers. Instead, we note that one of the nodes must have the smallest identifier. If $id(v_f) < id(v_g)$, then v_f is reached in the scan before v_g . A second time-forward is then performed, to forward the information of the node v_f to the point in time when the node v_g is reached, at which point the information on both nodes is known, and the request can be processed. The case is handled symmetrically for when $id(v_f) > id(v_g)$, as well as when $id(v_f) = id(v_g)$. Note that if multiple requests are made to process the same product node, the requests are reached at the same point in time, which therefore allows for only processing the request once.

For simplicity, Adiar uses two priority queues for the apply operation; one for forwarding request when no node information is held, and a second queue, which holds the information of some node. The reduce operation may similarly be computed using time-forward processing. The difference to the apply operation is that only a single BDD is scanned, and the ordering is reversed to process the BDD bottom-up.

In [125, Figure 2], restated in Figure 6.4, the product construction following the description of the apply operation above can be seen, for the input BDDs depicted in [125, Figure 1], which are restated in Figure 6.3. Note that the identifiers on the nodes are chosen such that they are consistent in ordering to the node labels and ordering by id within each layer.

6.3 Introducing Random Access

As noted in Section 6.2.1 above, Adiar uses two priority queues to perform time-forwarding in the apply operation. This was due to the recursive case, where the product node contains nodes of equal label. Here, the information had to be forwarded from the first node meet in the linear scan to the second node. However, if one of the input BDDs only contains layers with each have few nodes, then all nodes of any single layer may recite in internal memory, which directly grants access to all node information at once for one of the input BDDs. We shall denote such a BDD as *narrow*, specifically when all layers of a BDD contains $< M$ nodes. The access to all nodes in a layer at once directly allows for removing the second priority queue from

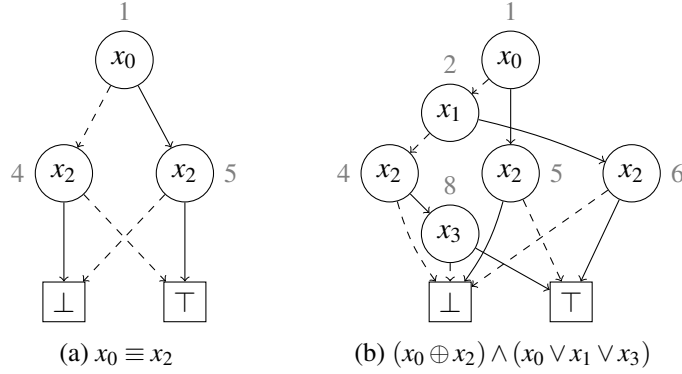


Figure 6.3: ([125, Figure 1]) Examples of Reduced and Ordered Binary Decision Diagrams. Terminals are drawn as boxes while internal nodes are drawn as circles with its decision variable. The high and low arcs are respectively drawn solid and dashed. Note that the displayed id of each node is chosen, s.t. node ids may be compared consistently to comparing by label and identifier.

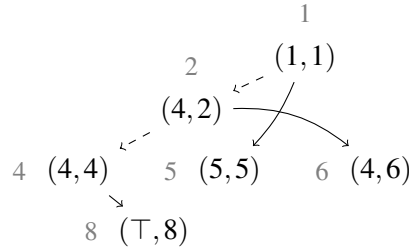


Figure 6.4: ([125, Figure 2]) Product Graph for BDDs in Figure 6.3a and Figure 6.3b.

the operation.

Assume that the apply operation is on input BDDs f and g , with g being narrow. The operation must then be altered slightly, as the time ordering in the first queue only needs to respect the ordering of the nodes of f . It must still respect the ordering on the layers. Request (v_f, v_g) is therefore first sorted by the smallest label of v_f and v_g , and secondly sorted by the identifier of v_f . The requests are then processed layer by layer, where each time a new layer is reached, the whole layer of g is read into the internal memory. When $label(v_f) = label(v_g)$, the request is processed when the identifier of v_f is reached in the scanning of f . At this time the information for v_f is available, and by random access, the information for v_g is available, and the request can therefore be processed.

Counter intuitively, removing the second time-forwarding step then improves the I/O cost, by introducing random access. The asymptotic improvement is, however, not improved, as any request in the second queue, which is now omitted, must still have been processed in the first queue, at the same asymptotic I/O cost. The result of the paper is therefore to study how the *actual* runtime is effected by this introduction

of random access.

The conclusion of the experiments is that random access provides a significant speedup of $\sim 15 - 25\%$, with the tendency that the speedup increases as the input BDDs grow.

6.4 Omitted Experiment

The inspiration to introduce random access into the apply operation, came from an observation by Sølvesten, stated in a GitHub issue to Adiar: “*When trying to compute the 8x8 Knight’s Tour, we ended up with a 1.4TiB sized ZDD where a single intersection operation with a tiny ZDD took more than an entire week. The tiny ZDD consists of two long chains. Hence, I believe one of the reasons it has been so slow is because an entire copy of the 1.4TiB ZDD was placed in the per-level priority queue to be forwarded to the second node of the chain. If we somehow can circumvent doing so (at the cost of maybe changing the order of execution), we can possibly gain lots in performance!*” [122].

A Zero-suppressed Decision Diagram (ZDD) is a BDD variant, where a missing node on some layer is not a *don’t care* node, but instead a node that enforces that the variable is $\perp$, i.e., the high arc goes to $\perp$ [100]. The apply operation have to be slightly altered to take into account how the missing nodes should be handled, i.e., a request for a product (v_f, v_g) , as the recursive request of a high arc, is replaced by $(v_f, \perp)$ when $\text{label}(v_f) < \text{label}(v_g)$, as g must then contain an omitted node at layer $\text{label}(v_f)$, where g enforces that the variable must be $\perp$ [126]. Symmetrically for when $\text{label}(v_f) > \text{label}(v_g)$.

The ZDDs are good at maintaining functions where the set of satisfying assignments is sparse. This makes them good for the *Knight’s Tour problem*, where one wish to compute the number of possible walks a knight can make on an $n \times n$ chessboard, where all nodes are visited only once, and the walk starts and ends in the same cell. The problem may be solved by ZDDs, by assigning a variable for each cell and point in time on a walk that cell may be visited. This leads to n^4 variables for an $n \times n$ board, as there are n^2 cells, which can be visited at n^2 points in time. The order of the variables are sorted first by the time and secondly by the cells. To handle duplicates, the tour is restricted to start in cell 1A at time 1, and must visit cell 2C at time 2, and finally visit cell 3B at time n^2 . First a single ZDD is constructed, which for all time steps ensures that only a single cell is visited, and that the step from time t to time $t + 1$ is a valid knight step. This ZDD does not need to remember information between many layers of time, and is therefore local in fashion. As each time may only visit a single cell, and at most 8 cells are reachable by a knight, the width of this ZDD is 8.

The time and steps have then been restricted. Finally, a ZDD is constructed for each of the n^2 cells, restricting each cell to be visited at most once. This ZDD needs to remember information across many layers, however, it only needs to remember if a single cell have been visited or not, and therefore has width 2. The intersection of all these $1 + n^2$ ZDD’s then represents the valid knight tours. Starting with the initial

ZDD constraining time and valid knight moves, each cell constraint is added once at a time. Note that as each of the intersected ZDD's for the cells have width 2, they are *narrow*, and the speedup from the random access optimization apply. From the final ZDD representing the valid set of states, the size of the state-space can be computed I/O efficiently, using the algorithm of Sølvsten et al. [123, 126]

The experiment was run for various smaller sizes of boards, which was included in the 'Combinatorial Counting' experiments of the paper. A separate machine was then used to run the experiment for the full board size of 8×8 . Unfortunately, the experiment never terminated before the paper was submitted and accepted at the conference SPIN'24. The experiment had at that point in time been executing for more than half a year, and had reached cell 6C. Assuming that the runtime of the experiment scales exponentially in the number of cells, as suggested by the data on the smaller boards, the experiment needed additionally ~ 9 years to conclude [124].

Another problem had also presented itself. The nodes in Adiar are encoded using a single 64-bit integer, with a bit denoting terminals, and the remaining bits split between representing the label and identifier of the node [127, Section 3.5]. The layout of the bits are chosen such that the lexicographical comparisons between nodes may be done directly by integer comparison. The downside of the encoding is that it limits the number of available labels and identifiers per layer. In the experiment, the size of the ZDD after reaching cell 6C had reached a point, where the widest layers may not be representable, and the ZDD may therefore had become corrupted. This therefore have to be addressed in Adiar, before the experiment may be run again.

Even though the addition of random access never improved the runtime enough to allow computing the number of knights tours on a regular chessboard, the optimization still lead to a general runtime improvement, and a valuable improvement step for the Adiar tool.

Chapter 7

The Contiguous Art Gallery Problem is Solvable in Polynomial Time

This chapter discusses the paper “The Contiguous Art Gallery Problem is Solvable in Polynomial Time”, which was published to arXiv [98]. This paper was created in collaboration with Magnus Christian Ring Merrild, Jens Kristian Refsgaard Schou, and Rolf Svenning from Aarhus University. The results of the paper was included in a merge of three submissions at SoCG’25 [22]. The other two concurrent papers contained in the merge are both available on arXiv [23, 108]. In the SoCG conference paper, most details of our paper are omitted due to page constraints. Additionally, the conference version contains the concurrent work of the other two submissions. The arXiv version of our submission contains all details, and only include our work. This arXiv paper is included in Chapter 15.

The *art gallery problem* has been studied in detail over multiple different variants. In the most basic version of the problem, first proposed by Victor Klee in 1973 [53, 105], the input is a simple polygon P of n vertices, and the goal is to compute a minimum size set of *guards*, which in union *cover* the whole polygon, i.e., from any interior point in P , there is a guard with line of sight to the given point. Another version opts for not covering the whole polygon P , but only the *boundary* ∂P , motivated by the art being hung on the walls of the gallery. This problem and its variants, are known to be hard, from some variants being NP-hard [92, 90, 105] to some even being $\exists\mathbb{R}$ -complete [2, 128]. The paper gives an overview of the complexity of the variants in [98, Table 1], which is restated in Table 7.1.

At the 2024 Canadian Conference on Computational Geometry Thomas C. Shermer introduced yet another variation, denoted as the *contiguous art gallery problem*, in which the guards must cover the boundary ∂P of the input polygon P , but where each guard placed must see a single *contiguous polygonal chain*, also denoted as a *contiguous interval*, part of the boundary. The paper studies this variant, and interestingly

Table 7.1: ([98, Table 1]). Summary of the work related to the contiguous art gallery problem, for polygons with n vertices where k^* is the size of the minimal decomposition. We abbreviate Art Gallery as AG, $\dagger$ refers to the guard location being restricted to vertices, and $\ddagger$ refers to the vertices of the pieces (the guarded area) being restricted to the vertices of the input polygon. Notice how the difficulty of the four problems increases along the diagonal from the bottom left vertex to the top right. The last row contains the results of the paper [98] discussed in this chapter.

Problem	Vertex restricted	Unrestricted	With holes
Standard AG	NP-hard $\dagger$ [92]	$\exists\mathbb{R}$ -complete [2]	$\exists\mathbb{R}$ -complete [2]
Boundary AG	NP-hard $\dagger$ [90]	$\exists\mathbb{R}$ -complete [128]	$\exists\mathbb{R}$ -complete [128]
Star-shaped partition	$\mathcal{O}(n^7 \log n)^{\ddagger}$ [84]	$\mathcal{O}(n^{105})$ [3]	NP-hard [105]
Contiguous AG	$\mathcal{O}(n^2 \log^2 n)^{\ddagger}$ [98] $\mathcal{O}(n^2 \log n)^{\dagger}$ [98]	$\mathcal{O}(k^* n^5 \log n)$ [98]	<i>Open</i>

shows that for this variant, and related vertex restricted versions, an optimal solution is computable using only polynomially many arithmetic operations. Additionally, the bit complexity of the arithmetic operations performed is shown to be polynomial boundable, which shows that the variant is contained in the simple complexity class P. The runtime results of the paper is summarized as the last row of [98, Table 1], which is restated in Table 7.1.

The merged paper [22] contains an overview of the contents of the three independent papers [23, 98, 108]. All three papers show that the contiguous art gallery problem is solvable in polynomial time. Interestingly, the three papers use three different approaches and algorithms to show this. Common to them all is that they use the notion of a *greedy* basic algorithm. In this chapter, I will present in details our solution [98], and in Section 7.6 shortly highlight the other two solutions of [23, 108].

We let P denote the input polygon consisting of n vertices $v_1, v_2, \dots, v_n$, which are connected by segments¹ from v_i to v_{i+1} , with v_n connecting to v_1 . These segments form the boundary ∂P , which we wish to cover with a minimum set of guards. Further, let k^* denote the size of an optimal contiguous guarding solution of P . We assume that the input polygon P is not guardable by a single guard, as this may be initially checked in $\mathcal{O}(n)$ time, by checking if P is star-shaped.

7.1 The GREEDYINTERVAL algorithm

As mentioned, the basic algorithm used in all three results is a *greedy* algorithm, which we denote as the GREEDYINTERVAL algorithm [98, Algorithm 2]. Given a point x

¹The paper [98] denotes this as ‘edges’, but to avoid confusion between the boundary and the segments of the polygon, I have removed the word ‘edge’ from this dissertation.

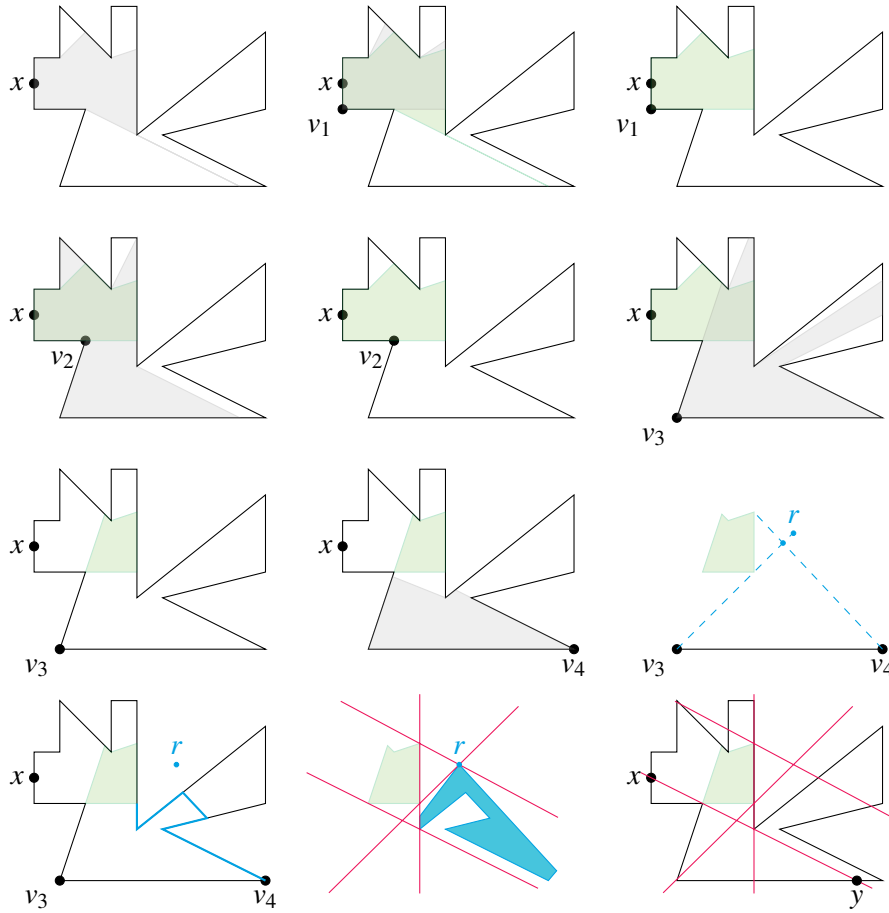


Figure 7.1: ([22, Figure 7]) The steps of the GREEDYINTERVAL algorithm animated. First successive iterations of computing visibility polygons (in gray) and intersecting with previous feasible regions (in green) and finally computing the blocking polygon (in blue), to find y through a common tangent.

on the boundary of the input polygon P , GREEDYINTERVAL computes the maximum point y clockwise² around the boundary, s.t. there exists a guard g , which can see the contiguous interval from x to y . An illustration of the steps of the GREEDYINTERVAL algorithm on an example is shown in [22, Figure 7] and restated in Figure 7.1.

Our paper [98] shows, that if there are e segments of P between x and y , then GREEDYINTERVAL is computable using $\mathcal{O}(en \log n)$ arithmetic operations [98, Theorem 3]. To show this runtime, several geometric tricks are used, most notably the notion of a *feasible region*, denoted by F , which is the area inside P consisting of all points that can see the whole contiguous boundary from x up to the current vertex v_i found clockwise along the boundary. This region, if not empty, therefore contains the

²In the merged paper [22], the GREEDYINTERVAL algorithm is defined to move counter-clockwise. All arguments stated from the clockwise results still hold by symmetry.

needed guard g . By then extending this region, by intersection the *visibility polygon* of the next vertex v_{i+1} along the boundary, it can be detected if there still exists a guard which can see the whole next segment, while still seeing the previous contiguous interval from x , thereby seeing the contiguous interval from x to v_{i+1} . The visibility polygon is computable in time $\mathcal{O}(n)$ and has size $\mathcal{O}(n)$ [63], and by showing that the feasible region has size $\mathcal{O}(n)$ [98, Lemma 4], then using a polygon clipping algorithm, this extension step takes $\mathcal{O}(n \log n)$ time [96].

If this extension then leads to the feasible region F being empty, it must then hold that there is no guard that can see from x up to and including the whole next segment. The furthest visible point y must therefore be on this segment, and the guard g must be contained in the feasible region F before the extension. In the initial submission, we showed that the guard must be a corner of F , and computing the furthest visible point y on the next segment for each corner of F , we obtain a set of candidate solutions, from which we compute the final value of y by picking the furthest point among the candidates. However, this naively uses $\mathcal{O}(n)$ time for each corner, and in the initial submission we did not bound the size of F by more than $\mathcal{O}(en)$. This step therefore takes time $\mathcal{O}(en^2)$, leading to a $\mathcal{O}(en^2 \log n)$ time GREEDYINTERVAL algorithm, when also accounting for the slower extension step.

However, doing the merge of the paper submissions, we managed to invent a better algorithm for this final step, using the common tangent algorithm of Abrahamsen and Walczak [4], which computes the common tangents between two non-overlapping polygons of size n in $\mathcal{O}(n)$ time. By inputting to this algorithm the feasible region F , and a cleverly crafted *blocking polygon* from the relevant parts of the polygon P , computed in time $\mathcal{O}(n \log n)$, there is only constantly many common tangents, one of which crosses the next segment of P at exactly the furthest visible point y . The runtime of the final step is therefore improved to $\mathcal{O}(n \log n)$ [98, Proposition 12]. In total, the GREEDYINTERVAL algorithm use $\mathcal{O}(en \log n)$ arithmetic operations [98, Theorem 3].

7.2 The REPEATEDGREEDY Algorithm

Our solution builds on the GREEDYINTERVAL algorithm, by starting at any point on the boundary of P , and then repeatedly applying the GREEDYINTERVAL algorithm to the last computed endpoint, to traverse around the polygon [98, Algorithm 1]. Specifically, given a starting point x , the GREEDYINTERVAL algorithm is used to compute the point y . Then continuing from y , GREEDYINTERVAL is used to compute the point z , and so on. After some number of applications of GREEDYINTERVAL, the REPEATEDGREEDY algorithm terminates, and the outputted solution of contiguous intervals and guards is the last computed *revolution* of the polygon. The main result of the paper is then showing that REPEATEDGREEDY terminates with an optimal solution in a polynomial number of applications of GREEDYINTERVAL. This goes by showing that the number of revolutions are polynomially bounded, as a single revolution of the GREEDYINTERVAL algorithm runs in time $\mathcal{O}(n^2 \log n)$ [98, Corollary 13].

One remark on the REPEATEDGREEDY algorithm, is that after just a single revolution, the computed solution is at most 1 off from being optimal, that is the returned solution is of size k , with $k^* \leq k \leq k^* + 1$ [98, Theorem 21]. The difficult problem is then to determine if $k = k^*$ or $k = k^* + 1$.

7.3 The Analysis of the REPEATEDGREEDY Algorithm

The analysis is performed in two steps. First, a combinatorial viewpoint is made on the REPEATEDGREEDY algorithm [98, Section 4], and then using the observations of the combinatorial viewpoint, a geometric viewpoint is made to complete the analysis [98, Section 5].

The combinatorial viewpoint simplifies the input polygon P to a circle, and defines how the GREEDYINTERVAL algorithm will map input points x to the output points y on this circular representation [98, Lemma 17 and 18]. Following this viewpoint, three stopping conditions, for when the last revolution of the REPEATEDGREEDY algorithm is optimal, is shown. Importantly, it is shown that once some revolution of REPEATEDGREEDY is optimal, then any later revolution is optimal, that is, optimally cannot be lost [98, Corollary 23]. We shall assume that we are given some optimal solution, which partitions the circle into *intervals*. This solution is only used for analyzing the behavior of the REPEATEDGREEDY algorithm, and is thus not visible to the algorithm. One property of the GREEDYINTERVAL algorithm, is that it will visit each of these intervals once doing a revolution. Inside each interval, what we denote as a *local sequence* [98, Definition 28] is constructed.

By then studying how the local sequences of non-optimal revolutions behave in the combinatorial viewpoint [98, Proposition 26], three conditions for when a revolution is optimal is obtained, denoted as the *stopping conditions*. These are as follows:

1. [98, Corollary 30]: The last three points of the local sequence moves clockwise along the interval, denoted as a *positive fingerprint*.
2. [98, Corollary 31]: The local sequence moves out of the interval containing it.
3. [98, Corollary 32]: Any point is repeated.

When any of these are met, the last revolution computed by REPEATEDGREEDY is optimal.

To then show that one of these will be met after polynomially many revolutions of REPEATEDGREEDY, the problem is observed from the geometric viewpoint. Here, the assumption on the given optimal solution is removed. This immediately makes it impossible to detect the second stopping condition, as the interval endpoints are unknown. Further, the first stopping condition becomes difficult to detect, as the local sequences are defined from the optimal solution. The intervals are redefined to be the intervals produced by the first revolution of REPEATEDGREEDY, which is enough to detect the first stopping condition. For the second stopping condition, the endpoints of the newly defined intervals are not enough to detect optimality. The condition is

instead translated into the notion of *segment jumping*, that is when a local sequence moves from one segment onto another. Clearly, if this happens $n + 1$ times, some segment is moved to twice, and as a consequence, some local sequence must have moved past the endpoint defined from an optimal solution [98, Corollary 35]. Moving to a new segment, as well as the other two stopping conditions, are denoted as a *progress condition*.

The remainder of the geometrical viewpoint section is then used on an involved case-analysis of the behavior of the REPEATEDGREEDY algorithm, which concludes in showing that a progress condition is met after at most $\mathcal{O}(k^*n^2)$ revolutions [98, Theorem 37]. A progress condition may then happen at most $n + 1$ times before optimality is reached, therefore concluding that after at most $\mathcal{O}(k^*n^3)$ revolutions have been performed, then REPEATEDGREEDY have found an optimal solution. As a revolution of GREEDYINTERVAL is computed in time $\mathcal{O}(n^2 \log n)$, then using $\mathcal{O}(k^*n^5 \log n)$ total arithmetic operations, REPEATEDGREEDY finds an optimal solution [98, Theorem 1].

7.4 Vertex Restricted Variants

The contiguous art gallery problem is a restriction of the boundary covering variant of the art gallery problem. By further restricting the contiguous art gallery problem to be vertex restricted, the problem becomes even simpler. There are two variations of the vertex restriction; either the guards are restricted to be on a vertex of the polygon, or the endpoints of the contiguous intervals are restricted to start and end in the vertices.

Useful to both problems, is the *circle-cover minimization problem* studied by Lee and Lee [91]. In this problem, the input consists of n arcs, with each arc covering some continuous part of a common circle, and the goal is to compute a minimum subset of the arcs, which in union covers the circle. Lee and Lee shows a simple $\mathcal{O}(n \log n)$ algorithm that computes the output arcs, consisting by first sorting the arcs by the endpoint of each arc in time $\mathcal{O}(n \log n)$, and then applying a linear scan over the arcs in time $\mathcal{O}(n)$, which then computes the optimal solution. The problem is therefore closely related to the contiguous art gallery problem, viewing the input polygon as the circle and the set of contiguous guardable intervals as the arcs. However, the solution of Lee and Lee does not immediately apply, as the set of contiguous guardable intervals of a polygon may be infinite. The size is, however, not infinite for the vertex restricted variants.

A solution to the guards being restricted to the vertices is then as follows, as stated in [98, Section A.2]. For each of the n vertices v_i of the polygon, the visibility polygon of v_i is computed in time $\mathcal{O}(n)$, and then intersected with the boundary of the polygon in $\mathcal{O}(n)$ time by a linear scan. This then yields a set of contiguous intervals visible to v_i , which is bounded in size by $\mathcal{O}(n)$. In total over all vertices there are therefore $\mathcal{O}(n^2)$ contiguous intervals, which is guarded by a vertex of the polygon. Applying the Lee and Lee algorithm, an optimal solution may be computed in $\mathcal{O}(n^2 \log n)$ time, as the sorting step is required.

The version, where the contiguous intervals are bounded to the vertices, seem

at first glance simpler, as the possible locations of the contiguous intervals are well-defined. However, checking if a contiguous interval is valid requires locating a guard, which may be at any interior point of the polygon, which thereby increases the complexity of the problem. A solution to the problem is presented in [98, Section A.1]. In the following, the underlying steps leading to the presented efficient solution are shown in detail.

The first and arguably most simple solution, is to observe that there are only $\Theta(n^2)$ such possible intervals. Each interval consists of $\mathcal{O}(n)$ segments from the polygon, and can therefore be checked for validity in time $\mathcal{O}(n^2 \log n)$, by computing and intersecting the visibility polygons of all vertices along the segments of the interval, and checking if the computed intersection is non-empty. In total, $\mathcal{O}(n^4 \log n)$ time is spent to compute a set of at most $\mathcal{O}(n^2)$ valid contiguous intervals, from which an optimal solution can be computed by the Lee and Lee algorithm. The constant may be improved, by not recomputing the visibility polygons of each vertex, but compute them as a $\mathcal{O}(n^2)$ preprocessing step. To further asymptotically improve the solution, observe that any contiguous interval contained in another valid contiguous interval, may be removed from the input set to the Lee and Lee algorithm, as the larger covering interval may be used to cover the smaller interval in any valid solution it would be part of. For each vertex in the polygon, it therefore suffices to compute from each vertex the longest clockwise contiguous interval to another vertex in the polygon. This may be computed simply by the GREEDYINTERVAL algorithm in time $\mathcal{O}(n^2 \log n)$ for each vertex, by skipping the final step of the algorithm, truncating the contiguous interval to span to a vertex of the polygon. In $\mathcal{O}(n^3 \log n)$ total time this yields n contiguous intervals, from which an optimal solution can be computed by the Lee and Lee algorithm.

Observe that if some vertex of this solution v_i can reach vertex v_j , then by the property of the GREEDYINTERVAL algorithm, then v_{i+1} must at least also reach at least v_j . The intersection of the visibility polygons between v_{i+1} and v_j are therefore recomputed, which may be avoided. To reuse the feasible region computed from v_i to v_j , to compute the feasible region from v_{i+1} to v_j , some solution must be created which is capable of *removing* the visibility polygon of v_i from the intersection. Notice that the feasible region is computed by intersecting polygons, which may be done in a divide-and-conquer manner. The result of Overmars [106] exactly shows how to convert such a static algorithm to a dynamic solution. In essence, the divide-and-conquer manner is a tree, where the leaves are the current set of elements, and each node is an application of the “conquer” step. In the regular divide-and-conquer manner, the output of these intermediate steps are discarded. If they are instead saved in each node of the tree, then when inserting or deleting a leaf in this tree, thereby updating the set of elements, then only the intermediate steps along a single path of the tree needs to be recomputed. Further, the tree may be balanced, thereby ensuring that the height is logarithmic. The elements of the set is then the visibility polygons of the contiguous interval. The visibility polygon of v_i may then be removed, and new visibility polygons from v_{j+1} may be inserted, to compute the contiguous interval from v_{i+1} , without entirely recomputing the intersection of all visibility polygons

between v_{i+1} and v_j . This leads to computing the contiguous intervals from each vertex in total time $\mathcal{O}(n^2 \log^2 n)$. Applying the Lee and Lee algorithm to these n intervals then solves the problem.

7.5 Bounding the Bit Complexity

The paper shows in detail, that the number of arithmetic operations needed to compute an optimal solution to the contiguous art gallery problem is polynomially bounded. However, in the initial submission, the bit complexity was not bounded, and the problem was therefore not shown to be in the complexity class P. The GREEDYINTERVAL algorithm may be bounded in polynomial bit complexity by the bit complexity of the polygon and the input point, and the output may be bounded. However, even if the complexity of the output grows by a small multiplicative constant, which does bound the complexity asymptotically, then by the nature of the REPEATEDGREEDY function, as GREEDYINTERVAL steps are performed on the output of the last GREEDYINTERVAL step, the bit complexity of REPEATEDGREEDY grows exponentially.

A definition of bit complexity is stated by Grötschel, Lovász and Schrijver [72], which is restated in final version of our paper in [98, Definition 49]. In essence, the definition denotes the bit complexity of a value, to be the number of bits used to represent the value. Specifically, for an integer v , the bit complexity is $1 + \lceil \log_2(|v| + 1) \rceil$, which accounts for the bits used to encode the value as well as the sign bit. The important part of this definition is that the bit complexity is in essence the logarithm of the value. Therefore, adding or subtracting integers only increases the bit complexity of the output by additively $\mathcal{O}(1)$ over the largest complexity of the two inputs. On the other hand, multiplying numbers increases the bit complexity to the sum of the complexity of the two numbers. This is shown in details in [98, Lemma 50]. Therefore, multiplying numbers is the culprit needing to be bounded, to bound the bit complexity increase of the GREEDYINTERVAL function.

Initially, an attempt to bound the complexity was made, but ultimately failed. This initial argument was as follows. As mentioned, it can be shown that all arithmetic computations performed by GREEDYINTERVAL is bounded by the input point and the intermediately computed points. It therefore suffices to bound the complexity of the computed points, which are the vertices of the feasible region F and the output point. All of these points are computed as the intersection point of two lines, each line defined by two points. Let the two lines be the extension of the segments $\overline{AB}$ and $\overline{CD}$, and the intersection point be E . Each point is represented as two fractions, one for the x value and the other for the y value. Each fraction is further composed of two whole numbers, one for the numerator and one for the denominator. The intersection is illustrated in Figure 7.2.

By using a formula for the intersection [70], the point E may be computed as

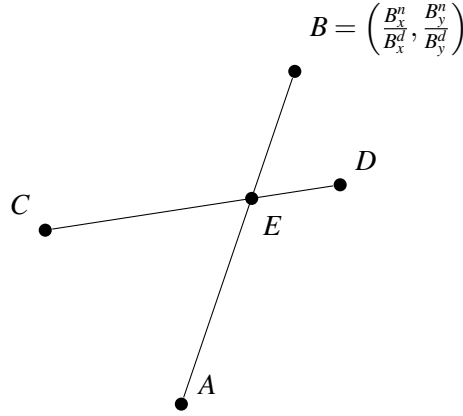


Figure 7.2: Initial setup of the intersection between four points. Note that it is not a requirement that the segments $\overline{AB}$ and $\overline{CD}$ intersect, only that their lines intersect.

follows:

$$\frac{E_x^n}{E_x^d} = \frac{\begin{aligned} &+A_x^d A_y^d B_x^n B_y^d C_x^d C_y^n D_x^n D_y^d - A_x^d A_y^d B_x^n B_y^d C_x^n C_y^d D_x^d D_y^n \\ &-A_x^d A_y^n B_x^n B_y^d C_x^d C_y^n D_x^n D_y^d + A_x^d A_y^n B_x^n B_y^d C_x^n C_y^d D_x^d D_y^n \\ &-A_x^n A_y^d B_x^d B_y^d C_x^d C_y^n D_x^n D_y^d + A_x^n A_y^d B_x^d B_y^d C_x^n C_y^d D_x^d D_y^n \\ &+A_x^n A_y^d B_x^d B_y^n C_x^d C_y^n D_x^n D_y^d - A_x^n A_y^d B_x^d B_y^n C_x^n C_y^d D_x^d D_y^n \end{aligned}}{\begin{aligned} &+A_x^d A_y^d B_x^n B_y^d C_x^d C_y^n D_x^n D_y^d - A_x^d A_y^d B_x^n B_y^d C_x^n C_y^d D_x^d D_y^n \\ &-A_x^d A_y^n B_x^n B_y^d C_x^d C_y^n D_x^n D_y^d + A_x^d A_y^n B_x^n B_y^d C_x^n C_y^d D_x^d D_y^n \\ &-A_x^d A_y^n B_x^d B_y^d C_x^d C_y^n D_x^n D_y^d + A_x^d A_y^n B_x^d B_y^d C_x^n C_y^d D_x^d D_y^n \\ &+A_x^n A_y^d B_x^d B_y^n C_x^d C_y^n D_x^n D_y^d - A_x^n A_y^d B_x^d B_y^n C_x^n C_y^d D_x^d D_y^n \end{aligned}}$$

$$\frac{E_y^n}{E_y^d} = \frac{\begin{aligned} &+A_x^d A_y^d B_x^n B_y^d C_x^d C_y^n D_x^n D_y^d - A_x^d A_y^d B_x^n B_y^d C_x^n C_y^d D_x^d D_y^n \\ &-A_x^d A_y^n B_x^n B_y^d C_x^d C_y^n D_x^n D_y^d + A_x^d A_y^n B_x^n B_y^d C_x^n C_y^d D_x^d D_y^n \\ &-A_x^d A_y^n B_x^d B_y^d C_x^d C_y^n D_x^n D_y^d + A_x^d A_y^n B_x^d B_y^d C_x^n C_y^d D_x^d D_y^n \\ &+A_x^n A_y^d B_x^d B_y^n C_x^d C_y^n D_x^n D_y^d - A_x^n A_y^d B_x^d B_y^n C_x^n C_y^d D_x^d D_y^n \end{aligned}}{\begin{aligned} &+A_x^d A_y^d B_x^n B_y^d C_x^d C_y^n D_x^n D_y^d - A_x^d A_y^d B_x^n B_y^d C_x^n C_y^d D_x^d D_y^n \\ &-A_x^d A_y^n B_x^n B_y^d C_x^d C_y^n D_x^n D_y^d + A_x^d A_y^n B_x^n B_y^d C_x^n C_y^d D_x^d D_y^n \\ &-A_x^d A_y^n B_x^d B_y^d C_x^d C_y^n D_x^n D_y^d + A_x^d A_y^n B_x^d B_y^d C_x^n C_y^d D_x^d D_y^n \\ &+A_x^n A_y^d B_x^d B_y^n C_x^d C_y^n D_x^n D_y^d - A_x^n A_y^d B_x^d B_y^n C_x^n C_y^d D_x^d D_y^n \end{aligned}}$$

As mentioned, adding and subtracting values is not an issue for the bit complexity, so the complexity of each term can be analyzed as just a single term of the summations. In each term of the summations, the input values of A , B , C , and D are each represented twice. This results in the output point E doubling the complexity of the input points, and ultimately shows that the complexity of the output of GREEDYINTERVAL is multiplicatively increased from the input. This rough analysis therefore leads to the REPEATEDGREEDY algorithm having exponential bit complexity in the number of GREEDYINTERVAL steps performed.

After the submission, a better argument bounding the bit complexity was found, which is included in [98, Section 6]. This uses the idea of *scalar points*, which instead of representing the complex points by four integers, instead represents it as a high complexity fractional scalar between two vertices of the input polygon, which are of low complexity. It can then be shown that each of the intersections performed only ever contains a single such scalar point, with the other three points being input points of the polygon. Let A be the scalar point, with scalar s and endpoints V and W . When the intersection is between the extension of the segments $\overline{AB}$ and $\overline{CD}$, then the intersection point E is a scalar point, for some scalar t and endpoints C and D . This refined intersection is illustrated in [98, Figure 39] and cloned in Figure 7.3.

Using the formula for intersection [70], and isolating for $t = \frac{t^n}{t^d}$, the scalar is computed as stated in [98, Lemma 51], and restated below:

$$\begin{aligned}
 t^n = & \quad s^n (+ B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^d W_y^d + B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^d W_y^n \\
 & \quad + B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d \\
 & \quad - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d \\
 & \quad - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d) \\
 & + s^d (+ B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d \\
 & \quad + B_x^n B_y^n C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d \\
 & \quad - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d) \\
 \\
 t^d = & \quad s^n (+ B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d \\
 & \quad + B_x^n B_y^n C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d \\
 & \quad - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d) \\
 & + s^d (+ B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d \\
 & \quad + B_x^n B_y^n C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d + B_x^n B_y^n C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d \\
 & \quad - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^d V_y^n W_x^n W_y^d - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d \\
 & \quad - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d)
 \end{aligned}$$

Notice that s only participates once in each term of the sum for the values of t . The bit complexity of the output scalar t can therefore be shown to be bounded by the bit complexity of the input scalar s increased additively by the complexity of the input polygon, which is of low complexity [98, Lemma 51]. The complexity of a scalar point when computed, is then shown to be bound by the complexity of the scalar increased additively by the complexity of the polygon [98, Lemma 52]. Building on these lemmas, it is shown that the complexity of the output point of GREEDYINTERVAL is bounded by the input point [98, Lemma 53], and that the internal bit complexity of GREEDYINTERVAL is bounded by the complexity of the

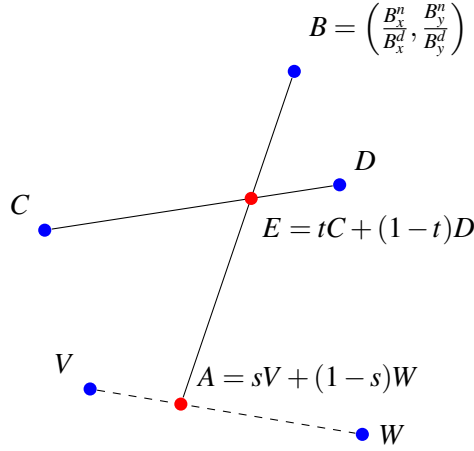


Figure 7.3: ([98, Figure 39]) Setup of the intersection. Blue points denote input vertices and the red points denote scalar points. Note that it is not a requirement that the segments $\overline{AB}$ and $\overline{CD}$ intersect, only that their lines intersect.

input point [98, Lemma 54]. Finally, it can be concluded that the total bit complexity of the arithmetic operations performed by REPEATEDGREEDY is bounded, as each step only increases the complexity by an additive term containing the input polygon complexity, thereby showing that the problem truly belongs in the complexity class P.

7.6 Concurrent Work

As previously mentioned, the merged paper [22] consists of three paper submissions [23, 98, 108]. In this section I will briefly mention the concurrent results of the other two paper submissions.

The result by Biniarz, Maheshwari, Mitchell, Odak, Polishchuk, and Shermer [23], shows that there exists a computable set of points Q of size $\mathcal{O}(n^3)$ inside the polygon P , where some optimal solution must contain a guard at some point of Q . This leads to a set of $\mathcal{O}(n^4)$ of guardable contiguous intervals visible to some guard of Q , and therefore a set S of $\mathcal{O}(n^4)$ starting points for computing a single revolution of REPEATEDGREEDY, where it must hold that some starting point of S leads to an optimal solution. Overall, this results in an algorithm computing an optimal solution using $\mathcal{O}(n^6 \log n)$ arithmetic operations. Note that the bit complexity arguments of our paper [98, Section 6] extends to this solution, showing that this algorithm is in P.

The result of Robson, Spalding-Jamieson, and Zheng [108], shows that the GREEDYINTERVAL function may be computable as a piecewise rational linear function of polynomial size. Effectively this gives a simple polynomial sized representation of the GREEDYINTERVAL function. By then composing this function with itself, a piecewise rational linear function is constructed, which for each input point takes two GREEDYINTERVAL steps. It can then be shown, that by continuing composing this function, the size remains polynomial. It can then be checked in polynomial time after

each composition if there is some input point which completes a revolution. As this is checked after each composition, the first time this check finds such a point, then an optimal solution is found in total polynomial time. Their paper also gives an argument to bound the bit complexity, showing that their algorithm is in P.

7.7 Open Problems

From a purely computational complexity standpoint, the problem is shown to be in P, and the work is therefore completed. From an algorithmic standpoint, one may then ask “but is the stated runtime optimal?”.

The analysis on the number of GREEDYINTERVAL steps needed by REPEATEDGREEDY for optimality presented in the paper is based on a very pessimistic worst-case analysis, which possibly sets a much too high upper bound. That the upper bound may be too high is also demonstrated by a C++ implementation of the REPEATEDGREEDY algorithm. In the over 2,000,000 random art gallery instances tested by the program, the algorithm never performed more than 4 revolutions before an optimal solution was detected as a repeated visited point. This leads us to hypothesize that $\log^{\mathcal{O}(1)}(n)$ or maybe even as few as $\mathcal{O}(1)$ revolutions of REPEATEDGREEDY is sufficient to compute an optimal solution. One path for future work is then to prove this, in order to improve the runtime.

Another path is to create faster GREEDYINTERVAL steps. Using the ideas in the vertex restricted contiguous intervals, a preprocessing step can be added, to allow for faster GREEDYINTERVAL steps. The idea is to preprocess for each vertex v_i , the furthest *counterclockwise* point p_i along the boundary that can reach v_i , i.e., the point of the boundary p_i , where any point before it cannot reach v_i in a *clockwise* guardable contiguous interval. Note that GREEDYINTERVAL is performing steps clockwise. The p_i ’s then partition the boundary of P into a set of intervals $[p_i, p_{i+1})$, where each interval may reach vertex v_i of P , but not v_{i+1} . Using the techniques of the vertex restricted contiguous intervals, the points p_i may be calculated efficiently in $\mathcal{O}(n^2 \log^2 n)$ preprocessing time. Let q_j be either a vertex of the polygon v_j contained in the interval from p_i to p_{i+1} , or be the point p_{i+1} . In the same preprocessing time, the feasible region F_j of each point q_j may be computed and stored. To then compute a GREEDYINTERVAL step for an input point x , a binary search is performed in $\mathcal{O}(\log n)$ time, which locates the interval $[p_i, p_{i+1})$ containing x , as well as the next clockwise point q_j , which by definition must be on the same segment as x . This also yields the feasible region F_j from q_j to v_i . This then yields a vertex v_i that x can reach clockwise, but where x cannot reach v_{i+1} . Extending F_j with the visibility polygon from x yields the feasible region from x to v_i , and the final step of GREEDYINTERVAL may be performed in time $\mathcal{O}(n \log n)$, as the final segment is known to be from v_i to v_{i+1} . As each GREEDYINTERVAL step is reduced to take $\mathcal{O}(n \log n)$ time, and as $\Theta(k^*)$ such steps are performed on a revolution, the time to perform a revolution is therefore reduced from $\mathcal{O}(n^2 \log n)$ to $\mathcal{O}(k^* n \log n)$.

In the concurrent work by Robson, Spalding-Jamieson, and Zheng [108], a repre-

sensation of the GREEDYINTERVAL function is given as a piecewise linear rational function, with polynomial many parts. After calculating this function, a GREEDYINTERVAL step may be performed in $\mathcal{O}(\log n)$ time, by binary searching to find the part containing the input point, and then evaluation the found rational function in $\mathcal{O}(1)$ time. This will then reduce the revolution time of REPEATEDGREEDY to $\mathcal{O}(k^* \log n)$. It is not given in the paper how this function is calculated, or how much preprocessing time this introduces. By using the techniques presented in the previous paragraph, this may be further studied.

As highlighted, the contiguous art gallery problem separates itself from the other art gallery variants, by being solvable in polynomial time. In analyzing the REPEATEDGREEDY solution, it is inherently obvious, that the simplicity of the problem stems from the contiguous intervals being *local*. That is, what a single guard sees in some part of the polygon affects mostly how the neighboring guards sees their intervals, and not all guards of the polygon. A new variant to study may then be how local the intervals needs to be, to let the problem stay in P. A natural generalization is to extend the problem such that each guard may see up to s contiguous intervals of the boundary. For which parameters of s is the problem in P? For $s = 1$ this is the variant studied in the paper. Our conjecture is that even for $s = 2$, the problem becomes hard, as the locality property disappears. Each guard may link two parts of the boundary together in a tradeoff, which may just be enough to encode a SAT instance, showing the problem to be NP-hard.

Stade [128] shows that the boundary guarding art gallery variant is $\exists\mathbb{R}$ -complete. He does so by a reduction with polygon gadgets. By analyzing these gadgets, one may be able to show that each guard of each gadget one see a constant number of contiguous intervals of the polygon, thereby showing an upper bound on s for when the new variant is $\exists\mathbb{R}$ -complete.

Finally, the problem of determining the complexity of the contiguous art gallery problem for polygons with holes remains open.

Chapter 8

On Finding Longest Palindromic Subsequences Using Longest Common Subsequences

This chapter discusses the paper “On Finding Longest Palindromic Subsequences Using Longest Common Subsequences”, which was published at ESA’24 [35]. This paper was created in collaboration with Gerth Stølting Brodal from Aarhus University and Rolf Fagerberg from the Southern University of Denmark. The included paper is the conference version, which fit all details. The paper is included in Chapter 16. The paper covers properties of the common longest common subsequence algorithm. The techniques may be extended to cover several other such algorithms, which I show as an extension in Chapter 19.

Both during my bachelors, masters and Ph.D. studies, I have been a teaching assistant in the first semester course “Algorithms and Data structures”. In this course, one of the taught subjects is *dynamic programming*, and as the first and main example used for teaching the techniques of dynamic programming is the *longest common subsequence* (LCS) problem. In this problem, the input is two strings s and t , and the goal is to compute a longest string r , such that r is a subsequence of both s and t . The original algorithm proposed to solve this problem is using dynamic programming, and was stated by Wagner and Fischer in 1974 [133].

After teaching the students dynamic programming, they are tasked with solving the *longest palindromic subsequence* (LPS) problem. In this problem, the input is a single string s , and the goal is to compute a longest string p , such that p is a subsequence of s and is a palindrome. We then further require the students to implement their solution in code. The students then find online, that the LPS problem may simply be solved by computing the LCS of the string s and its reverse s^R , which is then stated will be the

LPS¹.

There is some intuition behind this statement that makes it seem like it should hold. The palindrome must be a subsequence of both the string and its reverse, and it must be of the longest length, prompting us to believe that the statement holds. This is however, not always true. Take for example the string $s = ABCCAB$ and its reverse $s^R = BACCBA$, where there are the four LCS solutions $ACCA$, $BCCB$, $ACCB$, and $BCCA$. The first two options are palindromes of the longest length, and are thus the LPS solutions to s . However, the last two strings are not palindromes. What hinders the LCS algorithm from returning these strings?

The paper presents a proof to the claim, that the computed LCS of s and s^R is a LPS of s , when computing with the Wagner and Fischer algorithm. Further, the paper shows that for two different variants of the backtracking performed by the Wagner and Fischer algorithm, that one of them also returns palindromes, and that the other does not. The claim that the LCS of s and s^R being the LPS of s is therefore only true, if the right algorithm for computing the LCS is used.

The paper is split into three sections of results, which shows first the relation between LCS and LPS, without considering the underlying algorithm. Next, properties of the different variations of computing the LCS with Wagner and Fischer is discussed, and it is shown that two of the variations will output the same string, where the remaining algorithm may output a different string. Lastly it is shown that the two agreeing variations computes an LPS, where the last variation may not.

8.1 The Wagner and Fischer Algorithm

The dynamic programming solution of Wagner and Fischer [133] is constructed by recursively computing for the input strings s and t the LCS of a prefix of the two strings consisting of the first i symbols from s and j symbols from t . The variations then arises from how ties are broken in the recursive solution to the LCS. The recursion is shown in [35, Equation 1], and is as below.

$$lcs_{i,j} = \begin{cases} \varepsilon & \text{if } i = 0 \vee j = 0 \\ \text{longest}(lcs_{i-1,j}, lcs_{i,j-1}) & \text{if } i > 0 \wedge j > 0 \wedge s_i \neq t_j \\ \text{longest}(lcs_{i-1,j}, lcs_{i,j-1}, lcs_{i-1,j-1} \cdot s_i) & \text{if } i > 0 \wedge j > 0 \wedge s_i = t_j, \end{cases}$$

The function $\text{longest}(\dots)$ returns a longest string among the input strings, and it is how this function break ties between strings of equal longest length, that creates the variations.

Computing the actual LCS string for each pair of prefixes is expensive as old strings must be cloned to compute the new string. Instead, the algorithm is split into two parts, where first length of the LCS for each prefix pair (i, j) is computed

¹<https://www.geeksforgeeks.org/print-longest-palindromic-subsequence/>,
<https://www.educative.io/answers/longest-palindromic-subsequence-algorithm>,
 both accessed June 25, 2024.

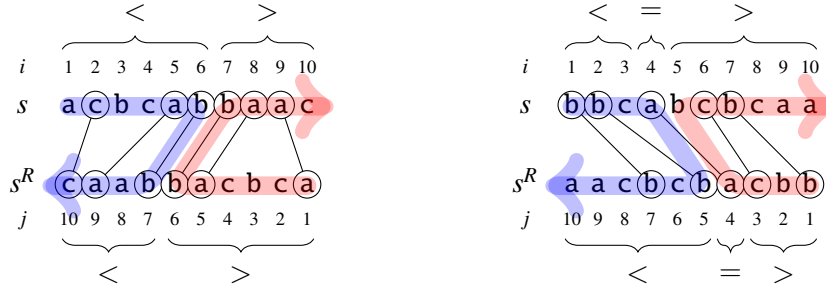


Figure 8.1: ([35, Figure 3]) Proof of [35, Lemma 1]: A common subsequence between s and s^R (shown as circled symbols) and its partition into two palindromic subsequences (blue and red), depending on whether an index is matched to itself (right, index 4) or not (left).

and stored in a table, and then secondly using the table to backtrack only along one solution, computing only a single LCS string. The tie breaking part of the algorithm is then moved from the computation of the strings, to the backtracking part of the algorithm.

8.2 Relationship between LCS and LPS

For this section, the algorithm computing the LCS is not needed, only that some LCS have been computed. Note that such a subsequence may be seen as a set of matches between equal symbols of the input strings. First, the relationship between LCS and LPS is established. This is completed by showing that the length of the LCS of s and s^R is equal to the length of the LPS of s . To show this, two inequalities on the lengths are shown, and used in combination to show equality, in [35, Lemma 2]

In [35, Lemma 1], it is shown that a solution to an LCS of length k of input strings s and s^R may be turned into two palindromic subsequences of combined length $2k$. Each symbol of the output LCS must correspond to a match between a location in s and a location in s^R . These locations correspond to a pair of indexes into s . The pairs may then be grouped into pairs going from a small index in s to a larger index, and from a large index to a small. If the length of the LCS is odd, then there is further a single pair consisting of the same index. This setup may be seen in [35, Figure 3] and cloned in Figure 8.1. Palindromic subsequences may then be created by following for each group the symbols in increasing order by the index in s . By the grouping and the increase in index, it must then hold that the generated string is a palindrome, and that it is a subsequence of both s and s^R .

It must then hold that at least one of these strings is of length at least k . This shows that $\text{LCS}(s, s^R) \leq \text{LPS}(s)$. The other direction follows by any palindromic substring p of s is a subsequence of s and s^R , as $p = p^R$. This then concludes that the lengths are equal.

As the length are equal, then [35, Lemma 3] concludes that both of the generated

palindromic subsequences in [35, Lemma 1] must have length k , as the length is otherwise longer than the LCS. This leads to an algorithm which can convert any $\text{LCS}(s, s^R)$ string into two (possible identical) palindromic subsequences of s . This then further raises suspicion to the claim of the internet: if a simple algorithm with a simple proof allows to convert the output of any LCS algorithm into an LPS string in linear time, then why would the internet claim that we do not need to?

8.3 Properties of LCS algorithms

Next, the paper covers properties of the three variations of the Wagner and Fischer algorithm [133]. For all algorithms, first the table of LCS lengths for each pair of prefixes is computed using dynamic programming. Next, the algorithms use the table for backtracking from the lower right to the top left. At each location, there may be multiple options for how to proceed. The variations are then based on how ties are broken in the backtracking. At each cell (i, j) , if the input symbols $s_i = t_j$, then the backtracking has the option to match the symbols in the output LCS string, and proceed to cell $(i - 1, j - 1)$. This choice is defined as following the $\nwarrow$ arrow. If the cell $(i - 1, j)$ has the same value in the table as cell (i, j) , then the length of the LCS string is the same when removing the last symbol of s . The backtracking may therefore proceed along a $\uparrow$ arrow. Symmetrically, the $\leftarrow$ is defined for when moving to cell $(i, j - 1)$. These arrow orientations corresponds to t being the horizontal string of the table, and s being the vertical, and the indexing is consisting with the table being stored in row wise order. The order of tie breaking between these three possible options, are defined as the *strategy* of the backtracking algorithm. Below is an overview of the three variations studied, and which strategy they use. Note that the backtracking used by Wagner and Fischer [133] is renamed to `ALGMATCH` for consistency naming with the other two variations.

Name	Strategy	Notes
<code>ALGMATCH</code>	$\langle \nwarrow, \uparrow, \leftarrow \rangle$	This is the same as the original backtracking of Wagner and Fischer. The algorithm prioritizes matching symbols when possible.
<code>ALGSKIP</code>	$\langle \uparrow, \leftarrow, \nwarrow \rangle$	Prioritizes skipping symbols from either input before matching.
<code>ALGCONTOUR</code>	$\langle \uparrow, \nwarrow, \leftarrow \rangle$	Follows the top most possible path.

Note that the $\uparrow$ and $\leftarrow$ may be swapped, to gain all six possible strategies. These are however behaving symmetrically, and is equal to swapping the two input strings to the algorithms.

At first glance, the two new algorithms `ALGSKIP` and `ALGCONTOUR` may seem like odd choices for how to backtrack. The `ALGSKIP` algorithm has the advantage, that only the table is needed, and no comparisons of symbols are needed upon backtracking, as the detection of $\uparrow$ and $\leftarrow$ may be made entirely from the table values. If neither

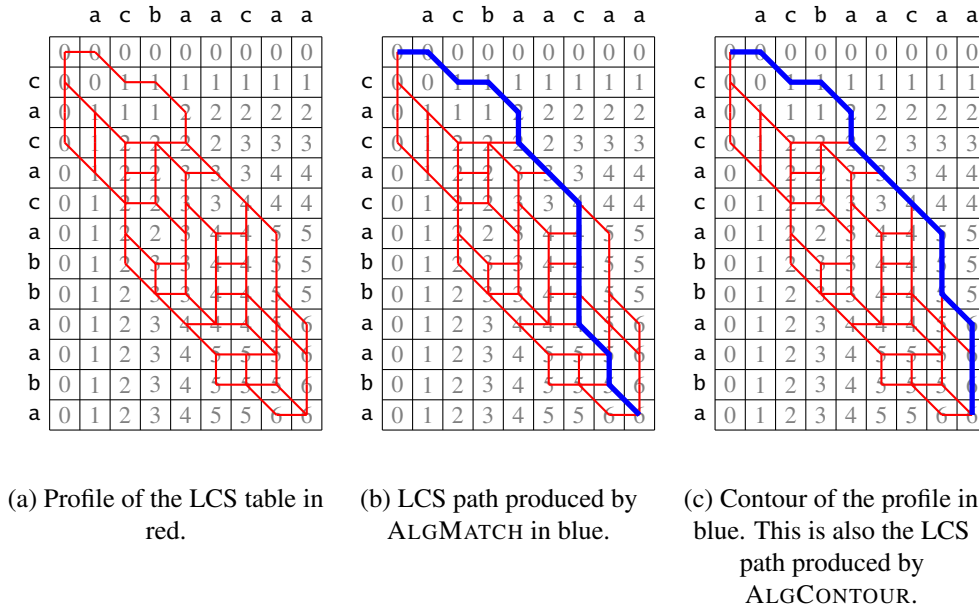


Figure 8.2: [35, Figure 4] LCS table of the strings $s = \text{cacacabbaaba}$ and $t = \text{acbaacaa}$, with the LCS values in gray and the profile overlayed.

are possible, then the $\nearrow$ must be an option, as the table is generated correctly. The ALGCONTOUR algorithm is used for the analysis, and is probably not used in practice.

To see the good thing about ALGCONTOUR, the paper defines what is known as the *profile* of the LCS table, which is the set of all valid LCS paths through the table [35, Definition 5]. Such a profile may be seen in [35, Figure 4(a)] and restated in Figure 8.2a. Here, the top most possible path is a natural path, as defined by [35, Definition 7], which is the path that ALGCONTOUR follows. This is seen in [35, Figure 4(c)] and restated in Figure 8.2c. In contrast, the path of the ALGMATCH seems more random, as seen in [35, Figure 4(b)] and restated in Figure 8.2b, and backtracks somewhere along the middle of the profile. Observe however, the output subsequences produced by ALGMATCH and ALGCONTOUR are both caacaa . The indexes in s where the subsequences are matched to are for ALGMATCH 1, 2, 4, 5, 10, 12 and for ALGCONTOUR 1, 2, 4, 5, 6, 9, which are as expected not the same. However, for the indexes matched in t , they are both 2, 4, 5, 6, 7, 8. The matching indexes are highlighted in Figure 8.3.

That ALGMATCH and ALGCONTOUR produces the same string is not just for one well-chosen input, but can be shown to hold for any pair of input strings s and t . This is shown in [35, Lemma 9], and is the main technical lemma of the paper. The proof goes by showing that ALGMATCH and ALGCONTOUR both output the symbols matches with the same indexes in t , by showing the for each column that they both either leave the column using a $\nearrow$ step or a $\leftarrow$ step. The proof uses by induction that

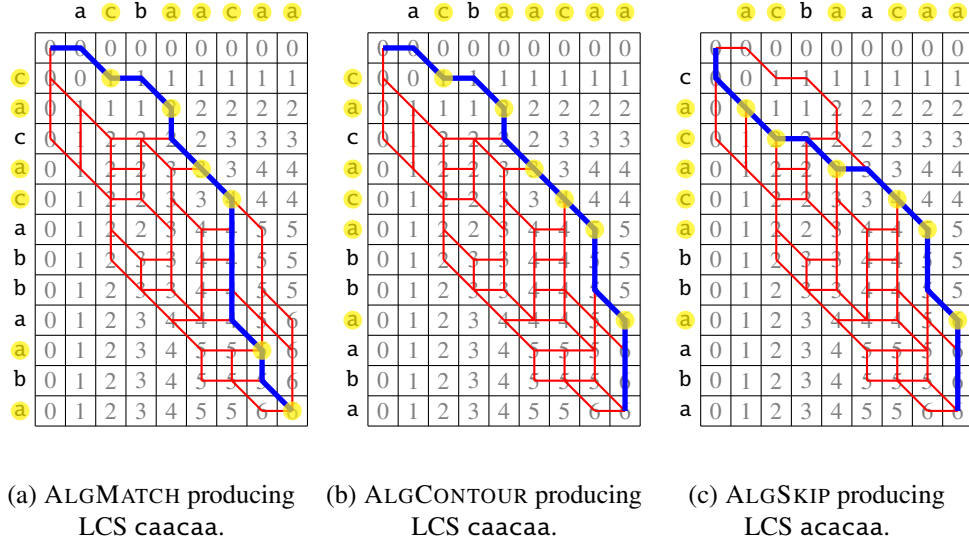


Figure 8.3: LCS table of the strings $s = \text{cacacabbaaba}$ and $t = \text{acbaacaa}$, with the LCS values in gray and the profile overlayed in red. The blue path, is the path taken by the given listed algorithm, with highlighted in yellow, the cells where a $\nwarrow$ step is made, and where the indexes into s and t the match corresponds.

they both start in the column at the same table value k , and that ALGCONTOUR starts on or above ALGMATCH.

In contrast, ALGSKIP does not produce the same output string as ALGMATCH and ALGCONTOUR. As an example, for the input strings $s = \text{cacacabbaaba}$ and $t = \text{acbaacaa}$, ALGSKIP produces the string acacaa , which does not match ALGMATCH and ALGCONTOUR, which can be seen in Figure 8.3.

8.4 Computing an LPS using an LCS algorithm

Finally, the result of the LCS algorithms are used to show that the algorithm of Wagner and Fischer [133] produces an LPS of s , when executed with input s and s^R . This is completed by first showing that the profile of the LCS table for s and s^R is symmetric [35, Definition 10, Lemma 11]. The symmetry is immediately apparent when observing the profile of s and s^R , as seen in [35, Figure 8a], with the profile restated in Figure 8.4. It is then shown that *any* symmetric path produces a palindromic string [35, Lemma 12]. As the profile is symmetric, it is easy to show that the contour produced by ALGCONTOUR is symmetric [35, Lemma 13], and therefore produces a palindromic string. As the string produced by ALGMATCH is the same as the output of ALGCONTOUR, then it must hold that ALGMATCH produces a palindromic string [35, Theorem 14]. In contrast, the output of ALGSKIP on s and s^R may be non-palindromic, as shown in [35, Lemma 15].

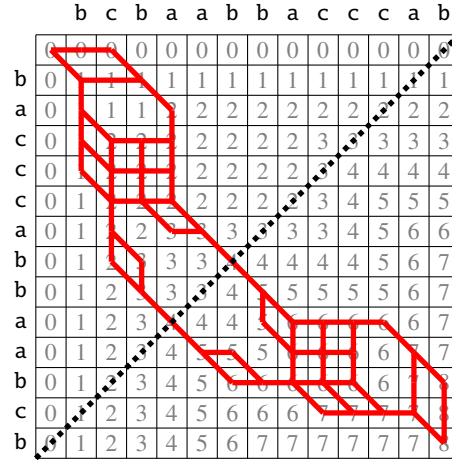


Figure 8.4: ([35, Figure 8(a)]) LCS table of the string bacccabbaabcb and the reversed string, including the profile (in red) with highlighted symmetry of the profile over the top-right to the bottom-left diagonal (dotted).

Table 8.1: ([35, Table 1]) The six different strategies to deterministically backtrack when computing $\text{LCS}(s, s^R)$. E.g., $\langle \nearrow, \uparrow, \leftarrow \rangle$ denotes that the prioritized backtracking order of moving from (i, j) is to $(i-1, j-1)$, $(i-1, j)$, $(j-1, j)$ (this corresponds to the Wagner and Fischer algorithm [133]). We prove in this paper that the four strategies on the left always report a palindrome, whereas the two strategies on the right might fail to report a palindrome.

LCS(s, s^R) is always a palindrome				LCS(s, s^R) can be non-palindromic			
$\langle \nearrow, \uparrow, \leftarrow \rangle$	$\langle \nearrow, \leftarrow, \uparrow \rangle$			$\langle \uparrow, \leftarrow, \nearrow \rangle$	$\langle \leftarrow, \uparrow, \nearrow \rangle$		
$\langle \uparrow, \nearrow, \leftarrow \rangle$	$\langle \leftarrow, \nearrow, \uparrow \rangle$						

The paper therefore conclude that for some strategies the LCS of s and s^R is a palindrome, and a solution to the LPS of s , but for some strategies, they are not. The claim by the internet, is therefore true and provable, but not for just any LCS algorithm. An overview of which variants produce palindromes, and which do not is given in [35, Table 1], which is restated in Table 8.1. Specifically, the taught method of Wagner and Fischer [133], both in the course and on the internet, the output is, as claimed, a palindrome.

Chapter 9

Pure Binary Finger Search Trees

This chapter discusses the paper “Pure Binary Finger Search Trees”, which was published at SOSA’25 [40]. This paper was created in collaboration with Gerth Stølting Brodal from Aarhus University. The included paper is the conference version, which allowed all details. The paper is included in Chapter 17. New extensions to the result, including new finger search algorithms and experiments, are included in Chapter 20.

A classical algorithm taught as the first algorithm to most computer science students is *Binary Search*. The algorithm is simple in nature – using a sorted list of size n , a value may be found using only $\mathcal{O}(\log n)$ comparisons, by repeatedly comparing the target value to the middle of the remaining candidates of the list. Binary Search was first described in lecture notes by John Mauchly in 1946 [97], and as noted by Knuth [86, Section 6.2.1, page 422], this is one of the first nonnumerical algorithms described in the literature. While Binary Search allows for efficient queries, updating the list is not efficient. This stems from the required ability to perform random access in constant time, which prompts the list to be an array. Updating the list with new values may therefore take $\Omega(n)$ time, as the elements of the list must be moved to make space for the new value. This led to the invention of *Binary Search Trees*, which orders the values into a tree in roughly the pattern that Binary Search would access the values [86, Section 6.2.2].

However, in a list, if some value x at some location is already known, then searching for another value y may be done from x using *Exponential Search*, created by Bentley and Yao [21]. When the known value x and the searched value y are d locations apart in the list, denoted as the *rank difference*, then y may be located in time $\mathcal{O}(\log d)$. This searching scheme does, however, not carry over to the binary search trees, as a search can only traverse the structure of the tree, and even when the elements have small rank difference, the search may need to traverse far up the tree before finding the subtree containing the value searched for.

Searching from a known location for another value is also known as *finger searching*, where the *finger* is some type of pointer into the data structure, pointing to the known value and its location. For the case of finger searching in a list, the finger to

x may simply be the index in the list where x resides. Not being able to search in binary trees lead to the invention of *finger search trees*, which allows for efficient finger searching in $\mathcal{O}(\log d)$ time, see, e.g., the *level-linked* (a, b) -trees by Huddleston and Mehlhorn [77].

9.1 Overview of the Paper

The paper presents a new type of Finger Search Trees, which allows for finger searching in $\mathcal{O}(\log d)$ time, and finger updates in amortized $\mathcal{O}(1)$ time. A finger update is inserting a predecessor or successor element to a node pointed to by a finger, or deleting the node at a finger. This result in itself is not new, as the level-linked (a, b) -trees by Huddleston and Mehlhorn [77] achieves the same time bounds. The novel result of the paper is that the runtimes are achieved in the *pure binary tree* model. In this model, each node is similar to that of an unbalanced binary search tree; they only contain a value and a pointer to their parent and left and right child. The model does not allow any additional nodes, only the nodes corresponding to an insertion of a value, which have not been deleted, is allowed. In contrast, most balanced trees contains additional information in the nodes of the tree, such as the red-black trees of Guibas and Sedgewick [74], where each node contains an additional bit denoting if the node is red or black.

The paper is split into four separate parts, where each part builds upon the previous, such that the abstraction increases. In the paper, the order is such that the highest level of abstraction is presented first. This order was chosen to allow readers to maintain a high level of abstraction before diving into the lower levels of details. In this dissertation, I will instead present them in a different order, which better aligns with how the result was obtained.

The work in the paper was inspired by previous work by Brodal. At FUN'24 Brodal published the paper “Bottom-Up Rebalancing Binary Search Trees by Flipping a Coin” [30], in which he tried to create balanced binary trees, which allowed for finger insertions at the leafs, without storing additional information in the nodes. The result is based on randomization, and shows two randomized rebalancing schemes, both of which runs in amortized $\mathcal{O}(1)$ time. For each scheme the paper shows that there exists insertion sequences of length n that shows the expected height to be $\mathcal{O}(\log n)$. However, there also exists simple sequences for each scheme where the height of the tree is $\Omega(n)$.

This prompted us to work on this result: Could we create deterministic balanced trees, with no information stored in the nodes, where the amortized insertion time from a leaf is $\mathcal{O}(1)$? In the following sections I will give an overview of the process which allowed us to answer yes to this question. An overview of the related work and the new result of the paper is stated in [40, Table 1], which is restated in Table 9.1.

Reference	Finger updates	Information stored
AVL trees [5]	$\mathcal{O}(\lg n)$	2 bits per node
Red-black trees [74]	$\mathcal{O}(\lg n)$	1 bit per node
Red-black trees + predecessor/successor links [131]	$\mathcal{O}_A(1)$	$\mathcal{O}(\lg n)$ bits per node
Scapegoat trees [7, 67]	$\mathcal{O}_A(\lg n)$	global tree size
Level linked B-trees [77]	$\mathcal{O}_A(1)$	$\mathcal{O}(\lg n)$ bits per node
BST with $\mathcal{O}(\lg^2 n)$ sized bags [93]	$\mathcal{O}(1)$	$\mathcal{O}(\lg n)$ bits per node
Treaps [119]	$\mathcal{O}_E(1)$	$\mathcal{O}_E(1)$ bits per node
Seidel random BST [118]	$\mathcal{O}_E(\lg^2 n)$	none
Splay trees [120]	$\mathcal{O}_A(\lg n)$	none
Encoded 2-3 trees [47, 48]	$\mathcal{O}(\lg n)$	none
<i>Our Paper</i> [40]	$\mathcal{O}_A(1)$	none

Table 9.1: ([40, Table 1]) Results for selected balanced BSTs. $\mathcal{O}_A$ and $\mathcal{O}_E$ denote amortized and expected bounds, respectively. All search trees have logarithmic height, and support searches in logarithmic time, except [118, 119] are randomized and bounds are expected, and bounds for [120] are amortized, with no bound on the tree height.

9.2 Encoding extra Information in Metanodes

That the extra information in the nodes of a binary tree may be removed is not a new idea. In 1978 Brown described a way of pairing pure nodes together in what he denoted as *supernodes* [47, 48]. Each supernode then encodes a bit in the rotational order of the two nodes, which describes enough information to make an encoding of a balanced 2-3 tree. See Figure 9.1 for an illustration of this encoding scheme. Here x and y are pure nodes, containing two consecutive values of the tree, which form a supernode. The supernode, independent on the inner rotation of x and y , has a parent and left and right child, all of which are supernodes. As x and y contain consecutive values, then the supernodes form a valid binary tree, with the value of each supernode being the range from x to y . Depending on which node is the root of the supernode, this may encode a single bit b .

As the tree may contain an odd number of nodes, Brown allows half supernodes at the bottom of the tree, consisting of only a single node. The 2-3 trees does not allow for amortized $\mathcal{O}(1)$ insertion time from a leaf. However, the supernodes may be used to create red-black trees, which Tarjan proved supports insertion from a leaf in amortized $\mathcal{O}(1)$ time [131]. What the encoding of Brown does not allow, is to detect for any given pure node if it is the root of a supernode, which is critical for supporting fingers into the tree at pure nodes, for both queries and updates. However, as the insertions of Brown proceeds from the root in an initial search for a leaf, the root of the supernodes may be detected on this downwards search, and similarly for the upwards fixup.

We therefore needed to create a new constant size structure on continuous pure

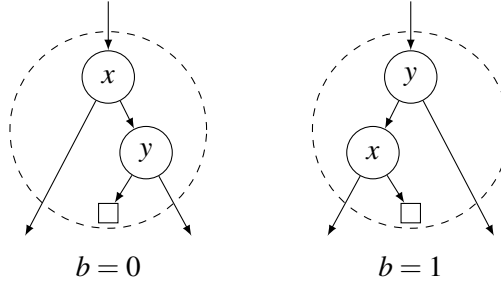


Figure 9.1: Encoding scheme of Brown [48], where two consecutive nodes x and y are grouped in a supernode, marked by the dashed circle. Depending on the inner orientation of x and y , a bit b may be encoded.

nodes, which allows for determining if a given node is the root of the structure. To avoid overlap with the structure of Brown, we let *metanode* denote the constant size structure of pure nodes, which represents a single node in the encoded tree. We let our metanodes be of a variable size, such that all pure nodes of the tree is contained in a metanode, unless fewer than a constant number of pure nodes is contained in the tree, in which case all operations must run in $\mathcal{O}(1)$ time. The metanodes therefore has a buffer, which allows a metanode to split into two metanodes, when the buffer is sufficiently large, as well as merge with a neighboring metanode. Additionally, the extra pure nodes allowed us to encode multiple bits into each metanode. There exists countless encoding schemes of metanodes, and in the paper we show one of them [40, Section 5], which is restated in Figure 9.2.

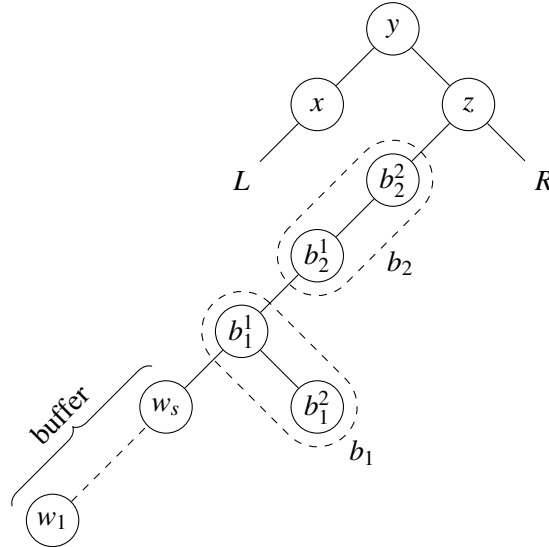


Figure 9.2: ([40, Figure 12]) A metanode with root y , encoding two bits $b_1 = 1$ and $b_2 = 0$, and L and R are the roots of two metanodes (possibly NIL).

This extended metanode allows for maintaining fingers to any pure node, and from

the pure node find the root of the metanode containing it in $\mathcal{O}(1)$ time [40, Lemma 1]. These metanodes may then be used as the nodes of an encoded red-black tree, by using a bit to encode the color of the metanode. An insertion at a leaf may therefore proceed by first detecting the root of the metanode containing the leaf, to then insert the new value in the metanode, while preserving the structure. If the buffer of the metanode then becomes large enough to split the metanode in two metanodes, the new metanode may then be inserted as a leaf in the encoded red-black tree, which then takes amortized $\mathcal{O}(1)$ time, where the metanodes structure may be detected along the fixup procedure.

The metanodes was chosen as above to allow for further reducing the size of each pure node. Notice that in the structure of the metanode, each pure node only has a right child if it also has a left child. We can therefore reduce the number of pointers in each pure node, by instead of storing a pointer to the parent and left and right child, instead only store a pointer to the left child, and a combined sibling/parent pointer, which either points to the right sibling if the pure node has one, and otherwise points to the parent. This is not a new idea, as described in [40, Section 6], it was described by, among other sources, by Knuth [86, page 477]. Importantly, as the left pointer is always to the left child, we do not need a bit or comparison to distinguish for a node with a single child, if the child is a left or right child.

The results presented in this section are not novel, and we therefore decided to extend the scope to allow for finger insertions of a predecessor or successor at any pure node, and not only the leafs, using a finger to the pure node, in amortized $\mathcal{O}(1)$ time.

9.3 Folding a Balanced Binary Tree

Recall that the metanodes presented in the previous section, allows creating encoded trees where each encoded node has a pointer to the parent and left and right child, as well as allowing for an arbitrary constant number of bits in each encoded node. To solve the new problem of efficient finger insertions, we therefore may assume the tree to contain simple nodes with constantly many bits. Note specifically that the goal was not yet to allow for efficient finger deletions or finger searching.

First we observed the following: in the balanced red-black trees, insertions consists of two steps. First, a downwards search is performed, to find a leaf of the tree, where the newly inserted value would reside, and the value is then simply inserted at this location. This may then break the balance property of the tree, and the insertion therefore secondly performs an upwards fixup routine on the leaf, to restore the balance. The search takes time $\mathcal{O}(\log n)$, for a tree of n values, as the tree is balanced, whereas the fixup takes amortized $\mathcal{O}(1)$ time [131]. This is also the case for finger insertions at any internal node, where the initial search is proportional to the height of the node at the finger, but the fixup remains amortized $\mathcal{O}(1)$. The goal is therefore to reduce this initial search.

Upon a finger insertion at some node v , we only allow for inserting a predecessor

or successor to v . The search may therefore be faster, as we do not need to ensure the ordering, as this is given by the update. For simplicity the following only discusses inserting a successor u . If this simple operation is supported, predecessor insertions may be performed by swapping u into the location where v resides in $\mathcal{O}(1)$ time, and then inserting v as the successor of u . The newly inserted successor u must have a value larger than v , but a value smaller than all nodes residing in the right subtree of v . Intuitively, in the binary tree structure, the newly inserted value u resides on the edge between v and its right subtree, see, e.g., [40, Figure 3] for an illustration. The leaf to insert u into is therefore at the bottom of a straight left path in the right subtree, which may be $\mathcal{O}(\log n)$ away. If we could delay the insertion, until $\mathcal{O}(\log n)$ elements have been inserted as the successor of v , we can then traverse down the $\mathcal{O}(\log n)$ height of the tree, to insert all elements at once, allowing the amortized time for each element to be $\mathcal{O}(1)$. On the journey of finding the folding presented in the paper [40, Section 4], we tried a couple of approaches as presented below.

9.3.1 First Idea: Adding Buffers to Edges

The first idea was to add a buffer onto each edge, and upon inserting a successor to v , the newly inserted element is rotated into the buffer. Each buffer must then recursively be a balanced tree with possible buffers on the inner edges, to allow inserting the whole tree into the right subtree of v , by joining the two trees [130]. This requires being able to recognize if a node is the buffer root on an edge or part of the outer tree. However, this may be maintained by using an additional bit, which by the metanodes, we are allowed to have. Notice that we cannot maintain counters in the tree, so we do not know when the buffer size is $\approx \log n$. We can however use that the black height of the buffer only increases when the root changes color from red to black, in which case we know a minimum number of elements contained in the tree. We can then traverse down the left spine of the right subtree of v in the outer tree, to check if the size is large enough. As the number of inserted elements must roughly double each time this happens, it may still be amortized away.

Next, we noticed that upon successor insertions in the outer tree, the newly added elements to the buffer would always be a new minimum element in the buffer. The height of the tree in the buffer is however $\mathcal{O}(\log \log n)$, which is not constant. It does reduce the height from the outer tree. The second idea was therefore to invert the left spine of the buffer, such that the bottom of the spine, which contains the smallest location in the buffer, is close to the root of the buffer. This does increase the distance between the root of the buffer, and its original left child, which we need access to in constant time, upon a rebalancing operation reaching the root. Notice however that it will only do so if the rebalancing has traversed the whole height of the tree, which allows for traversing to the left child of the root, without an increase of the amortized time.

9.3.2 Final Construction

Taking a step back, one may notice that we have created buffers, which allows for inserting a new smallest element into a tree, from the root in amortized $\mathcal{O}(1)$ time. This is exactly what we wanted to do in the right subtree of v when inserting a successor. We may therefore discard the recursive structure of buffers on the edges. One problem the reversing of the left spine of a tree completely, is that the height of the tree is doubled. To allow inserting successors at each node of the tree, each left spine of each subtree must be reversed, prompting the tree to have height $\mathcal{O}(\log^2 n)$. Additionally, the long distance between the root of each subtree to the left child prompts problems with the amortized rebalancing time.

We therefore finally created the *folding* of a spine as presented in the paper [40, Section 4], which allows for the distance between a node to its parent and left and right child in the unfolded version of the tree, to be $\mathcal{O}(1)$ in the folded tree. The fold consists of alternating nodes from the upper and lower part of the spine. The folding of a left spine may be seen in [40, Figure 7]. Additionally, the fold provides that the distance from the root of a subtree to the bottom of the left spine is $\mathcal{O}(1)$, which allows for finger inserting a successor to any node in $\mathcal{O}(1)$ time. Using a long case analysis, we further show that the rebalancing performed upon insertions may be performed on the folded trees, while maintaining the folded structure, in amortized $\mathcal{O}(1)$ time. We further show that finger deletions in the folded trees are possible in amortized $\mathcal{O}(1)$ time.

The folding of left spines allows for efficiently locating the successor leaf to a node in time $\mathcal{O}(1)$. However, it does not allow for locating the location of the successor node already contained in the tree in time $\mathcal{O}(1)$, as the node may reside further upwards in the tree. By alternating between folding the left and right spine of subtrees, location both the predecessor and successor in the tree may be supported in $\mathcal{O}(1)$ time.

In total, we have created a representation of a balanced binary tree, where access to the parent, left and right child, as well as the predecessor and successor of each node is available in $\mathcal{O}(1)$ worst-case time, all while each node of the tree is pure. The tree allows for finger updates in amortized $\mathcal{O}(1)$ time.

9.4 Supporting Finger Searching

The final step of the paper, was showing that finger searching is possible in a balanced tree, when access to the predecessor and successor is given in $\mathcal{O}(1)$ time. One such algorithm is presented in [40, Section 3]. Specifically, finger searching starts at a finger pointing to a node v in the tree, and searches for a value e contained in some unknown node u . Note that as the predecessor and successor may be accessed in $\mathcal{O}(1)$ time, then if e is not contained in the tree, we may freely return either the predecessor or successor node to value e .

The algorithm searches for the value e in a continuous range of $\Theta(2^h)$ nodes from v in time $\mathcal{O}(h)$. By then exponentially increasing h , then for the final search

$h = \mathcal{O}(\log d)$, and the search for u is performed in time $\mathcal{O}(\log d)$.

9.5 Worst-case Lower Bounds and a New Tree Structure

The finger updates of the tree are amortized $\mathcal{O}(1)$ time. One may ask: can we do better than amortized? As the tree allows access to the predecessor and successor in $\mathcal{O}(1)$ time, all while each pure node contains only a value and two pointers, this resembles a doubly linked list, which allows for updates in worst-case $\mathcal{O}(1)$ time. However, our model is that of pure binary trees, and we still want to maintain that searching from the root of the tree takes time $\mathcal{O}(\log n)$, i.e., that the tree is balanced. In the paper [40, Section 8] we show that under these two constraints, even just insertions in leafs in worst-case $\mathcal{O}(1)$ time is not possible.

In building this lower bound, the initial model was just that each node contains a value and two pointers, and not that the nodes must form a binary search tree. Under this more relaxed constraint, the bound proved impossible to prove, which lead us to discover the *Linked list metanodes* as described in [40, Section 9]. In this structure, the pure nodes of the tree is contained in a global single linked list. Each pure node then contains a value, a pointer *next* pointing to the next pure node in the linked list, and an additional *info* pointer, i.e., a single value and two pointers. Using the *info* pointer, the linked list may be partitioned into metanodes, where the metanodes are recognizable from any pure node it contains, by devising a special structure on the head and tail pure node of each metanode. This structure is depicted in [40, Figure 14], and for easy reference, the structure may also be seen in Figure 9.3. The intermediate pure nodes may then freely use their *info* pointer to point to other metanodes, or by being set or not encode a bit. This directly allows the metanodes to encode a red-black tree with explicit predecessor and successor pointers.

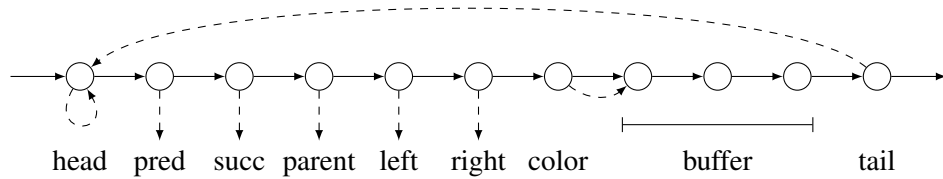


Figure 9.3: ([40, Figure 14]) A linked-list metanode representing a node in a red-black tree with predecessor and successor links. Horizontal arrows are *next* pointers, and dashed arrows are *info* pointers.

Part II

Publications



Space-Efficient Functional Offline-Partially-Persistent Trees with Applications to Planar Point Location

Gerth Stølting Brodal^(), Casper Moldrup Rysgaard^(),
Jens Kristian Refsgaard Schou^(), and Rolf Svenning^()

Department of Computer Science, Aarhus University, Aarhus, Denmark
{gerth,rysgaard,jkrs,rolfsvenning}@cs.au.dk

Abstract. In 1989 Driscoll, Sarnak, Sleator, and Tarjan presented general space-efficient transformations for making ephemeral data structures persistent. The main contribution of this paper is to adapt this transformation to the functional model. We present a general transformation of an ephemeral, linked data structure into an offline, partially persistent, purely functional data structure with additive $\mathcal{O}(n \log n)$ construction time and $\mathcal{O}(n)$ space overhead; with n denoting the number of ephemeral updates. An application of our transformation allows the elegant slab-based algorithm for planar point location by Sarnak and Tarjan 1986 to be implemented space efficiently in the functional model using linear space.

Keywords: Data structures · Functional · Persistence · Point location

1 Introduction

The functional model has many well-known advantages such as modulation, shared resources, no side effects, and easier formal verification [4, 14]. These advantages are given by restricting the model to only use functions and immutable data. As all data are immutable, side effects in functions are not possible, allowing modules to work independently of the context they are used in and reducing the complexity of formal verification [14].

In 1999 Okasaki [19] gave a seminal work on techniques for designing efficient (purely) functional data structures, and our result follows this line of research. Adapting existing data structures to the functional model is non-trivial since modifications are prohibited. However, it also means that updates do not destroy earlier versions of the data structure making functional data structures inherently persistent, but not necessarily space efficient. The focus of this paper is to adapt existing imperative techniques for persistence to the functional model in a space-efficient manner.

We introduce a purely functional framework that adapts classical tree structures to support offline partial persistence with an additive overhead. By offline partial persistence, we mean that all updates are made before queries. This

Work supported by Independent Research Fund Denmark, grant 9131-00113B.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
P. Morin and S. Suri (Eds.): WADS 2023, LNCS 14079, pp. 644–659, 2023.
https://doi.org/10.1007/978-3-031-38906-1_43

Table 1. Previous and new results for planar point location, where † are expected bounds and * are results based on persistent data structures.

Reference	Construction	Query	Space	Model
David Kirkpatrick [17]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative
Seidel [26]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative †
Dobkin and Munro [11]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n \log n)$	Imperative *
Richard Cole [8]	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative *
Sarnak and Tarjan [24]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative *
Sarnak and Tarjan [24]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$	Functional *
<i>New</i>	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Functional *

restriction allows us to store update information without immediately being able to handle queries efficiently. A data structure that is not persistent, i.e., does not support queries and updates in previous versions, is said to be *ephemeral*. We show that by recording when an ephemeral tree structure is updated, we can build a query structure that can efficiently answer queries to previous versions of the structure. In the imperative paradigm, it is possible to efficiently interweave updates and queries [12, 24], but in the functional paradigm, it incurs a multiplicative logarithmic space overhead, which we show how to circumvent, in the offline setting.

Planar point location is a classic computational geometry problem [13, 17, 24]. Given a planar straight-line graph with n edges (interchangeably line segments) and report the region containing a query point q . The task is to create a data structure that supports these queries while minimizing the construction time, query time, and space of the data structure. Sarnak and Tarjan [24] showed that the planar point location problem can be solved elegantly using *partially-persistent* sorted sets, that in the functional setting can be solved with balanced search trees using path copying resulting in a space usage of $\mathcal{O}(n \log n)$. As an application of our technique, we show how the algorithm of Sarnak and Tarjan can be implemented in the functional model to only use space $\mathcal{O}(n)$. For an overview of results for the planer point location problem see Table 1.

Below we state sufficient conditions for a functional or imperative data structure to be augmented with a functional support structure that supports offline partial persistence queries.

Definition 1 (TUNA conditions).

- T:* The data structure forms a rooted tree of constant degree d .
- U:* Updates create $\mathcal{O}(1)$ new edges and nodes.
- N:* No cycles are created by updates when considering the edges that have been created across all versions.
- A:* Attribute values of nodes are static, i.e., the information stored in a node is not changed after its insertion. This does not include fields pointing to the children.

646 G. S. Brodal et al.

This definition is not too restrictive, we show that binary search trees (BST), Treaps [3], Red/Black Trees [5], and Functional Random Access Arrays [18] all can be modified to satisfy Definition 1, without asymptotically significant overhead. In Sect. 5 we discuss how some of these requirements can be relaxed or generalized further.

Theorem 1. *For any ephemeral, linked data structure that satisfies the TUNA conditions, an equivalent, functional, offline-partially-persistent data structure preserving the asymptotic update and query times, can be created, with an additive construction overhead of time $\mathcal{O}(n \log n)$ and space $\mathcal{O}(n)$ for a series of n updates.*

The main idea behind Theorem 1 is to store the update information in a list, including edge insertion and deletion timestamps. After all the updates have been applied, a bottom-up topological sort produces a directed acyclic graph (DAG) that supports queries to any previous version of the data structure. This is essentially implementing the node copying approach of [24], while carefully avoiding the creation of cycles.

Building upon [24], Theorem 1 immediately implies a state-of-the-art functional planar point location solution, summarised in the following corollary.

Corollary 1. *There exists a purely functional solution to the planar point location problem with construction time $\mathcal{O}(n \log n)$, query time $\mathcal{O}(\log n)$, and space $\mathcal{O}(n)$.*

We implemented unbalanced binary search trees in the purely functional programming language Haskell and report on some experiments in Sect. 6.

1.1 Persistence

A data structure is said to be *persistent* if it is possible to query previous versions of it and *ephemeral* if it is only the current version that is available. A *partially-persistent* data structure is *persistent* and allows updates only to the latest version. The stronger notion of *full persistence* implies that any version can be both queried and updated. An update to a persistent data structure never changes an existing version, but instead creates a new version derived from the version the update is applied to. In this way, the different versions of the partial persistent structure form a version list, whereas full persistence forms a version tree. General transformations to make data structures persistent were studied by Driscoll et al. [12] and Overmars [21, 22]. In this paper, we focus on *offline* partial persistence for linked data structures, where all updates are performed before all queries, which was also explored in [11].

A naive idea to achieve partial persistence is to store a copy of every previous version. If the underlying structure is a list, then this approach generates an overhead of $\Omega(n^2)$ space for n insertions into an initially empty list. To improve upon this, the crucial observation is that when structures have a large overlap between updates, it is possible to reuse large parts of the previous versions and

greatly reduce space usage. To achieve linear space, the notions of *fat nodes* and *timestamps* were introduced by Driscoll et al. [12], where each pointer field in a node is replaced by a list of pointers and each pointer in the list has an associated timestamp, denoting when the pointer was updated. Specifically, for a binary search tree, each node will, instead of containing a pointer to the **left** and **right** subtree, contain a list of timestamped **left** pointers and a list of timestamped **right** pointers. An update to a given pointer now adds a new pointer to the pointer list with the current timestamp, resulting in $\mathcal{O}(1)$ space overhead per update. As the pointer in the version is the last in the list, there is no overhead in finding the active pointer in the current version. To locate the correct pointer at a given older timestamp, a binary search on the list of pointers can be performed, which then imposes a multiplicative $\mathcal{O}(\log n)$ overhead on queries.

In [24] the notion of *path copying* was introduced for BSTs, where all nodes on the path to the node being updated are copied. Any existing pointer along the path can then point freely to parts of the old structure. For BSTs, this has a space overhead of the length of the path, i.e., for balanced BSTs an $\mathcal{O}(\log n)$ space overhead per update. It does however impose no overhead on the query time, apart from initially finding the correct root to query.

Thus, the fat node technique has a query time overhead, whereas path copying has a space overhead. By combining these two techniques, it is possible to have no overhead on query time and space. In [12, 24] the authors achieve this by introducing the *node copying* technique for partially-persistent general pointer-based data structures. Here nodes are allowed to hold a constant number of additional time-stamped pointers. When some operation requires a node to update a pointer, the timestamp of the old pointer is updated to end at the current time. We denote a pointer that has ended as *expired*. If the pointer is replaced by a new pointer, then the new pointer is placed in one of the free extra pointer slots of the node, with a timestamp starting at the current time. If there is no free pointer slot *node copying* is performed, where the non-expired pointers and the new pointer are copied to a new node. Any pointer in another node to the copied node at the current time must be split, which may cause node copying to cascade up the structure. However, an amortization argument [24] shows that this technique only has an additive $\mathcal{O}(n)$ overhead in space, when the indegree of every node in the underlying structure is constant. Finally, as the number of pointers in each node is constant the overhead on the query time is also constant.

1.2 Persistence and the Functional Model

The functional programming paradigm is well suited for persistence as stated by Okasaki: “A *distinctive property of functional data structures is that they are always persistent*” [19]. In purely functional programming, there are no side effects and variables are immutable, meaning that any modifications to a structure S are obtained by creating a new structure S' without altering S . In this new

648 G. S. Brodal et al.

structure S' , substructures may be references to (immutable) old substructures from S .

Purely functional data structures are consequently particularly interesting when persistence is critical, and it is natural that they are less efficient than imperative data structures, since they inherently solve a more general problem. We note that there has been some work in describing to what degree mutability increases the capabilities (efficiency) of a language, and Pippenger [23] gave an example of a problem with a logarithmic-factor separation under certain conditions. However, for many data structure problems, purely functional solutions have been developed that match their imperative counterparts with only constant overhead. Examples include optimal *confluently* persistent deques which were developed over a number of papers [7, 9, 16] and optimal priority queues [6].

1.3 Functional vs. Space Efficient Imperative Persistence

The fat node and node copying persistence techniques mentioned in Sect. 1.1 rely upon the imperative paradigm's ability to modify pointers in the nodes in a graph structure where nodes can have multiple ingoing edges. As the functional model cannot mutate pointers, directly translating these solutions leads to significant overhead in the update time, as, even with path copying, all ancestors of an updated node must be remade to point to the newly created node(s). For this reason, we focus our attention on data structures with a tree structure where ancestors appear on a single path to the root.

1.4 Planar Point Location

Dobkin and Lipton [10] solved the planar point location problem by drawing vertical lines through every node resulting in vertical slabs. For every slab, the line segments spanning the slab are stored in a BST. This method allows efficient queries by performing a binary search horizontally for the slab, and then a binary search vertically in the slab for the region. This gives an overall query time of $\mathcal{O}(\log n)$. The drawback of this method is that each line could potentially be stored in almost every slab, resulting in $\Theta(n^2)$ space. A number of different results [8, 13, 17, 24], show that the space can be reduced to $\mathcal{O}(n)$ (non-functional) without affecting the query time. The solution by Cole [8] is particularly interesting as it exploits that neighboring slabs are very similar, meaning that the problem can be reduced to creating persistent, sorted sets. This observation is vital to the work on persistent search trees by Sarnak and Tarjan [24]. On the other hand, the approach by Kirkpatrick [17] is completely different and is based on repeatedly triangulating the graph and removing a constant fraction of the nodes with degree at most 11. Then a DAG is created bottom-up based on the overlap between two consecutive triangulations. Since the DAG is created bottom-up, similarly to our approach described in Sect. 3.2, we conjecture that this approach could be adapted to the purely functional model and leave it as an open problem.

2 Initial Analysis of Binary Search Trees

In this section, we introduce ephemeral (i.e., non-persistent) unbalanced binary search trees (BSTs) and illustrate how the fat node technique can be adapted to the functional paradigm. The main obstacle in making the adaptation is to avoid cycles between nodes. We solve this by making new copies of nodes that should be moved above their parent. In Sect. 5 these techniques are extended to cover balanced search trees.

2.1 Ephemeral Binary Search Trees

Any BST node is either an empty leaf or a node containing an element and two pointers to a left and right sub-tree, which are in turn also BSTs. Forthwith, these pointers will be denoted as *edges*. Likewise, let T and T' denote BSTs over a totally ordered set of elements X and let $x \in X$ denote an element. BSTs are ordered such that all elements in the left subtree are smaller than the element in the node, and all elements in the right subtree are larger. BSTs support many operations; we focus on the following three basic operations:

- **Insert**(T, x): Insert x into T and return the resulting tree T' .
- **Delete**(T, x): Delete x from T and return the resulting tree T' .
- **Search**(T, x): Return the smallest element x' in T , such that $x \leq x'$.

All operations can be implemented in time linear in the height of the tree. We call operations that modify the data structure *updates* (**Insert** and **Delete**). We call operations that query the data structure without changing it *queries* (**Search**). When constructing the data structure, we consider a sequence of n updates $(u_1, \dots, u_n)$ and for version $0 \leq t \leq n$ updates $u_1, \dots, u_t$ have been applied.

2.2 Fat Node Binary Search Trees

In this subsection, we describe how to adapt imperative unbalanced BSTs to adhere to the TUNA conditions (Definition 1), using fat nodes. Most importantly, the **Delete** update is changed slightly from the classic behavior since it can cause nodes to be reordered in the tree.

When performing **Insert**, assuming that x is not present in T , a path to the correct leaf position is found, and a new node, containing x is created at the position of that leaf. The difference between the old and the new tree is a single edge at the bottom of the tree. By adding *creation timestamps* on the edges to represent creation time, queries can detect if a particular edge should be considered to exist in a specific version or not.

When performing **Delete**, if the deleted element x is at the bottom of the tree, then, in effect, the edge e from its parent p to the deleted node ceases to exist. We record this by adding an *ending timestamp* to the edge. If the deleted element x is in some internal node, see Fig. 1 (Left), then a predecessor

650 G. S. Brodal et al.

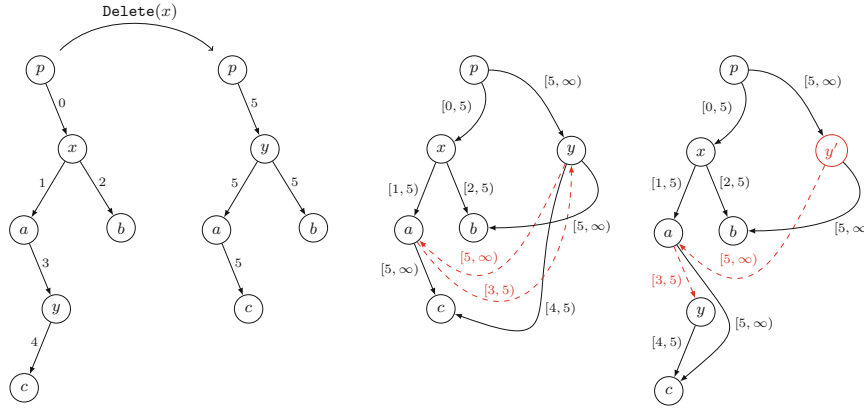


Fig. 1. We let a single number above an edge denote its creation time. In the graphs, the interval above an edge is the living span of the edge. Recall that a persistent search tree query is a timestamp/version and a value. Left: a deletion in a BST. Middle: the resulting graph contains the dashed cycle between y and a . Right: by creating a new copy y' of y the resulting graph remains a DAG and is TUNA compliant.

or successor y (depending on implementation) in the subtree rooted at x is found and moved to the deleted spot. This introduces up to four new edges.

When transforming the BST into the functional model, the predecessor cannot be directly moved and reused, as it might be moved above its parent, which would create a cycle in the structure when also considering expired edges, as seen in Fig. 1 (Middle). We avoid creating cycles by replacing x with a copy of its predecessor which breaks the cycle as seen in Fig. 1 (Right). However, it remains a crucial issue that b (and c) still have two parents, when also considering the expired edges, which cannot be updated efficiently in the functional model (see Sect. 1.3).

3 Freezer and Query Structure Construction

In this section we present our main contribution: the concept of a *Freezer*, which stores update information as it comes in, and how to use it to build a query DAG after all updates have occurred, using fat nodes and node copying. We give an amortized analysis argument that the total number of node copies is linear in the number of updates and from the proof we deduce that the TUNA conditions 1 are sufficient to prove Theorem 1.

3.1 The Freezer

The underlying ephemeral structure forms a tree, but when performing updates using the TUNA-compliant fat node method (see Fig. 3) to get persistence, then the data structure forms a DAG. This is problematic since in the functional model

a DAG cannot be maintained efficiently using path copying, as any node in the data structure which can reach an updated node would also have to be copied to reflect the change. Thus, we store all edges not present in the most recent version of the data structure separately in a list which we call the *freezer*. That is, only the most recent version of the data structure is explicitly maintained. This is sufficient for partial persistence since updates are only allowed in the most recent version. Finally, after all the updates have been applied, the DAG is built bottom-up. For this reason, persistence is restricted to offline.

To store the edges in the freezer, a unique *id* is assigned to each node and each edge in the freezer is stored as a 5-tuple $(id_{\text{from}}, id_{\text{to}}, field, t_{\text{start}}, t_{\text{end}})$. The ids are used to identify the nodes the edge connects, the field denotes which field of the node with id id_{from} the edge originates from (in the BST this is either **left** or **right**), and the time stamps t_{start} and t_{end} denote the *living span* of the edge as the half-open interval $[t_{\text{start}}, t_{\text{end}})$.

The freezer in addition stores which node is the root at any given time, and a map from ids to the value contained in the node with the given id. Further, note that storing the deletion time of an edge can be omitted and instead be read from the starting time of the next edge in the corresponding field of the same node, with the modification that empty fields have an edge to a special *Nil* leaf.

3.2 Offline Construction of the Fat Node Query DAG

In this section, we describe how to construct the fat node DAG structure from the expired edges in the freezer and the final non-expired structure.

Any edge that ends in the freezer must have been created in some version of the tree as the result of an update operation. An **Insert** operation creates one new edge, and a **Delete** operation creates at most four new edges. Thus, after all n updates, the freezer contains $\mathcal{O}(n)$ edges. Similarly, each **Insert** and **Delete** creates at most one new node, so the number of nodes is $\mathcal{O}(n)$.

Using Kahn's algorithm [15], which topologically sorts the nodes by repeatedly extracting nodes with outdegree zero, we build the DAG bottom-up. The while loop of the imperative algorithm is replaced by a recursive function in the functional algorithm, for each iteration calling with the new values needed. As mentioned, by copying nodes when they were moved around by the updates we avoid introducing cycles (see Figs. 1 and 3), which is required for Kahn's algorithm. By having a map from node ids to the values contained in the nodes, it is possible to explicitly build the DAG over the edges of the freezer. This creates the DAG of fat nodes.

The construction time of, a non-functional implementation of, Kahn's algorithm is linear in the number of nodes and edges. This however relies on being able to effectively fetch the ingoing and outgoing edges of nodes, and reduce the outdegree of nodes in time $\mathcal{O}(1)$. The construction relies on efficient maps from ids to nodes. As random access is not part of the functional model, maps with $\mathcal{O}(1)$ lookup time and update times do not exist. We instead use balanced trees

652 G. S. Brodal et al.

which introduces an overhead of $\mathcal{O}(\log n)$ for each of the operations, yielding a DAG construction time of $\mathcal{O}(n \log n)$. The space usage remains $\mathcal{O}(n)$.

3.3 Bounding Node Outdegree of the Query DAG

Having arbitrary outdegree of fat nodes affects the query time, as stated in Sect. 1.1. However, by limiting the number of extra pointers in each node, limited node copying similar to [24] can be implemented to remove the query overhead, while still maintaining linear space. The difference here is that in [24] the copy is performed during the update phase as soon as there are too many pointers in a node. We restrict ourselves to the offline setting and as such do not need to be able to handle queries before all updates have been applied, this means that we can allow nodes to become arbitrarily fat in the update phase.

Let d be the degree of the underlying ephemeral structure. After the update phase, we handle the high degree by recursively splitting the fat nodes into multiple nodes, each with degrees $\mathcal{O}(d)$ by interleaving the node splitting idea with Kahn's algorithm. As discussed in Sect. 3.2 we visit the nodes of the freezer bottom-up in topological order. Recall that the freezer contains edges, so nodes are inferred. When visiting a node v , that has an outdegree larger than $d + e$, for a parameter $e = \mathcal{O}(d)$ indicating the extra edges we allow every node to hold, we split it into a left node v_l to represent “the past” and a right node v_r to represent “the future”, essentially performing node copying. Note that some edges will be active in both of these nodes and thus are duplicated, similar to regular node copying. We perform the split such that the outdegree of the left node is at most $d + e$, and that the left and right nodes in total represent the original node. If the outdegree of v_r is larger than $d + e$ we recursively split it until v has been split into a number of nodes each of outdegree at most $d + e$. By requiring $1 \leq e \leq cd$, for some constant c , we ensure that the query time is proportional to that of the underlying ephemeral structure since the new nodes will contain $\Theta(d)$ edges. We call this procedure *node copying* and the following lemma shows that the space remains linear in the number of updates n .

Lemma 1. *The number of nodes introduced by node copying is $\mathcal{O}(n)$ when $1 \leq e \leq cd$ for some constant c .*

Proof. We define the potential function

$$\Phi = \sum_v \max\{0, O_v - (d + e)\},$$

where O_v is the number of outgoing edges from the node v , and the terms in the sum indicate the number of outgoing edges above the threshold $d + e$. Observe that the potential is nonnegative and that $\Phi = \mathcal{O}(n)$ when initializing *node copying* due to property U.

We now analyze how the potential changes when we split a node v , which must have $O_v > d + e$, into a left node v_l and a right node v_r . We consider the edges (consisting of a start and end timestamp) in nondecreasing order by

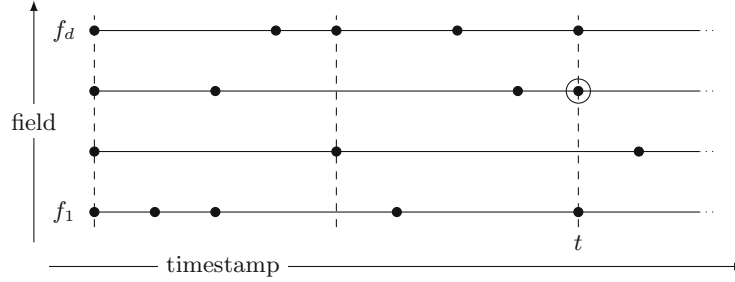


Fig. 2. Illustration of the fields of a node over time. The number of fields is $d = 4$ and $e = 4$. Each horizontal line represents one of the fields, with the dots denoting times when the value of the field change. In other words, the segments between dots represent time intervals where the field is unaltered. The vertical dashed lines are the times when the node is split. The dot with the circle around it is the start of the $(d + e + 1)$ th edge (when counting only edges after the previous split) introducing the split at time t . This results in $s_1 = 1$, $s_2 = 2$ and $a = 1$.

their start timestamp. The split is performed at the $(d + e + 1)$ th smallest start timestamp t . We include all edges with start timestamps strictly less than t in v_l . In v_r we include all edges with an end timestamp strictly larger than t .

Expanding on this, at time t we let s_1 denote the number of edges with start timestamp t among the first $d + e$ edges, and let s_2 be the number of edges with start timestamp t not among the first $d + e$ edges. Together $s_1 + s_2$ is the number of edges changing at time t , or equivalently the number of fields updated at time t . Furthermore, we call an edge *active* if its living span contains the splitting time, that is $t_{start} < t < t_{end}$, and let a denote the number of active edges at time t . See Fig. 2 for an example. Since all nodes have degree d exactly¹, we have $d = s_1 + s_2 + a$, where $s_2 \geq 1$ from the edge where we perform the split. The number of outgoing edges in v_l is $O_{v_l} = d + e - s_1 \leq d + e$. Likewise, the number of outgoing edges in v_r is $O_{v_r} = O_v - O_{v_l} + a = O_v - d - e + s_1 + d - s_1 - s_2 \leq O_v - (e + 1)$, where the inequality follows from $s_2 \geq 1$.

Finally, at time t there is at most one incoming edge to v from a parent v_p , since the underlying structure forms a tree. This edge must potentially be split in two which increases the outdegree of the parent by one. The difference in potential before and after this split is then at most:

$$\begin{aligned} \Delta\Phi &= \Delta\Phi_{v_p} + \Phi_{v_l} + \Phi_{v_r} - \Phi_v \\ &\leq 1 + 0 + \max\{0, O_v - (e + 1) - (d + e)\} - (O_v - (d + e)). \end{aligned}$$

Now there are two cases depending on which term is larger in the maximum. We first look at the case when $0 \leq O_v - (e + 1) - (d + e)$. We call this case A and here we get $\Delta\Phi \leq -e$. The other case we call case B and here we get

¹ Each node has exactly d fields and each field always holds an edge to either another node or to Nil.

654 G. S. Brodal et al.

$\Delta\Phi \leq 0$ since $\Phi_v \geq 1$. Furthermore, since $O_v - (e + 1) - (d + e) < 0$ implies $O_{v_r} \leq O_v - (e + 1) < e + d$ then no further splits of v_r are needed. We can now upper bound the number of splits by the number of times these two cases occur. Since the potential never increases, case A can occur at most $\mathcal{O}(n/e)$ times. Next, since we split nodes bottom-up in the topological order, we never add edges to a node after it has been split. Thus, as case B only happens when we perform the last split of a node, it can occur at most $\mathcal{O}(n)$ times, once for each node in the freezer. Combining the two cases we see that the number of splits is $\mathcal{O}(n)$. $\square$

4 Proof of Theorem 1

Let D be a data structure that satisfies the TUNA conditions (Definition 1). We identify each node by a unique id and store timestamps with each edge denoting their living span. The freezer, see Sect. 3.1, stores edges on the form $(id_{\text{from}}, id_{\text{to}}, field, t_{\text{start}}, t_{\text{end}})$, where $field$ denotes the outgoing field of the edge from the node with id id_{from} , and the edge was live in the version interval $[t_{\text{start}}, t_{\text{end}})$. The freezer further records what id the root of the data structure has for each time step, as well as what static value is associated with each id.

Applying an update to the structure can be done without saving the old structure, as condition A ensures that all nodes still present contain the same value. Furthermore, condition T ensures that the outgoing number of edges is d , leading to a constant number of updates to the freezer for each node updated, resulting in updates having unaltered asymptotic running time.

After applying all updates, recording the relevant information in the freezer, and obtaining tree T , condition U ensures that the freezer contains $\mathcal{O}(n)$ edges. For ease of argument, we then enter all edges from T into the freezer. Note that the number of elements in the freezer remains $\mathcal{O}(n)$.

To build the query DAG, apply a modified version of Kahn's algorithm [15] as described in Sect. 3.2, to perform a bottom-up topological sort of the graph induced by the edges in the freezer in time $\mathcal{O}(n \log n)$. Kahn's algorithm requires that the graph is acyclic, which condition N ensures. First, give each node, defined by id, the value stored in the freezer and the edges with matching id_{from} . Second, we employ the fat node technique [12] by allowing nodes to have $d + e$ edges. A fat node can overflow, if it gets more than $d + e$ edges. When an overflow is encountered the node is split into two nodes with the same value and the same parent but only a subset of the edges. Lemma 1 guarantees that splits only cause limited cascading while keeping nodes of degree $\mathcal{O}(d)$ and thus within a constant factor of their degree in the ephemeral structure as promised by condition T.

We now have a list of roots, sorted by time, that can be used to access previous versions of D . For queries, we assume the search starts from the relevant root. Otherwise, the relevant root can be found in $\mathcal{O}(\log r)$ time, where r is the total number of roots. Timestamps on edges represent the versions in which the edge was present, so it is easy to adapt queries to the DAG to only take into account the relevant edges among the $\Theta(d)$ present edges in the node, leading to the same asymptotic running time for queries. $\square$

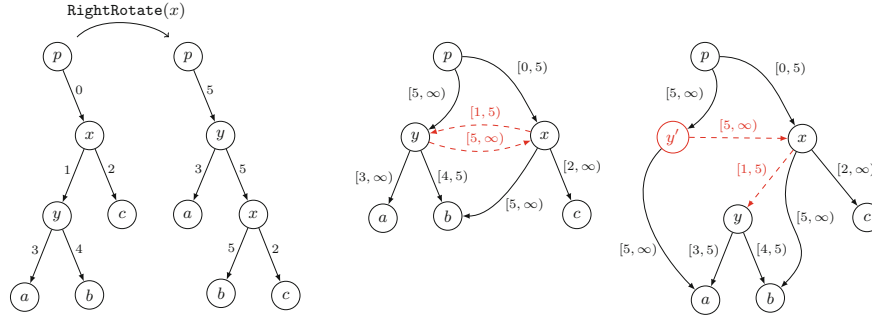


Fig. 3. Left: a rotation in a BST. Middle: the resulting graph contains the dashed cycle between x and y . Right: by creating a new copy y' of y the resulting graph remains a DAG and is TUNA compliant. We use the same notation as in Fig. 1.

5 Further Applications

Okasaki [18] introduces functional *random access arrays*, achieving, for an array of size n , worst-case lookup and update time $\mathcal{O}(\min\{i, \log n\})$, where i is the index of the queried element. This data structure easily satisfies the TUNA conditions, and can therefore be made offline partially persistent with only a constant factor space overhead.

Condition U of Definition 1 ensures that each of the n updates makes $\mathcal{O}(1)$ edges, which then totals to $\mathcal{O}(n)$ edges in the final structure. In the analysis of the space complexity, it is however not important exactly how many edges each update adds, and in fact, the more general property holds that the space usage is linear in the number of edges created by the ephemeral structure. Condition U can therefore be relaxed to each update producing for example amortized or expected $\mathcal{O}(1)$ new edges. This allows for balanced *Red-Black trees* [5] to fulfill the TUNA conditions even if the colors should be stored for persistent queries, as they make amortized $\mathcal{O}(1)$ color changes for each update [27]. The simpler functional implementation of Red-Black trees by Okasaki [20] makes amortized $\mathcal{O}(1)$ changes per insertion by a similar argument. See Fig. 3 on how rotations can be handled. Similarly, *Treaps* [3] fulfill the relaxed TUNA conditions, as each update makes expected $\mathcal{O}(1)$ rotations and therefore $\mathcal{O}(1)$ new edges.

Condition A ensures that all nodes can be reused and that storing edges is sufficient to produce the query DAG. The underlying tree is always represented explicitly, and as updates always operate on the current structure, it is possible to relax condition A, to allow for dynamic update information in each node. This information can be altered during the update phase and is not needed to produce the query DAG. Recall that our structure imposes additive $\mathcal{O}(n \log n)$ construction time independent of original update complexity. This allows for *AVL-trees* [1] to fulfill the TUNA conditions, as the balance value (the height of the subtree rooted at the node) is only used for balancing during the updates, and updates only perform amortized $\mathcal{O}(1)$ rotations [2].

656 G. S. Brodal et al.

6 Implementation and Experiments

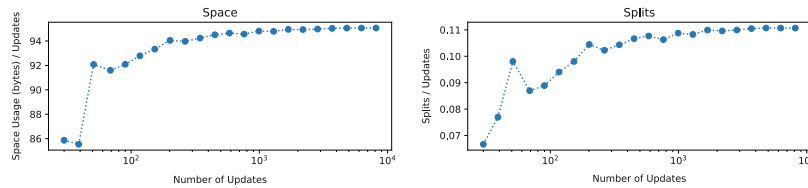


Fig. 4. Space usage experiments. Elements 1 to n are inserted into a persistent BST in order to create a path. Element $n + 1$ was then inserted and deleted n times, to introduce cascading node splits up the path, for a total of $3n$ updates.

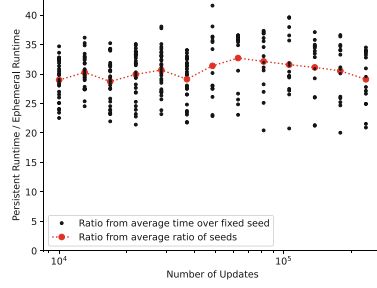
The construction described in this chapter has been implemented and tested in Haskell². Experiments were performed on WSL Ubuntu 20.04.6 on Windows 10.0.19044, with Processor 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80 GHz, 4 Core(s), 8 Logical Processor(s), 16 GB RAM, running `ghc` haskell compiler version 8.6.5, without any compiler flags. Each runtime measurement is the average of a constant number of runs on the same input. A random input is generated by providing the algorithm with a seed to a pseudo-random number generator.

For the following, let the version of a data structure made in our offline-partially-persistent framework be denoted as *persistent* and the version made in regular Haskell without necessarily saving the root of the structure be denoted *ephemeral*. The implementation uses time intervals for the edges in the freezer, and not only a start timestamp. Edges in the current structure are therefore not recorded in the freezer before they are removed from the live structure. Therefore, if any larger part of the structure is to be removed, it must be done so recursively, to correctly enter all of the edges into the freezer. Note that this does not alter the amortized running time of updates.

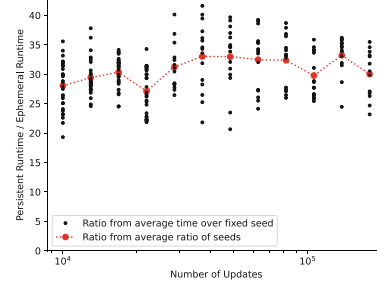
To test the *correctness* of the implementation, we apply various deterministic and random sequences of updates to both an ephemeral and a persistent unbalanced binary search tree, saving the roots of the ephemeral versions. We then construct the query DAG as described in Sect. 3.2 based on the persistent version. Then, for each timestamp, we tested if both versions produced the same tree. The test did not reveal any errors.

To measure space usage of a data structure, Haskell provides the function `recursiveSizeNF` which recursively measures the size of the object in bytes, i.e., traverses the whole pointer structure on the heap. Note that parts of the structure reachable from multiple places only are counted once. The space of the persistent data structure is measured after building the query DAG. Note that due to [25], a sequence of uniformly random insertions in an unbalanced binary search tree produces expected depth $\Theta(\log n)$.

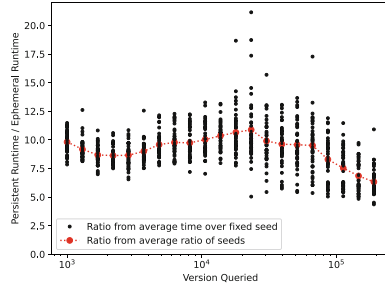
² Available at <https://github.com/Crowton/Persistent-Functional-Trees>.



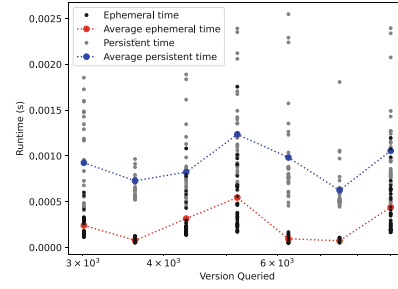
(a) Inserting elements 1 to n in random order.



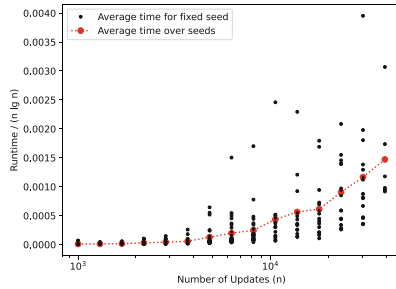
(b) Inserting and deleting elements 1 to n in random order.



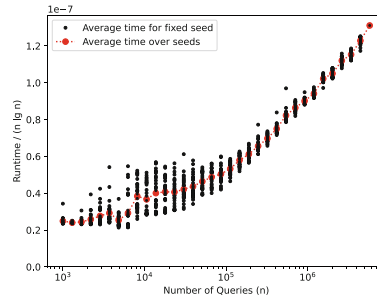
(c) Relative running time over querying all elements of a persistent BST in order at different timestamps. The tree is created by inserting elements 1 to 200000 in random order.



(d) Elements 1 to n are inserted to make a path. Element $n + 1$ is inserted and deleted n times, for $n = 3000$. Time measured for querying element $n + 1$ at different timestamps.



(e) DAG building running time experiment. Elements 1 to n were inserted and deleted from a persistent BST in random order. Time was measured on the DAG building alone.



(f) Sanity experiment. Querying all elements of an ephemeral perfectly balanced BST of size n in random order.

Fig. 5. Running time experiments.

658 G. S. Brodal et al.

The experiments for space usage, runtime of updates, and queries follow the expected result from the theory, see Figs. 4 and 5a–d. The experiment for the DAG building runtime appeared to be more than the theoretical $\mathcal{O}(n \log n)$ from the plot, see Fig. 5e. We are unable to find an explanation for the overhead. To ascertain the source of the overhead, we ran a sanity experiment to test if this overhead occurred on simpler problems. We inserted 1 to n into an unbalanced ephemeral BST where the insertion order created an almost perfectly balanced tree, via careful insertion of elements. Then all elements were queried in random order. This simple experiment has the clean theoretical runtime of $\mathcal{O}(n \log n)$ but similarly turned out not to produce an $\mathcal{O}(n \log n)$ plot in practice, see Fig. 5f, leading to the conjecture that the extra overhead in runtime is not from the program itself, but the Haskell compiler, specific implementations of underlying structures, and/or the environment the code is executed in.

The runtime experiments of updates and queries showed no issues with extra logarithmic factors, as only the relative runtime of the ephemeral and persistent implementation is compared. Here we found a constant factor difference, and thus the experiments did not disprove Theorem 1.

References

1. Adel'son-Vel'skii, G.M., Landis, E.M.: An algorithm for the organization of information. *Soviet Math. Doklady* **3**, 1259–1263 (1962)
2. Amani, M., Lai, K.A., Tarjan, R.E.: Amortized rotation cost in AVL trees. *Inf. Process. Lett.* **116**(5), 327–330 (2016). <https://doi.org/10.1016/j.ipl.2015.12.009>
3. Aragon, C.R., Seidel, R.: Randomized search trees. In: 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October–1 November 1989, pp. 540–545. IEEE Computer Society (1989). <https://doi.org/10.1109/SFCS.1989.63531>
4. Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM* **21**(8), 613–641 (1978). <https://doi.org/10.1145/359576.359579>
5. Bayer, R.: Symmetric binary B-trees: data structure and maintenance algorithms. *Acta Informatica* **1**, 290–306 (1972). <https://doi.org/10.1007/BF00289509>
6. Brodal, G.S., Okasaki, C.: Optimal purely functional priority queues. *J. Funct. Program.* **6**(6), 839–857 (1996). <https://doi.org/10.1017/S095679680000201X>
7. Buchsbaum, A.L., Tarjan, R.E.: Confluently persistent dequeues via data-structural bootstrapping. *J. Algorithms* **18**(3), 513–547 (1995). <https://doi.org/10.1006/jagm.1995.1020>
8. Cole, R.: Searching and storing similar lists. *J. Algorithms* **7**(2), 202–220 (1986). [https://doi.org/10.1016/0196-6774\(86\)90004-0](https://doi.org/10.1016/0196-6774(86)90004-0)
9. Demaine, E.D., Langerman, S., Price, E.: Confluently persistent tries for efficient version control. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 160–172. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69903-3_16
10. Dobkin, D., Lipton, R.J.: Multidimensional searching problems. *SIAM J. Comput.* **5**(2), 181–186 (1976). <https://doi.org/10.1137/0205015>
11. Dobkin, D.P., Munro, J.I.: Efficient uses of the past. In: Proceedings of the 21st Annual Symposium on Foundations of Computer Science, SFCS 1980, pp. 200–206. IEEE Computer Society, USA (1980). <https://doi.org/10.1109/SFCS.1980.18>

12. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. In: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing, STOC 1986, pp. 109–121. Association for Computing Machinery, New York (1986). <https://doi.org/10.1145/12130.12142>
13. Edelsbrunner, H., Guibas, L.J., Stolfi, J.: Optimal point location in a monotone subdivision. *SIAM J. Comput.* **15**(2), 317–340 (1986). <https://doi.org/10.1137/0215023>
14. Hughes, J.: Why functional programming matters. In: Turner, D.A. (ed.) *Research Topics in Functional Programming*. The UT Year of Programming Series, pp. 17–42. Addison-Wesley (1990). <https://doi.org/10.1093/comjnl/32.2.98>
15. Kahn, A.B.: Topological sorting of large networks. *Commun. ACM* **5**(11), 558–562 (1962). <https://doi.org/10.1145/368996.369025>
16. Kaplan, H., Tarjan, R.E.: Persistent lists with catenation via recursive slow-down. In: Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, pp. 93–102 (1995). <https://doi.org/10.1145/225058.225090>
17. Kirkpatrick, D.: Optimal search in planar subdivisions. *SIAM J. Comput.* **12**(1), 28–35 (1983). <https://doi.org/10.1137/0212002>
18. Okasaki, C.: Purely functional random-access lists. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA 1995, pp. 86–95. Association for Computing Machinery, New York (1995). <https://doi.org/10.1145/224164.224187>
19. Okasaki, C.: *Purely Functional Data Structures*. Cambridge University Press, Cambridge (1999)
20. Okasaki, C.: Red-black trees in a functional setting. *J. Funct. Program.* **9**(4), 471–477 (1999). <https://doi.org/10.1017/s0956796899003494>
21. Overmars, M.H.: Searching in the past II: general transforms. Technical report. RUU (1981)
22. Overmars, M.H.: *The Design of Dynamic Data Structures*. LNCS, vol. 156. Springer, Heidelberg (1983). <https://doi.org/10.1007/BFb0014927>
23. Pippenger, N.: Pure versus impure lisp. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1996, pp. 104–109. Association for Computing Machinery, New York (1996). <https://doi.org/10.1145/237721.237741>
24. Sarnak, N., Tarjan, R.E.: Planar point location using persistent search trees. *Commun. ACM* **29**(7), 669–679 (1986). <https://doi.org/10.1145/6138.6151>
25. Sedgewick, R., Flajolet, P.: *An Introduction to the Analysis of Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston (1996)
26. Seidel, R.: A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom.* **1**(1), 51–64 (1991). [https://doi.org/10.1016/0925-7721\(91\)90012-4](https://doi.org/10.1016/0925-7721(91)90012-4)
27. Tarjan, R.E.: Amortized computational complexity. *SIAM J. Algebraic Discret. Methods* **6**(2), 306–318 (1985). <https://doi.org/10.1137/0606031>

External Memory Fully Persistent Search Trees*

Gerth Stølting Brodal

Aarhus University
Department of Computer Science
Denmark
gerth@cs.au.dk

Casper Moldrup Rysgaard

Aarhus University
Department of Computer Science
Denmark
rysgaard@cs.au.dk

Rolf Svenning

Aarhus University
Department of Computer Science
Denmark
rolfsvenning@cs.au.dk

ABSTRACT

We present the first fully-persistent external-memory search tree achieving amortized I/O bounds matching those of the classic (ephemeral) B-tree by Bayer and McCreight. The insertion and deletion of a value in any version requires amortized $O(\log_B N_v)$ I/Os and a range reporting query in any version requires worst-case $O(\log_B N_v + K/B)$ I/Os, where K is the number of values reported, N_v is the number of values in the version v of the tree queried or updated, and B is the external-memory block size. The data structure requires space linear in the total number of updates. Compared to the previous best bounds for fully persistent B-trees [Brodal, Sioutas, Tsakalidis, and Tschlas, SODA 2012], this paper eliminates from the update bound an additive term of $O(\log_2 B)$ I/Os. This result matches the previous best bounds for the restricted case of partial persistent B-trees [Arge, Danner and Teh, JEA 2003]. Central to our approach is to consider the problem as a dynamic set of two-dimensional rectangles that can be merged and split.

CCS CONCEPTS

• Theory of computation → Data structures design and analysis.

KEYWORDS

B-trees, range queries, external memory, full persistence, semi-dynamic ray shooting.

ACM Reference Format:

Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. 2023. External Memory Fully Persistent Search Trees. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC '23)*, June 20–23, 2023, Orlando, FL, USA. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3564246.3585140>

1 INTRODUCTION

The B-tree of Bayer and McCreight [6] is the classic external-memory data structure for storing a dynamic set S of N totally ordered values. It supports the insertion and deletion of values in $O(\log_B N)$ I/Os and range reporting queries in $O(\log_B N + K/B)$

I/Os, i.e., reporting all values contained in a value range $[x, y]$, where K is the number of values reported and B is the external-memory block size.

In this paper, we consider the problem of storing a dynamic set in external memory *fully persistently*, i.e., all versions of the set are emembered, any previous version can be queried, and any previous version can be updated resulting in a new version of the set. We present the first fully persistent search trees matching the asymptotic I/O bounds of the classic *ephemeral* (i.e., non-persistent) B-trees in the amortized sense.

Throughout this paper, we will assume the I/O model of Aggarwal and Vitter [2] consisting of an internal memory being able to hold M values, and an infinite external memory. One I/O can transfer B consecutive values between internal and external memory, and the I/O complexity of an algorithm is the number of I/Os performed. Computation can only be performed in internal memory, and the only allowed operations on values are comparisons.

1.1 Interface of a Fully Persistent Search Tree

The interface to a fully persistent search tree consists of the below operations. Each version is identified by a unique version identifier v (a positive integer). We let S_v denote the set of values at version v . The versions form a *version tree* T , where the root (version 1) stores the initial set, and a new version w becomes a child of an existing version v in T , if S_w is derived as a *clone* of S_v . Updates (insertions and deletions) can only be applied at leaves of the version tree T , i.e., versions that have not been cloned yet. These versions are said to be *unlocked*. A version that has been cloned cannot be updated further and is said to be *locked*. All versions can be queried and cloned. If one needs to update a locked version (an internal node of the version tree), one has to clone the version and apply the update to the new unlocked version (leaf of the version tree).

CLONE(v) Creates a new version w , where $S_w = S_v$. Returns the new version identifier w . Version w becomes a child of v in the version tree T , i.e., after the operation version v is locked and version w is unlocked.

INSERT(v, x) Adds x to S_v . Requires version v is unlocked.

DELETE(v, x) Removes x from S_v . Requires version v is unlocked. If x is not contained in S_v , nothing is changed.

RANGE(v, x, y) Returns all values in $S_v \cap [x, y]$ in increasing order.

SEARCH(v, x) Returns the predecessor of x in S_v .

We let N_v denote the number of values in S_v , and N the total number of updates done to all versions. Figure 1 (top) illustrates a version tree. In the following examples, we illustrate values as integers, but our construction works for any comparison based values.

*Work supported by Independent Research Fund Denmark, grant 9131-00113B.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

STOC '23, June 20–23, 2023, Orlando, FL, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9913-5/23/06...\$15.00

<https://doi.org/10.1145/3564246.3585140>

1.2 Previous Work

For the non-persistent setting in internal memory, a set can be stored using a standard balanced binary search tree, e.g., AVL-trees [1], red-black trees [20], and (a, b) -trees [21]. These all support insertions and deletions in $O(\log_2 N)$ time, and range reporting queries in $O(\log_2 N + K)$ time. In external memory, the B-tree [6] is the classic data structure of choice and supports updates in $O(\log_B N)$ I/Os and range reporting queries in $O(\log_B N + K/B)$ I/Os. A number of papers have since explored the update-query trade-off starting with [10] giving a structure known as the B^ϵ -tree for any $0 < \epsilon < 1$ with amortized $O\left(\frac{1}{\epsilon B^{1-\epsilon}} \log_B N\right)$ I/Os for updates and $O\left(\frac{1}{\epsilon} \log_B N\right)$ I/Os for queries. The B^ϵ -tree operates similarly to a B-tree but with fanout B^ϵ and updates are flushed down the tree using buffers, leading to the amortized bound. In [8, 9] the amortized bound was improved to high probability and finally in [15] to worst-case.

The notion of (data structural) full persistence for internal memory data structures was coined by Driscoll, Sarnak, Sleator and Tarjan in [18], where they also described how to achieve a fully persistent version of red-black trees. Previous to this Sarnak and Tarjan [25] had presented partially persistent red-black trees, where partial refers to the limitation that only the most recent version of the red-black tree can be updated, i.e., all versions of the red-black tree form a version tree consisting of a single path with the most recent version of the red-black tree being the single leaf. Demaine, Iacono and Langerman [16] considered the notion of *retroactive* data structures, where updates can be applied to any version in the version tree, and updates are automatically recursively applied to all derived version.

Adopting the notions of partial and full persistence to external-memory search trees has been done in a sequence of papers. See Table 1. The results of Arge, Danner and Teh 2003 [4] and Brodal, Sioutas, Tsakalidis and Tsihlias [11, 12] achieve the best bounds for partial and full persistence, respectively. Both results achieve optimal bounds for range reporting, i.e., $O(\log_B N_v + K/B)$ I/Os (the N in the bounds of [4] for partial persistence can be reduced to N_v by applying global rebuilding after a linear number of updates). The update I/O bound of [4] matches the $O(\log_B N)$ I/Os for B-trees, whereas the full persistence result of [11] has an additive $O(\log_2 B)$ I/Os overhead per update. This paper eliminates this additive term and achieves update bounds matching those of (ephemeral) B-trees and [4] for partial persistence.

1.3 Contribution

The main contribution of this paper are fully-persistent external-memory search trees, achieving the following bounds, matching those of (ephemeral) external-memory B-trees.

THEOREM 1.1. *There exist external-memory fully-persistent search trees supporting INSERT and DELETE in amortized $O(\log_B N_v)$ I/Os, CLONE in worst-case $O(1)$ I/Os, SEARCH in worst-case $O(\log_B N_v)$ I/Os, and RANGE in worst-case $O(\log_B N_v + K/B)$ I/Os. The space usage is linear in the number of updates.*

It should be noted that for the case where $N_v = B^{O(1)}$, the update bounds are $O(\log_B N_v) = O(1)$ I/Os, whereas the best previous

update bound was $O(\log_2 B) = O(\log_2 N_v)$ I/Os [11]. Our range reporting queries can be extended to support an argument k , and to only report in sorted order the first k (or last k) values in a value range $[x, y]$ using $O(\log_B N_v + k/B)$ I/Os, by simply truncating the reporting when k values have been found.

The basic ideas of the construction are the following. As in [18], we use a linearization of the version tree into a version list obtained by a pre-order traversal of the version tree. Values are associated with half-open liveness intervals of the version list. See Figure 1 (bottom). To adapt this to external memory setting, we consider these (value, liveness interval) pairs as vertical segments in a two-dimensional plane, and partition the plane into rectangles, possible splitting segments into multiple smaller disjoint segments. See Figure 2. During insertions and deletions the partition into rectangles is dynamically updated by splitting rectangles in either the version dimension (corresponding to the node splitting idea in [18]) or value dimension (corresponding to splitting a leaf in a B-tree), or by joining two horizontally adjacent rectangles (corresponding to joining two adjacent leaves in a B-tree). See Figure 3. A range reporting query $\text{RANGE}(v, x, y)$ simply identifies all rectangles intersected by the horizontal query segment $[x, y] \times \{v\}$, and reports the values of all vertical segments in these rectangles intersecting the horizontal query segment. Invariants ensure that for all rectangles intersected, except for two, a constant fraction of the values are part of the answer to the query. This is inspired by the filtering search of Chazelle [13].

Essential to our result is the application of a two-dimensional orthogonal point location (vertical ray shooting) structure to identify the rectangle containing a (value, version) point. Point location queries must be supported in worst-case $O(\log_B N)$ I/Os, whereas segment insertions are allowed to be significantly slower, as high as amortized $O(B \log_B^2 N)$ I/Os. The best external-memory bounds for dynamic orthogonal point location are by Munro and Nekrich [24], who support updates and queries with $O(\log_B N \log \log_B N)$ I/Os, and by Arge, Brodal and Rao [3], who support queries with $O(\log_B^2 N)$ I/Os and updates with $O(\log_B N)$ I/Os (in internal memory $O(\log n)$ time updates and queries were achieved by Giora and Kaplan [19]). To achieve queries with $O(\log_B N)$ I/Os, we describe a specialized insertion-only external-memory point location structure, that also crucially avoids the comparison of version identifiers. The internal-memory [18] and external-memory [11] persistent search trees make essential use of the dynamic data structure by Dietz and Sleator [17] to answer order queries among two version identifiers in the version list in $O(1)$ time and I/Os. This is in fact the bottleneck causing the additive $O(\log_2 B)$ I/Os on updates in [11]. We avoid the use of [17] by using an external-memory colored predecessor data structure, that essentially stores multiple predecessor data structures in a single B-tree.

1.4 Outline of Paper

In Section 2 we present our geometric interpretation of the fully persistence search tree problem and describe our data structure for the static case. In Sections 3–5 we present a simplified dynamic result, Theorem 1.2 below, where we assume an upper bound $\bar{N}$ of the total number of updates to be performed and use $\bar{N}$ as a parameter in our construction, and CLONE uses $O(\log_B \bar{N})$ I/Os. In

Table 1: I/O bounds for previous and new results for partial and full persistent search trees in external memory. All structures use linear space.

	Reporting	Update
Partial persistence		
Lomet and Salzberg [23]	$O(N_v/B)$	$O(\log_B N_v \log_B N)$
Becker, Gschwind, Ohler, Seeger, Widmayer [7]	$O(\log_B N + K/B)$	$O(\log_B^2 N_v)$ am.
Varman and Verma [26]		
Arge, Danner, Teh [4]	$O(\log_B N + K/B)$	$O(\log_B N)$ am.
Full persistence		
Lanka and Mays [22]	$O((\log_B N_v + K/B) \log_B N)$	$O(\log_B^2 N_v)$ am.
Brodal, Sioutas, Tsakalidis, Tsihlias [11]	$O(\log_B N_v + K/B)$	$O(\log_B N_v + \log_2 B)$ am.
<i>This paper</i>	$O(\log_B N_v + K/B)$	$O(\log_B N_v)$ am.

Table 2: I/O bounds of the operations for each improvement of the structure. All structures use linear space.

	CLONE	INSERT and DELETE	SEARCH	RANGE
Geometric Structure (Sections 2, 3, 4, 5)	$O(\log_B \bar{N})$	$O(\log_B \bar{N})$	$O(\log_B \bar{N})$	$O(\log_B \bar{N} + K/B)$
Partitioning the Version Tree (Section 6)	$O(\log_B N_v)$	$O(\log_B N_v)$	$O(\log_B N_v)$	$O(\log_B N_v + K/B)$
Lazy Clones (Section 7)	$O(1)$	$O(\log_B N_v)$	$O(\log_B N_v)$	$O(\log_B N_v + K/B)$

Section 3 we describe the dynamic version of our data structure and in Sections 4 and 5 we describe our colored predecessor and point location structures, respectively. In Sections 6–7 we then show how to improve these bounds to those of Theorem 1.1. In Section 6 we show how to make the I/O bounds depend on the size N_v of a version v , instead of the upper bound parameter $\bar{N}$, by splitting the version tree into multiple version trees, where all versions in a version tree have approximately equal sizes. In Section 7 we show how to reduce the cost of CLONE from $O(\log_B N_v)$ I/Os to $O(1)$ I/Os by performing *lazy clones*, i.e., the actual cloning is postponed until the first update is performed on a version. An overview of the I/O bounds for each of these improvements is presented in Table 2.

THEOREM 1.2. *Let $\bar{N}$ be a parameter giving an upper bound on the total number of updates. Then there exist external-memory fully-persistent search trees that support INSERT, DELETE and CLONE in amortized $O(\log_B \bar{N})$ I/Os, SEARCH in worst-case $O(\log_B \bar{N})$ I/Os, and RANGE in worst-case $O(\log_B \bar{N} + K/B)$ I/Os. The space usage is linear in the number of updates.*

2 STATIC DATA STRUCTURE

Our implementation of a persistent search tree takes a geometric approach. In Section 2.1 we introduce the version tree and version list, concepts that we borrow from [18], and in Section 2.2 we give a geometric interpretation of range reporting queries. In Section 2.3 we introduce our rectangular two-dimensional space partition in the static setting, and in Section 2.4 we summarize the main properties of the rectangular partition that should be maintained by the dynamic structure in Section 3. In Section 2.5 we describe how to support queries at the high level.

2.1 Version Tree and Version List

Following [18], we define the version tree T as follows. The root is version 1. Whenever a new version w is created, it is assigned

the smallest unused positive integer value so far. If w is created by cloning v , then w becomes the leftmost child of v . This approach guarantees if T contains s versions, then these are versions $\{1, 2, \dots, s\}$, that version identifiers are unique, the version identifiers of the children of a node are increasing from right to left, and the version identifiers along a root-to-leaf path are increasing.

Similarly to [18], we derive a *version list* L from T by a preorder left-to-right traversal of T . As version 1 is the root of the version tree T , it is also the first version in L . We assume that L is terminated by version 0. Given two versions v and w , we in the following let $[v, w[$ denote the versions in the version list from v up to w , but excluding w . Note that in the version list $[1, 0[$ includes all nodes of the version list, except for the terminating version 0. With each value x inserted into version v we store the *liveness interval* $[v, \text{succ}(v)[$, where $\text{succ}(v)$ is the successor of v in the version list. If version w is created by cloning version v , then w becomes the leftmost child of v in the version tree, i.e., w should be inserted immediately after v in the version list. Crucial to the definition of the liveness interval, which is also used in [18], is that the later versions w created below v in the version tree are exactly the versions that will be contained in the liveness interval of x , if x was inserted in version v .

In Figure 1 (bottom) the liveness intervals are shown for all values inserted in the example in Figure 1 (top). E.g., the value 2 is inserted into version 1 with liveness interval $[1, 0[$, since only version 1 exists when 2 is inserted. The value 5 is inserted into versions 2 and 8, with liveness intervals $[2, 0[$ and $[8, 6[$, respectively (versions 3, 4 and 5 did not yet exist when 5 was inserted into version 2). Finally, value 7 is inserted in version 1 with liveness interval $[1, 0[$ but again deleted in versions 3 and 7 for liveness intervals $[3, 0[$ and $[7, 6[$, respectively. The result is that value 7 only exists for the remaining version intervals $[1, 7[$ and $[6, 3[$, i.e., versions 1, 2 and 6 in the current version tree.

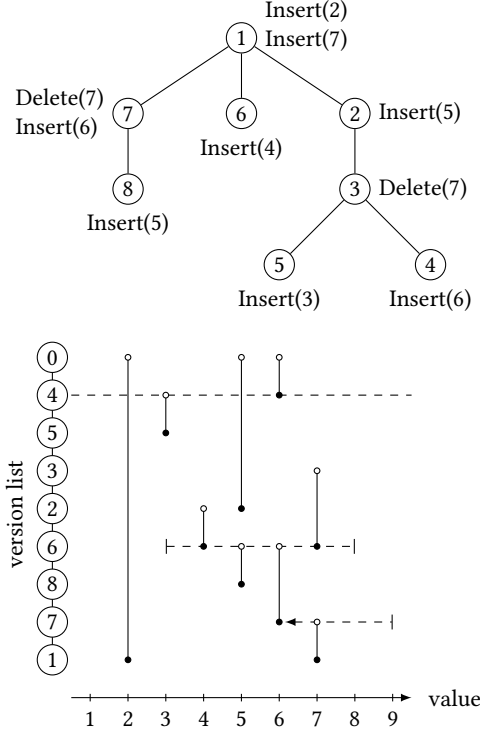


Figure 1: (Top) Version tree illustrating 8 versions of a set after $N = 10$ updates. Versions 1, 2, 3, and 4 contain the sets $\{2, 7\}$, $\{2, 5, 7\}$, $\{2, 5\}$, $\{2, 5, 6\}$, respectively. Updates to a version are shown next to the node of the version tree. (Bottom) The version list, consisting of a left-to-right preorder traversal of the version tree terminated by 0, and the half-open liveness intervals of versions containing a value. Note that the value 7 only is contained in versions 1, 6 and 2 of the set. The topmost dashed line shows that version 4 of the set contains $\{2, 5, 6\}$, the dashed line segment at version 6 shows that the range reporting query $\text{RANGE}(6, 3, 8)$ has result $\{4, 7\}$, and the bottommost dashed arrow that the query $\text{SEARCH}(7, 9)$ has result 6.

2.2 Geometry of Updates and Queries

Our problem has a natural geometric interpretation in a two-dimensional space, see Figure 1 (bottom). Consider the plane where x -values correspond to values, and the y -value corresponds to the order in the version list. $\text{INSERT}(v, x)$ corresponds to inserting the line segment $\{x\} \times [v, \text{succ}(v)[$, whereas $\text{DELETE}(v, x)$ to deleting the line segment $\{x\} \times [v, \text{succ}(v)[$, possibly splitting a longer line segment into two disjoint segments. Crucial for the implementation of deletions is that the length of the interval to be deleted (at the time of deletion) is only between two adjacent versions in the version list. The values in $\mathcal{S}_v$ are exactly those vertical line segments intersecting the horizontal line at version v , and the answer to the range reporting query $\text{RANGE}(v, x, y)$ are exactly the vertical segments intersected by $[x, y] \times \{v\}$, and the answer to $\text{SEARCH}(v, x)$ is the rightmost vertical segment intersected by $[-\infty, x] \times \{v\}$.

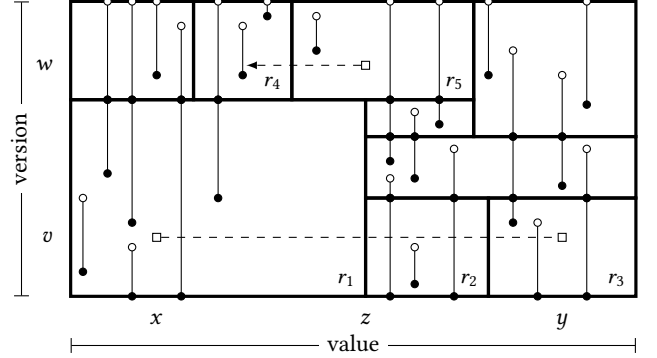


Figure 2: The partitioning of the plane into rectangles, a query $\text{RANGE}(v, x, y)$ represented by the dashed line between two square endpoints spanning rectangles r_1 , r_2 , and r_3 , and a query $\text{SEARCH}(w, z)$ represented by the dashed arrow originating from a square endpoint, spanning rectangles r_5 and r_4 . Note that black dots on vertical segments correspond to the upper endpoint of the segment in the rectangle below and the lower endpoint of the segment in the rectangle above.

2.3 Static Rectangular Partitioning

Assume N updates have been performed on a fully persistent search tree. In the following we present a static structure storing the resulting versions using the geometric representation discussed in Section 2.2 and supports queries in $O(\log_B N)$ I/Os. In Section 3 we extend this structure to support updates.

A key cornerstone of our structure is the parameter

$$R = \Theta(B \log_B N) .$$

A crucial property that we will make repeated use of is that, given a list of R segments in external memory stored in $O(R/B)$ blocks, the list can be scanned using $O(R/B) = O(\log_B N)$ I/Os.

We partition the value $\times$ version plane into disjoint rectangles $r_1, r_2, \dots, r_k$, see Figure 2. We denote the full plane by $[-\infty, +\infty[\times [1, 0[$ (by slight misuse of notation we let $[-\infty, +\infty[$ denote the whole value range), and each rectangle r is of the form $[x^r, y^r[\times [v^r, w^r[$, where $x^r < y^r$ are values in the value range, possibly $x^r = -\infty$ and $y^r = \infty$, and v^r and w^r are versions, where w^r is strictly after v^r in the version list. The vertical segments corresponding to liveness intervals of a value (Section 2.1) are split into shorter segments such that segments never overlap with two rectangles, i.e., a segment $\{x\} \times [v, w[$ is repeatedly split into segments $\{x\} \times [v, v^r[$ and $\{x\} \times [v^r, w[$, if a rectangle $r = [x^r, y^r[\times [v^r, w^r[$ is intersected by the segment, i.e., $x^r \leq x < y^r$ and $v^r \in [v, w[$. For a rectangle r , we let S^r denote the set of vertical segments $\{x\} \times [v, w[$ contained in r .

If the rectangle partition is constructed such that each rectangle contains $O(R)$ segments, each rectangle is stored in $O(R/B)$ blocks in external memory, and we have a point location structure that can report the rectangle containing a given a query point using $O(\log_B N)$ I/Os (e.g., the structure in Section 5), then all segments in the rectangle containing a query point can be reported using $O(\log_B N + R/B) = O(\log_B N)$ I/Os.

As seen in Figure 2 a range reporting query may need to report values from multiple rectangles. Assuming that we for each rectangle can output $\Theta(R)$ values, except for the at most two rectangles containing the endpoints of the range reporting query, then each rectangle can be found using $O(\log_B N)$ I/Os by the point location structure, and the I/O cost for querying each of these rectangles can be charged to the output. The total number of I/Os for a range reporting query becomes $O(\log_B N + K/B)$ I/Os, where K is the total number of values reported by the range reporting query.

We require that each non-rightmost rectangle contains $\Theta(R)$ segments spanning all versions of the rectangle. We denote these *spanning segments*, i.e., segments $\{x\} \times [v^r, w^r]$ contained in a rectangle $r = [x^r, y^r] \times [v^r, w^r]$, where $x^r \leq x < y^r$. This ensures that all rectangles considered by a SEARCH and RANGE query for a given version contain $\Theta(R)$ values, except for possibly the rightmost rectangle. Note that for SEARCH queries, the predecessor must be reported, and by the spanning requirement, if the predecessor does not exist in the current rectangle, then it must either exist in the left neighboring rectangle for this version, or no predecessor exists. For each rectangle the segments are stored in increasing value order, so that range reporting queries can report values in increasing order.

To be able to efficiently compare versions within a rectangle $r = [x^r, y^r] \times [v^r, w^r]$, we maintain the *local version list* $L^r \subseteq L \cap [v^r, w^r]$ for r , that contains the two versions v^r and w^r and all versions that define endpoints of segments in S^r . Since $|S^r| = O(R)$, we also have $|L^r| = O(R)$. The versions in L^r are stored in the same order as in the global version list L . With each version $v \in L^r$ we store the *local version* $\pi^r(v)$ of v in L^r , that just is the position of v in L^r , i.e., an integer in the range $1..|L^r|$. The important property is that v is before w in L^r if and only if $\pi^r(v) < \pi^r(w)$, where the later is a simple comparison between two integers.

We store the segments in S^r only using the local versions. Segment $\{x\} \times [v, w]$ in S^r is stored as the triple $(x, \pi^r(v), \pi^r(w))$, and S^r is stored in increasing value order in $O(|S^r|/B)$ blocks in external memory. When a new version v is added to L^r , all versions w after v in L^r get their local version $\pi^r(w)$ increased by one, and similarly all segments in S^r need to be checked if the local version of their endpoints must be increased by one.

If the version v of a query point (x, v) is not the endpoint of any of the segments in the rectangle r containing (x, v) , i.e., $v \notin L^r$, let u be the predecessor of v and w the successor of v in the local version list. Since no version between u and w are endpoints of any segments and all segments are inclusive in the first version and exclusive in the second version, all versions between u and w must have the same values in the value range of r . Since $u \in L^r$, we can use $\pi^r(u)$ to compare against the local versions of the segment endpoints in S^r during a query. In Section 4 we describe a structure that can find the predecessor version in a local version list using $O(\log_B N)$ I/Os, which thus does not increase the I/O cost of queries.

2.4 Structural Requirements

The below list summarizes the main properties required by our rectangular partition from the previous sections. In Section 3 we explain how to maintain them dynamically.

- A rectangle stores $O(R)$ segments.

- The number of spanning segments in a rectangle is $\Theta(R)$, except for rightmost rectangles.
- The segments are stored in increasing value order.
- For each rectangle and segment in a rectangle we store the local versions.

Additionally, we need two more structures. The first structure we call a *colored predecessor* structure which relates global and local versions. The second structure is a *point location* structure which allows us to navigate the rectangular partitioning. Crucial to both structures is that they should avoid comparing the relative order of versions in the global version list, as we also do internally in the rectangles by using local version lists. That is, we need data structures for the following two problems:

- Colored predecessor problem (Section 4): Given a version v and a rectangle r find the predecessor version u of v in the local version list L^r using $O(\log_B N)$ I/Os. For this problem, we think of each rectangle as a unique color.
- Point location (Section 5): Given a query point (x, v) find the rectangle containing the query point using $O(\log_B N)$ I/Os.

2.5 Queries

To perform $\text{RANGE}(v, x, y)$ we first perform a point location query to find the rectangle r containing the point (x, v) and perform a colored predecessor query to find the predecessor u of v in L^r . We find the vertical segments in r intersected by the horizontal segment $[x, y] \times \{u\}$ by scanning through all triples (z, i, j) in S^r and report those z where $x \leq z \leq y$ and $i \leq \pi^r(u) < j$, i.e., the local version interval contains $\pi^r(u)$. If $y^r < y$ we recursively call $\text{RANGE}(v, y^r, y)$. The properties of the partitioning ensure that for each rectangle visited, we will report $\Theta(R)$ values, except possibly the leftmost and rightmost rectangles considered.

To perform $\text{SEARCH}(v, x)$, similarly to $\text{RANGE}(v, x, y)$ we first locate the rectangle containing the point (x, v) , but now we instead traverse left until the first segment intersection with $[-\infty, x] \times \{v\}$ is identified, which will be the predecessor of x in S_v (how to move to the rectangle to the left of another rectangle is described in Section 3.2). It is guaranteed that at most two rectangles are required to be scanned due to spanning segments requirement.

Assuming the I/O bounds for the colored predecessor structure in Section 4 and the point location structure in Section 5 are $O(\log_B N)$, we from the discussion in this section get that RANGE and SEARCH queries can be performed in $O(\log_B N + K/B)$ and $O(\log_B N)$ I/Os, respectively.

3 UPDATES

In Section 2.3 we described how to efficiently perform queries given a rectangular partition of the plane, assuming some structures exist to report rectangles and version predecessors, where each rectangle had to meet a list of requirements as summarized in Section 2.4. In this section, we discuss how to turn these requirements into invariants, such that the structure can be made dynamic, i.e., allow for cloning versions and performing insertions and deletions of values in the different versions. We assume, that on initialization we are given a constant $\bar{N}$, which is an upper bound on the total

number of updates to be performed on the structure, and let

$$R = B \log_B \bar{N}.$$

When performing a CLONE operation, the geometric view remains the same, apart from a horizontal part being “stretched” to make room for the new version. All information stored in the rectangles remains the same, as the new version is not contained in any segment endpoints. The new version must therefore only be inserted in the global version list L and the colored predecessor structure from Section 4. Assuming that a direct pointer to the version cloned in L is provided, the new version can be inserted into L using worst-case $O(1)$ I/Os. A new version can be inserted in the colored predecessor structure using amortized $O(\log_B \bar{N})$ I/Os. In total the CLONE operation requires amortized $O(\log_B \bar{N})$ I/Os.

The INSERT operation is performed as discussed in Section 2.2 by inserting a value x into the structure at version v is equivalent to adding the segment $\{x\} \times [v, \text{succ}(v)[$. Similarly, to perform a DELETE operation of a value x at version v , then the segment $\{x\} \times [v, \text{succ}(v)[$ is to be removed. Note that this potentially means partitioning an already present segment $\{x\} \times [u, w[$, where $u \leq v < w$, into two new segments, where the piece of the segment between versions v and $\text{succ}(v)$ is removed.

As $\text{succ}(v)$ is the version immediately after v in the version list, and all rectangles are exclusive the top version, then the segment $\{x\} \times [v, \text{succ}(v)[$ exists entirely within one rectangle. This rectangle r can be found as the rectangle containing the point (x, v) using the point location structure of Section 5 with $O(\log_B \bar{N})$ I/Os. L^r can then be updated with versions v and $\text{succ}(v)$, which are either inserted or deleted to make L^r reflect the current segment endpoints. The locations in L^r can be found using the colored predecessor structure in Section 4 with $O(\log_B \bar{N})$ I/Os. As new versions are inserted into L^r , this may require the colored predecessor structure to be updated using $O(\log_B \bar{N})$ I/Os for each newly inserted version. Next, the segments of S^r must be updated for the segment $\{x\} \times [v, \text{succ}(v)[$, as well as the local version of all segments in S^r . Note that the local version of all versions present after v in L^r is incremented or decremented by either 0, 1 or 2, depending on how many versions were inserted or deleted from L^r . The segments of S^r can thus be updated in a single scan using $O(\log_B \bar{N})$ I/Os by the requirement on the size of S^r .

Updates can thus be performed using $O(\log_B \bar{N})$ I/Os. They may however break the requirements that were placed on the rectangles to ensure that queries are efficient. In order to make sure that these requirements are met, they are listed in Section 3.1 as invariants, where the asymptotic bounds are replaced by defined constants.

3.1 Invariants

For a rectangle $r = [x^r, y^r[\times [v^r, w^r[$ we define a *spanning segment* to be a segment $\{x\} \times [v^r, w^r[\in S^r$, i.e., the segment represents a value x , which is presented in all versions r spans. The following invariants are then placed on the rectangle:

- I_1 : $|S^r| \leq 2R$, i.e., a rectangle stores at most $2R$ segments.
- I_2 : The number of spanning segments in r is at least $\frac{R}{4}$, except if $y^r = +\infty$, i.e., r is a rightmost rectangle, where there is no lower bound on the number of spanning segments.

I_3 : The segments in S^r are stored in sorted order by the values the segments represent.

I_4 : The list L^r contains precisely the two versions v^r and w^r , and the versions of all endpoints of segments in S^r .

Invariants I_3 and I_4 can easily be maintained upon an update (see Section 3.7). Invariants I_1 and I_2 may however be broken by an update. For insertions, only invariant I_1 may break, and for deletions, both invariants may break. Note that deletions can break both invariants at once.

In order to handle invariant I_1 , the issue is the rectangle now contains too many segments. This can be fixed by repeatedly splitting the rectangle into two smaller rectangles, until they all do not contain too many segments. These splits may be done either vertically or horizontally, as illustrated in Figure 3.

The invariant I_2 ensures that each rectangle contains sufficiently many spanning segments. If there are too few spanning segments, denoted as an *underflowing* rectangle, and the rectangle is not a rightmost rectangle, then there must exist some rectangles, which are the right neighbors of the underflowing rectangle. These rectangles must then either contain sufficiently many spanning segments or be rightmost rectangles themselves. By performing horizontal splits to align the rectangles, then the underflowing rectangle can be joined with the right neighbors in order to become a rightmost rectangle or to then contain sufficiently many spanning segments. See Figure 4 for an illustration of how this works. This figure only covers the basic idea of merging, and it later turns out, that more cases for merging is needed, which can be seen in Figure 5. Note that merge is performed by making multiple primitive rectangle transformations.

A merge operation takes time proportional to its number of neighbors. Therefore, we place an upper bound t_N on the number of right neighbors, for some integer constant $t_N \geq 2$. When making a merge operation, the top and bottom right neighbors are split, which if they are close to underflowing will not be an issue, as only one of the two rectangles they are split into will remain close to underflowing. If however there only is one right neighbor, and it is close to underflowing, it may be split twice, and make two rectangles, which are close to underflow. In this case, one of the resulting rectangles may be merged with all of its left neighbors, which is symmetrical to merging to the right, see Figure 5b. Note that there must then also be an upper bound t_N on the number of left neighbors.

This thus adds an extra invariant for each rectangle r , with N_r^R and N_r^L denoting the number of neighbors to the right and left of rectangle r respectively.

I_5 : $N_r^R \leq t_N$ and $N_r^L \leq t_N$, i.e., the number of neighboring rectangles to rectangle r is at most t_N on both sides.

Note that horizontal splits may increase the number of neighbors in some rectangles. However, if some rectangle has too many neighbors, then it can be split horizontally, to distribute the number of neighbors between two rectangles, and thus decrease the number of neighbors each rectangle has.

3.2 Finding the Neighbors

For the construction, we must be able to find the neighbors of a given rectangle for merging and detecting neighbor overflow. To

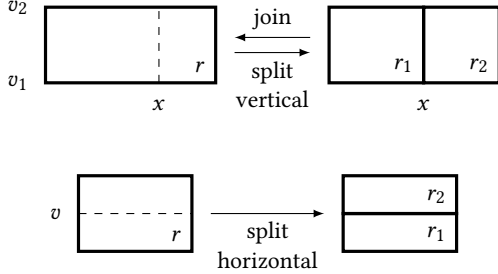


Figure 3: Primitive rectangle transformations: Vertical split and join at value x , and horizontal split at version v .

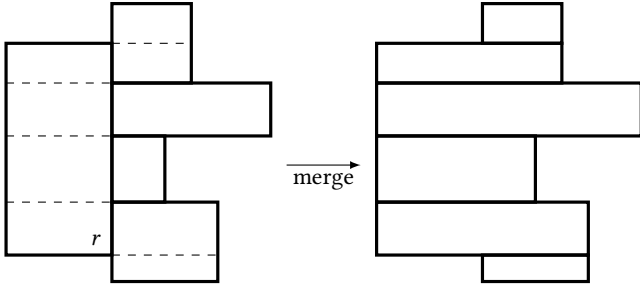


Figure 4: Merging of a rectangle r with its right neighboring rectangles. The dashed lines represent the location that the rectangles are horizontally split, before they are vertically joined.

find the neighbors to the right of a rectangle, we use that the rectangles are exclusive in the top and right coordinates. By repeatedly querying the endpoints using the point location structure from Section 5, all right rectangles can be found from the bottom up. To find the left rectangles, we need a point inside the left neighbor. Such a point can be found by maintaining a search tree over all rectangle start points and searching for the predecessor of the current left side of the rectangle. Each query uses $O(\log_B \bar{N})$ I/Os, and as the number of neighbors is bounded by the constant t_N , the total to find all neighbors is $O(\log_B \bar{N})$ I/Os.

3.3 Potential Functions

Updates to the persistent structure, when ignoring the potential cost of updating the partition of rectangles to restore the rectangle invariants, can be done with $O(\log_B \bar{N})$ I/Os. Splitting and merging rectangles, in order to maintain the invariants, then uses further I/Os, which affect the update time. We amortize the cost of this rebalancing over the updates. In this section, we state the potential function Φ driving the analysis of our structure.

Horizontal and vertical splits are triggered by the rectangle having too many segments. Vertical splits are performed if there are many spanning segments and horizontal splits if there are many non-spanning segments. There must thus be a large potential when there are many spanning and non-spanning segments. The merge is triggered by an underflow in the number of spanning segments. The potential must therefore be large when there are few spanning

segments. Finally, rectangles are split when there are too many neighbors on one side, so the potential must be large when there are many neighbors on the same side.

This leads to the following potential function Φ , which is the weighted sum of four potential functions, with S_r denoting the number of spanning segments in rectangle r , and I_r denoting the number of internal endpoints of segments in r , i.e., endpoints of segments, which are not equal to the bottom or top version of r . The constants c_O , c_U , c_I , and c_N are determined in Section 3.6.

$$\Phi = (c_O \cdot \Phi_O + c_U \cdot \Phi_U + c_I \cdot \Phi_I + c_N \cdot \Phi_N) \cdot R \cdot \log_B \bar{N}$$

The four subpotential functions are again defined as $\Phi_O = \sum_r \Phi_O^r$, $\Phi_U = \sum_r \Phi_U^r$, $\Phi_I = \sum_r \Phi_I^r$, and $\Phi_N = \sum_r \Phi_N^r$, where

$$\Phi_O^r = \max \{0, S_r/R - 1/2\}$$

$$\Phi_U^r = \begin{cases} 0 & \text{if } y^r = +\infty \\ \max \{0, 1/2 - S_r/R\} & \text{otherwise} \end{cases}$$

$$\Phi_I^r = \begin{cases} \max \{0, I_r/R - 1/2\} & \text{if } y^r = +\infty \\ \max \{0, (I_r/R - 1/2) (9 - 16 \cdot \min \{S_r/R, 1/2\})\} & \text{otherwise} \end{cases}$$

$$\Phi_N^r = \max \{0, N_r^L - t_N/2\} + \max \{0, N_r^R - t_N/2\}$$

For a rectangle r , we denote the four potentials Φ_O^r , Φ_U^r , Φ_I^r and Φ_N^r for *overflow*, *underflow*, *internal endpoint* and *neighbor potentials*, respectively.

The max functions ensure that the potential is never negative. E.g., for spanning segments, subtracting $R/2$ from S_r insures that each rectangle does not have any potential for the first $R/2$ spanning segments, and then linear potential on the rest. Similarly for the rest of the potential functions.

The potential function Φ_I^r for internal points contains the same base function on the internal point count, but then depending on if the rectangle can contain underflow potential, this base function is multiplied by a function dependent on the amount of underflow. This leads to the value of the function being scaled larger when there is more underflow. As for all non-rightmost rectangles, it must hold that $S_r \geq R/4$, then the *scale value* $9 - 16 \cdot \min \{S_r/R, 1/2\}$ is in the interval $[1, 5]$.

Note that cloning and queries do not alter the potential. The potential must therefore be analyzed only for INSERT and DELETE updates. Any update can be split into two parts: updating a segment in some rectangle and a potential re-partitioning of the rectangles.

For each of the rebalancing operations, the colored predecessor structure and point location query structure need to be updated for the new rectangles. In Section 4 we describe that inserting a new rectangle into the colored predecessor structure requires amortized $O(R \log_B \bar{N})$ I/Os, as each rectangle contains $\Theta(R)$ versions, and each version can be inserted in amortized $O(\log_B \bar{N})$ I/Os. Further, we describe in Section 5 that inserting new rectangles into the point location query structure requires amortized $O(R \log_B \bar{N})$ I/Os. As both operations must be performed with the released potential, then a rebalancing operation needs to release $R \log_B \bar{N}$ potential to pay for the work. In Section 3.5 and 3.6 we show that the factor preceding $R \log_B \bar{N}$ in Φ decreases by at least 1 for every rebalancing operation.

For each update operation without rebalancing, Φ may increase by no more than $O(\log_B \bar{N})$ for updates to use amortized $O(\log_B \bar{N})$ I/Os. We achieve this by showing in Section 3.4 that the factor preceding $R \log_B \bar{N}$ in Φ increases by $O(1/R)$. Table 3 summarizes how the potential changes with and without rebalancing.

3.4 Update without Rebalancing

When performing update operations without rebalancing the rectangles, the partitioning of the rectangles remains unaltered, and thus the neighbors of all rectangles are unchanged. This results in $\Delta\Phi_N = 0$ for both update operations.

INSERT. For an insert operation, a segment $\{x\} \times [v, \text{succ}(v)[$ is inserted into a rectangle $r = [x^r, y^r] \times [v^r, w^r]$. This may create a spanning segment if the rectangle only spans a single version, or the inserted value is inserted into the only version in the rectangle, where it was non-present, e.g., r already contains segments $\{x\} \times [v^r, v[$ and $\{x\} \times [\text{succ}(v), w^r]$. This leads to $\Delta\Phi_O \leq 1/R$. By creating a spanning segment, the underflow of spanning segments may decrease but not increase, resulting in $\Delta\Phi_U \leq 0$. The inserted segment may also create at most two new internal points in the rectangle. As the scale value is at most 5, then $\Delta\Phi_I \leq 10/R$.

DELETE. For a delete operation, a segment $\{x\} \times [v, \text{succ}(v)[$ is removed from a rectangle $r = [x^r, y^r] \times [v^r, w^r]$. This can then break or delete a spanning segment, which increases the underflow of segments, leading to $\Delta\Phi_U \leq 1/R$. As the number of spanning segments can only decrease, then $\Delta\Phi_O \leq 0$. By breaking a segment, up to two new internal endpoints may be created in the rectangle. The two new internal points then increase the potential function by at most $10/R$. The decrease of spanning segments in the internal point potential function then leads to a further increase, since the scale value of Φ_I^r can increase. As the number of segments is at most $2R$, and at least $R/4$ are spanning, then the number of internal points is at most $7/2 \cdot R$, leading to an increase of at most $(7/2 - 1/2) \cdot 16/R = 48/R$. In total $\Delta\Phi_I \leq 58/R$.

3.5 Rebalancing Rectangles

If an update in rectangle r causes the invariants to be violated, we perform three phases of rebalancing operations to restore them. First, if the update is a deletion that causes the number of spanning segments to fall below $R/4$ then we perform a single merge to restore invariant I_2 . The resulting rectangles contain up to $6R + 1$ segments as at most 3 rectangles are joined. Second, invariant I_1 is restored in all rectangles containing more than $2R$ segments by performing $O(t_N)$ vertical and horizontal splits as described in Section 3.1. The first two phases may cause rectangles to have many neighbors. In phase three, we fix this by making neighbor splits until all rectangles have fewer than t_N neighbors, which restores invariant I_5 . If the update is an insertion or deletion that causes the rectangle to contain more than $2R$ segments but not too few spanning segments, rebalancing starts from the second phase. In the remainder of this section, we show how the different subpotential functions change for every rebalancing operation.

Vertical split. We perform a vertical split only if the number of segments is $|S^r| > 2R$, and the number of spanning segments is

$S_r \geq R$. The split is performed at the median value of the spanning segments. This evenly partitions the spanning segments, such that both new rectangles contain $\geq R/2$ spanning segments. Thus, $\Delta\Phi_U = 0$. For Φ_O , due to the max function and that there now are two rectangles, the potential is released from the threshold in both the new rectangles, i.e., $\Delta\Phi_O = -1/2$.

For the internal points, there is no guarantee as to how they are distributed, but in any case, then $\Delta\Phi_I \leq 0$, since the number of internal points is unchanged.

When splitting the rectangle, all left neighbors become left neighbors of the left rectangle, and all right neighbors become right neighbors of the right rectangle, thus not altering the neighbor potential. Since $t_N \geq 2$, the two new rectangles being neighbors do not yield any increase in potential, and thus $\Delta\Phi_N = 0$.

Horizontal split. We perform a horizontal split only if the number of segments is $|S^r| > 2R$, and the number of spanning segments is $S_r < R$. Since each of the at least R non-spanning segments contributes at least one internal endpoint, then $I_r \geq R$. The median version among the internal endpoints is then found using the local versions, at which the split will occur. This evenly partitions the internal points into the new top and bottom rectangles, with some endpoints landing on the split line, which will no longer be internal points. If the rectangle has no underflow potential, a similar analysis as for vertical splits leads to $\Delta\Phi_I = -1/2$ and $\Delta\Phi_U = 0$.

If the rectangle has some underflow potential, then the scale value of the internal point potential function is larger than one. During the split, new spanning segments may be created, but worst-case no new spanning segments are created, and the underflow potential is transferred to both new rectangles. This results in $\Delta\Phi_U \leq 1/2 - S_r/R$ and $\Delta\Phi_I \leq -1/2 (9 - 16S_r/R)$. As the rectangle, in this case, has underflow potential, then it must be a non-rightmost rectangle, and thus $R/4 \leq S_r < R/2$.

When splitting horizontally, all spanning segments in the original rectangle become spanning in both new rectangles. Any other segment may become spanning in at most one of the two rectangles. In total, each segment may yield one extra spanning segment compared to before the split. As there are at most $6R + 1$ segments, and there is no potential for the first $R/2$ new spanning segments, then $\Delta\Phi_O \leq (6R + 1 - R/2)/R \leq 6$, assuming that $R/2 \geq 1$.

Consider one side of neighbors. The split distributes these neighbors among the top and bottom rectangle, with at most one becoming the neighbor of both. None of the two new rectangles can have more neighbors than the original rectangle. If one rectangle has the same number of neighbors as the original rectangle, then the other has one neighbor. The neighbor potential thus does not increase internally among the rectangles. However, the split leads to the one neighboring rectangle at the split point, getting an additional neighbor. Considering both sides, this leads to $\Delta\Phi_N \leq 2$.

Neighbor split. A neighbor split occurs when the number of neighbors on one side of a rectangle is strictly larger than t_N , the number of segments is $\leq 2R$ and the number of spanning segments is $S_r \geq R/4$ if the rectangle is non-rightmost. The median version is selected among the versions where the neighbor changes and a horizontal split is made at this version. This distributes the neighbors evenly among the new top and bottom rectangle, thus decreasing the potential. However, for the other side, one split is made, leading

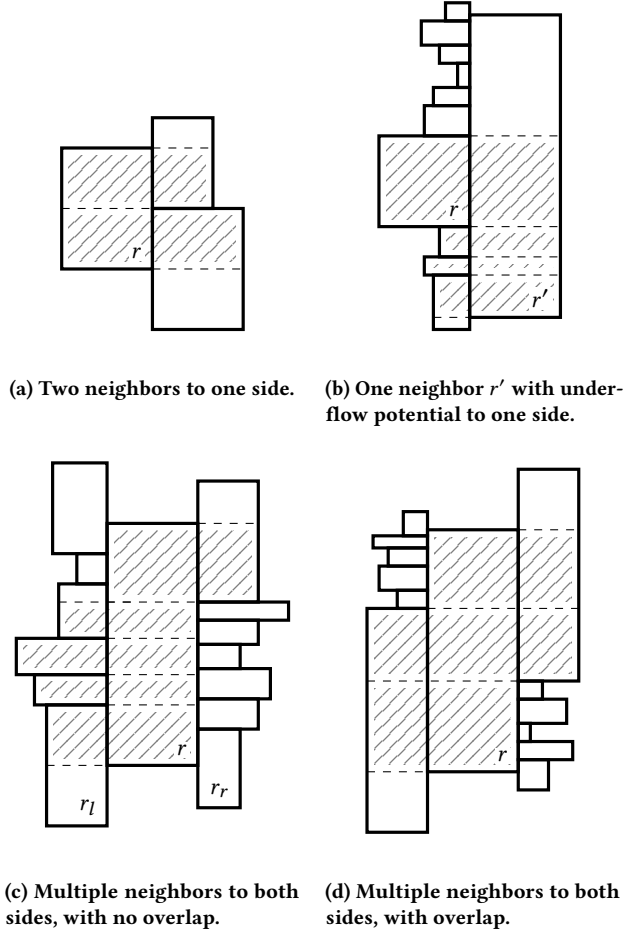


Figure 5: The different merge cases, with the underflowing rectangle being r , apart from when r is a leftmost rectangle. The dashed lines represent the location where the rectangles are horizontally split. The grayed areas indicate the newly joined rectangles.

to a potential increase of at most one neighbor. In total, this results in $\Delta\Phi_N \leq -t_N/2 + 1$.

By similar arguments as for horizontal splits, the at most $2R$ segments lead to $\Delta\Phi_O \leq 2R/R - 1/2 = 3/2$ and $\Delta\Phi_U \leq 1/2 - R/4/R = 1/4$. Note, however, that they cannot both increase. Since the internal points are distributed among the top and bottom rectangle and the scale value can only decrease, then $\Delta\Phi_I \leq 0$.

Merge. A merge occurs, when a rectangle r is underflowing, i.e., when $S_r < R/4$ and the rectangle is not a rightmost rectangle. Note that the number of segments in the rectangle is at most $2R + 1$ and that all neighbors that are non-rightmost rectangles contain at least $R/4$ spanning segments, and at most $2R$ segments. There are then multiple cases for how the merge is performed, in order to make sure not too many new neighbors are created. These cases can be seen in Figures 4 and 5.

Two neighbors on some side. This case is seen on Figure 5a. If one side of r contains exactly two neighbors, the neighbors on this side can be split, as well as r itself, to align the rectangles for joining. The three splits result in $\Delta\Phi_N \leq 3$. One half of the split neighbors are not joined, and using similar arguments as earlier, they cannot increase the potential of the internal points or underflow. The joined rectangles must contain at least $R/2$ spanning segments and thus have no underflow potential. This removes the underflow of r , resulting in $\Delta\Phi_U \leq -1/4$.

There are at most $R/2$ internal points in r with no internal end-point potential, which now potentially get potential. However, as they are placed into rectangles with at least $R/2$ spanning segments, the scale value is 1. The potential on the already existing internal points may decrease, due to the increase in spanning segments and implied decrease in scale value, however, it cannot increase. Therefore $\Delta\Phi_I \leq 1/2$.

The neighbor rectangles contain at most $2R$ spanning segments, and their overflow potential each increase by at most $2/3$. Rectangle r contains at most $2R+1$ segments. As these segments are distributed into two rectangles, each of which may contain $\geq R/2$ spanning segments, then all distributed segments may be spanning. The spanning segments of r , of which there are $< R/4$, may then become spanning in both new rectangles. The rest of the segments can only be spanning in one of the new rectangles. The extra $+1$ segment arrives from some old spanning segment that was broken and has a gap of at least one version. Therefore at most $2R - R/4$ of the remaining non-spanning segments may become new spanning segments. This results in $\Delta\Phi_O \leq 5 + 1/4$.

Note that this case is a special case of the basic merge seen in Figure 4.

One neighbor on some side. This case is shown in Figure 5b. If some side has one neighbor r' , and the other does not have two neighbors, then it is favorable to join with r' , as this limits the neighbor increase in r' . To join r with r' , r' may need to be split twice (at the top and bottom of r), leaving two additional rectangles with the same underflow as r' . If r' has no underflow, this is not a problem, and the merge can stop by simply joining r into the split middle section of r' . For the following, assume that r' has underflow potential.

In this case, one of the two created rectangles from r' must have its underflow potential removed to release the potential needed for the operation. Consider the same side of r' as the side r is on. This side must contain some neighbors above and below r . Choose the half, which contains the least number of neighbors, and split the part of r' to this half, and join all of the rectangles, which then must remove the underflow. It must hold that this half has at most $t_N/2$ neighbors, leading to at most $t_N/2 + 2$ splits when factoring in the extra splits to align the neighbors with r' and the split from r . By a similar argument as above, it must hold that $\Delta\Phi_U \leq -1/4$, as the underflow potential of r is released.

The at most $2R$ segments of r' are split into at most $t_N/2 + 2$ rectangles, where they may be new spanning segments in all but one of them, increasing the overflow potential by $\leq t_N + 2$. Rectangle r was underflowing and had less than $R/4$ spanning segments, and therefore the first $R/4$ spanning segments added to r from r' do not increase the potential. The last split rectangle increases the

potential by $3/2$ by a similar argument as above. In total resulting in $\Delta\Phi_O \leq t_N + 2 - 1/4 + 3/2 = t_N + 3 + 1/4$.

Consider the internal points of r' , where some are added to r and the rectangles below or above r . By a similar argument as above, it also holds that $\Delta\Phi_I \leq 1/2$.

Assume that r' has k neighbors on the side r is on, and only consider the change in the neighbor potential for these rectangles. Then at most $\lceil k/2 \rceil$ neighbors will change from being a neighbor of r' into being a neighbor of other rectangles. The rounding is when k is odd, as there may be $\lfloor k/2 \rfloor$ neighbors on either side of r , and r is one of the rectangles which no longer is a neighbor of r' . Worst case, they all become neighbors of the same rectangle, increasing the neighbor potential in this rectangle by at most $\lceil k/2 \rceil$. The difference in potential of r' is at most $\max\{0, \lceil k/2 \rceil - t_N/2\} - \max\{0, k - t_N/2\}$. Lastly, an aligning split increases the neighbor potential by at most 1. In total, the potential increase is maximized for $k = t_N/2$ resulting in $\Delta\Phi_N \leq t_N/4 + 2$.

Note that this case uses the basic merge seen in Figure 4 twice, once for merging r into r' , and once to merge the top or bottom rectangle of r' into its neighbors.

Multiple neighbors on both sides. In this case, many splits must be made. If r is joined with the neighbors to the left, this may increase the number of neighbors for the rectangles on the right side. If, however, all of these right neighbors only originally have r as a left neighbor, then the neighboring potential for these rectangles cannot increase in total, as the neighbor potential r has on the right is released.

This holds for all right neighbors, except for the top and bottom. If r is split to align with these, it can then freely be split in between. To not increase the number of neighbors of the top and bottom rectangle on the left, the splits are made to align with the side, which covers most of r . This can be seen in Figure 5c, where r is split at the top of r_l instead of at the top of r_r , as r_l covers most of r . The top of both rectangles cannot however be directly compared, apart from checking if they are identical, but by making a point query at (x^{r_r}, w^{r_l}) , i.e., at the top value of r_l , but at the left side of r_r , then this point will be inside r_r only if the top of r_l is below the top of r_r . Similarly for the top neighbors.

The selected top and bottom neighbors may overlap with the part of r they cover, as seen in Figure 5d, further creating two cases. This happens only when the selected neighbors are from each side. If they do not overlap but exactly cover r , then this reduces to the case with two neighbors. Note that overlapping can be checked with a point query, like above. For both cases, it holds that $\Delta\Phi_U \leq -1/4$, as the underflow potential of r is removed, and no new underflow potential is introduced, and $\Delta\Phi_I \leq 1/2$ by similar arguments as above, as the internal points of r are distributed into rectangles with no underflow potential.

No overlap. This is the case illustrated in Figure 5c. In this case, four splits are initially made, two on the selected neighbors and two in r , and potentially a fifth split to align the space between the top and bottom split. All joined rectangles in between become neighbors of rectangles that previously only had r as a neighbor, therefore not increasing the potential. In total, the neighbor potential may only increase for the 3 splits outside of r , resulting in $\Delta\Phi_N \leq 3$. Rectangle r is split at most $t_N - 1$ times. The increase in overflow potential

for the other three split rectangles is at most $3/2$ each, by a similar argument as above. In r there are at most $R/4$ spanning segments, which may be spanning at most t_N rectangles, and the remaining segments may be spanning in at most $t_N - 1$ rectangles, by similar argument as above. In total resulting in $\Delta\Phi_O \leq 2t_N + 3 + 3/4$.

Overlap. This is the case illustrated in Figure 5d. In this case, the selected neighbors overlap vertically. By further splitting these neighbors, r can be joined into three rectangles. The splits in the neighbors result in $\Delta\Phi_N \leq 4$. By similar arguments as above, the increase in overflow potential for r is at most $4 + 1/4$. In the neighboring rectangle, there are at most $2R$ segments, which all may become spanning in two new rectangles. Each of the three rectangles does not have potential for the first $R/2$ spanning segments, resulting in total $\Delta\Phi_O \leq 10 + 3/4$.

Leftmost rectangles. Rectangles at the rightmost side do not have underflow potential and thus cannot trigger a merge. Rectangles at the leftmost side, however, only have neighbors on the right side. In this case, r must be merged with the right neighbors, however, as it has no left neighbors, the neighbor potential on this side cannot increase. This is done as seen in Figure 4. By similar arguments as earlier, it can be shown that $\Delta\Phi_U \leq -1/4$, $\Delta\Phi_I \leq 1/2$, $\Delta\Phi_N \leq 2$ and $\Delta\Phi_O \leq 2t_N + 1 + 1/4$.

3.6 Potential Function Constants

The differences in the subpotential functions, as calculated in Section 3.4 and Section 3.5 can be seen summarized in Table 3. For the insert and delete rows at the top, the differences are without rebalancing.

In Φ the subpotential functions are a linear combination. In the table, each rebalancing operation row must decrease by 1 to release $R \log_B \bar{N}$ potential. In order to find working constants c_N , c_O , c_U and c_I , the rows of the table can be viewed as constraint for a linear program, where each row must be less than -1 . Some values are dependent on t_N , which is a constant. In order to find a solution, the linear program is tried solved for increasing values of t_N , until a solution is found. For the constraint from horizontal split with underflow potential, $\Delta\Phi_U$ and $\Delta\Phi_I$ are not constants since the scale value can change. However, they are linear bounds in S_r . By creating two constraints for the minimum and maximum value of S_r , which is $R/4$ and $R/2$ respectively as there must be underflow potential in this case, the program may be solved.

The objective function to minimize in the linear program, is chosen to be the sum of the constants. This yields the following constants

$$t_N = 22 \quad c_N = 573 \quad c_O = 2 \quad c_U = 22904 \quad c_I = 2754.$$

These constants can then be used to check if the constraint from horizontal split with underflow potential is satisfied, by calculating the left side of the constraint:

$$\begin{aligned} & 573 \cdot 2 + 2 \cdot 6 + 22904 \cdot (1/2 - S_r/R) + 2754 \cdot (-1/2 \cdot (9 - 16S_r/R)) \\ & = 217 - 872S_r/R. \end{aligned}$$

This value is large, when S_r is small. The smallest value of S_r is $R/4$, resulting in the maximum value of the left side being -1 . The constraint is therefore satisfied, and the solution to the constants is valid.

Table 3: Difference in the subpotential functions for the different update operations and rebalancing operations.

Operation	$\Delta\Phi_N$	$\Delta\Phi_O$	$\Delta\Phi_U$	$\Delta\Phi_I$
Insert	0	$\leq 1/R$	≤ 0	$\leq 10/R$
Delete	0	≤ 0	$\leq 1/R$	$\leq 58/R$
Vertical split	0	$-1/2$	0	≤ 0
Horizontal split ($\Phi_U^r = 0$)	≤ 2	≤ 6	0	$-1/2$
($\Phi_U^r > 0$)	≤ 2	≤ 6	$\leq 1/2 - S_r/R$	$\leq -1/2 \cdot (9 - 16S_r/R)$
Neighbor split	$\leq -t_N/2 + 1$	$\leq 3/2$	$\leq 1/4$	≤ 0
Merge (Figure 5a, $N_r^L = 2 \vee N_r^R = 2$)	≤ 3	$\leq 5 + 1/4$	$\leq -1/4$	$\leq 1/2$
(Figure 5b, $N_r^L = 1 \vee N_r^R = 1$)	$\leq t_N/4 + 2$	$\leq t_N + 3 + 1/4$	$\leq -1/4$	$\leq 1/2$
(Figure 5c, $N_r^L \geq 3 \wedge N_r^R \geq 3$, no overlap)	≤ 3	$\leq 2t_N + 3 + 3/4$	$\leq -1/4$	$\leq 1/2$
(Figure 5d, $N_r^L \geq 3 \wedge N_r^R \geq 3$, overlap)	≤ 4	$\leq 10 + 3/4$	$\leq -1/4$	$\leq 1/2$
(Figure 4, $x^r = -\infty \wedge N_r^R \geq 3$)	≤ 2	$\leq 2t_N + 1 + 1/4$	$\leq -1/4$	$\leq 1/2$

3.7 Maintaining Structure inside a Rectangle

The remaining invariants to maintain are I_3 and I_4 . For update operations without rebalancing, the updated segments can be inserted into S^r to maintain the order. An update may only add new versions to r , and using the colored predecessor structure, the new versions can be inserted into L^r at the right position. The segments can then be scanned, to update the local versions of the endpoints.

For the rebalancing operations, first note that S^r can maintain the value order during horizontal and vertical splits, as the segments may be partitioned in order, and for vertical joins all segments in one rectangle are before all rectangles in the other, and the lists can then simply be concatenated.

For a horizontal split L^r is split at some version, and the resulting segments in the upper rectangle can be scanned to update the local versions. For a vertical split, L^{r_1} and L^{r_2} must be reconstructed from the endpoints of the segments in the rectangles. As binary merge-sort on $O(R)$ elements uses $O(R/B \log_2 R) = O(R \log_B \bar{N})$ I/Os, then $O(R)$ elements can be sorted a constant number of times using the released potential. This allow for constructing L^{r_1} and L^{r_2} for the resulting rectangles from the endpoints of the segments, by sorting them by the local version in L^r .

When vertically joining two rectangles, two version lists L^{r_1} and L^{r_2} needs to be merged into L^r . Firstly, the start and end of the two list must be equal, as the top and bottom version of the rectangles are equal. Using the colored predecessor structure, before it is updated, the versions may be merged, as the predecessor query can be used to determine the ordering. As the list contains $O(R)$ versions, and each predecessor uses $O(\log_B \bar{N})$ I/Os, the resulting $O(R \log_B \bar{N})$ I/Os can be payed by the released potential. The local ordering on the segments can then be done by a constant number of sorts and scans.

3.8 Space Usage

Each update operation increases the value of the subpotential functions by $O(1/R)$. Each rebalancing operation decreases the subpotential functions by at least 1, and creates $O(1)$ new rectangles. There are at most $\bar{N}$ updates performed on the structure, resulting in $O(\bar{N}/R)$ rectangles. Each rectangle uses $O(R/B)$ blocks of space, i.e., the total space usage is $O(\bar{N}/R) \cdot O(R/B) = O(\bar{N}/B)$ blocks.

4 COLORED PREDECESSOR QUERIES

In this section we sketch our solution to the colored predecessor problem (details are available in the full version of the paper). That is, given a version v and a rectangle r , find the predecessor version of v in the local version list L^r . We consider each rectangle to represent a unique *color*. The global version list L is ordered from left-to-right such that the predecessor of a version is to the left of it. We overload the notation of r such that it also defines the color corresponding to the rectangle r and let $C = \{1, 2, 3, \dots\}$ be the set of colors. In the point location structure in Section 5 we use the simple case with only a single color ($|C| = 1$) to find the predecessor of a version among the bottom versions of the rectangles. The following theorem states our result for the colored predecessor problem.

THEOREM 4.1. *Let N be a parameter giving an upper bound on the total number of updates to the global version list L and all local version lists L^r . Then there exists a data structure that given a version $v \in L$ and a color r , can find the predecessor u of v in L^r in worst-case $O(\log_B N)$ I/Os. The structure supports insertions of versions in both L and L^r and deletions from L^r in amortized $O(\log_B N)$ I/Os. The space usage is linear in the number of updates.*

We store all versions from the global version list L and all local version lists L^r at the leaves of a B-tree ordered by the global version list, i.e., the same version can appear multiple times. For every local version $v_r \in L^r$, we conceptually color the path from the leaf containing v_r to the root by the color r , i.e., an internal node can have up to $|C|$ colors. The predecessor of a version v among the versions in L^r can now be found by following the path from v towards the root until a node of color r is found that has child of color r that is to the left of the search path, from which the search reverses towards the leaves following the rightmost nodes of color r (starting with a child to the left of the path). To avoid storing a color at every node on the path to the root we observe that when considering the colored paths from the root towards the leaves it suffices to only color the nodes where paths branch. The number of branches is $O(N)$. To achieve query complexity $O(\log_B N)$ we use the idea of *down pointers* [27] to avoid searching in every node among its colors. Finally, we make the structure dynamic using a weight-balanced B-tree [5].

5 POINT LOCATION

In this section, we sketch how to find the unique rectangle containing a query point (v, x) , where v is a version and x is a value (details are available in the full version of the paper). Recall that we consider a disjoint rectangular partition of the plane as described in Section 2.3. At first, this appears to be an orthogonal planar point location problem. However, a crucial difference in our setting is that versions (corresponding to the vertical axis) are ordered according to their position in the version list. Because of this, it is hard to simply apply existing algorithms. Instead, we formulate it as a *dynamic ray shooting* problem on the bottom segments of each rectangle. Throughout this section, we assume that given a version v we can quickly find the predecessor of v among the versions represented by bottom segments. This is a special case of the colored predecessor problem we solve in Section 4 and allows us to assume that v is a version corresponding to a bottom segment of some rectangle and that for insertions of new segments, we know its correct position among the existing segments, with regards to the global ordering of versions. The bounds we get are summarized in Theorem 5.1

THEOREM 5.1. *There exists a semi-dynamic ray shooting data structure for non-overlapping horizontal segments, supporting ray shooting queries in worst-case $O(\log_B S)$ I/Os and insertions in amortized $O(B \log_B^2 S)$ I/Os, where S is the number of segments inserted. The space usage is $O(S \log_B S)$ blocks. Furthermore, segments are only compared on the vertical axis for equality, that is testing if they are at the same height.*

In Section 3.8 it was shown that $S = O(\bar{N}/R)$ resulting in the promised bound for point location queries in Section 2.4, and the insertions are within the potential released, as described in Section 3.3. Note that the space usage is $O(\bar{N}/B)$ blocks.

To see how this theorem applies to our problem we first observe that it suffices to consider insertions in the structure. Each segment corresponds to the bottom of a rectangle and thus the primitive rectangle transformations (Figure 3) and consequently the merge rectangle transformations (Figures 4 and 5) define how the set of segments change. Crucially, in all cases any point covered by a segment must still be covered, i.e., the area covered by segments only increases. Thus, using only insertions we can make sure that a given ray shooting query finds some segment at the correct version, by only inserting the segments that cover previously uncovered areas. For every version we maintain the real set of segments created by rectangle transformations in a B-tree, sorted on the horizontal axis. Since there are at most S segments we can now find the correct segment to report with an overhead of only $O(\log_B S)$. The overhead for insertions is similar.

Our approach is to store the segments in a segment tree T . For every node u in T , we have a secondary structure with a subset of the versions corresponding (primarily) to segments that span the horizontal interval defined by u . For a query (v, x) we find the predecessor of v in all the secondary structures on the path down to the leaf in T which contains x in its interval, as one of these predecessors will be the segment we are looking for. To determine the real predecessor among the candidate predecessors from the path

we perform consecutive predecessor queries efficiently using a variation of fractional cascading [14]. Finally, since we need a dynamic version of the structure we use a weight-balanced B-tree [5].

6 PARTITIONING THE VERSION TREE

In this section, we prove the following theorem which describes how to improve the I/O bounds in Theorem 1.2, such that they depend on the size N_v of the accessed version v instead of the upper bound $\bar{N}$ on the total number of updates.

THEOREM 6.1. *Assume we have a data structure for external-memory fully-persistent search trees supporting a sequence of at most $\bar{N}$ updates, for a constant $\bar{N}$, that supports INSERT, DELETE and CLONE in amortized $O(\log_B \bar{N})$ I/Os, SEARCH in worst-case $O(\log_B \bar{N})$ I/Os, and RANGE in worst-case $O(\log_B \bar{N} + K/B)$ I/Os, and uses space linear in the number of updates. Then there exist external-memory fully-persistent search trees supporting INSERT, DELETE and CLONE in amortized $O(\log_B N_v)$ I/Os, SEARCH in worst-case $O(\log_B N_v)$ I/Os, and RANGE in worst-case $O(\log_B N_v + K/B)$ I/Os, and uses space linear in the number of updates.*

The basic idea of our approach is to split the version tree into smaller version trees by cutting out subtrees after a certain number of updates, such that all versions in a version tree have approximately the same size.

To be more precise, let T be a version tree rooted at version v_0 . We let $N_0(T)$ be the initial number of insertions in version v_0 before other updates are performed at version v_0 . We let $upd_T(v_0)$ denote the number of insertions, deletions and clones (i.e., $|T| - 1$) in T , excluding the initial $N_0(T)$ insertions in version v_0 . For a subtree T_v rooted at a non-root node v , we let $upd_T(v)$ denote the number of insertions, deletions and clones (i.e., $|T_v|$) in T_v .

Let c be a constant, where $0 < c < 1$. We maintain the invariant $upd_T(v_0) \leq cN_0(T)$. Crucially, this ensures that $|N_v - N_0(T)| \leq cN_0(T)$ for any version v in T . We split T when $upd_T(v_0) = \lfloor cN_0(T) \rfloor$. We say that a version v is *small* if $N_v < \frac{2(2-c)}{c(1-c)}$. Small versions can be maintained naively as a list of values. If a new version tree arising from a split has a small version as the root, we get rid of the tree and instead represent all small versions in the tree naively. Each version in the tree that is not small becomes a version tree by itself. As soon as a version that is maintained naively is not small, it is converted into a version tree.

Choosing $\bar{N} = (1+c)N_0(T)$ ensures $\bar{N} \leq \frac{1+c}{1-c}N_v$ for all versions v in T , i.e., $O(\log_B \bar{N}) = O(\log_B N_v)$ and the bounds in Theorem 6.1 follow, ignoring the cost for splitting version trees. The split of T will result in four version trees T' , T_H , T_L and T_R , some of which may be empty, and we rebuild each of them by simply performing all the updates again to an initial empty version tree. We assume that the version tree and the history of all insertions and deletions for each version are maintained explicitly so that we can easily repeat all operations. This introduces a constant space overhead for each update.

To split T , we first find a *heavy subtree* T_v rooted at a node v with many updates, but where none of the subtrees at the children are heavy. This ensures that by cutting T_v from T many updates must be performed before the next split of T . More formally, the node v must satisfy the following,

- (1) Node v is *heavy*, that is $\text{upd}_T(v) > \frac{c}{2}N_0(T)$, and
- (2) $\text{upd}_T(w) \leq \frac{c}{2}N_0(T)$ for all children w of v .

We can always find a node v satisfying the conditions above by following a path of heavy nodes from the root v_0 towards the leaves since the leaves always satisfy the second condition. We now describe the four resulting trees and show that a linear number of updates must be performed in each of them before they are split again.

We construct the subtree $T' = T \setminus T_v$ only if $v_0 \neq v$, where T_v is the subtree of T rooted at v . In version v_0 of T , there might be values that have been inserted but have been canceled again by deletions. In T' the initial insertions in version v_0 are all the insertions in version v_0 of T , including the initial insertions, that have not been canceled by a deletion. Version v_0 of T' contains no deletions. For all other versions in T' , the updates are the same as in T . The number of insertions left in v_0 in T' defines the new $N_0(T') \geq N_0(T) - (\text{upd}_T(v_0) - \text{upd}_T(v)) > N_0(T) - (cN_0(T) - \frac{c}{2}N_0(T)) = (1 - \frac{c}{2})N_0(T)$, as all updates, but the updates in v may be deletions in v_0 . Similarly, the number of updates in T' is now $\text{upd}_{T'}(v_0) = \text{upd}_T(v_0) - \text{upd}_T(v) < cN_0(T) - \frac{c}{2}N_0(T) = \frac{c}{2}N_0(T)$. Therefore

$$\frac{\text{upd}_{T'}(v_0)}{N_0(T')} < \frac{\frac{c}{2}N_0(T)}{(1 - \frac{c}{2})N_0(T)} = \frac{c}{2 - c},$$

and a gap of at least $\lfloor cN_0(T') \rfloor - \text{upd}_{T'}(v_0) \geq (c - \frac{c}{2-c})N_0(T') - 1$ future updates must occur in T' before we need to split T' . Note $c - \frac{c}{2-c} > 0$ for all $0 < c < 1$. If v_0 is not small, i.e., $\frac{2(2-c)}{c(1-c)} \leq N_0(T')$, we always have a gap of at least one update. For $c = \frac{1}{2}$ the gap is of size $\frac{1}{6}N_0(T') - 1$ and $12 \leq N_0(T')$.

We split the subtree T_v into at most three version trees T_H , T_L , and T_R , depending on the number of children of v . Version v exists in all three version trees, but when an operation subsequently refers to version v , we let it refer to version v in T_H . The version tree T_H consists of v and the subtree at the child of v containing the most updates. If v has no children, T_H is still created but only contains v . Next, we greedily partition the remaining children of v into two sets L and R , such that the total number of updates in each is at most $\frac{c}{2}N_0(T)$. We then create the two subtrees T_L and T_R , which are rooted at v with children L and R , respectively. These version trees are only created if they have at least one child. Similarly to the case for the root v_0 of T' , for the root v of T_H , T_L , and T_R , we examine all the insertions and deletions on the path from v_0 to v in T and keep only the insertions that are not canceled by a deletion on the path. The resulting set of insertions is the initial set of insertions for v of size $N_0(T_H) = N_0(T_L) = N_0(T_R)$. We define d to be the number of deletions on the path from v to v_0 . Since $\text{upd}_{T_H}(v) \leq \frac{c}{2}N_0(T)$ by the second condition in the definition of v , and $\text{upd}_{T_L}(v) = \sum_{w \in L} \text{upd}_T(w) \leq \frac{c}{2}N_0(T)$ (similarly for T_R) by the choice of L and R , the analysis becomes the same for all three version trees with root v . Here we consider the analysis for T_L and note that $d \leq \text{upd}_T(v_0) - \sum_{w \in L} \text{upd}_T(w)$. The number of initial insertions in v in T_L is exactly the number of initial insertions in v_0 in T , that are not deleted by the d deletions on the path from v to v_0 , i.e., $N_0(T_L) = N_0(T) - d$. Therefore

$$\begin{aligned} \frac{\text{upd}_{T_L}(v)}{N_0(T_L)} &\leq \frac{\sum_{w \in L} \text{upd}_T(w)}{N_0(T) - (\text{upd}_T(v_0) - \sum_{w \in L} \text{upd}_T(w))} \\ &\leq \frac{\frac{c}{2}N_0(T)}{(1 - c)N_0(T) + \frac{c}{2}N_0(T)} = \frac{c}{2 - c}. \end{aligned}$$

Thus, we get the same gap as for T' .

Finally, we argue about the resulting I/O and space bounds. Each of the four resulting version trees created by a split contains at most $(1 + \frac{c}{2})N_0(T)$ updates, i.e., by assumption can be constructed using $O(N_0(T) \log_B N_0(T))$ I/Os. Since there have been performed at least $(c - \frac{c}{2-c})N_0(T) - 1$ updates since T was created, we can charge the cost of splitting T to these updates, yielding an additional amortized $O(\log_B N_0(T))$ cost per update. Since all versions in T have size at least $(1 - \frac{c}{2})N_0(T)$, we can restate the I/O cost as amortized $O(\log_B N_v)$. Similarly, the additional $O(N_0(T_H)/B)$ blocks of space overhead when splitting T for introducing v and all $N_0(T_H)$ initial updates to v in all three version trees T_H , T_L and T_R , can be charged to the at least $(c - \frac{c}{2-c})N_0(T) - 1$ updates to T since T was constructed, i.e., the space usage remains linear.

In the above analysis we showed that by setting $c = \frac{1}{2}$, we are guaranteed for each of the resulting trees, e.g., T' , at least $\frac{1}{6}N_0(T') - 1$ updates to T' are required before T' is required to be split. Any choice of $0 < c < 1$ works, and at least $(c - \frac{c}{2-c})N_0(T') - 1$ updates are required before a split. The best choice is $c = 2 - \sqrt{2}$, where the lower bound becomes $(3 - 2\sqrt{2})N_0(T') - 1 = 0.1715N_0(T') - 1$, that is slightly better than the $\frac{1}{6}N_0(T') - 1 = 0.1667N_0(T') - 1$ lower bound achieved by setting $c = \frac{1}{2}$.

7 LAZY CLONES

In this section we prove the following theorem which describes how to improve the amortized I/O bound on CLONE operations in Theorem 6.1, such that CLONE operations use worst-case constant I/Os.

THEOREM 7.1. *Assume we have a data structure for external-memory fully-persistent search trees that supports INSERT, DELETE and CLONE in amortized $O(\log_B N_v)$ I/Os, SEARCH in worst-case $O(\log_B N_v)$ I/Os, and RANGE in worst-case $O(\log_B N_v + K/B)$ I/Os, and uses space linear in the number of updates. Then there exist external-memory fully-persistent search trees supporting INSERT and DELETE in amortized $O(\log_B N_v)$ I/Os, CLONE in worst-case $O(1)$ I/Os, SEARCH in worst-case $O(\log_B N_v)$ I/Os, and RANGE in worst-case $O(\log_B N_v + K/B)$ I/Os. The space usage is linear in the number of updates.*

The idea of the construction to reduce the I/O bound for CLONE operations, is to postpone the actual cloning to the first update to the version and to charge the cost for the cloning to the later update instead. Any version v , which has not been the subject of any updates, must contain the same values as the (locked) parent version u it has been cloned from. Therefore, any clone or query performed on version v can be performed on version u . Hence version v does not need to be explicitly created in the structure. When version v is cloned, it will become locked, and a new unlocked version w is created that is identical to both v and u . Thus, the same result is obtained as if w was cloned from u .

In this way, *lazy clones* can be implemented using a single layer of references. Each version is either a *real* version existing in the underlying structure or a *lazy* version pointing to a locked real version containing the same values. When making a clone w of a lazy version v pointing to a real version u , version v can no longer receive updates and is locked. Version v remains lazy and is never explicitly constructed. The new version w becomes lazy, pointing to the real version u . As a `CLONE` operation always creates a lazy version from some real version, and a real version can be found by traversing at most 1 pointer, then `CLONE` operations use worst-case $O(1)$ I/Os. Any `INSERT` or `DELETE` operation must first check if the operation is performed on a real or lazy version, which uses $O(1)$ additive I/Os. If the version is lazy, it must first be made real, which uses amortized $O(\log_B N_v)$ I/Os, by performing a `CLONE` operation on the underlying data structure given to the construction. After this, the update can be performed in amortized $O(\log_B N_v)$ I/Os, resulting in total amortized $O(\log_B N_v)$ I/Os. Similarly, `SEARCH` and `RANGE` operations on lazy versions are instead performed on the equivalent real versions. The overhead is $O(1)$ additive I/Os. Finally, storing the single layer of references from lazy versions to real versions requires only additive linear space. This concludes Theorem 7.1.

8 CONCLUSION AND FUTURE WORK

This paper presents external-memory fully-persistent B-trees with I/O bounds (Theorem 1.1) matching those of classical B-trees [6]. A natural open question is whether this result can be extended to the update-query trade-off regime by buffering updates as was done for classical B-trees. Adopting our solution seems plausible but nontrivial since it likely requires buffering the point location structure. In the B^e tree, queries can be performed efficiently since all the relevant buffered updates are on the path in the tree from the root to the query position. However, that would not be the case in the segment tree, which our point location structure is built upon. Another direction is to improve the amortized bounds to instead hold with high probability or worst-case. The main obstacle here seems to be how to handle rectangle rebalancing. These open questions are similar to the improvements to the classical B-tree mentioned in Section 1.2, and likely some of those techniques can also be deployed here.

REFERENCES

- [1] Georgy M. Adelson-Velsky and Evgenii M. Landis. 1962. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)* 146 (1962), 263–266. English translation by Myron J. Ricci in *Soviet Mathematics - Doklady*, 3:1259–1263, 1962.
- [2] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127. <https://doi.org/10.1145/48529.48535>
- [3] Lars Arge, Gerth Stølting Brodal, and S. Srinivasa Rao. 2012. External Memory Planar Point Location with Logarithmic Updates. *Algorithmica* 63, 1 (2012), 457–475. <https://doi.org/10.1007/s00453-011-9541-2>
- [4] Lars Arge, Andrew Dannar, and Sha-Mayn Teh. 2003. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithmics* 8 (2003), 22 pages. <https://doi.org/10.1145/996546.996549>
- [5] Lars Arge and Jeffrey Vitter. 2003. Optimal External Memory Interval Management. *SIAM J. Comput.* 32 (09 2003), 1488–1508. <https://doi.org/10.1137/S009753970240481X>
- [6] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indices. *Acta Informatica* 1 (1972), 173–189. <https://doi.org/10.1007/BF00288683>
- [7] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. 1996. An Asymptotically Optimal Multiversion B-Tree. *The VLDB Journal* 5, 4 (1996), 264–275. <https://doi.org/10.1007/s007780050028>
- [8] Michael A. Bender, Rathish Das, Martin Farach-Colton, Rob Johnson, and William Kuszmaul. 2020. Flushing Without Cascades. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5–8, 2020*, Shuchi Chawla (Ed.). SIAM, 650–669. <https://doi.org/10.1137/1.9781611975994.40>
- [9] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Simon Muraas, Tyler Mayer, Cynthia A. Phillips, and Helen Xu. 2017. Write-Optimized Skip Lists. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) (PODS '17). Association for Computing Machinery, New York, NY, USA, 69–78. <https://doi.org/10.1145/3034786.3056117>
- [10] Gerth Stølting Brodal and Rolf Fagerberg. 2003. On the Limits of Cache-Obliviousness. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing* (San Diego, CA, USA) (STOC '03). Association for Computing Machinery, New York, NY, USA, 307–315. <https://doi.org/10.1145/780542.780589>
- [11] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. 2012. Fully Persistent B-trees. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17–19, 2012*. SIAM, 602–614. <https://doi.org/10.1137/1.9781611973099.51>
- [12] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. 2020. Fully persistent B-trees. *Theoretical Computer Science* 841 (2020), 10–26. <https://doi.org/10.1016/j.tcs.2020.06.027>
- [13] Bernard Chazelle. 1986. Filtering Search: A New Approach to Query-Answering. *SIAM J. Comput.* 15, 3 (1986), 703–724. <https://doi.org/10.1137/0215051>
- [14] Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional Cascading: I. A Data Structuring Technique. *Algorithmica* 1, 2 (1986), 133–162. <https://doi.org/10.1007/BF01840440>
- [15] Rathish Das, John Iacono, and Yakov Nekrich. 2022. External-memory dictionaries with worst-case update cost. [arXiv:2211.06044](https://arxiv.org/abs/2211.06044) [cs.DS]
- [16] Erik D. Demaine, John Iacono, and Stefan Langerman. 2007. Retroactive Data Structures. *ACM Transactions on Algorithms* 3, 2, Article 13 (May 2007), 20 pages. <https://doi.org/10.1145/1240233.1240236>
- [17] P. Dietz and D. Sleator. 1987. Two Algorithms for Maintaining Order in a List. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) (STOC '87). ACM, New York, NY, USA, 365–372. <https://doi.org/10.1145/28395.28434>
- [18] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1989. Making Data Structures Persistent. *J. Comput. System Sci.* 38, 1 (1989), 86–124. [https://doi.org/10.1016/0022-0000\(89\)90034-2](https://doi.org/10.1016/0022-0000(89)90034-2)
- [19] Yoav Giora and Haim Kaplan. 2009. Optimal Dynamic Vertical Ray Shooting in Rectilinear Planar Subdivisions. *ACM Transactions on Algorithms* 5, 3, Article 28 (July 2009), 51 pages. <https://doi.org/10.1145/1541885.1541889>
- [20] Leonidas J. Guibas and Robert Sedgwick. 1978. A Dichromatic Framework for Balanced Trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16–18 October 1978*. IEEE Computer Society, 8–21. <https://doi.org/10.1109/SFCS.1978.3>
- [21] Scott Huddleston and Kurt Mehlhorn. 1982. A New Data Structure for Representing Sorted Lists. *Acta Informatica* 17 (1982), 157–184. <https://doi.org/10.1007/BF00288968>
- [22] Sitarum Lanka and Eric Mays. 1991. Fully Persistent B+-trees. *SIGMOD Records* 20, 2 (April 1991), 426–435. <https://doi.org/10.1145/119995.115861>
- [23] David B. Lomet and Betty Salzberg. 1993. Exploiting A History Database for Backup. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB '93)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 380–390. <https://dl.acm.org/doi/10.5555/645919.672672>
- [24] J. Ian Munro and Yakov Nekrich. 2019. Dynamic Planar Point Location in External Memory. In *35th International Symposium on Computational Geometry, SoCG 2019, June 18–21, 2019, Portland, Oregon, USA (LIPIcs, Vol. 129)*, Gill Barequet and Yusu Wang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 52:1–52:15. <https://doi.org/10.4230/LIPIcs.SocG.2019.52>
- [25] Neil Sarnak and Robert Endre Tarjan. 1986. Planar Point Location Using Persistent Search Trees. *Commun. ACM* 29, 7 (1986), 669–679. <https://doi.org/10.1145/6138.6151>
- [26] Peter J. Varman and Rakesh M. Verma. 1997. An Efficient Multiversion Access Structure. *IEEE Transactions on Knowledge and Data Engineering* 9, 3 (1997), 391–409. <https://doi.org/10.1109/69.599929>
- [27] Dan E. Willard. 1985. New Data Structures for Orthogonal Range Queries. *SIAM J. Comput.* 14, 1 (1985), 232–253. <https://doi.org/10.1137/0214019>

Received 2022-11-07; accepted 2023-02-06

Buffered Partially-Persistent External-Memory Search Trees

Gerth Stølting Brodal  

Aarhus University, Denmark

Casper Moldrup Rysgaard  

Aarhus University, Denmark

Rolf Svenning  

Aarhus University, Denmark

Abstract

We present an optimal partially-persistent external-memory search tree with amortized I/O bounds matching those achieved by the non-persistent B^ε -tree by Brodal and Fagerberg [SODA 2003]. In a partially-persistent data structure each update creates a new version of the data structure, where all past versions can be queried, but only the current version can be updated. All operations should be efficient with respect to the size N_v of the accessed version v . For any parameter $0 < \varepsilon < 1$, our data structure supports insertions and deletions in amortized $\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v)$ I/Os, where B is the external-memory block size. It also supports successor and range reporting queries in amortized $\mathcal{O}(\frac{1}{\varepsilon} \log_B N_v + K/B)$ I/Os, where K is the number of values reported. The space usage of the data structure is linear in the total number of updates. We make the standard and minimal assumption that the internal memory has size $M \geq 2B$. The previous state-of-the-art external-memory partially-persistent search tree by Arge, Danner and Teh [JEA 2003] supports all operations in worst-case $\mathcal{O}(\log_B N_v + K/B)$ I/Os, matching the bounds achieved by the classical B-tree by Bayer and McCreight [Acta Informatica 1972]. Our data structure successfully combines buffering updates with partial persistence. The I/O bounds can also be achieved in the worst-case sense, by slightly modifying our data structure and under the requirement that the memory size $M = \Omega(B^{1-\varepsilon} \log_2(\max_v N_v))$. For updates, where the I/O bound is $o(1)$, we assume that the I/Os are performed evenly spread out among the updates (by performing buffer-overflows incrementally). The worst-case result slightly improves the memory requirement over the previous ephemeral external-memory dictionary by Das, Iacono, and Nekrich (ISAAC 2022), who achieved matching worst-case I/O bounds but required $M = \Omega(B \log_B N)$.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Data structures design and analysis

Keywords and phrases B-tree, buffered updates, partial persistence, external memory

Funding All authors are funded by Independent Research Fund Denmark, grant 9131-00113B.

1 Introduction

Developing data structures for storing a set of values from a totally ordered set subject to insertions, deletions, successor and predecessor queries, and range reporting queries is a fundamental problem in computer science. The classical solution in external-memory is the B-tree by Bayer and McCreight [5] which supports all the operations in worst-case $\mathcal{O}(\log_B N + N/K)$ I/Os, where N is the current size of the set, K is the number of reported values, and B is the external-memory block size. While the B-tree achieves the optimal number of I/Os for queries, for any $0 < \varepsilon < 1$, the B^ε -tree by Brodal and Fagerberg [10] significantly improves update efficiency by attaching buffers to the internal nodes of a B-tree. This design supports updates with amortized $\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N)$ I/Os. The $\frac{1}{\varepsilon} B^{1-\varepsilon}$ factor improvement over traditional B-trees is significant when considering typical parameters of, e.g., $\varepsilon = 1/2$ and $B = 1000$ [4] and the B^ε -tree has found important applications in high-performance industry software such as TokuDB [9] and BetrFS [24].

Although the B^ε -tree optimizes update efficiency, it is *ephemeral*, like most dynamic data structures, meaning that each update overwrites the previous version, and only the current version can be queried. In many applications, maintaining access to past versions is beneficial or even essential. A *persistent* data structure supports such accesses, and in their seminal 1989 paper, Driscoll, Sarnak, Sleator, and Tarjan introduced general techniques for making ephemeral data structures persistent [19]. In particular, a *partially-persistent* data structure supports queries in all past versions of the data structure but only the current version can be updated. Multiple authors have adapted these techniques to the external-memory model, developing partially-persistent B-trees that support updates and queries in worst-case $\mathcal{O}(\log_B N_v + K/B)$ I/Os, where N_v is size of the accessed version v , matching the performance of classical B-trees [3, 6, 32].

In this paper, we present the first buffered partially-persistent external-memory search tree that retains the optimal update and query performance of the (ephemeral) B^ε -tree. Our approach combines buffering techniques, which are essential for efficient updates in external memory, with a geometric view of persistence.

1.1 The External-Memory Model

For problems on massive amounts of data that do not fit in internal memory, the standard model of computation is the I/O-model by Aggarwal and Vitter [1]. In this model, all computation occurs in an internal memory of size M , while an infinite external memory is used for storage. Data is transferred between internal and external memory in blocks of B consecutive elements, with each transfer counting as an I/O. The I/O complexity of an algorithm is defined as the total number of I/Os it performs, and the space usage is the maximum number of external-memory blocks used at any given time. The only operation we allow on stored values are comparisons and we follow the standard assumption that the parameters $B \geq 2$ and $M \geq 2B$. Aggarwal and Vitter proved that the optimal bound for sorting in external memory is $\text{sort}(N) = \Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os [1]. An algorithm is called *cache oblivious* if it is designed without explicit knowledge of B and M but is still analyzed in the I/O model for arbitrary values of these parameters, assuming an optimal offline cache replacement strategy [21]. Some authors make stronger assumptions on the size of the internal memory, such as the *tall-cache assumption* $M \geq B^{1+\delta}$, for some constant $\delta > 0$. For cache-oblivious algorithms, a tall-cache assumption is necessary to achieve optimal comparison-based external-memory sorting [11].

Being considerate of the I/O-behavior of algorithms can be crucial in practice, as demonstrated by Streaming B-trees [8], the generation of massive graphs for the LFR benchmark [23, 26, 27], and the FlashAttention algorithm used in Transformer models [14].

1.2 Interface of a Partially-Persistent Search Tree

A partially-persistent search tree stores an ordered set of values supporting the below interface (in our examples we use integers, but our data structure works for any totally ordered set). Each version is identified by a unique integer version identifier v , with zero being the initial version and the *current* version denoted by v_c . Further, we let $\mathcal{S}_v$ denote the set of values contained in version v , and N_v the size of $\mathcal{S}_v$. Initially $v_c = 0$ and $\mathcal{S}_{v_c} = \emptyset$. Updates (insertions and deletions) can only be performed on the current set $\mathcal{S}_{v_c}$, and any update advances the current version identifier, i.e., each version of the set $\mathcal{S}_v$ only differs from the previous version $\mathcal{S}_{v-1}$ by at most a single value. Queries can be performed on any version.

INSERT(x) Creates $\mathcal{S}_{v_c+1} = \mathcal{S}_{v_c} \cup \{x\}$, increments v_c , and returns v_c .

■ **Table 1** Overview of results on the I/O complexity of ephemeral and partial persistent search trees. Results marked by “am.” hold amortized, and results marked by “rand.” are randomized and hold with high probability. All other results hold in worst case. The parameter ε must satisfy $0 < \varepsilon < 1$. All results assume $M = \Omega(B)$, further $\dagger$ assumes $B = \Omega(\log N)$ and $M = \Omega(\max\{B \log^{\Theta(1)} N, B^2\})$; $\ddagger$ assumes $M = \Omega(B \log B N)$; and $*$ assumes $M = \Omega(B^{1-\varepsilon} \log_2(\max_v N_v))$. For both queries and updates in [7, 15], we include the multiplicative dependency on $\frac{1}{\varepsilon}$ (that can be omitted when treating ε as a constant), allowing, for example, setting $\varepsilon = \frac{1}{\log_2 B}$. All ephemeral results use space linear in N and all partial persistence results use space linear in the total number of updates.

	Range Query	Update
Ephemeral		
Bayer and McCreigh [5]	$\mathcal{O}(\log_B N + K/B)$	$\mathcal{O}(\log_B N)$
Brodal and Fagerberg [10]	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N + K/B)$ am.	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N)$ am.
Bender, Das, Farach-Colton, Johnson, and Kuszmaul [†] [7]	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N + K/B)$	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N)$ rand.
Das, Iacono, and Nekrich [‡] [15]	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N + K/B)$	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N)$
Partial Persistent		
Becker, Gschwind, Ohler, Seeger, and Widmayer [6]	$\mathcal{O}(\log_B N_v + K/B)$	$\mathcal{O}(\log_B N_v)$
Varman and Verma [32]		
Arge, Danner, and Teh [3]		
<i>This paper (Theorem 1)</i>	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N_v + K/B)$ am.	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v)$ am.
<i>This paper (Theorem 2)*</i>	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N_v + K/B)$	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v)$
<i>This paper (Theorem 3)</i>	$\mathcal{O}(\frac{1}{\varepsilon} \log_B N_v + \gamma + K/B)$	$\mathcal{O}(\frac{1}{B^{1-\varepsilon}} (\frac{1}{\varepsilon} \log_B N_v + \gamma))$
	$\gamma = \text{sort}(B^{1-\varepsilon} \log_2 N_v) = \frac{\log_2 N_v}{B^\varepsilon} \log_{M/B} \frac{\log_2 N_v}{B^\varepsilon}$	

DELETE(x) Creates $\mathcal{S}_{v_c+1} = \mathcal{S}_{v_c} \setminus \{x\}$, increments v_c , and returns v_c .

RANGE(v, x, y) Reports all values in $\mathcal{S}_v \cap [x, y]$ in increasing order.

SEARCH(v, x) Returns the successor of x in $\mathcal{S}_v$, i.e., $\min\{y \in \mathcal{S}_v \mid x \leq y\}$.

1.3 Previous Work

In internal memory, the *fat node* and *node copying* techniques can make any ephemeral linked data structure partially-persistent with constant overhead in both time and space, as long as the in-degree of each node in the ephemeral structure is constant [19]. Becker, Gschwind, Ohler, Seeger, and Widmayer [6] and Varman and Verma [32] adapted these techniques to B-trees in external-memory. An elegant application of partial persistence appears in the design of linear space planar point location data structures [31]. In this setting, the underlying set consists of segments which are partially ordered (only a pair of segments intersected by a vertical line can be compared). To adapt this approach to the external-memory setting, Arge, Danner, and Teh strengthened the partially-persistent B-tree to require only a total order on values alive at any given version, leading to a static external-memory point-location structure [3].

A different approach to persistence is to interpret it geometrically (Figure 1), modeling it as a data structure problem on a dynamic set of vertical (or horizontal) segments. Kolovson and Stonebraker explored this perspective [25], though their reliance on R-trees led to poor

performance guarantees [22]. More recently, Brodal, Rysgaard, and Svenning [12] leveraged this geometric approach to develop *fully persistent* B-trees, which allow both queries and modifications to all past versions in $\mathcal{O}(\log_B N_v)$ I/Os. In a fully persistent data structure, updating a version corresponds to cloning it and then applying the modification to the newly cloned version, ensuring that existing versions remain unaffected. Such behavior contrasts with *retroactive data structures* [16], where updates recursively propagate to cloned versions.

Concurrently with the work on persistent data structures in external-memory, there were significant improvements to external-memory data structures by leveraging buffering techniques to always process multiple updates and/or queries together. These include the Buffer Tree by Arge [2] which can form the basis for external-memory sorting, priority queues and batched dynamic algorithms [20] in amortized $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os per operation. For a batched operation the answer might not be immediately returned, which is often sufficient, e.g., in many geometric plane-sweep algorithms where only the end result matters. For standard (non-batched) data structures, a line of work has investigated the update-query trade-off, beginning with the Buffered Repository Tree [13] performing updates in amortized $\mathcal{O}\left(\frac{1}{B} \log_B N\right)$ I/Os and queries in $\mathcal{O}(\log_2 N)$ I/Os. This was later generalized by the B^ε -tree [10] which for $\varepsilon \approx 0$ corresponds to the Buffered Repository Tree and for $\varepsilon \approx 1$ to the standard B-tree. The amortized performance of the B^ε -tree was improved to high-probability [7] and worst-case [15] I/O bounds using stronger assumptions on the size of B and M (see Table 1).

1.4 Contribution

Combining the two lines of research on persistence and buffered data structures has remained an open challenge for the past 20 years, likely due to their seemingly conflicting principles. Persistence requires maintaining access to past versions without affecting their structure, while buffers essentially hold updates to past versions before applying them. Our work demonstrates that that these two ideas can be effectively unified by developing partially-persistent external-memory search trees that achieve bounds matching those of ephemeral B^ε -trees.

► **Theorem 1.** *Given any parameter $0 < \varepsilon < 1$ and $M \geq 2B$, there exist partially-persistent external-memory search trees over any totally ordered set, that support INSERT and DELETE in amortized $\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v\right)$ I/Os, SEARCH in amortized $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N_v\right)$ I/Os, and RANGE in amortized $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N_v + K/B\right)$ I/Os. Here N_v denotes the number of values contained in version v , and K the number of values reported by RANGE. The space usage is linear in the total number of updates.*

The query SEARCH can trivially also answer a member query “ $x \in \mathcal{S}_v$?” by checking if $\text{SEARCH}(v, x)$ returns x . Our data structure can further also support predecessor queries instead of successor queries, as well as strict predecessor and successor queries, i.e., the returned value should be strictly smaller or larger than the query value x . The structure can also handle the case when $\mathcal{S}_0 \neq \emptyset$, where the initial structure can be constructed using $\mathcal{O}(1 + |\mathcal{S}_0|/B)$ I/Os (essentially this is Section 2.6, where a structure is constructed for a given set). Our data structure is stated as maintaining a set of values, but it can easily be extended to support dictionaries storing key-value pairs (each segment in Figure 1 and Figure 2 now stores a key-value pair, where the first axis now are keys; changing the value for key x at version v starts a new vertical segment at (x, v) with the new value).

In Section 3 we describe how to convert the amortized I/O bounds of Theorem 1 to worst-case bounds under the assumption that $M = \Omega(B^{1-\varepsilon} \log_2(\max_v N_v))$ (Theorem 2),

a weaker or equal assumption on the memory size than used in [7] and [15] for high-probability and worst-case bounds for B^ε -trees, respectively. Under the weakest assumption that $M \geq 2B$, we achieve the worst-case bounds in Theorem 3 with an additional term of at most $\mathcal{O}(\text{sort}(B^{1-\varepsilon} \log_2 N_v))$ I/Os, where $B^{1-\varepsilon} \log_2 N_v$ is an upper bound on the number of buffered updates on a root-to-leaf path that should be flushed to the leaf, and $\text{sort}(N) = \Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ denotes the number of I/Os to sort N values [1]. For updates, where the I/O bound is $o(1)$, we assume that the I/Os are performed evenly spread out among the updates.

► **Theorem 2.** *Given any parameter $0 < \varepsilon < 1$ and $M = \Omega(B^{1-\varepsilon} \log_2(\max_v N_v))$, there exist partially-persistent external-memory search trees over any totally ordered set, that support INSERT and DELETE in worst-case $\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v\right)$ I/Os, SEARCH in worst-case $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N_v\right)$ I/Os, and RANGE in worst-case $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N_v + K/B\right)$ I/Os. Here N_v denotes the number of values contained in version v , and K the number of values reported by RANGE. The space usage is linear in the total number of updates.*

► **Theorem 3.** *Given any parameter $0 < \varepsilon < 1$ and $M \geq 2B$, there exist partially-persistent external-memory search trees over any totally ordered set, that support INSERT and DELETE in worst-case $\mathcal{O}\left(\frac{1}{B^{1-\varepsilon}} \left(\frac{1}{\varepsilon} \log_B N_v + \gamma\right)\right)$ I/Os, SEARCH in worst-case $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N_v + \gamma\right)$ I/Os, and RANGE in worst-case $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N_v + \gamma + K/B\right)$ I/Os, where $\gamma = \text{sort}(B^{1-\varepsilon} \log_2 N_v)$. Here N_v denotes the number of values contained in version v , and K the number of values reported by RANGE. The space usage is linear in the total number of updates.*

Note that, for example, when $N_v = 2^{\mathcal{O}(B^\varepsilon)}$ then $B^{1-\varepsilon} \log_2 N_v = \mathcal{O}(B)$ and $\gamma = \mathcal{O}(1)$ and the I/O bounds of Theorem 3 match those of Theorem 2, with only the assumption $M \geq 2B$. This observation can be further strengthened, as when $\gamma = \mathcal{O}\left(\frac{1}{\varepsilon} \log_B N_v\right)$ the bounds I/O match similarly, which holds when $N_v = 2^{B^\varepsilon \left(\frac{M}{B}\right)^{\mathcal{O}\left(\frac{B^\varepsilon}{\varepsilon \log_2 B}\right)}}$.

Outline of Data Structure Previous work on partially-persistent search trees in external memory directly adapted the general pointer-based transformations for persistence [19]. In contrast, our approach embraces the geometric interpretation of partial persistence (see Figure 1) similar to that of [12], where the state of the data structure is embedded in a two-dimensional plane with values on the first axis and versions on the second axis. Under this interpretation, each update corresponds to the start or end of a vertical segment in the plane. Since partial persistence updates are applied to the current version, it always affects the top of the plane. Successor and predecessor queries correspond to horizontal ray shooting to the right and left, respectively, and range queries to reporting the intersections between a horizontal query segment among vertical segments.

To efficiently update and query the geometric view, we partition the plane into rectangles, each containing $\Theta\left(\frac{1}{\varepsilon} B \log_B \bar{N}\right)$ vertical segments in lexicographic order. For now we assume that all versions have size $\Theta(\bar{N})$, for a fixed $\bar{N}$ (this assumption is lifted using global rebuilding, see Section 2.6). In the geometric persistent view, a vertical segment crossing multiple rectangles is split into multiple smaller segments, one for each rectangle, and each smaller segment is inserted into one rectangle.

At a high level, our data structure is divided into two parts. The top part consists of all the open rectangles containing the current set S_{v_c} , which may still be updated. The entry point of this data structure is a B^ε -tree T on the value axis to facilitate buffered updates and to find the relevant rectangle(s) for updates and queries. Since updates are buffered, the geometric view stored in the rectangles may be incomplete, since buffered updates (segment

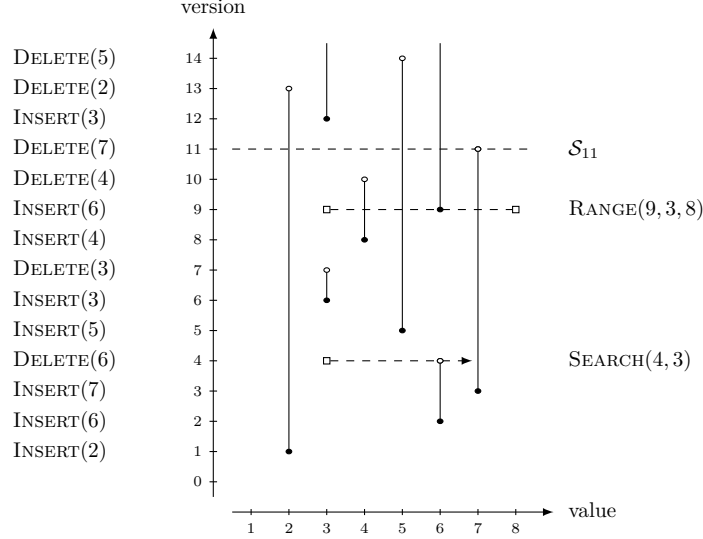


Figure 1 (Left) A list of updates performed on an initially empty set and (Right) the geometric interpretation of the updates. Vertical lines illustrate the interval of versions containing a value. Note that the value 4 is contained in versions $[8, 10[$, i.e., versions 8 and 9, whereas the value 3 is contained in version 6 and versions $[12, \infty[$. The topmost dashed line shows that version 11 of the set is $\mathcal{S}_{11} = \{2, 5, 6\}$, the dashed line segment at version 9 shows that the result of the query $\text{RANGE}(9, 3, 8)$ is $\{4, 5, 6, 7\}$, and the bottommost dashed arrow shows that the result of the successor search $\text{SEARCH}(4, 3)$ is 7.

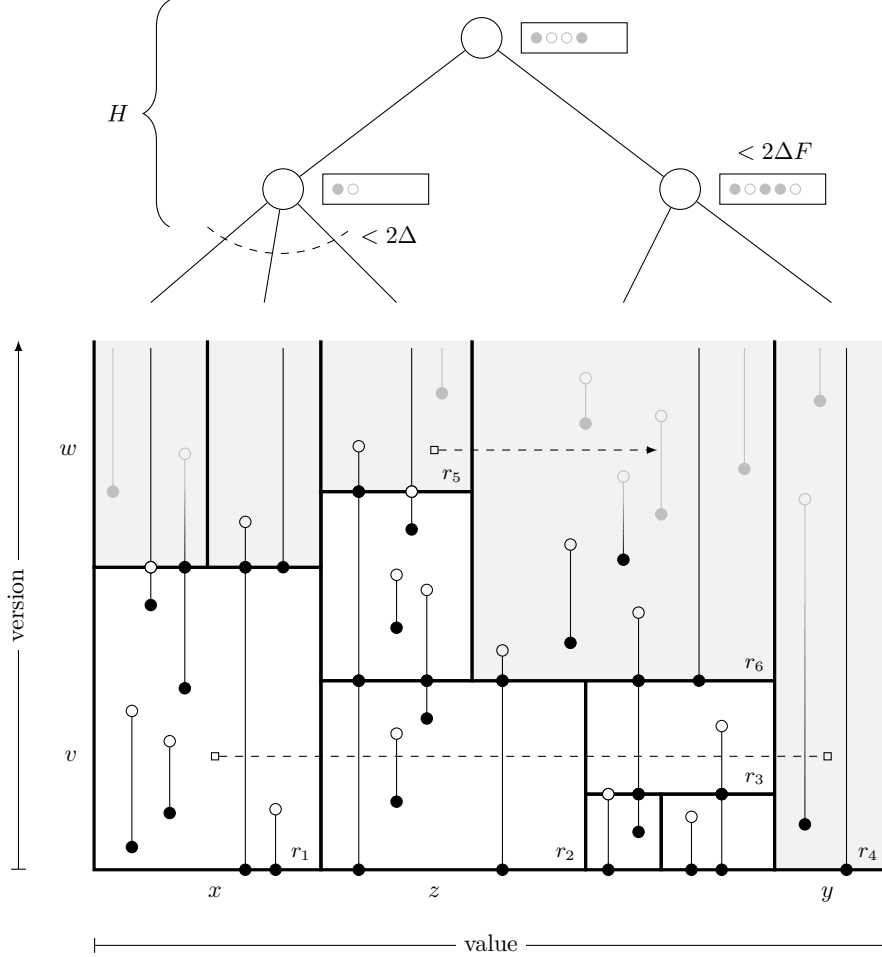
endpoints) will first be added to the rectangle when buffers are flushed. The bottom part consists of all the finalized rectangles, i.e., rectangles which can be queried but not updated. The entry point to the bottom part is a data structure P to find the relevant rectangle(s) for a query. This corresponds exactly to a point location problem and we implement P as an external-memory adaption of the classical planar point location solution using partial persistence [3, 31], more specifically a B-tree with path copying during updates.

2 The Buffered Persistent Data Structure

In this section, we describe our partially-persistent B^ε -tree structure. Versions are identified by the integers $0, 1, 2, \dots$, where v_c denotes the identifier of the current version. We let $\mathcal{S}_v$ denote the set at version v , where values are from some totally ordered set. The initial set $\mathcal{S}_0 = \emptyset$, and $\mathcal{S}_{v+1} = \mathcal{S}_v \cup \{x\}$ if the $v+1$ 'th update is $\text{INSERT}(x)$, and $\mathcal{S}_{v+1} = \mathcal{S}_v \setminus \{x\}$ if the $v+1$ 'th update is $\text{DELETE}(x)$. Note that $\mathcal{S}_{v+1} = \mathcal{S}_v$ if the $(v+1)$ 'th update inserts a value already in $\mathcal{S}_v$ or is deleting a value not in $\mathcal{S}_v$.

2.1 Geometric Interpretation of Partial Persistence

The problem has a natural geometric interpretation in a two dimensional space, with the first dimension representing the values and the second dimension representing the versions, see Figure 1. On this two dimensional plane, a value x existing in versions $[v, w[$, can be represented by the vertical line segment $\{x\} \times [v, w[$, i.e., x is inserted in version v and deleted in version w . If $x \in \mathcal{S}_{v_c}$, then $w = +\infty$ (x has not been deleted yet).



■ **Figure 2** (Top) A B^ε -tree of the open rectangles and (Bottom) the geometric interpretation of the updates, split into multiple rectangles, where gray rectangles are open rectangles. The black endpoints represent updates present in the rectangle and the gray endpoints represent buffered updates present in the buffers at the internal nodes of the B^ε -tree. A query $\text{RANGE}(v, x, y)$ is represented as the dashed line between two square endpoints, spanning rectangles r_1 , r_2 , r_3 , and r_4 , and a successor query $\text{SEARCH}(w, z)$ is represented as the dashed arrow from a square endpoint, spanning rectangles r_5 and r_6 . Black dots on vertical segments correspond to the upper endpoint of the segment in the rectangle below and the lower endpoint of the segment in the rectangle above.

2.2 Partitioning the Plane into Rectangles

We consider the sequence of versions partitioned into intervals $[v_0, v_1[, \dots, [v_{k-1}, v_k[, [v_k, \infty[$, for some versions $0 = v_0 < v_1 < \dots < v_k \leq v_c$. In Section 2.6 we show how to maintain the version intervals. In the following we consider the interval $[\bar{v}, \infty[$ containing the current version v_c of the set. Let $\bar{N} = |\mathcal{S}_{\bar{v}}|$. We allow up to $c \cdot \bar{N}$ partial persistent updates during this interval of versions for a constant $0 < c < 1$, i.e., all versions $v, \bar{v} \leq v \leq v_c$, satisfy $(1 - c) \cdot \bar{N} \leq |\mathcal{S}_v| \leq (1 + c) \cdot \bar{N}$.

Our data structure is built around four central parameters:

$$\Delta = \lceil B^\varepsilon \rceil \quad H = 1 + \lceil \log_\Delta \bar{N} \rceil \quad F = \lceil B^{1-\varepsilon} \rceil \quad R = H \cdot 2\Delta \cdot F$$

The basic idea is to have a B^ε -tree T of degree at most 2Δ (and degree at least Δ , if only

insertions can be performed, and degree at least 1 if deletions are allowed), where leaves (open rectangles) store between $4R$ and $10R$ updates (see Section 2.4) and all leaves are at the same layer. In Section 2.5 we prove that H is an upper bound on the height of T (number of nodes on a root-to-leaf path, excluding the leaves). Each internal node of T will have a buffer of at most $2\Delta F = \Theta(B)$ updates yet to be applied to the leaves of the subtree rooted at the node. Note that R is an upper bound on the total number of buffered updates along a root-to-leaf path in T . The essential property of the parameters is that $R/B = \Theta(H) = \Theta(\frac{1}{\epsilon} \log_B \bar{N})$.

The geometric plane defined in Section 2.1 is partitioned into a number of axis aligned rectangles $[x, y] \times [v, w]$, such that the number of updates in each rectangle is $\Theta(R)$. For each rectangle we store a list of the updates in the rectangle in lexicographical order by first the value and secondly the version of that update. Note that this groups equal values consecutively in the list. To allow efficiently locating a rectangle for a given version and value, we store a list indexed by version identifier, where we for each version v store a pointer to the root of a B-tree P_v over the rectangles left-to-right containing $\mathcal{S}_v$ (see Section 2.4). Further, we require that each rectangle contains $\Omega(R)$ values which are present in all versions the rectangle spans. We denote such a value as *spanning*. If $\bar{N} = \mathcal{O}(R)$, all updates are stored in a single list.

New updates are buffered, to achieve I/O efficient update bounds. The topmost rectangles, which cover the current version v_c , are all *open*, with all other rectangles being *closed*. Crucially, new updates are always performed in the current version. We maintain the invariant that for a buffered update, i.e., an update that has not yet been flushed to the corresponding rectangle, the corresponding rectangle must be open.

For the open rectangles, we store a B^ϵ -tree T , such that recent updates to the open rectangles are buffered. We let the maximum degree of an internal node in T be 2Δ . Each internal node of T contains a buffer of up to $2\Delta F$ updates, sorted lexicographically by value and version. Additionally, each update stores if the update is an insertion or deletion. Consider a full buffer, i.e., it contains at least $2\Delta F$ updates, where each update should be *flushed* to one of the at most 2Δ children. Then, there must exist a subset of size at least F updates, which should be flushed to the same child.

The setup is illustrated on Figure 2. The vertical black and gray lines represent the version intervals containing a value. The black lines represent updates present in the list of updates contained in that rectangle, while the gray lines and endpoints represent updates contained in buffers of T , which are illustrated at the top of the figure.

2.3 Handling Queries and Updates

When performing $\text{SEARCH}(v, x)$, first the rectangle r covering point (x, v) in the plane must be found. By using the B-tree P_v associated with version v , r can be found using $\mathcal{O}(H)$ I/Os. If r is closed, then all updates inside r are contained in the sorted list of updates stored in r , and these can be scanned in $\mathcal{O}(R/B)$ I/Os. If r is open, then the result of the successor query may be affected by buffered updates, which are not stored in r . The rectangle r must therefore be *actualized*, by merging all updates in buffers on the path from the root to r with the updates in r . The details of this operation is described in Section 2.4, where the actualize operation is shown to have an amortized $\mathcal{O}(H)$ I/Os. After r is actualized, the query continues by scanning the updates of r . If the result of the successor query is not contained in the rectangle r , then by the spanning requirement, the result of the query must be in the neighboring rectangle to the right, that similarly is actualized if it is open. In total, the operation spends amortized $\mathcal{O}(H + R/B) = \mathcal{O}(\frac{1}{\epsilon} \log_B \bar{N})$ I/Os. Note that the operation

easily can be modified to support member, predecessor, and strict predecessor or successor queries.

A $\text{RANGE}(v, x, y)$ query is performed very similar to a SEARCH query. The query may however touch more than two rectangles. Note that for the at most two rectangles containing the endpoints of the query we do not necessarily report all values they contain at version v . These rectangles can be found using amortized $\mathcal{O}(H + R/B)$ I/Os by the argument above. For each intermediate rectangle accessed (and possibly actualized if it is open), then by the spanning requirement, a constant fraction of the updates scanned result in reported values. Since accessing a rectangle takes amortized $\mathcal{O}(H + R/B)$ I/Os, and each rectangle contains $\Theta(R) = \Theta(B \cdot H)$ values at version v , then amortized $\mathcal{O}(1/B)$ I/Os are spent for each value reported for an intermediate rectangle. In total, a RANGE query reporting K values takes amortized $\mathcal{O}(H + R/B + K/B) = \mathcal{O}(\frac{1}{\varepsilon} \log_B \bar{N} + K/B)$ I/Os.

Each update, either an INSERT or DELETE , is applied to the current version v_c of the set. The B^ε -tree T contains all buffered updates to the open rectangles, which cover the current version v_c . For an update operation, a tuple with the update and v_c is added to the root buffer of T , which is stored in internal memory. In Section 2.4 it is shown that adding the update to the root buffer and handling possible *buffer overflows* takes amortized $\mathcal{O}(H/F) = \mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B \bar{N})$ I/Os.

2.4 Flushing Buffers

To argue about the amortized cost of flushing the content of buffers down the tree T , we let the potential of each buffered update be $1/F$ multiplied by the height of the buffer the update is stored in, with the root buffer being at the largest height. One unit of released potential can cover $\mathcal{O}(1)$ I/Os. When adding an update to the tree, the root buffer is always stored in internal memory, and therefore no I/Os are needed to access it. However, the potential is increased by at most $1/F \cdot H$, and the operation therefore uses amortized $\mathcal{O}(H/F)$ I/Os.

Buffer Overflows. Each buffer at an internal node of T contains at most $2\Delta F$ updates. If a buffer contains more than $2\Delta F$ updates, then a *buffer overflow* is performed. Since each node of T has at most $2\Delta - 1$ children, at least F updates from the buffer must be to the same child. These updates can be moved together to the buffer of that child.

A buffer overflow can happen in two cases. Either when an update is placed into the root buffer as the result of an update operation, or when updates are placed into a buffer because the parent buffer is overflowing. Moving exactly F updates out of a buffer, is always sufficient to make an overflowing buffer non-overflowing again. A buffer overflow therefore only moves down a single path of T .

An overflowing buffer can be stored in $\mathcal{O}(1)$ blocks, since $2\Delta F + F = \mathcal{O}(B)$, and therefore the F updates to remove can be found in $\mathcal{O}(1)$ I/Os. If the overflowing updates are moved to a child buffer, these can be inserted via a merge in $\mathcal{O}(1)$ I/Os. As F updates are moved one layer down the tree, then the potential decreases by 1, which is enough to cover the $\mathcal{O}(1)$ I/O cost of the overflow operation.

If the child is not an internal node of T , but an open rectangle, then merging the F overflowing updates into the list of updates in the rectangle takes $\mathcal{O}(R/B) = \mathcal{O}(H)$ I/Os. As buffer overflows only move down a single path of the tree, then $\Omega(H)$ overflows must have occurred before the overflow reaches the open rectangle. Merging the overflow into the list of updates in the open rectangle therefore does not increase the asymptotic amortized number of I/Os performed.

Actualizing. When *actualizing* an open rectangle, all buffered updates to that open rectangle must be moved into the rectangle. Note that all relevant updates are in the buffers on the root-to-leaf path in T to the open rectangle. Each of these buffers contains at most $2\Delta F = \mathcal{O}(B)$ buffered updates. The total number of buffered updates on the path is at most $H \cdot 2\Delta \cdot F = R$. For each node on the path from the root down to the rectangle the following is done. Let U be the updates on the path from the layers above in sorted order. Initially U is empty. To extend U for each layer top-down, U is merged with the relevant updates of the next buffer. This requires $\mathcal{O}(1 + |U|/B)$ I/Os, by scanning U and the buffer. Since the U updates are moved one layer down, they release potential $|U|/F \geq |U|/B$, that can cover $\Theta(|U|/B)$ I/Os, i.e., the amortized cost for actualizing one level of the tree is $\mathcal{O}(1)$ I/Os. As there are $\mathcal{O}(H)$ layers of the tree, the at most R relevant updates can be found in sorted order in amortized $\mathcal{O}(H)$ I/Os. They can then be merged with the updates in the open rectangle in $\mathcal{O}(R/B)$ I/Os. In total, the actualize operation requires amortized $\mathcal{O}(H + R/B) = \mathcal{O}(H)$ I/Os.

Finalizing. Each open rectangle is allowed to receive between R and $2R$ updates before it is *finalized*, converting it into a closed rectangle. When finalizing an open rectangle, all buffered updates to the rectangle are removed from T and merged with the rectangle, to ensure that all buffered updates in T are only to open rectangles. We will now argue that open rectangles receive at most $2R$ updates in total by finalizing the rectangle as soon as R updates have been added to it. A finalize operation can be triggered from an actualize operation or from a buffer overflow. Note that in both cases, the number of updates in the rectangle before the operation is at most $R - 1$.

An actualize operation may add at most R buffered updates to a rectangle, i.e., at most $2R - 1$ total updates are placed in a finalized rectangle. If the rectangle receives an update from a buffer overflow, then the overflow must have been triggered by an update in the root buffer. Buffered updates to add to the rectangle can only be the R updates in buffers on the path, and the new update, which in total is at most $(R - 1) + R + 1 = 2R$ updates to add to the open rectangle. Thus, by finalizing a rectangle as soon as it receives at least R updates, it will contain between R and $2R$ updates.

Spanning Requirement. We require that the first version of a rectangle contains $[4R, 8R[$ values. When finalizing the rectangle, $[R, 2R]$ updates have been performed and therefore at least $2R$ of the initial values are still present, that is, span all versions of the rectangle. This ensures that the $\Omega(R)$ spanning values requirement is met.

When finalizing a rectangle, it holds that $[2R, 10R[$ values are contained in the rectangle at version v_c . New open rectangles must be created to span the value range of the closed rectangle, where the values present at version v_c are contained. If the count is in $[4R, 8R[$, then a single rectangle suffices. If $[8R, 10R[$ values are present, then the range is *split* in two rectangles at the median value, both with $[4R, 5R[$ values, and T must be updated as described below. Otherwise, version v_c of the rectangle contains $[2R, 4R[$ values. A sibling rectangle is finalized, to allow for a *merge* of the rectangles to occur. Note that the early finalizing of the sibling preserves the $\Omega(R)$ spanning values requirement of the sibling. The combined present values is then $[4R, 14R[$. A split may need to be performed, i.e., the result is one or two new open rectangles.

Updating the B^ε -Tree T . After finalizing open rectangles, the B^ε -tree must be updated accordingly. If an open rectangle was split, then a new child is added to the parent node

in T of the updated rectangle. If this increases the degree of the node to 2Δ , it is split by distributing its children into two new nodes, each with degree Δ . Its buffer is also partitioned so that each buffered update is placed in the buffer containing its relevant child. Splitting the node further introduces a new child in the parent of the node. Note that this may cascade up the tree, but only on the path towards the root. If a merge of the rectangle occurred, then a child is deleted from the parent. When merging rectangles, the merged rectangles must be siblings in the tree. The rectangle is merged with the left or right neighbor rectangle, which has a closest lowest common ancestor with the rectangle in T , to ensure that the value range of existing nodes only increase. This may cause the degree of nodes to be below Δ . Notably, we do not merge internal nodes of T , as this could create a large buffer that requires multiple flushes in different directions, known as *flushing cascades* [7]. We instead allow nodes to have a degree down to 1, where deleting the last child results in deleting the path of consecutive degree-one from the child towards the root. As we show in Section 2.5, this does not affect the asymptotic height of the tree.

When finalizing a rectangle, only the path from the finalized rectangle to the root may be affected. We therefore create the new tree T via *path copying*, which preserves the old T . The buffers of the copied nodes are moved, such that all buffers are present in the tree for the current version. We maintain a list indexed by version identifier, that for any version v stores a pointer to the root of T for version v , i.e., the required tree P_v .

Updating the rectangles and the B^ε -tree T upon finalizing therefore requires scanning $\mathcal{O}(R)$ values and traversing a constant number of paths of length $\mathcal{O}(H)$ in T , which takes $\mathcal{O}(R/B + H) = \mathcal{O}(R/B)$ I/Os. As $\Omega(R)$ updates must be applied to a rectangle before finalizing it, this does not increase the asymptotic amortized cost of an update operation. Queries may also finalize rectangles, but already require amortized $\mathcal{O}(H)$ I/Os, causing no asymptotic query overhead.

Space Usage. When finalizing a rectangle, $\Omega(R)$ updates must have occurred in that rectangle. New rectangles are then created, which in total copies $\mathcal{O}(R)$ updates, and one path of the tree is copied. As the height of the tree is at most $H = \mathcal{O}(R/B)$, and the updates of the rectangles are stored in lists, the newly allocated space is $\mathcal{O}(R/B)$, which can be amortized over the $\Omega(R)$ updates required for the finalization to happen. In addition to the updates, initially $\bar{N}$ values are stored across $\mathcal{O}(\bar{N}/R)$ rectangles in lists, and a balanced initial B^ε -tree is built on these initial rectangles, causing an initial space of $\mathcal{O}(\bar{N}/B)$ blocks. As the structure allows for at most $c \cdot \bar{N}$ updates, the space usage is therefore in total $\mathcal{O}(\bar{N}/B)$ blocks.

2.5 Bounding the Tree Height

In this section we show that H is an upper bound on the height of the B^ε -tree T .

We define the *weight* w_i of a node at height i in T to be the number of updates on values in the value range of the node. The rectangles are at height 0 of the tree, with the nodes of the tree starting at height 1. The updates are both the $\bar{N}$ initial values as well as the up to $c \cdot \bar{N}$ additional updates. It holds that the weight of a node is the sum of the weights of its children. By induction on the number of updates we show that $w_i \geq B\Delta^i$ for all nodes at all heights, except for the root. First note that the inequality holds for $i = 0$, as any rectangle contains at least $R \geq B$ updates when it was created. Initially, $\bar{N}$ updates are distributed into at most $\bar{N}/R$ rectangles, where the number of updates in each rectangle is at least $R \geq B$. Each internal node initially has degree $[\Delta, 2\Delta]$, except for the root that has degree $[2, 2\Delta]$. By induction on the tree height i , it holds that the initial tree

satisfies $w_i \geq B\Delta^i$, except for the root. Each update affects some path of the tree. If the tree is not updated, then the weights of the nodes on the path can only grow, and therefore the inequality holds. If rectangles are merged, then one rectangle disappears together with all the ancestors having only this single rectangle as a leaf. The other rectangle and its ancestors up to the least common ancestor of the two merged leaves get their value ranges expanded. It follows that the surviving nodes of a merge only can have their value range increase, and therefore the inequality holds. If a split occurs in any node at height i , then the degree of the node before the split is 2Δ . The node is split in two nodes at height i , each with Δ children. The weight of each of the two nodes is therefore at least $\Delta \cdot w_{i-1} \geq \Delta \cdot B\Delta^{i-1} = B\Delta^i$. It therefore holds that $w_i \geq B\Delta^i$.

Since the number of updates is at most $(1+c) \cdot \bar{N}$, we have $B\Delta^i \leq (1+c) \cdot \bar{N}$ for all nodes at height i , except for the root. Since by definition $2 \leq \Delta \leq B$ and $c < 1$, we have $\Delta^{i+1} \leq 2\bar{N}$, i.e., $i \leq \log_\Delta(2\bar{N}) - 1 \leq \log_\Delta \bar{N}$. The height of T is then at most the largest value of i satisfying this inequality, plus one for the root, i.e., the height of T is at most $1 + \log_\Delta \bar{N} \leq 1 + \lceil \log_\Delta \bar{N} \rceil = H$.

2.6 Global Rebuilding

The data structure above allows for an initial set of $\bar{N}$ values to receive up to $c \cdot \bar{N}$ persistent updates, for a constant $0 < c < 1$. For any version v , we have $(1-c) \cdot \bar{N} \leq N_v \leq (1+c) \cdot \bar{N}$, i.e., $N_v = \Theta(\bar{N})$. Therefore, the asymptotic costs of all operations also hold with $\bar{N}$ replaced by N_v .

To allow for more than $c \cdot \bar{N}$ updates, we create multiple copies of the data structure above using global rebuilding [28, 29]. Whenever the current data structure reaches $c \cdot \bar{N}$ updates, a new data structure is created with initial set $\mathcal{S}_{v_c}$ and $\bar{N}_{\text{new}} = |\mathcal{S}_{v_c}|$ (and new H and R parameters), with a new set of rectangles and a new B^ε -tree T , where all buffers are empty. We compute $\mathcal{S}_{v_c}$ by performing $\text{RANGE}(v_c, -\infty, \infty)$ in amortized $\mathcal{O}(\frac{1}{\varepsilon} \log_B \bar{N} + K/B) = \mathcal{O}(\bar{N}/B)$ I/Os. The new data structure can be build using $\mathcal{O}(\bar{N}/B)$ I/Os by a single scan of the sorted list containing $\mathcal{S}_{v_c}$.

In the old data structure $c \cdot \bar{N}$ updates have been performed before this rebuild is performed. By amortizing the rebuild cost over these updates, the amortized cost of each update is increased by $\mathcal{O}(1/B)$ I/Os, i.e., the asymptotic amortized cost of an update is not increased. As the space usage of the new data structure is $\mathcal{O}(\bar{N}/B)$ blocks, a similar argument can be used to amortize the space usage over the updates, maintaining a linear space usage in the total number of updates. This concludes the proof of Theorem 1.

3 Worst-Case Bounds

In this section, we describe how to achieve worst-case I/O guarantees instead of amortized under progressively weaker assumptions on the internal memory size M . Previous approaches to improving the amortized performance of ephemeral B^ε -trees, both in the randomized [7] and worst-case [15] setting, assumed at least that $M = \Omega(\frac{1}{\varepsilon} B \log_B \bar{N}) = \Omega(HB)$, which allows all buffers on a path to be sorted in internal memory, i.e., $\mathcal{O}(\text{sort}(HB)) = \mathcal{O}(H)$ I/Os. First, in Section 3.1, we show that if $M = \Omega(HB)$, our persistent structure can be deamortized without asymptotic overhead. Then, in Section 3.2, we relax the assumption to $M = \Omega(B^{1-\varepsilon} \log_2 \bar{N})$ using the *subtracting game* studied by Dietz and Raman [17]. This represents a factor $B^\varepsilon / \log_2 B$ improvement on the assumption for the size of the internal memory. Finally, in Section 3.3, we show worst-case results when only assuming $M \geq 2B$,

which introduces a small additive overhead on all operations. We employ *the zeroing game* by Dietz and Sleator [18, Theorem 5] to avoid a multiplicative overhead for RANGE queries.

3.1 Large Internal Memory Assumption

We first consider the case when $M = \Omega(HB)$. When actualizing a rectangle (see Section 2.3), the buffers at the $\mathcal{O}(H)$ nodes along the root-to-leaf path, with a total size of $\mathcal{O}(HB)$, are merged to produce a sorted list of updates to apply to the rectangle. As shown in Section 2.4, this can be done in amortized $\mathcal{O}(H)$ I/Os, by merging the buffers top-down. In the worst case, this requires $\mathcal{O}(H^2)$ I/Os. By instead merging the buffers using an external memory sorting algorithm, the worst-case number of I/Os can be improved to $\mathcal{O}(\text{sort}(HB))$. Previous approaches to improving the amortized performance of B^ε -trees in the randomized [7] and worst-case [15] settings both assumed at least that $M = \Omega(HB)$, in which case the sorting term trivially disappears by performing the sorting internally after reading the H buffers into internal memory. The remaining challenge was handling flushing cascades, which occur when merging internal nodes of T results in large buffers requiring many flushes in different directions. For our structure, we avoid this issue by never merging internal nodes, and instead maintain the height of T using global rebuilding. For the remainder of this section, we assume a large internal memory of size $M = \Omega(HB)$ and describe how to achieve worst-case guarantees by incrementally performing amortized work.

Queries. Finding and actualizing a relevant rectangle for a query takes $\mathcal{O}(H + \text{sort}(HB)) = \mathcal{O}(H)$ I/Os when $M = \Omega(HB)$. The worst case for a SEARCH and RANGE query is therefore $\mathcal{O}(H)$ and $\mathcal{O}((1 + K/R)H) = \mathcal{O}(H + K/B)$ I/Os, respectively. Note that for a RANGE query, for each rectangle that intersects the query, except for the leftmost and rightmost ones, $\Omega(R)$ values are reported due to the $\Omega(R)$ spanning values in each rectangle.

Updates. When performing an update, it may be the $\lceil c \cdot \bar{N} \rceil$ 'th update, which triggers a global rebuild of the structure based on a new $\bar{N}$, which uses $\mathcal{O}(\bar{N}/B)$ I/Os, as described in Section 2.6. However, by performing the global rebuilding incrementally [28, 29] over the next $\Theta(\bar{N})$ updates, this does not increase the asymptotic worst-case number of I/Os of each update. While initializing the new structure there are still updates happening which must then be applied before it can take over. By performing updates to the new structure at a sufficiently fast rate compared to the live structure this ensures that they stay within a constant factor of each other in size until the new structure takes over. Therefore, only the I/O cost of an update without global rebuilding needs to be considered.

Updates are inserted in the root buffer of the B^ε -tree, as described in Section 2.4. By keeping the root buffer in internal memory this uses no I/Os. If the root buffer overflows, it may cause buffer overflows along a root-to-leaf path down to an open rectangle, which may then be finalized by performing an actualize operation followed by a path copy. The update therefore requires $\mathcal{O}(H)$ I/Os in total under the large internal memory assumption. However, each time the root buffer overflows, F updates are removed from it, meaning this occurs at most every F th update. Thus, when the root buffer overflows, we incrementally apply the update to the structure over the next F updates, ensuring that $\mathcal{O}(H/F)$ I/Os are performed per update in the worst case. To not interfere with the incremental work, we place new updates in a separate buffer while it is in progress and merge them with the root buffer when it is finished. If a path copy has occurred, the root pointers of the F most recent versions must be updated to the new root. Since they are stored together in an array indexed by their version identifier this takes $\mathcal{O}(1)$ I/Os. If a query occurs while an update is

being performed incrementally, we complete the update before executing the query. This does not increase the asymptotic worst-case number of I/Os for queries.

3.2 Smaller Buffers on All Paths Using the Subtraction Game

In the previous section, we described how as long as the internal memory can hold all values on a root-to-leaf path towards the same open rectangle, there is no overhead on worst-case queries and updates compared to the amortized bounds. To lower the possible number of such values, we will slightly change the flushing strategy described in Section 2.4 where we only performed a flush when a buffer overflowed. Instead, for every F 'th update, we flush along an entire root-to-leaf path, always flushing towards the child where most of the updates are going. We still flush at most F values, which preserves the property that internal nodes of T contain at most $2\Delta F$ updates. In the following, we show that this flushing strategy guarantees that all buffers contain $\mathcal{O}(F \log_2 \Delta)$ updates going towards the same child, and therefore also the same leaf. This implies that the assumption $M = \Omega(HF \log_2 \Delta) = \Omega(B^{1-\varepsilon} \log_2 \bar{N})$ is sufficient to achieve no overhead for worst-case queries and updates. This is a factor $\Theta(B^\varepsilon / \log_2 B)$ improvement over the previous smallest assumption on M [15].

We can view each node as playing the *subtracting game* studied by Dietz and Raman [17] for the number of updates $x_1, x_2, x_3, \dots, x_{2\Delta}$ going towards each of its at most 2Δ children. In particular, when at most F updates are flushed towards a node, if there are δ_i new updates going towards the i th child, then variable x_i is increased by δ_i . Then we flush towards the child j where most of the updates are going which sets the variable $x_j = \max\{x_j - F, 0\}$. Following [17, Theorem 3] and Theorem 4 in the Appendix, scaled by a factor of F , this guarantees $x_i = \mathcal{O}(F \log_2 \Delta)$ for any i .

We also need to consider how merging and splitting in T impacts the games. Only leaves of T , corresponding to open rectangles, are merged. When an internal node is split, it corresponds to evenly distributing the x_i variables from one game to two new games, except for one variable that is split into two new variables, each with a smaller or equal value. When a leaf, i.e. an open rectangle, is merged or split, the one or two rectangles involved are first actualized, which sets their variables to zero, a stronger operation than subtracting. Thus, the variable for a new rectangle is always zero and variables on root-to-leaf paths to actualized rectangles may be decremented. In all cases and for all games, variables are either decremented without adding to the game, or a copy of an existing game is created, where all variables in the copy are equal or smaller in value than before. This concludes the proof of Theorem 2.

3.3 Improving Worst-Case RANGE Queries Using the Zeroing Game

In this section, we consider the small-memory setting with $M \geq 2B$, to overcome the theoretical limitation of the memory assumptions made in Sections 3.1 and 3.2. Actualizing a rectangle by merging the relevant updates on a root-to-leaf path to a rectangle requires $\mathcal{O}(H + \gamma)$ total I/Os, where $\gamma = \text{sort}(B^{1-\varepsilon} \log_2 \bar{N})$. The construction from the previous section directly results in worst-case SEARCH queries and updates in $\mathcal{O}(H + \gamma)$ and $\mathcal{O}(\frac{1}{F}(H + \gamma))$ I/Os, respectively. However, since RANGE queries are performed by repeatedly searching for the $\Theta(1 + K/R)$ rectangles intersecting the query, the worst-case number of I/Os is $\Theta((1 + K/R)(H + \gamma)) = \Theta\left(H + \gamma + \frac{K}{B} \left(1 + \frac{\varepsilon \log_2 B}{B^\varepsilon} \log_{M/B}(B^{-\varepsilon} \log_2 \bar{N})\right)\right)$, notably with a multiplicative non-constant overhead on the reporting term. In this section, we describe how to guarantee RANGE queries in worst-case $\Theta(H + \gamma + \frac{K}{B})$ I/Os when $M \geq 2B$.

The worst-case I/O cost of a RANGE query can be improved by merging all the buffered updates to the open rectangles intersecting the query in a top-down, layer-by-layer fashion. That is, by essentially actualizing all the open rectangles intersected by the RANGE query simultaneously. We denote the updates already present in the rectangles by the *partial output list*. Rather than applying the buffered updates to the rectangles, we merge them with the partial output list to obtain the final output. A given query $\text{RANGE}(v, x, y)$ reports $\Omega(R)$ values from each intermediate rectangles, i.e., all rectangles intersecting the query except for the two that contain the endpoints x and y . Thus, in $\mathcal{O}((1 + K/R)H + (R + K)/B) = \mathcal{O}(H + K/B)$ I/Os we can find all the relevant rectangles and compute the partial output list. To collect the buffered updates in T for the intermediate open rectangles in sorted order, we merge the updates down layer by layer. We only move down the updates to versions earlier or equal to v since only these can affect the query result. Once obtained, these updates are merged with the partial output list using linear I/Os to report the output of the RANGE query.

The buffered updates are stored in T , which contains the open rectangles at version v_c , however, the RANGE query is on the rectangles present at version v . Let T_v denote the B^ε -tree on open rectangles at version v , i.e., the state of T when version v was created. From T_v to T the tree may have changed, but no later updates are relevant for the query. Thus, the total number of relevant updates does not increase from T_v to T , and each update remains on the root-to-leaf path towards the open rectangle to which the update is relevant. The relevant updates in T can be collected in sorted lists ordered by layer by traversing each root-to-leaf path in T towards open rectangles intersecting the query using $\mathcal{O}((1 + K/R)H)$ I/Os. To bound the I/Os to move the updates down layer by layer, we show that the total number of updates is $\mathcal{O}(B^{1-\varepsilon} \log_2 \bar{N} + K/H)$. To this end, we need the additional invariant that all nodes of degree one have empty buffers, which we show how to obtain below. Let T'_v be the subtree of T_v consisting of all nodes on root-to-leaf paths to rectangles intersecting the query. Then split T'_v into two root-to-leaf paths p_x and p_y to x and y , respectively, along with all the subtrees hanging off p_x or p_y . For a node on p_x (symmetrically p_y) of degree at least two there may be one or more subtrees T_{sub} hanging off the node. Since only the nodes of T_{sub} with degree at least two have non-empty buffers, if T_{sub} has ℓ leaves, the number of buffered updates in T_{sub} is at most $\mathcal{O}((\ell - 1)B)$. Thus, the number of buffered updates in T'_v excluding degree one nodes on p_x and p_y is $\mathcal{O}(K/H)$. A degree one node on p_x and p_y may have a large degree in T_v , but since it only has one child in the direction of the query, due to the subtracting game, it stores at most $\mathcal{O}(F \log_2 \Delta)$ relevant updates. The number of degree one nodes on p_x and p_y is at most $2H$, and they together contribute $\mathcal{O}(B^{1-\varepsilon} \log_2 \bar{N})$ buffered updates, which we locate and sort separately using $\mathcal{O}(H + \gamma)$ I/Os. For the remaining $\mathcal{O}(K/H)$ buffered updates, we merge them layer by layer using $\mathcal{O}(H + K/B)$ I/Os. In total, the worst-case number of I/Os to perform a RANGE query is $\mathcal{O}(H + \gamma + K/B)$ I/Os.

Empty Buffers for Degree one Nodes. To ensure that each node of degree one has an empty buffer, we alter the buffer capacity of nodes to scale with the degree. Let the capacity of the buffer of a node with degree $d \geq 2$ be at most $F \cdot \min\{2d, 2\Delta\}$, with nodes of degree one having a buffer capacity of 0. When flushing a node, as the maximum number of updates in the buffer scales with the degree, then at least F updates going to the same child can be found when overflowing. When splitting a node, it must have degree 2Δ , resulting in the two new nodes having degree Δ , which therefore does not decrease the buffer capacity, and flushing is not needed. When a child of a node is removed due to merging rectangles, the degree of the node is decreased by one. If the degree remains at least two, at most two flushes

are required to get the buffer capacity within bounds. Otherwise, if the degree drops from two to one, at most four flushes are needed. To avoid cascading merges of rectangles, we do not finalize a rectangle once it receives a certain number of updates. Instead, we finalize the rectangle that has received the most updates, provided it has received at least R updates. This last condition ensures a bound on the space usage. Including the initial flush from the root buffer, then when a rectangle is finalized, at most $5F$ updates have been flushed into open rectangles.

Let U_i denote the number of updates to the i th rectangle, excluding the initial insertions. We extend the data structure to include an array over all open rectangles, where index j stores a blocked-linked-list of all rectangles where the number of updates is $U_i = j$. Each rectangle has a double linked pointer between its location in the array of lists and the rectangle. This allows for moving a rectangle to a new entry in the array, when it receives updates, as well as finding a rectangle which have received the most updates, by scanning the list.

To show that U_i is bounded by $\mathcal{O}(R)$, we apply the *zeroing game* of Dietz and Sleator [18], using the variables $x_i = \max\{0, \frac{U_i - R}{5F}\}$ if rectangle i is open and $x_i = 0$ if it is closed. For open rectangles, x_i count the number of units of $5F$ updates received beyond the first R updates. This ensures that the variables are incremented by at most 1 in total for each round, when at most 5 flushes of size at most F are flushed into the open rectangles. When finalizing a rectangle it becomes closed, which ensures that $x_i = 0$, matching the zeroing step. We bound the total number of rectangles by $\bar{N}$ and therefore also the number of variables. Following [18, Theorem 5] and a proof similar to Theorem 4 in the Appendix, we have that for any i then $x_i \leq \log_2 \bar{N} + 1$ at any time. Consequently, it follows that $U_i \leq 5F(\log_2 \bar{N} + 1) + R$. It can be shown that $5F(\log_2 \bar{N} + 1) \leq 2R$, when $\bar{N} \geq 8$, by unfolding R and simplifying the inequality to show that $\frac{\varepsilon \log_2 B}{B^\varepsilon} \left(1 + \frac{1}{\log_2 \bar{N}}\right) \leq \frac{4}{5}$. It therefore holds that each rectangle receives at most $U_i \leq 3R$ updates, due to the zeroing game. Thus, when finalizing a rectangle, at least R updates have been performed. Including the updates from the buffers on the path towards the rectangle, the total number of updates applied is between R and $4R$. By ensuring that each rectangle contains $[8R, 16R[$ initial values, the rebalancing operations are possible, and the spanning criteria remains satisfied.

An update therefore performs at most 5 flushes using $\mathcal{O}(H)$ I/Os, along with locating and finalizing a single rectangle in respectively $\mathcal{O}(R/B) = \mathcal{O}(H)$ and $\mathcal{O}(H + \gamma)$ I/Os. Performing this operation incrementally allows for updates to spend worst-case $\mathcal{O}(\frac{1}{F}(H + \gamma))$ I/Os. If a query happens while an incremental update is being performed, the incremental update is completed, using at most $\mathcal{O}(H + \gamma)$ I/Os, which does not increase the total cost of the query. When a SEARCH query happens, similar to the new RANGE query, we do not apply the relevant updates on the path to the open rectangle to avoid queries interfering with the zeroing game. This concludes the proof of Theorem 3.

4 Discussion and Open Problems

Global rebuilding, as described in Section 2.6, allows for constructing a partially-persistent set of any sorted set in a linear number of I/Os, without creating the set anew by a sequence of insertions. Symmetrically, it is possible to *purge* all versions of the set older than some threshold, without performing all updates anew. This problem was first motivated by Becker, Gschwind, Ohler, Seeger, and Widmayer [6]. As our data structure consists of multiple independent data structures covering disjoint version intervals, then all data structures which only cover versions to be purged can be removed efficiently by a linear number of I/Os. For

both use cases, the space usage is asymptotically linear in the size of the oldest stored set and the number of updates performed.

Further, global rebuilding allows for a crude fully persistent data structure, which supports efficient buffered updates and queries, but where cloning past versions requires a linear number of I/Os. The fully persistent data structure by Brodal, Rysgaard, and Svenning [12] allows cloning past versions in worst case $\mathcal{O}(1)$ I/Os. They do, however, not buffer updates, which therefore are amortized and a factor $\mathcal{O}(1/B^{1-\varepsilon})$ slower than our data structure. Our data structure is therefore better when there are many updates, but few clones of past versions happening. Further, our data structure is simpler. It remains an open problem to design buffered fully-persistent search-trees, which remains efficient for clone operations.

In Section 3 we showed how to achieve worst-case bounds matching those of ephemeral B^ε -trees, when $M = \Omega(B^{1-\varepsilon} \log_2 N)$. This is an improvement by a factor $\Theta(B^\varepsilon / \log_2 B)$ on the required lower bound on M over the worst-case results of B^ε -trees by Das, Iacono, and Nekrich [15]. It remains an open problem to show a worst-case I/O lower bound dependency on M or to find a structure with worst-case I/O guarantees matching the amortized I/O bounds for $M = 2B$.

References

- 1 Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988. doi:10.1145/48529.48535.
- 2 Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, September 2003. doi:10.1007/s00453-003-1021-x.
- 3 Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithmics*, 8:1.2-es, December 2004. doi:10.1145/996546.996549.
- 4 Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, 47:1–25, 2007. doi:10.1007/s00453-006-1208-z.
- 5 Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683.
- 6 Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996. doi:10.1007/s007780050028.
- 7 Michael A. Bender, Rathish Das, Martín Farach-Colton, Rob Johnson, and William Kuszmaul. Flushing without cascades. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 650–669. Society for Industrial and Applied Mathematics, 2020. doi:10.1137/1.9781611975994.40.
- 8 Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 81–92. Association for Computing Machinery, 2007. doi:10.1145/1248377.1248393.
- 9 Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to B-trees and write-optimization. *login.*, 40(5), 2015. URL: <https://www.usenix.org/publications/login/oct15/bender>.
- 10 Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages

- 546—554. Society for Industrial and Applied Mathematics, 2003. URL: <http://dl.acm.org/citation.cfm?id=644108.644201>.
- 11 Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, pages 307—315. Association for Computing Machinery, 2003. doi:10.1145/780542.780589.
- 12 Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. External memory fully persistent search trees. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, pages 1410–1423. Association for Computing Machinery, 2023. doi:10.1145/3564246.3585140.
- 13 Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. On external memory graph traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 859—860. Society for Industrial and Applied Mathematics, 2000.
- 14 Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.
- 15 Rathish Das, John Iacono, and Yakov Nekrich. External-memory dictionaries with worst-case update cost. In Sang Won Bae and Heejin Park, editors, *33rd International Symposium on Algorithms and Computation, ISAAC 2022, December 19-21, 2022, Seoul, Korea*, volume 248 of *LIPIcs*, pages 21:1–21:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ISAAC.2022.21.
- 16 Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. *ACM Trans. Algorithms*, 3(2):13–es, May 2007. doi:10.1145/1240233.1240236.
- 17 Paul F. Dietz and Rajeev Raman. Persistence, randomization and parallelization: On some combinatorial games and their applications (abstract). In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11-13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 289–301. Springer, 1993. doi:10.1007/3-540-57155-8_256.
- 18 Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 365–372. Association for Computing Machinery, 1987. doi:10.1145/28395.28434.
- 19 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. doi:10.1016/0022-0000(89)90034-2.
- 20 Herbert Edelsbrunner and Mark H. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6(4):515–542, 1985.
- 21 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1), January 2012. doi:10.1145/2071379.2071383.
- 22 Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57. Association for Computing Machinery, 1984. doi:10.1145/602259.602266.
- 23 Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. I/O-efficient generation of massive graphs following the LFR benchmark. *ACM Journal of Experimental Algorithmics*, 23, August 2018. doi:10.1145/3230743.
- 24 William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin

- Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: Write-optimization in a kernel file system. *ACM Transactions on Storage*, 11(4), November 2015. doi:10.1145/2798729.
- 25 Curtis P. Kolovson and Michael Stonebraker. Indexing techniques for historical databases. In *Proceedings of the Fifth International Conference on Data Engineering, February 6-10, 1989, Los Angeles, California, USA*, pages 127–137. IEEE Computer Society, 1989. doi:10.1109/ICDE.1989.47208.
 - 26 Andrea Lancichinetti and Santo Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Physical Review E*, 80:016118, July 2009. doi:10.1103/PhysRevE.80.016118.
 - 27 Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78:046110, October 2008. doi:10.1103/PhysRevE.78.046110.
 - 28 Mark H. Overmars. *The design of dynamic data structures*, volume 156. Springer Science & Business Media, 1983. doi:10.1007/BFb0014927.
 - 29 Mark H. Overmars and Jan van Leeuwen. Dynamization of decomposable searching problems yielding good worsts-case bounds. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 1981. doi:10.1007/BFB0017314.
 - 30 Rajeev Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Time*. PhD thesis, Department of Computer Science, University of Rochester, 1992.
 - 31 Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986. doi:10.1145/6138.6151.
 - 32 Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997. doi:10.1109/69.599929.

A Subtraction Game

Dietz and Raman [17] considered the following *subtraction game*. The game is played on n non-negative real variables $x_1, \dots, x_n$, initially all zero. The game progresses in *rounds*. Each round consists of an *increment step* followed by a *subtraction step*. In the increment step an adversary selects n non-negative real values $\delta_1, \dots, \delta_n$, where $\sum_{i=1}^n \delta_i \leq 1$, and sets $x_i \leftarrow x_i + \delta_i$. In the subtraction step a largest x_i is decremented by setting $x_i \leftarrow \max\{0, x_i - 1\}$. Dietz and Raman [17, Theorem 3] proved an upper bound on all variables of $1 + \ln n$. Below we improve this bound to be $H_{n-1} \leq 1 + \ln(n-1)$. Note that $H_{n-1} \leq \log_2 n$ for $n \geq 1$. The proof extends to the *zeroing game* of Dietz and Sleator [18, Theorem 5], where the subtraction step is replaced by a *zeroing step*, in which a largest x_i is set to 0. For both games, it holds that for all i , $x_i < H_{n-1} + 1$ after each step.

► **Theorem 4.** *The subtraction game on $n \geq 2$ variables guarantees all $x_i < H_{n-1} \leq 1 + \ln(n-1)$ after each round.*

Proof. For $1 \leq k \leq n$, let s_k denote a strict upper bound on the sum of the k largest variables in the game after any round. The goal is to find a small valid s_1 . Below we argue that $s_n = n - 1$ and $s_k = \frac{k}{k+1}(1 + s_{k+1})$, for $1 \leq k < n$, are valid upper bounds. By induction for decreasing k , we have $s_k \geq k$ for $1 \leq k < n$: $s_{n-1} = \frac{n-1}{n}(1 + s_n) = \frac{n-1}{n}(1 + n - 1) = n - 1$ and $s_k = \frac{k}{k+1}(1 + s_{k+1}) \geq \frac{k}{k+1}(1 + k + 1) > k$ for $1 \leq k < n - 1$. By induction for increasing k , we have $s_1 = \sum_{j=2}^k \frac{1}{j} + \frac{1}{k}s_k$ for $2 \leq k \leq n$, i.e., $s_1 = \sum_{j=2}^n \frac{1}{j} + \frac{n-1}{n} = \sum_{j=1}^{n-1} \frac{1}{j} = H_{n-1}$.

Initially, all x_i are zero, i.e., all $s_1, \dots, s_n$ are valid strict upper bounds. By induction on the number of rounds, we show that the upper bounds remain valid after each round.

Consider s_n , and assume there exists a round where the total sum is $< n - 1$ before the round and $\geq n - 1$ after the round. Since the total sum increases, the subtraction step must have decreased the total sum by < 1 . It follows that all variables after the increment step must be < 1 , and the subtraction step sets one variable to zero, i.e., the total sum after the increment step is $< n - 1$. This contradicts the assumption that the total sum is $\geq n - 1$ after the round, i.e., the total sum after each round is $< n - 1$.

Next, consider s_k , for $1 \leq k < n$. We first consider the case where the subtracted variable $x_i < 1$ before the subtraction step, i.e., all variables are < 1 . Then, after the subtraction step the sum of the k largest variables is $< k \leq s_k$. Otherwise, $x_i \geq 1$ after the increment step. If the variable x_i is among the k largest variables after the round, then the sum of the k largest variables can increase by at most $\sum_{j=1}^n \delta_j - 1 \leq 0$ in the round (the sum of the k largest variables after the round consists of the same variables as before the round, or variables with smaller value before the round), i.e., the sum of the k largest variables does not increase by the round. Otherwise, x_i is not among the k largest variables after the round, but x_i is the largest after the increment step, where the $k + 1$ largest variables have sum $< \sum_{j=1}^n \delta_j + s_{k+1}$. After subtracting x_i (i.e., after the round), the sum of the new k largest variables is $< \frac{k}{k+1} (\sum_{j=1}^n \delta_j + s_{k+1}) \leq \frac{k}{k+1} (1 + s_{k+1}) = s_k$. $\blacktriangleleft$

That the analysis is tight follows from the following strategy, as also described in the PhD thesis of Raman [30, Section 2.2.3]. We first perform sufficiently many initial rounds, where after r rounds one variable has value zero, say $x_1 = 0$, and all other $n - 1$ variables have value $\varepsilon_r = 1 - \left(1 - \frac{1}{n}\right)^r$. Note $\varepsilon_0 = 0$ and $\varepsilon_r \rightarrow 1^-$ for $r \rightarrow \infty$. In round r we let $\delta_1 = \varepsilon_{r-1} + (1 - \varepsilon_{r-1})/n$ and $\delta_2 = \dots = \delta_n = (1 - \varepsilon_{r-1})/n$. This ensures that the increment step makes all n values have value $\varepsilon_r = \varepsilon_{r-1} + (1 - \varepsilon_{r-1})/n$ before the subtraction step. By induction it follows that $\varepsilon_r = 1 - \left(1 - \frac{1}{n}\right)^r$. By performing a sufficient number of initial rounds, $n - 1$ variables can achieve value $1 - \varepsilon$ arbitrary close to one. In the next $n - 2$ rounds, for $j = n - 1, \dots, 2$, we distribute value $1/j$ to the j variables with maximum value $1 - \varepsilon + \sum_{i=j+1}^{n-1} \frac{1}{i}$. The final maximum value is $1 - \varepsilon + \sum_{i=2}^{n-1} \frac{1}{i} = H_{n-1} - \varepsilon$.

Lazy B-Trees

Casper Moldrup Rysgaard ✉ 

Aarhus University, Denmark

Sebastian Wild ✉ 

Philipps-University Marburg, Germany, and University of Liverpool, UK

Abstract

Lazy search trees (Sandlund & Wild FOCS 2020, Sandlund & Zhang SODA 2022) are sorted dictionaries whose update and query performance smoothly interpolates between that of efficient priority queues and binary search trees – automatically, depending on actual use; no adjustments are necessary to the data structure to realize the cost savings. In this paper, we design *lazy B-trees*, a variant of lazy search trees suitable for external memory that generalizes the speedup of B-trees over binary search trees wrt. input/output operations to the same smooth interpolation regime.

A key technical difficulty to overcome is the lack of a (fully satisfactory) external variant of *biased* search trees, on which lazy search trees crucially rely. We give a construction for a subset of performance guarantees sufficient to realize external-memory lazy search trees, which we deem of independent interest.

As one special case, lazy B-trees can be used as an external-memory priority queue, in which case they are competitive with some tailor-made heaps; indeed, they offer faster decrease-key and insert operations than known data structures.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases B-tree, lazy search trees, lazy updates, external memory, deferred data structures, database cracking

Funding Casper Moldrup Rysgaard: Independent Research Fund Denmark, grant 9131-00113B.

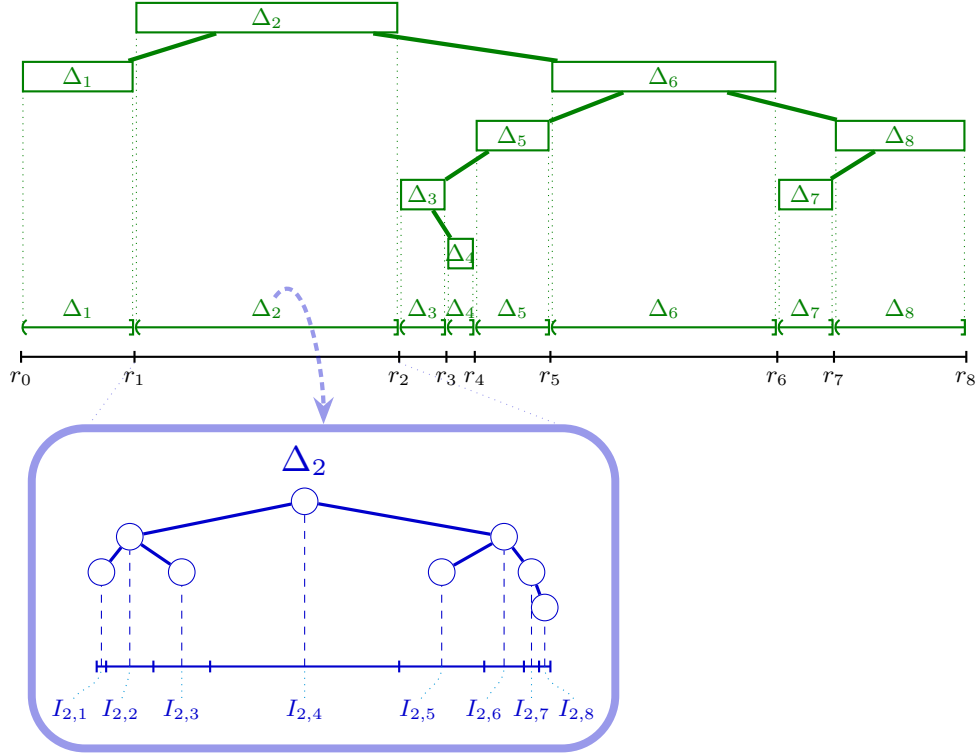
Sebastian Wild: EPSRC grant EP/X039447/1.

1 Introduction

We introduce the *lazy B-tree* data structure, which brings the adaptive performance guarantees of lazy search trees to external memory.

The binary search tree (BST) is a fundamental data structure, taught in every computer science degree and widespread in practical use. Wherever rank-based operations are needed, e.g., finding a k th smallest element in a dynamic set or determining the rank of an element in the set, i.e., its position in the sorted order of the current elements, augmented BSTs are the folklore solution: binary search trees using one of the known schemes to “balance” them, i.e., guarantee $\mathcal{O}(\log N)$ height for a set of size N , where we additionally store the subtree size in each node. On, say, AVL-trees, all operations of a sorted dictionary, i.e., rank, select, membership, predecessor, successor, minimum, and maximum, as well as insert, delete, change-key, split, and merge can all be supported in $\mathcal{O}(\log N)$ worst case time, where N is the current size of the set.

From the theory of efficient priority queues, it is well known that much more efficient implementations are possible if not all of the above operations have to be supported: When only minimum, insert, delete, decrease-key, and meld are allowed, all can be supported to run in constant time, except for $\mathcal{O}(\log N)$ delete. Lazy search trees [44, 46] (LSTs) show that we can get the same result with full support for all sorted-dictionary operations, so long as they are not used. When they do get used, lazy search trees give the optimal guarantees of any comparison-based data structure for the given query ranks, gracefully degrading from



■ **Figure 1** Schematic view of the original lazy search tree data structure [44].

priority-queue performance to BST performance, as more and more queries are asked. They therefore serve as a versatile drop-in replacement for both priority queues and BSTs.¹

The arguably most impactful application of sorted dictionaries across time is the database index. Here, economizing on accesses to slow secondary storage can outweigh any other costs, which gave rise to external-memory data structures; in particular B-trees and their numerous variants. They achieve a $\log_2(B)$ -factor speedup over BSTs in terms of input/output operations (I/Os), for B the block size of transfers to the external memory, for scanning-dominated operations such as a range query, even a factor B speedup can be achieved.

Much ingenuity has been devoted to (a) write-optimized variants of B-trees [3, 15], often with batched queries, and (b), adaptive indices [31], or more recently learned indices [36] that try to improve query performance via adapting to data at hand. *Deferred data structuring* – known as *database cracking* in the systems community – is a special case of (b) that, like lazy search trees, refines a sorted order (of an index) successively upon queries. Yet in the past, these two approaches often remained incompatible, if not having outright conflicting goals, making either insertions or queries fast. Lazy search trees are a principled solution to get both (as detailed in Section 1.2); unfortunately, standard lazy search trees are not competitive with B-trees in terms of their I/O efficiency, thus losing their adaptive advantages when data resides on secondary storage.

¹ We point out that a related line of work on “deferred data structures” also adapts to the queried ranks in the way LSTs do, but all data structures prior to LSTs had $\Omega(\log N)$ insertion time (with the assumption that any insertion is preceded by a query for said element). These data structures are therefore inherently unable to function as an efficient priority queue. Further related work is discussed in Section 1.3.

In this paper, we present *lazy B-trees*, an I/O-efficient variant of the original lazy search trees [44].² Lazy search trees consist of three layers (see Figure 1). The topmost layer consists of a biased search tree of *gaps* (defined in Section 1.2), weighted by their size. The second layer consists, for each gap Δ_i , of a balanced binary search tree of $\mathcal{O}(\log |\Delta_i|)$ *intervals* $I_{i,j}$. Intervals are separated by splitter elements (pivots as in Quicksort), but are unsorted within. The third layer represents intervals as a simple unordered list.

A key obstacle to external-memory variants of lazy search trees – both for the original version [44] and for optimal lazy search trees [46] – is the topmost layer. As discussed in Section 1.3, none of the existing data structures fully solves the problem of how to design a general-purpose *external-memory biased search tree* – and we likewise leave this as an open problem for future work. However, we show that for a slightly restricted set of performance guarantees sufficient for lazy search trees, a (to our knowledge) novel construction based on partitioning elements by *weight* with doubly-exponentially increasing bucket sizes provides an I/O-efficient solution. This forms the basis of our lazy B-trees, which finally bring the adaptive versatility of lazy search trees to external memory.

1.1 The External-Memory Model

In this paper we study the sorted dictionary problem in a hierarchical-memory model, where we have an unbounded external memory and an internal memory of capacity M elements, and where data is transferred between the internal and external memory in blocks of B consecutive elements. A block transfer is called an *I/O* (input/output operation). The I/O cost of an algorithm is the number of I/Os it performs. Aggarwal and Vitter [2] introduced this as the *external-memory model* (and proved, e.g., lower bounds for sorting in the model).

1.2 Lazy Search Trees

In this section we briefly describe lazy search trees [44]. Lazy search trees support all operations of a dynamic sorted dictionary (details in [44]). We consider the following operations (sufficient to base a full implementation on):

CONSTRUCT(S): Constructs the data structure from the elements of set S .³

INSERT(e): Adds element e to the set.

DELETE(ptr): Deletes the element at pointer ptr from the set.

CHANGEKEY(ptr, e'): Changes the element e at pointer ptr to e' .

QUERYELEMENT(e): Locates the predecessor e' to element e in the set, and returns rank r of e' , and a pointer to e' .

QUERYRANK(r): Locates the element e with rank r in the set, and returns a pointer to e .

Lazy search trees maintain the elements of the set partitioned into G “gaps”, $\Delta_1, \dots, \Delta_G$. All elements in Δ_i are weakly smaller than all elements in Δ_{i+1} , but the gaps are otherwise treated as unsorted bags. Boundaries between gaps (and new gaps) are introduced only via a query; initially we have a single gap Δ_1 . Every query is associated with its *query rank* r [45, §4], i.e., the rank of its result, and splits an existing gap into two new gaps, such that its query rank is the largest rank in the left one of these gaps. After queries with

² The follow-up work [46] replaces the second and third layer using *selectable priority queues*; we discuss why these are challenging to make external in Section 1.4.

³ **CONSTRUCT** and iterative insertion have the same complexity in internal memory, but in external memory, the bulk operation can be supported more efficiently.

■ **Table 1** Overview of results on Search Trees and Priority Queues for both the internal- and external-memory model. For the internal model the displayed time is the number of RAM operations performed, while for the external model the displayed time is the number of I/Os performed. Amortized times are marked by “am.” and results doing batched updates and queries are marked by “batched”. For priority queues, query is only for the minimum. All results use linear space.

	Insert	Query
Internal-Memory		
Balanced BST [1, 26]	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Priority Queue [14, 19]	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$
Lazy Search Tree [44]	$\mathcal{O}\left(\log \frac{N}{ \Delta_i } + \log \log N\right)$	$\mathcal{O}(\log N + x \log c)$ am.
Optimal LST [46]	$\mathcal{O}\left(\log \frac{N}{ \Delta_i }\right)$	$\mathcal{O}(\log N + x \log c)$ am.
External-Memory		
B-tree [8]	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\log_B N)$
B $^\varepsilon$ -tree [15]	$\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\right)$ am.	$\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N\right)$ am.
Buffer Tree [3, 4]	$\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ batched	$\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ batched
I/O-eff. heap [37]	$\mathcal{O}\left(\frac{1}{B} \log_2 \frac{N}{B}\right)$ am.	$\mathcal{O}\left(\frac{1}{B} \log_2 \frac{N}{B}\right)$ am.
x-treap heap [28]	$\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ am.	$\mathcal{O}\left(\frac{M^\varepsilon}{B} \log_{M/B}^2 \frac{N}{B}\right)$ am.
<i>This paper, LST</i> (Theorem 1)	$\mathcal{O}\left(\log_B \frac{N}{ \Delta_i } + \log_B \log_B \Delta_i \right)$	$\mathcal{O}\left(\log_B \min\{N, q\} + \frac{1}{B} \log_2 \Delta_i + \log_B \log_B \Delta_i + \frac{1}{B} x \log_2 c\right)$ am.
<i>This paper, PQ</i> (Corollary 2)	$\mathcal{O}(\log_B \log_B N)$	$\mathcal{O}\left(\frac{1}{B} \log_2 N + \log_B \log_B N\right)$ am.

ranks $r_1 < r_2 < \dots < r_q$, we thus obtain the gaps $\Delta_1, \dots, \Delta_{q+1}$ where $|\Delta_i| = r_i - r_{i-1}$ upon setting $r_0 = 0$ and $r_{q+1} = n$.

Lazy search trees support insertions landing in gap Δ_i in $\mathcal{O}(\log(N/|\Delta_i|) + \log \log |\Delta_i|)$ (worst-case) time and queries in $\mathcal{O}(x \log c + \log n)$ amortized time, where the query splits a gap into new gaps of sizes x and cx for some $c \geq 1$. Deletions are supported in $\mathcal{O}(\log n)$ (worst-case) time. To achieve these times, gaps are further partitioned into intervals, with an amortized splitting and merging scheme (see Section 3 for details).

1.3 Related Work

We survey key prior work in the following; for a quick overview with selected results and comparison with our new results, see also Table 1.

(External) Search trees. Balanced binary search trees (BSTs) exist in many varieties, with AVL trees [1] and red-black trees [26] the most widely known. When augmented with subtree sizes, they support all sorted dictionary operations in $\mathcal{O}(\log N)$ worst-case time. Simpler variants can achieve the same via randomization [47, 39] or amortization [48]. In external memory, B-trees [8], often in the leaf-oriented flavor as B $^+$ -trees and augmented with subtree sizes, are the benchmark. They support all sorted-dictionary operations in $\mathcal{O}(\log_B N)$ I/Os. By batching queries, buffer trees [3, 4] substantially reduce the cost to amortized $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os, but are mostly useful in an offline setting due to the long delay from query to answer. B $^\varepsilon$ -trees [15] avoid this with a smaller buffer of operations per node to achieve amortized $\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\right)$ I/Os for updates and $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N\right)$ I/Os for queries (with immediate answers), where $\varepsilon \in (0, 1]$ is a parameter.

Dynamic Optimality. The dynamic-optimality conjecture for Splay trees [48] resp. the GreedyBST [38, 41, 21] algorithm posits that these methods provide an instance-optimal binary-search-tree algorithm for any (long) sequence of searches over a static set of keys in a binary search tree. While still open for Splay trees and GreedyBST, the dynamic optimality of data structures has been settled in some other models: it holds for a variant of skip lists [13], multi-way branching search trees, and B-trees [13]; it has been refuted for tournament heaps [42]. As in lazy search trees, queries clustered in time or space allow a sequence to be served faster. Unlike in lazy search trees, insertions can remain expensive even when no queries ever happen close to them. A more detailed technical discussion of similarities and vital differences compared to lazy search trees appears in [44, 45].

(External) Priority Queues. When only minimum-queries are allowed, a sorted dictionary becomes a priority queue (PQ) (a.k.a. “heap”). In internal memory, all operations except delete-min can then be supported in $\mathcal{O}(1)$ amortized [25] or even worst-case time [14, 19]. In external memory, developing efficient PQs has a long history. For batch use, buffer-tree PQs [4] and the I/O-efficient heap of [37] support k operations of insert and delete-min in $\mathcal{O}(\frac{k}{B} \log_{M/B} \frac{N}{M})$ I/Os, with N denoting the maximum number of stored keys over the k operations. The same cost per operation can be achieved in a worst-case sense [16] (i.e., B subsequent insert/delete-min operations cost $\mathcal{O}(\log_{M/B} \frac{N}{M})$ I/Os).

None of these external-memory PQs supports decrease-key. The I/O-efficient tournament trees of [37] support a sequence of insert, delete-min, and decrease-key with $\mathcal{O}(\frac{1}{B} \log_2 \frac{N}{B})$ I/Os per operation. A further $\log \log N$ factor can be shaved off [32] (using randomization), but that, surprisingly, is optimal [23]. A different trade-off is possible: insert and decrease-key are possible in $\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ amortized I/Os at the expense of driving up the cost for delete/delete-min to $\mathcal{O}(\frac{M^\epsilon}{B} \log_{M/B}^2 \frac{N}{B})$ [28].

(External) Biased Search Trees. Biased search trees [9] maintain a sorted set, where each element e has a (dynamic) weight $w(e)$, such that operations in the tree spend $\mathcal{O}(\log \frac{W}{w(e)})$ time, for W the total weight, $W = \sum_e w(e)$. Biased search trees can be built from height-balanced trees ((2, b)-globally-biased trees [9]) or weight-balanced trees (by representing high-weight elements several times in the tree [40]), and Splay trees automatically achieve the desired time in an amortized sense [48, 40].

None of the original designs are well suited for external memory. Feigenbaum and Tarjan [24] extended biased search trees to (a, b) -trees for that purpose. However, during the maintenance of (a, b) -trees, some internal nodes may have a degree much smaller than a (at least 2), which means that instead of requiring $\mathcal{O}(N/B)$ blocks to store N weighted elements, they require $\mathcal{O}(N)$ blocks in the worst case.⁴ The authors in [24] indeed leave it as an open problem to find a space-efficient version of biased (a, b) -trees. Another attempt at an external biased search tree data structure is based on deterministic skip lists [5]. Yet again, the space usage seems to be $\Omega(N)$ blocks of memory.⁵

⁴ In particular, in [24], they distinguish internal nodes between minor and major, minor being the nodes that have degree $< a$ or have a small rank. All external nodes are major.

⁵ Unfortunately, it remains rather unclear from the description in the article exactly which parts of the skiplist “towers” of pointers of an element are materialized. Unlike in the unweighted case, an input could have large and small weight alternating, with half the elements of height $\approx \log_b W$. Fully materializing the towers would incur $\Omega(n \log_b W)$ space; otherwise this seems to require a sophisticated scheme to materialize towers on demand, e.g., upon insertions, and we are not aware of a solution.

6 Lazy B-Trees

To our knowledge, no data structure is known that achieves $\mathcal{O}(\log_B \frac{W}{w(e)})$ I/Os for a dynamic weighted set in external memory while using $\mathcal{O}(N/B)$ blocks of memory.

Deferred Data Structures & Database Cracking. Deferred data structuring refers to the idea of successively establishing a query-driven structure on a dataset. This is what lazy search trees do upon queries. While the term is used more generally, it was initially proposed in [35] on the example of a (sorted) dictionary. For an offline sequence of q queries on a given (static) set of N elements, their data structure uses $\mathcal{O}(N \log q + q \log N)$ time. In [20], this is extended to allow update operations in the same time (where q now counts all operations). The refined complexity taking query *ranks* into account was originally considered for the (offline) multiple selection problem: when q ranks $r_1 < \dots < r_q$ are sought, leaving gaps $\Delta_1, \dots, \Delta_{q+1}$, $\Theta(\sum_{i=1}^{q+1} |\Delta_i| \log(N/\Delta_i))$ comparisons are necessary and sufficient [22, 33]. For multiple selection in external memory, $\Theta(\sum_{i=1}^{q+1} \frac{|\Delta_i|}{B} \log_{M/B} \frac{N}{|\Delta_i|})$ I/Os are necessary and sufficient [7, 17], even cache-obliviously [17, 18].

Closest to lazy search trees is the work on *online dynamic multiple selection* by Barbay et al. [6, 7], where online refers to the query ranks arriving one by one. As pointed out in [44], the crucial difference between all these works and lazy search trees is that the analysis of dynamic multiple selection assumes that every insertion is preceded by a query for the element, which implies that insertions must take $\Omega(\log N)$ time. (They assume a nonempty set of elements to initialize the data structure with, for which no pre-insertion queries are performed.) Barbay et al. also consider online dynamic multiple selection in external memory. By maintaining a B-tree of the pivots for partitioning, they can support updates – again, implicitly preceded by a query – at a cost of $\Theta(\log_B N)$ I/Os each.

In the context of adaptive indexing of databases, deferred data structuring is known under the name of *database cracking* [30, 29, 27]. While the focus of research is on systems engineering, e.g., on the partitioning method [43], some theoretical analyses of devised algorithms have also appeared [50, 49]. These consider the worst case for q queries on N elements similar to the original works on deferred data structures.

1.4 Contribution

Our main contribution, the lazy B-tree data structure, is summarized in Theorem 1 below.

► **Theorem 1 (Lazy B-Trees).** *There exists a data structure over an ordered set, that supports*

- *CONSTRUCT(S) in worst-case $\mathcal{O}(|S|/B)$ I/Os,*
- *INSERT in worst-case $\mathcal{O}(\log_B \frac{N}{|\Delta_i|} + \log_B \log_B |\Delta_i|)$ I/Os,*
- *DELETE in amortized $\mathcal{O}(\log_B \frac{N}{|\Delta_i|} + \frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os,*
- *CHANGEKEY in worst-case $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os if the element is moved towards the nearest queried element but not past it, and amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os otherwise, and*
- *QUERYELEMENT and QUERYRANK in amortized*
 $\mathcal{O}(\log_B \min\{N, q\} + \frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B} x \log_2 c)$ *I/Os.*

Here N denotes the size of the current set, $|\Delta_i|$ denotes the size of the manipulated gap, q denotes the number of performed queries, and x and cx for $c \geq 1$ are the resulting sizes of the two gaps produced by a query. The space usage is $\mathcal{O}(N/B)$ blocks.

From the above theorem, the following corollary can be derived, which states the performance of lazy B-trees when used as a priority queue.

► **Corollary 2** (Lazy B-Trees as external PQ). *A lazy B-tree may be used as a priority queue, to support, within $\mathcal{O}(N/B)$ blocks of space, the operations*

- *CONSTRUCT(S) in worst-case $\mathcal{O}(|S|/B)$ I/Os,*
- *INSERT in worst-case $\mathcal{O}(\log_B \log_B N)$ I/Os,*
- *DELETE in amortized $\mathcal{O}(\frac{1}{B} \log_2 N + \log_B \log_B N)$ I/Os,*
- *DECREASEKEY in worst-case $\mathcal{O}(\log_B \log_B N)$ I/Os, and*
- *MINIMUM in amortized $\mathcal{O}(\frac{1}{B} \log_2 N + \log_B \log_B N)$ I/Os.*

The running times of lazy B-trees when used as a priority queue are not competitive with heaps targeting sorting complexity (such as buffer trees [3, 4]); however these data structures do not (and cannot [23]) support decrease-key efficiently. By contrast, for very large N , lazy B-trees offer exponentially faster decrease-key and insert than previously known external priority queues, while only moderately slowing down delete-min queries.

Our key technical contribution is a novel technique for partially supporting external biased search tree performance, formally stated in Theorem 3 below. In the language of a biased search tree, it supports searches (by value or rank) as well as incrementing or decrementing⁶ a weight $w(e)$ by 1 in $\mathcal{O}(\log_B(W/w(e)))$ I/Os for an element e or weight $w(e)$; inserting or deleting an element, however, takes $\mathcal{O}(\log_B N)$ I/Os irrespective of weight, where N is the number of elements currently stored. Unlike previous approaches, the space usage is the $\mathcal{O}(N/B)$ blocks throughout. A second technical contribution is the streamlined potential-function-based analysis of the interval data structure of lazy search trees.

We mention two, as yet insurmountable, shortcomings of lazy B-trees. The first one is the $\log \log N$ term we inherit from the original Lazy search trees [44]. This cost term is in addition to the multiple-selection lower bound and thus not necessary. Indeed, it was in internal memory subsequently removed [46], using an entirely different representation of gaps, which fundamentally relies on soft-heap-based selection on a priority queue [34]. The route to an external memory version of this construction is currently obstructed by two road blocks. First, we need an external-memory soft heap; the only known result in this direction [11] only gives performance guarantees when $N = \mathcal{O}(B(M/B)^{M/2(B+\sqrt{M/B})})$ and hence seems not to represent a solution for the general problem. Second, the selection algorithm from [46] requires further properties of the priority queue implementation, in particular a bound on the fanout; it is not clear how to combine this with known external priority queues.

The second shortcoming is that – unlike for comparisons – we do not get close to the I/O-lower bound for multiple-selection with lazy B-trees. Doing so seems to require a way of buffering as in buffer trees, to replace our fanout of B by a fanout of M/B . This again seems hard to achieve since an *insertion* in the lazy search tree uses a *query* on the gap data structure, and a *query* on the lazy search tree entails an *insertion* into the gap data structure (plus a re-weighting operation).

Outline. The remainder of the paper is structured as follows. Section 2 describes the gap data structure (our partial external biased search tree) and key innovation. Section 3 sketches the changes needed to turn the interval data structure from [44] into an I/O-efficient data structure; the full details, including our streamlined potential function, appear in Appendix A. In Section 4, we then show how to assemble the pieces into a proof of our main result, Theorem 1. We conclude in Section 5 with some open problems. To be self-contained, we include proofs of some technical lemmas used in the analysis in Appendix B.

⁶ General weight changes are possible in that time with $w(e)$ the minimum of the old and new weight.

2 The Gap Structure: A New Restricted External Biased Search Tree

In this section we present a structure on the gaps, which allows for the following operations. Let N denote the total number of elements over all gaps and G denote the number of gaps. Note that $G \leq N$, as no gaps are empty. We let Δ_i denote the i th gap when sorted by element, and $|\Delta_i|$ denote the number of elements contained in gap Δ_i .

GAPBYELEMENT(e) Locates the gap Δ_i containing element e .

GAPBYRANK(r) Locates the gap Δ_i containing the element of rank r .

GAPINCREMENT(Δ_i) / **GAPDECREMENT**(Δ_i) Changes the weight of gap Δ_i by 1.

GAPSPLIT($\Delta_i, \Delta'_i, \Delta'_{i+1}$) Splits gap Δ_i into the two non-overlapping gaps Δ'_i and Δ'_{i+1} , s.t. the elements of Δ_i are equal to the elements of Δ'_i and Δ'_{i+1} .

For the operations, we obtain I/O costs, as described in the theorem below.

► **Theorem 3** (Gap Data Structure). *There exists a data structure on a weighted ordered set, that supports **GAPBYELEMENT**, **GAPBYRANK**, **GAPINCREMENT** and **GAPDECREMENT** in $\mathcal{O}(\log_B \frac{N}{|\Delta_i|})$ I/Os, and **GAPSPLIT** in $\mathcal{O}(\log_B G)$ I/Os. Here N denotes the total size of all gaps, $|\Delta_i|$ denotes the size of the touched gap and G denotes the total number of gaps. The space usage is $\mathcal{O}(G/B)$ blocks.*

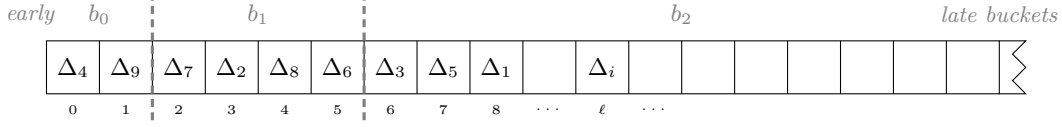
► **Remark 4** (Comparison with biased search trees). A possible solution would be an external-memory version of biased search trees, but as discussed in the introduction, no fully satisfactory such data structure is known. Instead of supporting all operations of biased trees in full generality, we here opt for a solution solving (just) the case at hand. The solution falls short of a general biased search trees, as the insertion or deletion costs are not a function of the *weight* of the affected gap, but the *total number* G of gaps in the structure. Moreover, we only describe how to change the weight of gaps by 1; however, general weight changes could be supported, with the size of the change entering the I/O cost, matching what we expect from a general biased search tree.

Note that the gaps in the structure are non-overlapping, and that their union covers the whole element range. The query for an element contained in some gap is therefore a predecessor query on the left side of the gaps, however, as their union covers the entire element range, the queries on the gap structure behave like *exact queries*: we can detect whether we have found the correct gap and can then terminate the search. (By definition, there are no unsuccessful searches in our scenario, either.)

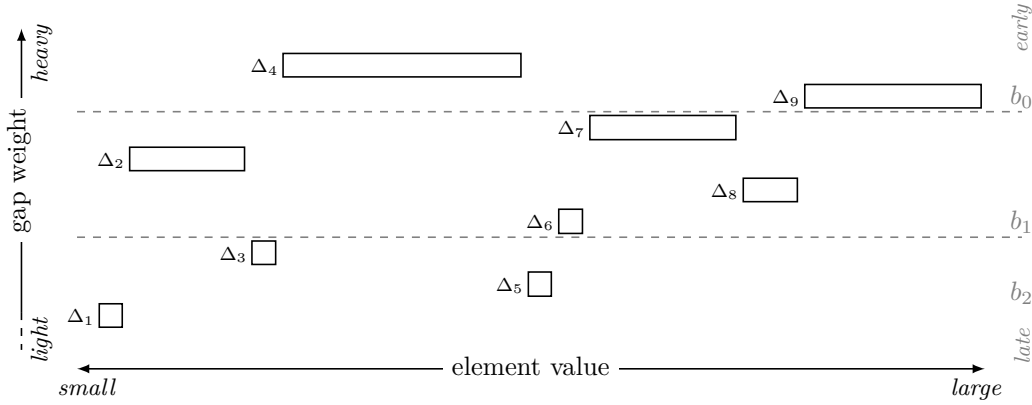
For consistency with the notation of biased search trees, we write in the following $w_i = |\Delta_i|$ and $W = \sum_i w_i$. Note that $W = N$. Consider a conceptual list, where the gaps are sorted decreasingly by *weight*, and let gap Δ_i be located at some index ℓ in this list. This conceptual list is illustrated in Figure 2, and Figure 3 gives a two-dimensional view of the list, which considers both the gap weight as well as the gaps ordered by element. As the total weight before index ℓ in the conceptual list is at most W and the weight of each gap before index ℓ is at least w_i , then it must hold that $\ell \leq \frac{W}{w_i}$. If we were to search for a gap of known *weight*, it can therefore be found with an exponential search [10] in time $\mathcal{O}(\log \ell) = \mathcal{O}(\log \frac{W}{w_i})$. However, searches are based on *element values* (and, e.g., for insertions, without knowing the target gap's weight), so this alone would not work.

2.1 Buckets by Weight and Searching by Element

Instead, the gaps are split into buckets b_j , s.t. the weight of all gaps in bucket b_j is greater than or equal to the weight of all gaps in bucket b_{j+1} . We further impose the invariant



■ **Figure 2** The conceptual list of the gaps. The gaps are sorted by decreasing weight, with the heaviest gap (largest weight), Δ_4 at index 0. Gap Δ_i with weight w_i is stored at index $\ell \leq W/w_i$ in the list. The gaps are split into buckets $b_0, b_1, b_2, \dots$ of doubly exponential size.



■ **Figure 3** Two dimensional view of the first gaps of Figure 2 represented by rectangles, with the width of the rectangle denoting the weight (size) of the gap. The horizontal axis sorts gaps by element value; the vertical axis by weight. Dashed lines show bucket boundaries.

that the size of (number of gaps in) bucket b_j is *exactly* B^{2^j} for all but the last bucket, which may be of any (smaller) size. Each bucket is a B-tree containing the gaps in the bucket, sorted by element value. The time to search for a given element in bucket b_j is therefore $\mathcal{O}(\log_B(|b_j|)) = \mathcal{O}(2^j)$ I/Os. For consistent wording, we will in the following always use *smaller/larger* to refer to the order by element values, *lighter/heavier* for gaps of smaller/larger weight/size, and *earlier/later* for buckets of smaller/larger index. Note that earlier buckets contain fewer, but heavier gaps.

GAPBYRANK. A search for a gap proceeds by searching in buckets $b_0, b_1, b_2, \dots$ until the desired gap is found in some bucket b_k . To search in all buckets up until b_k requires $\sum_{j=0}^k \mathcal{O}(2^j) = \mathcal{O}(2^k)$ I/Os, which is therefore up to constant factors the same cost as searching in only bucket b_k . Consider some gap Δ_i , which has the ℓ th heaviest weight, i.e., it is located at index ℓ , when sorting all gaps decreasingly by weight. By the invariant, the buckets are sorted decreasingly by the weight of their internal elements. Let the bucket containing Δ_i be b_k . It must then hold that the sizes of the earlier buckets does not allow for Δ_i to be included, but that bucket b_k does. Therefore,

$$\sum_{j=0}^{k-1} |b_j| < \ell \leq \sum_{j=0}^k |b_j|.$$

As $|b_j| = B^{2^j}$, the sums are asymptotically equal to the last term of the sum up to constant factors (Lemma 10 in Appendix B). It then holds that $\ell = \mathcal{O}(B^{2^k})$ and $\ell = \Omega(B^{2^{k-1}}) = \Omega((B^{2^k})^{1/2})$. Thus $\log_B(\ell) = \Theta(\log_B |b_k|)$, and conversely $2^k = \Theta(\log_B \ell)$. The gap Δ_i can thus be found using $\mathcal{O}(\log_B \frac{W}{w_i})$ I/Os, concluding the GAPBYELEMENT operation.

2.2 Updating the Weights

Both explicit weight changes in **GAPINCREMENT**/**GAPDECREMENT** as well as the **GAPSPLIT** operation require changes to the weights of gaps. Here, we have to maintain the sorting of buckets by weight.

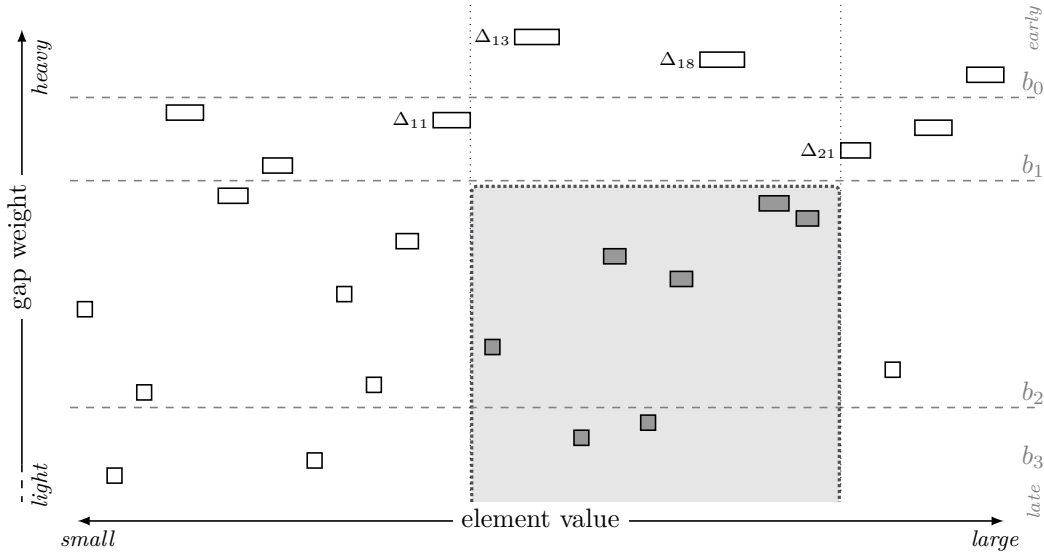
GAPINCREMENT/GAPDECREMENT. When performing **GAPINCREMENT** on gap Δ_i , the weight is increased by 1. In the conceptual list, this may result in the gap moving to an earlier index, with the order of the other gaps the same. This move is made in the conceptual list (Figure 2) by swapping Δ_i with its neighbor. As the buckets are ordered by element value, swapping with a neighbor (by weight) in the same bucket does not change the bucket structure. When swapping with a neighbor in an earlier bucket, however, the bucket structure must change to reflect this. Following the conceptual list (Figure 2), this gap is a lightest (minimum-weight) gap in bucket b_{k-1} . This may then result in Δ_i moving out of its current bucket b_k , into some earlier bucket. As the gap moves to an earlier bucket, some other gap must take its place in bucket b_k . Following the conceptual list, it holds that this gap is a lightest (minimum-weight) gap in bucket b_{k-1} , and so on for the remaining bucket moves.

When performing **GAPDECREMENT**, the gap may move to a later bucket, where the heaviest gap in bucket b_{k+1} is moved to bucket b_k .

In both cases, a lightest gap in one bucket is swapped with a heaviest gap in the neighboring (later) bucket. To find lightest or heaviest gaps in a bucket, we augment the nodes of the B-tree of each bucket with the values of the lightest and heaviest weights in the subtree for each of its children. This allows for finding a lightest or heaviest weight gap in bucket b_j efficiently, using $\mathcal{O}(\log_B |b_j|)$ I/Os. Further, this augmentation can be maintained under insertions and deletions.

Upon a **GAPINCREMENT** or **GAPDECREMENT** operation, the desired gap Δ_i can be located in the structure using $\mathcal{O}(\log_B \frac{W}{w_i})$ I/Os, and the weight is adjusted. We must then perform swaps between the buckets, until the invariant that buckets are sorted decreasingly by weight holds. This swapping may be performed on all buckets up to the last touched bucket. For **GAPINCREMENT** the swapping only applies to earlier buckets as the weight of Δ_i increases, and the height of the tree in the latest bucket is $\mathcal{O}(\log_B \frac{W}{w_i})$. For **GAPDECREMENT** the swapping is performed into later buckets, but only up to the final landing place for weight $w_i - 1$. If $w_i \geq 2$, the height of the tree in the last touched bucket is $\mathcal{O}(\log_B \frac{W}{w_i-1}) = \mathcal{O}(1 + \log_B \frac{W}{w_i})$. If $w_i = 1$, gap Δ_i is to be deleted. The gap is swapped until it is located in the last bucket, from where it is removed. We have $\mathcal{O}(\log_B \frac{W}{w_i}) = \mathcal{O}(\log_B W)$, as $w_i = 1$, whereas the height of the last bucket is $\mathcal{O}(\log_B G) = \mathcal{O}(\log_B W)$, as $G \leq W$. Therefore both **GAPINCREMENT** and **GAPDECREMENT** can be performed using $\mathcal{O}(\log_B \frac{W}{w_i})$ I/Os.

GAPSPLIT. When **GAPSPLIT** is performed on gap Δ_i , the gap is removed and replaced by two new gaps Δ'_i and Δ'_{i+1} , s.t. $|\Delta_i| = |\Delta'_i| + |\Delta'_{i+1}|$. In the conceptual list, the new gaps must reside at later indexes than Δ_i . As the sizes of the buckets are fixed, and the number of gaps increases, some gap must move to the last bucket. Similarly to **GAPINCREMENT** and **GAPDECREMENT**, the order can be maintained by performing swaps of gaps between consecutive buckets: First, we replace Δ_i (in its spot) by Δ'_i ; if this violates the ordering of weights between buckets, we swap Δ'_i with the later neighboring bucket, until the correct bucket is reached. Then we insert Δ'_{i+1} into the last bucket and swap it with its earlier neighboring bucket until it, too, has reached its bucket. Both processes touch at most all $\log_B \log_2 G$ buckets and spend a total of $\mathcal{O}(\log_B G)$ I/Os.



■ **Figure 4** Illustration of “holes”. The figure shows the two-dimensional view of gaps described in Figure 3 (gap widths not to scale). The *hole* between Δ_{11} and Δ_{21} in bucket b_1 , marked by the dotted line, contains the total weight of all gaps marked in gray. These are precisely the gaps that fall both (a) between Δ_{11} and Δ_{21} by element value and (b) in later buckets by (lighter) weight. Note specifically that Δ_{13} and Δ_{18} are *not* counted in the hole, as they reside in an earlier bucket.

2.3 Supporting Rank Queries

The final operation to support is `GAPBYRANK`. Let the *global rank* of a gap Δ_i , $r(\Delta_i) = |\Delta_1| + \dots + |\Delta_{i-1}|$, denote the rank of the smallest element located in the gap, i.e., the number of elements residing in gaps of smaller elements. The *local rank* of a gap Δ_i in bucket b_j , $r_j(\Delta_i)$, denotes the rank of the gap among elements located in bucket b_j only.

Computing the rank of a gap. To determine the global rank of a gap Δ_i , the total weight of all smaller gaps must be computed. This is equivalent to the sum of the local ranks of Δ_i over all buckets: $r(\Delta_i) = \sum_j r_j(\Delta_i)$. First we augment the B-trees of the buckets, such that each node contains the total weight of all gaps in the subtree. This allows us to compute the local rank $r_j(\Delta_i)$ of a gap Δ_i inside a bucket b_j using $\mathcal{O}(\log_B |b_j|)$ I/Os. The augmented values can be maintained under insertions and deletions in the tree within the stated I/O cost. When searching for the gap Δ_i , the local rank $r_j(\Delta_i)$ of all earlier buckets b_j up to the bucket b_k , which contains Δ_i , can then be computed using $\mathcal{O}(\log_B \frac{W}{w_i})$ I/Os in total.

It then remains to compute the total size of gaps smaller than Δ_i in all buckets after b_k , i.e., the sum of the local ranks $r_\ell(\Delta_i)$ for all later buckets b_ℓ , $\ell > k$, to compute in total the global rank $r(\Delta_i)$. As these buckets are far bigger, we cannot afford to query them. Note that any gaps in these later buckets must fall between two *consecutive* gaps in b_k , as the gaps are non overlapping in element-value space. Denote the space between two gaps in a bucket as a *hole*. We then further augment the B-tree to contain in each hole of bucket b_j the total size of gaps of *later* buckets contained in that hole, and let each node contain the total size of all holes in the subtree. See Figure 4 for an illustration of which gaps contributes to a hole.

This then allows computing the global rank $r(\Delta_i)$ of gap Δ_i , by first computing the local rank $r_j(\Delta_i)$ in all earlier buckets b_j , i.e., for all $j \leq k$, and then adding the total size of all smaller holes in b_k . The smaller holes in b_k exactly sum to the local ranks $r_\ell(\Delta_i)$ for all later

12 Lazy B-Trees

buckets b_ℓ for $\ell > k$. As this in total computes $\sum_j r_j(\Delta_i)$, then by definition, we obtain the global rank $r(\Delta_i)$ of the gap Δ_i .

GAPBYRANK. For a GAPBYRANK operation, some rank r is given, and the gap Δ_i is to be found, s.t. Δ_i contains the element of rank r , i.e., the global rank $r(\Delta_i) \leq r < r(\Delta_{i+1})$. The procedure is as follows. First, the location of the queried rank r is computed in the first bucket. If this location is contained in a gap Δ_i of the bucket, then Δ_i must be the correct gap, which is then returned. Otherwise, the location of r must be in a hole. As the sizes of the gaps contained in the first bucket is not reflected in the augmentation of later buckets, the value of r is updated to reflect the missing gaps of the first bucket: we subtract from t the local rank $r_0(\Delta_i)$ of the gap Δ_i immediately after the hole containing the queried rank r . This adjusts the queried rank r to be relative to elements in gaps of later buckets. Put differently, the initial queried rank r is the queried global rank, which is the sum local ranks; we now remove the first local rank again. This step is recursively applied to later buckets, until the correct gap Δ_i is found in some bucket b_k , which contains the element of the initially queried global rank r . As the correct gap Δ_i must be found in the bucket b_k containing it, the GAPBYRANK operation uses $\mathcal{O}(\log_B \frac{W}{w_i})$ I/Os to locate it.

Maintaining hole sizes. The augmentation storing the sizes of all holes in a subtree can be updated efficiently upon insertions or deletions of gaps in the tree, or upon updating the size of a hole. However, computing the sizes of the holes upon updates in the tree is not trivial. If a gap is removed from a bucket, the holes to the left resp. right of that gap are merged into a single hole, with a size equal to the sum of the previous holes. If the removed gap is moved to a later bucket, the hole must now also include the size of the removed gap. A newly inserted gap must, by the non-overlapping nature of gaps, land within a hole of the bucket, which is now split in two. If the global rank of the inserted gap is known, then the sizes of the resulting two holes can be computed, s.t. the global rank is preserved.

In the operations on the gap structure, GAPBYELEMENT or GAPBYRANK do not change the sizes of gaps, so the augmentation does not change. Upon a GAPINCREMENT or GAPDECREMENT operation, the size of some gap Δ_i changes. This change in size can be applied to all holes containing Δ_i in earlier buckets using $\mathcal{O}(\log_B \frac{W}{w_i})$ I/Os in total. Then swaps are performed between neighboring buckets until the invariant that buckets are sorted by weight, holds again. During these swaps, the sizes of gaps do not change, which allows for computing the global rank of the gaps being moved, and update the augmentation without any overhead in the asymptotic number of I/Os performed. The total number of I/Os to perform a GAPINCREMENT or GAPDECREMENT operation does not increase asymptotically, and therefore remains $\mathcal{O}(\log_B \frac{W}{w_i})$.

When a GAPSPLIT is performed, a single gap Δ_i is split. As the elements in the two new gaps Δ'_i and Δ'_{i+1} remain the same as those of the old gap Δ_i , there cannot be another gap between them. The value in all smaller holes therefore remains correct. Moving Δ'_{i+1} to the last bucket then only needs updating the value of the holes of the intermediate buckets, which touches at most all $\log_B \log_2 G$ buckets spending $\mathcal{O}(\log_B G)$ I/Os in total. Swaps are then performed, where updating the augmentation does not change the asymptotic number of I/Os performed.

Space Usage. To bound the space usage, note that the augmentation of the B-trees at most increase the node size by a constant factor. Since the space usage of a B-tree is linear

in the number of stored elements, and since each gap is contained in a single bucket, the space usage is $\mathcal{O}(G/B)$ blocks in total. This concludes the proof of Theorem 3.

3 The Interval Structure

A gap Δ_i contains all elements in the range the gap covers. Note that by Theorem 1, a query on the gap for an element must be *faster* for elements *closer* to the border of a gap; information-theoretically speaking, such queries reveal less information about the data. It is therefore not sufficient to store the elements in the gap in a single list.

External memory interval data structure. We follow the construction of Sandlund and Wild [44], in their design of the interval structure. In this section we present the construction, and argue on the number of I/Os this construction uses. The main difference from the original construction is in the speed-up provided by the external-memory model; namely that scanning a list is faster by a factor B , and that B-trees allows for more efficient searching. These differences are also what allows for slightly improving the overall I/O cost of the original construction, when moving it to the external-model. Due to space constraints, the full analysis can be found in Appendix A.

We allow for the following operations:

INTERVALSINSERT(e): Inserts element e into the structure.

INTERVALSDELETE(ptr): Deletes the element e at pointer ptr from the structure.

INTERVALSCHANGE(ptr, e'): Changes the element e at pointer ptr to the element e' .

INTERVALSPLIT(e) or INTERVALSPLIT(r): Splits the set of intervals into two sets of intervals at element e or rank r .

A gap has a “*sidedness*”, which denotes the number of sides the gap has had a query. Denote a side as a *queried side*, if that rank has been queried before (cf. [44]). If there have been no queries yet, the (single) gap is a 0-sided gap. When a query in a 0-sided gap occurs, two new gaps are created which are both 1-sided. Note that “1-sided” does not specify which side was queried – left or right. When queries have touched both sides, the gap is 2-sided.

We obtain the following I/O costs for the above operations.

► **Theorem 5 (Interval Data Structure).** *There exists a data structure on an ordered set, maintaining a set of intervals, supporting*

- *INTERVALSINSERT in worst-case $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os,*
- *INTERVALSDELETE in amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os,*
- *INTERVALSCHANGE in worst-case $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os, if the element is moved towards the nearest queried side or amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os otherwise, and*
- *INTERVALSPLIT in amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B} x \log_2 c)$ I/Os.*

Here $|\Delta_i|$ denotes the number of elements contained in all intervals, and x and cx for $c \geq 1$ are the resulting sizes of the two sets created by a split. The space usage is $\mathcal{O}(|\Delta_i|/B)$ blocks.

Intervals in external memory. Let the gap Δ_i contain multiple non-overlapping intervals $I_{i,j}$, which contain the elements located in the gap. The elements of the intervals are sorted between intervals, but not within an interval. Intervals therefore span a range of elements with known endpoints. Each such interval is a blocked-linked-list containing the elements of the interval. Additionally, we store an augmented B-tree over the intervals, allowing for efficiently locating the interval containing a given element (using $\mathcal{O}(\log_B(\#\text{intervals}))$ I/Os).

14 Lazy B-Trees

The B-tree is augmented to hold in each node the total sizes of all intervals in the subtrees of the node, which allows for efficient rank queries.

By packing all intervals into a single blocked-linked-list, and noting that there can be no more intervals than there are elements, the space usage of the intervals and the augmented B-tree over the intervals is $\mathcal{O}(|\Delta_i|/B)$ blocks.

Intuition of interval maintenance of [44]. Intuitively, there is a trade-off in maintaining intervals: having many small intervals reduces future query costs since these are typically dominated by the linear cost of splitting one interval; but it increases the time to search for the correct interval upon insertions (and other operations). Lazy search trees handle this trade-off as follows. To bound the worst case insertion costs, we enforce a hard limit of $\mathcal{O}(\log_2 |\Delta_i|)$ on the number of intervals in a gap Δ_i , implemented via a merging rule. To amortize occasional high costs for queries, we accrue potential for any intervals that have grown “too large” relative to their proximity to a gap boundary.

Given the logarithmic number of intervals, the best case for queries would be to have interval sizes grow exponentially towards the middle. This makes processing of intervals close to the boundary (i.e., close to previous queries) cheap; for intervals close to the middle, we can afford query costs linear in Δ_i . It can be shown that for exponentially growing intervals, the increase of the lower bound from any query allows us to precisely pay for the incurred splitting cost. However, the folklore rule of having each interval, say, 2–4 times bigger than the previous interval seems too rigid to maintain. Upon queries, it triggers too many changes.

Merging & Potential. We therefore allow intervals to grow bigger than they are supposed to be, but charge them for doing so in the potential. By enforcing the merging rule when the number of elements in the gap decreases, we maintain the invariant that the number of intervals is $\mathcal{O}(\log_2 |\Delta_i|)$, which allows for locating an interval using $\mathcal{O}(\log_B \log_2 |\Delta_i|) = \mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os. Enforcing the merging rule is achieved by scanning the intervals using the B-tree, and it can be shown that this operation uses amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i|)$ I/Os. Merging intervals takes $\mathcal{O}(1)$ I/O, but by adding extra potential to intervals, this step of the merge can be amortized away.

Upon this, we can show that INTERVALSINSERT uses $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os, and that INTERVALSDELETE uses amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os. When performing INTERVALSCHANGE, moving the element from one interval to another uses $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os. However, the potential function causes an increase of $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i|)$ I/Os in the amortized cost when an element is moved away from the closest queried side of the gap.

When a query occurs, an interval $I_{i,j}$ is split around some element or rank. The interval can be located using the B-tree over intervals on either element or rank in $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os. For splitting around an element, the interval is scanned and partitioned using $\mathcal{O}(|I_{i,j}|/B)$ I/Os. For splitting around a rank, deterministic selection [12] is applied, which uses $\mathcal{O}(|I_{i,j}|/B)$ I/Os (see, e.g., [17, 18]). In both cases the number of I/Os grows with the interval size. Analyzing the change in the potential function upon this split, we can show that INTERVALSPLIT uses amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B} x \log_2 c)$ I/Os.

This (together with the details from Appendix A) concludes the proof of Theorem 5.

4 Lazy B-Trees

In this section, we combine the gap structure of Section 2 and the interval structure of Section 3, to achieve an external-memory lazy search-tree. Recall our goal, Theorem 1.

► **Theorem 1** (Lazy B-Trees). *There exists a data structure over an ordered set, that supports*

- *CONSTRUCT(S) in worst-case $\mathcal{O}(|S|/B)$ I/Os,*
- *INSERT in worst-case $\mathcal{O}(\log_B \frac{N}{|\Delta_i|} + \log_B \log_B |\Delta_i|)$ I/Os,*
- *DELETE in amortized $\mathcal{O}(\log_B \frac{N}{|\Delta_i|} + \frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os,*
- *CHANGEKEY in worst-case $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os if the element is moved towards the nearest queried element but not past it, and amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os otherwise, and*
- *QUERYELEMENT and QUERYRANK in amortized $\mathcal{O}(\log_B \min\{N, q\} + \frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B} x \log_2 c)$ I/Os.*

Here N denotes the size of the current set, $|\Delta_i|$ denotes the size of the manipulated gap, q denotes the number of performed queries, and x and cx for $c \geq 1$ are the resulting sizes of the two gaps produced by a query. The space usage is $\mathcal{O}(N/B)$ blocks.

We use the I/O bounds shown in Theorems 3 and 5 to bound the cost of the combined operations. These operations are performed as follows.

A CONSTRUCT(S) operation is performed by creating a single gap over all elements in S , and in the gap create a single interval with all elements. This can be done by a single scan of S , and assuming that S is represented compactly/contiguously, this uses $\mathcal{O}(|S|/B)$ I/Os.

To perform an INSERT(e) operation, GAPBYELEMENT is performed to find the gap Δ_i containing element e . Next, INTERVALSINSERT is performed to insert e into the interval structure of gap Δ_i . Finally, GAPINCREMENT is performed on Δ_i , as the size has increased. In total this uses worst-case $\mathcal{O}(\log_B \frac{N}{|\Delta_i|} + \log_B \log_B |\Delta_i|)$ I/Os.

Similarly, upon a DELETE(ptr) operation, INTERVALSDELETE is performed using ptr , to remove the element e at the pointer location. Next GAPDECREMENT is performed on the gap Δ_i . In total this uses amortized $\mathcal{O}(\log_B \frac{N}{|\Delta_i|} + \frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os.

A CHANGEKEY operation may only change an element, s.t. it remains within the same gap (otherwise we need to use DELETE and INSERT). This operation therefore does not need to perform operations on the gap structure, but only on the interval structure of the relevant gap. The operation is therefore performed directly using the INTERVALSCHANGE operation.

The QUERYELEMENT and QUERYRANK operations are performed in similar fashions. First, GAPBYELEMENT or GAPBYRANK is performed to find the relevant gap Δ_i containing the queried element. Next INTERVALSPLIT is performed on the interval structure, which yields two new gaps Δ'_i and Δ'_{i+1} , which is updated into the gap structure using GAPSPLIT. Note that the number of gaps is bounded by the number of elements N , but also by the number of queries q performed, as only queries introduce new gaps. In total, the QUERYELEMENT and QUERYRANK operations uses amortized $\mathcal{O}(\log_B \min\{N, q\} + \frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B} x \log_2 c)$ I/Os.

The space usage of the gap structure is $\mathcal{O}(G/B) = \mathcal{O}(N/B)$ blocks. For gap Δ_i , the space usage of the interval structure is $\mathcal{O}(|\Delta_i|/B)$ blocks. If $|\Delta_j| = \mathcal{O}(B)$, for some Δ_j , we cannot simply sum over all the substructures. By instead representing all such “lightweight” gaps in a single blocked-linked-list, we obtain that the space usage over the combined structure is $\mathcal{O}(N/B)$ blocks. The gaps are located in the combined list using the gap structure, and updates or queries to the elements of the gap may then be performed using $\mathcal{O}(1)$ I/Os, allowing the stated time bounds to hold. Similarly, an interval structure may be created for a gap, when it contains $\Omega(B)$ elements using $\mathcal{O}(1)$ I/Os. The stated time bounds therefore still applies to the lightweight gaps.

This concludes the proof of Theorem 1.

16 Lazy B-Trees

Priority Queue Case. When a lazy B-tree is used as a priority queue, the queries are only performed on the element of smallest rank, and this element is deleted before a new query is performed. If this behavior is maintained, we first note that the structure only ever has a single 1-sided gap of size N , with the queried side to the left. This allows for INSERT to be performed in worst-case $\mathcal{O}(\log_B \log_B N)$ I/Os, as the search time in the gap structure reduces to $\mathcal{O}(1)$. Similarly, DELETE is performed using amortized $\mathcal{O}(\frac{1}{B} \log_2 N + \log_B \log_B N)$ I/Os. To perform the DECREASEKEY operation, a CHANGEKEY operation is performed. As the queried side is to the left, this must move the element towards the closest queried side, leading to a DECREASEKEY operation being performed using worst-case $\mathcal{O}(\log_B \log_B N)$ I/Os. Finally, MINIMUM is performed as a QUERYRANK(r) with $r = 1$. As this must split the gap at $x = 1$ and $cx = N - 1$, the operation is performed using amortized $\mathcal{O}(\frac{1}{B} \log_2 N + \log_B \log_B N)$ I/Os. This concludes the proof of Corollary 2.

5 Open Problems

The internal lazy search trees [44] discuss further two operations on the structure; *SPLIT* and *MERGE*, which allows for splitting or merging disjoint lazy search tree structures around some query element or rank. These operations are supported as efficient as queries in the internal-model. In the external-memory case, the gap structure designed in this paper, however, does not easily allow for splitting or merging the set of gaps around an element or rank, as the gaps are stored by element in disjoint buckets, where all buckets must be updated upon executing these extra operations. By sorting and scanning the gaps, the operations may be supported, but not as efficiently as a simple query, but instead in sorting time relative to the number of gaps. It remains an open problem to design a gap structure, which allows for efficiently supporting *SPLIT* and *MERGE*.

In the external-memory model, bulk operations are generally faster, as scanning consecutive elements saves a factor B I/Os. One such operation is *RANGEQUERY*, where a range of elements may be queried and reported at once. In a B-tree, this operation is supported in $\mathcal{O}(\log_B N + k/N)$ I/Os, when k elements are reported. The lazy B-tree designed in this paper allows for efficiently reporting the elements of a range in unsorted order, by querying the endpoints of the range, and then reporting all elements of the gaps between the endpoints. Note that for the priority-queue case, this allows reporting the k smallest elements in $\mathcal{O}(\frac{1}{B} k \log_2 \frac{N}{k} + \frac{1}{B} \log_2 N + \log_B \log_B N)$ I/Os. If the elements must be reported in sorted order, sorting may be applied to the reported elements. However, this effectively queries all elements of the range, and should therefore be reflected into the lazy B-tree. As this introduces many gaps of small size, the I/O cost increases over simply sorting the elements of the range. It remains an open problem on how to efficiently support sorted *RANGEQUERY* operations, while maintaining the properties of lazy B-trees.

In the internal-memory model, an optimal version of lazy search trees was constructed [46], which gets rid of the added $\log \log N$ term on all operations. It remains an open problem to similarly get rid of the added $\log_B \log_B N$ term for the external-memory model.

Improvements on external-memory efficient search trees and priority queues use buffering of updates to move more data in a single I/O, thus improving the I/O cost of operations. The first hurdle to overcome in order to create buffered lazy B-trees is creating buffered biased trees used for the gap structure. By buffering updates to the gaps, it must then hold that the weights are not correctly updated, which imposes problems on searching; both by element and rank. It remains an open problem to overcome this first hurdle as a step towards buffering lazy B-trees.

References

- 1 Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)*, 146:263–266, 1962.
- 2 Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988. doi:10.1145/48529.48535.
- 3 Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Workshop on Algorithms and Data Structures (WADS)*, page 334–345. Springer, 1995. doi:10.1007/3-540-60220-8_74.
- 4 Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. doi:10.1007/S00453-003-1021-X.
- 5 Amitabha Bagchi, Adam L Buchsbaum, and Michael T Goodrich. Biased skip lists. *Algorithmica*, 42:31–48, 2005.
- 6 Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jon Sorenson. Dynamic online multiselection in internal and external memory. In *WALCOM: Algorithms and Computation*, pages 199–209. Springer, 2015. doi:10.1007/978-3-319-15612-5_18.
- 7 Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jon Sorenson. Near-optimal online multiselection in internal and external memory. *Journal of Discrete Algorithms*, 36:3–17, 2016. WALCOM 2015.
- 8 Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683.
- 9 Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985.
- 10 Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976. doi:10.1016/0020-0190(76)90071-5.
- 11 Alka Bhushan and Sajith Gopalan. *External Memory Soft Heap, and Hard Heap, a Meldable Priority Queue*, page 360–371. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-32241-9_31.
- 12 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.
- 13 Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and B-trees. In *Symposium on Discrete Algorithms (SODA)*, pages 1106–1114. SIAM, 2008.
- 14 Gerth Stølting Brodal. Worst-case efficient priority queues. In *Symposium on Discrete Algorithms (SODA)*, 1996.
- 15 Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554. Society for Industrial and Applied Mathematics, 2003.
- 16 Gerth Stølting Brodal and Jyrki Katajainen. Worst-case efficient external-memory priority queues. In Stefan Arnborg and Lars Ivansson, editors, *Algorithm Theory — SWAT’98*, pages 107–118, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. doi:10.1007/BFb0054359.
- 17 Gerth Stølting Brodal and Sebastian Wild. Funnelselect: Cache-oblivious multiple selection. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *European Symposium on Algorithms (ESA)*, pages 25:1–25:17, 2023. doi:10.4230/LIPIcs.ESA.2023.25.
- 18 Gerth Stølting Brodal and Sebastian Wild. Deterministic cache-oblivious funnelselect. In *Scandinavian Symposium on Algorithm Theory (SWAT)*, 2024. arXiv:2402.17631, doi:10.4230/LIPIcs.SWAT.2024.17.
- 19 Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict fibonacci heaps. In *Symposium on Theory of Computing (STOC)*, STOC’12, page 1177–1184. ACM, May 2012. doi:10.1145/2213977.2214082.

18 Lazy B-Trees

- 20 Yu-Tai Ching, Kurt Mehlhorn, and Michiel H.M. Smid. Dynamic deferred data structuring. *Information Processing Letters*, 35(1):37 – 40, 1990.
- 21 Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Pătraşcu. The geometry of binary search trees. In *Symposium on Discrete Algorithms SODA 2009*, pages 496–505. SIAM, 2009.
- 22 David Dobkin and J. Ian Munro. Optimal time minimal space selection algorithms. *Journal of the Association for Computing Machinery*, 28(3):454–461, 1981.
- 23 Kasper Eenberg, Kasper Green Larsen, and Huacheng Yu. Decreasekeys are expensive for external memory priority queues. In *Symposium on Theory of Computing (STOC)*, page 1081–1093. ACM, June 2017. doi:10.1145/3055399.3055437.
- 24 Joan Feigenbaum and Robert E Tarjan. Two new kinds of biased search trees. *Bell System Technical Journal*, 62(10):3139–3158, 1983.
- 25 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987. doi:10.1145/28869.28874.
- 26 Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3.
- 27 Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: towards robust adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment*, 5(6):502–513, February 2012. doi:10.14778/2168651.2168652.
- 28 John Iacono, Riko Jacob, and Konstantinos Tsakalidis. External memory priority queues with decrease-key and applications to graph algorithms. In *European Symposium on Algorithms (ESA)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ESA.2019.60.
- 29 Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, 2010.
- 30 Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Conference on Innovative Data Systems Research (CIDR)*, 2007.
- 31 Stratos Idreos, Stefan Manegold, and Goetz Graefe. Adaptive indexing in modern database kernels. In *International Conference on Extending Database Technology (EDBT)*, EDBT ’12, page 566–569. ACM, March 2012. doi:10.1145/2247596.2247667.
- 32 Shunhua Jiang and Kasper Green Larsen. A faster external memory priority queue with decreasekeys. In *Symposium on Discrete Algorithms (SODA)*, page 1331–1343. Society for Industrial and Applied Mathematics, January 2019. doi:10.1137/1.9781611975482.81.
- 33 Kanella Kaligosi, Kurt Mehlhorn, J. Ian Munro, and Peter Sanders. Towards optimal multiple selection. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 103–114. Springer, 2005. doi:10.1007/11523468_9.
- 34 Haim Kaplan, László Kozma, Or Zamir, and Uri Zwick. Selection from heaps, row-sorted matrices, and $x + y$ using soft heaps. In *Symposium on Simplicity in Algorithms (SOSA)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/OASICS.SOSA.2019.5.
- 35 Richard M. Karp, Rajeev Motwani, and Prabhakar Raghavan. Deferred data structuring. *SIAM Journal on Computing*, 17(5):883–902, 1988.
- 36 Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *International Conference on Management of Data, SIGMOD/PODS ’18*, page 489–504. ACM, May 2018. doi:10.1145/3183713.3196909.
- 37 Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing, SPDP 1996, New Orleans, Louisiana, USA, October 23-26, 1996*, pages 169–176. IEEE Computer Society, 1996. doi:10.1109/SPDP.1996.570330.
- 38 Joan M. Lucas. Canonical forms for competitive binary search tree algorithms. Technical Report DCS-TR-250, Rutgers University, 1988.

- 39 Conrado Martínez and Salvador Roura. Randomized binary search trees. *J. ACM*, 45(2):288–323, 1998. doi:10.1145/274787.274812.
- 40 Kurt Mehlhorn. *Data Structures and Efficient Algorithms, Volume 1: Sorting and Searching*. Springer, 1984.
- 41 J. Ian Munro. On the competitiveness of linear search. In *ESA 2000*, pages 338–345. Springer, 2000. doi:10.1007/3-540-45253-2_31.
- 42 J. Ian Munro, Richard Peng, Sebastian Wild, and Lingyi Zhang. Dynamic optimality refuted – for tournament heaps. working paper, 2019. URL: <https://www.wild-inter.net/publications/unordered-dynamic-optimality>, arXiv:1908.00563.
- 43 Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin Kersten. Database cracking: fancy scan, not poor man’s sort! In *International Workshop on Data Management on New Hardware (DaMoN)*, SIGMOD/PODS’14, page 1–8. ACM, June 2014. doi:10.1145/2619228.2619232.
- 44 Bryce Sandlund and Sebastian Wild. Lazy search trees. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 704–715. IEEE, 2020. doi:10.1109/FOCS46700.2020.00071.
- 45 Bryce Sandlund and Sebastian Wild. Lazy search trees, 2020. arXiv:2010.08840.
- 46 Bryce Sandlund and Lingyi Zhang. Selectable heaps and optimal lazy search trees. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1962–1975, 2022. doi:10.1137/1.9781611977073.78.
- 47 Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. doi:10.1007/BF01940876.
- 48 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.
- 49 Yufei Tao. Maximizing the optimality streak of deferred data structuring (a.k.a. database cracking). In *International Conference on Database Theory (ICDT)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.ICDT.2025.10.
- 50 Fatemeh Zardbani, Peyman Afshani, and Panagiotis Karras. Revisiting the theory and practice of database cracking. In *International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2020. doi:10.5441/002/EDBT.2020.46.

A

 The Interval Structure

In this appendix, we give the full description and detailed analysis of the interval data structure outlined in Section 3. We follow the construction of Sandlund and Wild [44], in their design of the interval structure, with some improvements for external memory. To give a self-contained analysis of the number of I/Os used by all operations, we reproduce the analysis from [44, 45] and spell out all cases explicitly.

A gap Δ_i contains all elements in the range the gap covers. Recall that a query on the gap for an element must be faster for elements closer to the border of a gap. We allow for the following operations:

INTERVALSINSERT(e): Inserts element e into the structure.

INTERVALSDELETE(ptr): Deletes the element e at pointer ptr from the structure.

INTERVALSCHANGE(ptr, e'): Changes the element e at pointer ptr to the element e' .

INTERVALSPLIT(e) or INTERVALSPLIT(r): Splits the set of intervals into two sets of intervals at element e or rank r .

A gap has a “sidesness”, which denotes the number of sides the gap has had a query. Denote a side as a *queried side*, if that rank has been queried before (cf. [44]). If there have been no queries yet, the (single) gap is a 0-sided gap. When a query in a 0-sided gap occurs, two new gaps are created which are both 1-sided. Note that both gaps which have a single query on either the left or right side is called 1-sided. When queries have touched both sides, the gap is 2-sided.

We obtain the following I/O costs for the above operations.

► **Theorem 6.** *There exists a data structure on an ordered set, that maintains a set of intervals, supporting*

- *INTERVALSINSERT in worst-case $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os,*
- *INTERVALSDELETE in amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os,*
- *INTERVALSCHANGE in worst-case $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os, if the element is moved towards the nearest queried side or amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os otherwise, and*
- *INTERVALSPLIT in amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B} x \log_2 c)$ I/Os.*

Here $|\Delta_i|$ denotes the number of elements contained in all intervals, and x and cx for $c \geq 1$ are the resulting sizes of the two sets produced by a split. The space usage is $\mathcal{O}(|\Delta_i|/B)$ blocks.

The structure is as follows. Let the gap Δ_i contain multiple non-overlapping intervals $I_{i,j}$, which contains the elements located in the gap. The elements of the intervals are sorted between intervals, but not within an interval. Intervals therefore span a range of elements with known endpoints. Each such interval is a blocked-linked-list containing the elements of the interval. Additionally, we store an augmented B-tree over the intervals, allowing for efficiently locating the interval containing a given element (using $\mathcal{O}(\log_B (\#\text{intervals}))$ I/Os). The B-tree is augmented to hold in each node the total size of all intervals in the subtree of the node, which allows for efficient rank queries.

By packing all intervals into a single blocked-linked-list, and noting that there can be no more intervals than there are elements, the space usage of the intervals and the augmented B-tree over the intervals is $\mathcal{O}(|\Delta_i|/B)$ blocks.

A.1 Invariants and Potential

Intuitively, there is a trade-off in maintaining intervals: having many small intervals reduces future query costs (typically dominated by the linear cost of splitting one interval), but

increases the time to search for the correct interval upon insertions (and other operations). Lazy search trees handle this trade-off as follows. To bound the worst case insertion costs, we enforce a hard limit number on the number of intervals via a merging rule. To amortize occasional high costs for queries, we accrue potential ($\Phi_{i,j}$ below) for any intervals that are “too large” relative to their proximity of a gap boundary.

Moreover, for queries we would like to have interval sizes grow exponentially towards the middle, as this allows for efficiently manipulating the intervals closest to the edge. This folklore rule seems too rigid to maintain upon queries, though, as many changes can be necessary.

A.1.1 Merging Intervals

We maintain the following invariant on the number of intervals:

Invariant (#Int): For a gap Δ_i , the number of intervals $I_{i,j}$ is at most $4 \log_2 |\Delta_i| + 2$.

This allows for locating an interval using $\mathcal{O}(\log_B \log_2 |\Delta_i|) = \mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os. This I/O cost also applies to recomputing the sizes of subtrees when updating an interval. To bound the I/O cost of the operations, we define the following for the analysis. Let $\mathbf{o}(I_{i,j})$, the “outside (elements) of $I_{i,j}$ ”, denote the number of elements between interval $I_{i,j}$ and a *queried* side of the gap; if both sides have been queried, $\mathbf{o}(I_{i,j})$ is the towards the *closest* side. We denote an interval to be *left* if the closest edge is to the left and *right* otherwise. Note for a 0-sided gap, the outside is always 0. For a 1-sided gap with a query on the left (resp. right), all intervals are left (resp. right).

When an insertion occurs in the gap, the interval containing the inserted element is located, and the inserted element is appended to the list of the interval. This increases $|\Delta_i|$ by one, but does not create any new intervals, so (#Int) remains satisfied.

When a deletion occurs, $|\Delta_i|$ decreases, which may invalidate (#Int). We therefore need a way to reduce the number of intervals, which we do by simply *merging* two adjacent intervals. Recall that elements within intervals are not sorted, so merging two intervals effectively just means “forgetting” about the pivot element currently separating them; here, we need to concatenate the list of elements. The rule in lazy search trees is as follows, using the outside as a measure for how close to the middle an interval is:

Merge Rule (M): If $|I_{i,j}| + |I_{i,j+1}| < \mathbf{o}(I_{i,j})$, merge $I_{i,j}$ and $I_{i,j+1}$.

This rule only applies between intervals of the same type; a left interval is never merged with a right interval.

That is, an interval $I_{i,j}$ is merged with its inner neighbor, if they combined contain fewer elements than outside of the intervals; they are therefore intuitively “too small” for where they reside in the gap. Note that we do *not* treat this rule as an invariant; we do not require it to be true at all times, but only use it to reduce the number of intervals when explicitly invoked in operation INTERVALSMERGE.

Suppose that INTERVALSMERGE has just merged all interval pairs that satisfy (M). We then obtain bounds on the number of intervals as stated in the following two lemmas.

► **Lemma 7** (The Merge rule on one side yields logarithmic internals). *Let a set of N elements partitioned in intervals that satisfy rule (M). Let further I be one of the intervals and $k = \mathbf{o}(I)$ be its number of outside elements. Then there are at most $\max\{1, 2 \log_2 \frac{N}{k} + 2\}$ intervals inside of I and on the same side as I .*

22 Lazy B-Trees

Proof. The proof goes by induction in N . When $N = 1$, there can be at most one interval. If $N < k$, there can be at most one interval, as the merge rule would otherwise be violated. In both cases, the claim is satisfied.

For $N \geq k$, we may assume the induction hypothesis on all smaller values of N . If there are less than three intervals, the bound holds from the additive constant. Otherwise, let a , b and c be the size of the first three intervals in order. By the merge rule, it holds that $a + b \geq k$. By definition $\mathbf{o}(I_c) = k + a + b \geq 2k$. The number of intervals is then the two intervals a and b , and the number of intervals from c , which by induction is $\leq 2 + 2 \log_2 \frac{N-a-b}{\mathbf{o}(I_c)} + 2 \leq 2 \log_2 \frac{N}{2k} + 4 = 2 \log_2 \frac{N}{k} + 2$. This concludes the proof. $\blacktriangleleft$

► **Lemma 8** ((M) implies (#Int)). *Let a gap Δ_i adhere to rule (M) on intervals. Then there are at most $4 \log_2 |\Delta_i| + 2$ intervals.*

Proof. The proof goes by case analysis on the sidesness of Δ_i . If the gap is 0-sided, there must be one interval. If the gap is 1-sided, the first interval has size at least 1, and there must then be at most $2 \log_2 \Delta_i + 2$ remaining intervals by Lemma 7. In both cases the bound holds.

Otherwise, the gap must be 2-sided, and the merge rule holds from both left and right. The outermost intervals have size at least 1. Let L and R denote the number of elements in left and right intervals respectively. By Lemma 7, the number of intervals is at most

$$2 + (2 \log_2 L + 2) + (2 \log_2 R + 2) = 6 + 2 \log_2 (L \cdot R) \leq 6 + 2 \log_2 \frac{\Delta_i^2}{4} = 4 \log_2 \Delta_i + 2. \quad \blacktriangleleft$$

A.1.2 Potential

When a query occurs, an interval $I_{i,j}$ is split around some element or rank. The interval can be located using the B-tree over intervals on either element or rank in $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os. For splitting around an element, the interval is scanned and partitioned using $\mathcal{O}(|I_{i,j}|/B)$ I/Os. For splitting around a rank, deterministic selection [12] is applied, which uses $\mathcal{O}(|I_{i,j}|/B)$ I/Os (see, e.g., [17, 18]). In both cases the number of I/Os grows with the interval size.

We therefore need a potential function, which balances the time to split an interval with how close to the border it is, which is defined using the outside of an interval. For 1-sided gaps, the outside however only measures toward one side of the gap, and we therefore need further potential for the 1-sided intervals. Let N_{01} be the number of elements contained in total in 0- and 1-sided gaps. Further, we need potential to efficiently merge intervals in a scan.

We let the potential be

$$\Phi = \frac{1}{B} N_{01} + \sum_{i,j} \Phi_{i,j}$$

and let

$$\Phi_{i,j} = 1 + \frac{1}{B} \max \{ |I_{i,j}| - \mathbf{o}(I_{i,j}), 0 \}.$$

Using this potential function, we can analyze the amortized I/O cost of applying the merge rule on all intervals, which we denote as the INTERVALSMERGE operation. This operation uses that scanning all keys of a B-tree can be done efficiently.

► **Lemma 9** (Merging runtime). *Let the number of intervals contained in the gap Δ_i be k . Then INTERVALSMERGE uses amortized $\mathcal{O}(k/B)$ I/Os.*

Proof. When INTERVALSMERGE is performed, first $\mathbf{o}(I_{i,j})$ is computed for each $I_{i,j}$ in $\mathcal{O}(k/B)$ I/Os, by scanning the B-tree. The B-tree may then be scanned again, to perform the merges on the intervals, and finally a new B-tree over the resulting intervals can be built. As the number of intervals can only decrease, the B-tree can be build using $\mathcal{O}(k/B)$ I/Os.

Let $I_{i,j}$ and $I_{i,j+1}$ be a pair of intervals which are merged during the operation. It must hold that $|I_{i,j}| + |I_{i,j+1}| < \mathbf{o}(I_{i,j})$. The merge is performed by following the pointers to the intervals and concatenating the lists, using $\mathcal{O}(1)$ I/O. Let $I_{i,j'}$ be the resulting interval of the merge. The potential then decreases by $\Phi_{i,j}$ and $\Phi_{i,j+1}$, and increases by $\Phi_{i,j'}$. From the merge rule and the definition of the outside, it follows that

$$\begin{aligned}\Phi_{i,j} &= 1 + \frac{1}{B} \max \{|I_{i,j}| - \mathbf{o}(I_{i,j}), 0\} &= 1 \\ \Phi_{i,j+1} &= 1 + \frac{1}{B} \max \{|I_{i,j+1}| - (\mathbf{o}(I_{i,j}) + |I_{i,j}|), 0\} &= 1 \\ \Phi_{i,j'} &= 1 + \frac{1}{B} \max \{(|I_{i,j}| + |I_{i,j+1}|) - \mathbf{o}(I_{i,j}), 0\} &= 1.\end{aligned}$$

The difference in potential is therefore $\Delta\Phi = -1$, which covers the I/Os performed by the merge, which concludes the proof. $\blacktriangleleft$

A.2 Updating Elements of the Intervals

Upon an INTERVALSINSERT operation, a new element e is added to the gap Δ_i . The interval $I_{i,j}$ containing e can be located using the B-tree, and the size of the interval incremented by 1 using $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os. As previously discussed, the intervals need not be merged, as the invariant on the interval count is satisfied. In the potential function Φ , the outside of all inner intervals grows by 1, which does not increase the value of Φ . The size of $I_{i,j}$ grows by 1, and N_{01} grows by at most 1, which in total increases Φ by at most $\frac{2}{B}$. INTERVALSINSERT can therefore be performed in both amortized and worst-case $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os.

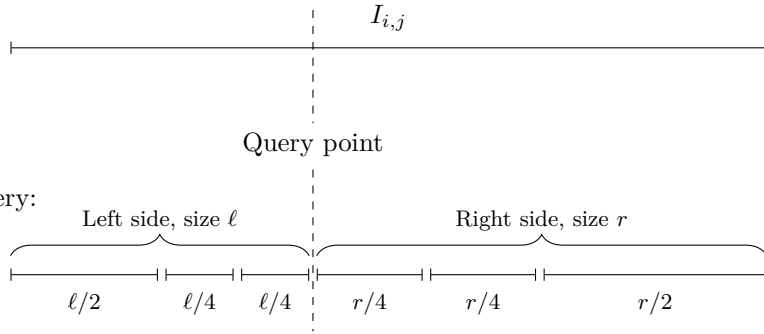
When an INTERVALSDELETE operation occurs, a pointed to element is removed from some interval $I_{i,j}$. The size is then decremented by 1, and the B-tree must be updated to reflect so. If this decreases the size to 0, the interval is removed, which in total uses $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os. In contrast to the INTERVALSINSERT operation, upon removing an element, the outside of all inner intervals decreases, which makes the potential function Φ grow by at most $\frac{1}{B}$ for each such interval, of which there are $\mathcal{O}(\log_2 |\Delta_i|)$. In addition, the number of elements decreases without the number of intervals decreasing, which may leading to breaking the invariant on the bound on the number of intervals. Note that merging a single interval is enough to ensure that the number of intervals stays bounded, and even so, only after linearly many deletes, the merge of a single interval needs to occur. However, as the potential increases by $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i|)$, and the INTERVALSMERGE uses the same amount of amortized I/Os, it is efficient enough to run INTERVALSMERGE after every delete. In total, INTERVALSDELETE uses amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os.

When INTERVALSCHANGE(ptr, e') is performed, the element e at pointer ptr is changed to element e' . This new element must then be placed in the correct interval, which can be done using the B-tree over intervals using $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os. This then changes the size of at most two intervals, with one increasing and the other decreasing, which increases the potential by at most $\frac{1}{B}$. This may also change the outside size of intervals. If the element is moved towards the closest queried side, then the outside may increase, which does not increase the potential. Otherwise, the outside may grow by one, for each interval the element is moved past, which increases the potential by at most $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i|)$. In total,



■ **Figure 5** The query splits the gap in two, where some side contains x elements and the other contains cx elements, for some $c \geq 1$.

Interval before query:



Intervals after query:

■ **Figure 6** A query in interval $I_{i,j}$ splits the interval into six new intervals.

INTERVALSCHANGE uses worst-case $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os, if the element is moved towards the closest queried side, and otherwise amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|)$ I/Os.

A.3 Splitting the Intervals

When a query occurs in gap Δ_i , the gap is split at the query element into two new gaps Δ'_i and Δ'_{i+1} . Let the resulting gap sizes be x and cx for some $c \geq 1$ (see Figure 5). We assume here and throughout that the left resulting gap has size x ; the other case is symmetric.

Upon a INTERVALSPLIT operation, the interval $I_{i,j}$ containing the query element must be split, and the remaining intervals distributed into the two new gaps. First, INTERVALSMERGE is applied on gap Δ_i , to ensure structure on the intervals. Then interval $I_{i,j}$ is split around the query element into a left and right side of elements with respectively smaller and larger elements. Let ℓ be the number of elements on the left side. Using deterministic selection, the left side is partitioned into three intervals of sizes $\ell/2$, $\ell/4$ and $\ell/4$, such that the sizes are doubling from the query point and outwards. The right side is similarly split, resulting in six new intervals covering the elements of $I_{i,j}$. See Figure 6 for an illustration of this split. From these new intervals, along with the remaining intervals of gap Δ_i , two new gaps Δ'_i and Δ'_{i+1} are created, containing the intervals to the left and right side of the query point respectively. To finally ensure invariant (#Int) on the two new gaps, we lastly apply INTERVALSMERGE on them both, which concludes the steps of the INTERVALSPLIT operation.

We next analyze the amortized I/O cost of INTERVALSPLIT. The interval $I_{i,j}$ can be located using the B-tree in $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os. Splitting the interval is done using a constant number of scans and deterministic selections on the interval, which uses $\mathcal{O}(\frac{1}{B} |I_{i,j}|)$ I/Os in total. When performing INTERVALSMERGE on Δ_i , Δ'_i , and Δ'_{i+1} , then by Lemma 9, the invariant on the number of intervals (#Int), and that splitting interval $I_{i,j}$ creates $\mathcal{O}(1)$ new intervals, it holds that these merges uses amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i|)$ I/Os in total. Note that

these costs is independent on where $I_{i,j}$ resides in the structure, and the sidesness of gap Δ_i . The change in potential however depends heavily on these properties. We shall therefore use case analysis for the change in potential. These cases are on if the split interval is on the left or right, and on the sidesness of the gap Δ_i . We only analyze the difference in potential under the splitting of interval $I_{i,j}$, as the INTERVALSMERGE operation already is amortized.

A.3.1 Gap Δ_i is 2-sided.

The split must result in two new 2-sided gaps. Therefore, N_{01} is unchanged, and can be disregarded for the potential difference. As the gap is 2-sided, $\mathbf{o}(I_{i,j})$ is the smaller distance to the two sides of the gap. It holds that $I_{i,j}$ overlaps the query point, and therefore the distance to the left side (in Δ_i) is at most x , and the distance to the right side (in Δ_i) is at most cx . Therefore, $\mathbf{o}(I_{i,j}) \leq x$.

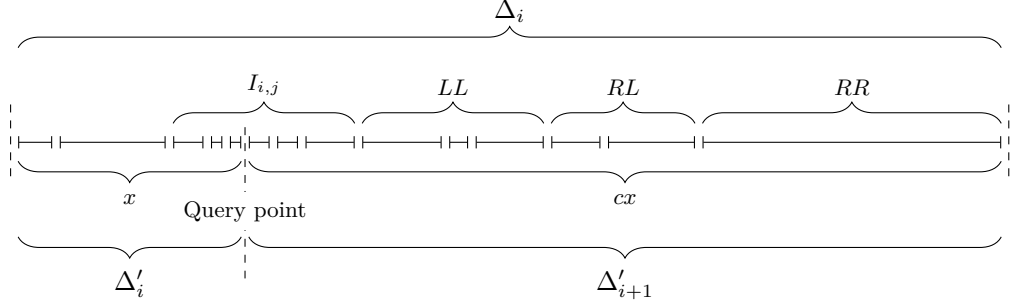
There are three types of intervals: the intervals resulting from splitting $I_{i,j}$, the intervals to the left of $I_{i,j}$ in Δ_i , and the intervals to the right of $I_{i,j}$ in Δ_i . Let us first consider the potential on the intervals resulting from splitting $I_{i,j}$. We denote this change in potential as $\Delta\Phi_I$. The potential on $I_{i,j}$ is $\Phi_{i,j}$, which is removed. Then six new intervals are introduced. Consider the intervals created on the left side of the query point, and let the total number of elements in these three intervals be ℓ . The rightmost one of these intervals is next to a query point in the new gap, and thus its outside is 0. The potential on this interval is therefore $\frac{1}{B}\ell/4$. The middle interval has the right interval as its outside, and as they are both of size $\ell/4$, there is no potential on this interval. The leftmost interval may have either side as its outside; in the worst-case, it has a potential of $\frac{1}{B}\ell/2$. This holds symmetrically for the new intervals on the right. As the size of the left and right in total is $|I_{i,j}|$, the total worst-case potential on the new intervals is $\frac{3}{4}\frac{1}{B}|I_{i,j}|$. In total, the change in potential on the intervals created from $I_{i,j}$ is

$$\begin{aligned} \Delta\Phi_I &\leq 5 && \text{The number of intervals created vs removed} \\ &- \frac{1}{B} \max\{|I_{i,j}| - \mathbf{o}(I_{i,j}), 0\} && \text{Removing } I_{i,j} \\ &+ \frac{3}{4}\frac{1}{B}|I_{i,j}| && \text{Introducing new intervals} \end{aligned}$$

To simplify this, we distinguish cases on the size of $I_{i,j}$, and compare it to the I/O cost of splitting the interval. If $|I_{i,j}| \leq x$, then $\Delta\Phi \leq 5 + \frac{3}{4}\frac{1}{B}x = \mathcal{O}(\frac{1}{B}x)$. Otherwise, $\Delta\Phi \leq 5 + \frac{1}{B}x - \frac{1}{4}\frac{1}{B}|I_{i,j}|$. The amortized cost of the split, including only the potential on the intervals of $I_{i,j}$, is therefore $\mathcal{O}(\frac{1}{B}x)$ I/Os. The total amortized cost must also cover the remaining intervals.

For the intervals to the left of $I_{i,j}$, let the change in potential be denoted as $\Delta\Phi_L$. In total there are at most x elements located on the left side. As there is no new intervals introduced, the potential function on the left intervals is therefore bounded by the number of elements, and thus $\Delta\Phi_L \leq \frac{1}{B}x$. For the change in potential for the intervals on the right of $I_{i,j}$, denoted $\Delta\Phi_R$, we must further distinguish whether $I_{i,j}$ is a left or right interval.

Case 1: $I_{i,j}$ is a left interval. We partition the intervals on the right into three categories: the intervals that were a left interval in Δ_i and remain a left interval in Δ'_{i+1} (LL), the intervals that were right intervals in Δ_i and become left intervals in Δ'_{i+1} (RL), and the intervals that were right intervals in Δ_i and remain right intervals in Δ'_{i+1} (RR). Note that there cannot be any intervals that were left and became right, as the number of elements



■ **Figure 7** The categories of intervals upon splitting a 2-sided gap, with $I_{i,j}$ being a left interval.

to the left decreases; if this was the side with the fewer elements before, it must remain so. These categories are illustrated on Figure 7.

First note that the number of intervals and their sizes do not change from Δ_i to Δ'_{i+1} . The only change is in the size of the outsides. For the intervals in RR , their outside does not change, and therefore their potential does not change.

Note that INTERVALSMERGE was performed before the split, and therefore rule (M) is satisfied on the intervals in Δ_i . Also note that splitting another interval does not alter the outside values of other intervals. Therefore, for the intervals in LL , x elements are removed from their outside in Δ'_{i+1} . We must then bound the number of intervals in LL , to bound the change in the potential. As the outside of the leftmost interval in LL has an outside of at least x in Δ_i , and there are at most cx elements in LL , as they are all contained in Δ'_{i+1} , then by Lemma 7, the number of intervals in LL is at most $2 \log_2 \frac{cx}{x} + 2 = \mathcal{O}(\log_2 c)$. The potential increase of LL is therefore bounded by $\mathcal{O}(\frac{1}{B}x \log_2 c)$.

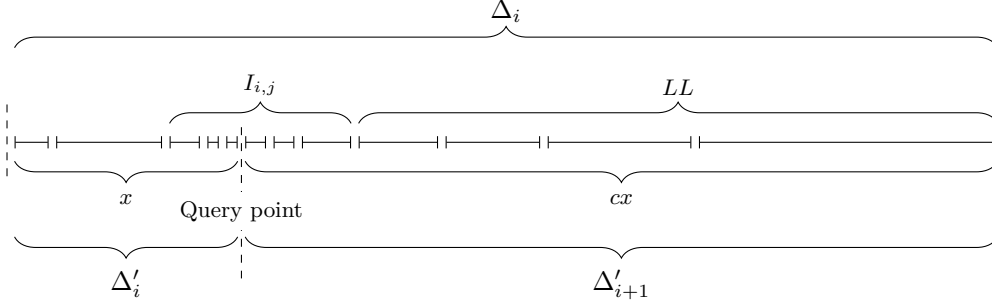
For the intervals in RL , as they were a right interval before in Δ_i , but become a left interval in Δ'_{i+1} due to x elements being removed from the left, their outside can decrease by at most x , when moved to Δ'_{i+1} . Consider all but the rightmost interval of RL . As these are left intervals of Δ'_{i+1} without the last left interval, it must hold that they are contained entirely in the left half of the cx elements of Δ'_{i+1} . As they are right intervals in Δ_i , then their old outside is to the right in Δ_i , and therefore the outside of the last considered interval is in Δ_i at least the right half of the cx elements. By Lemma 7, the number of intervals in RL , including the rightmost interval, is bounded by $2 \log_2 \frac{cx/2}{cx/2} + 2 + 1 = 3$. The potential increase of RL is therefore bounded by $\frac{3}{B}x$.

In total, $\Delta\Phi_R = \mathcal{O}(\frac{1}{B}x \log_2 c)$.

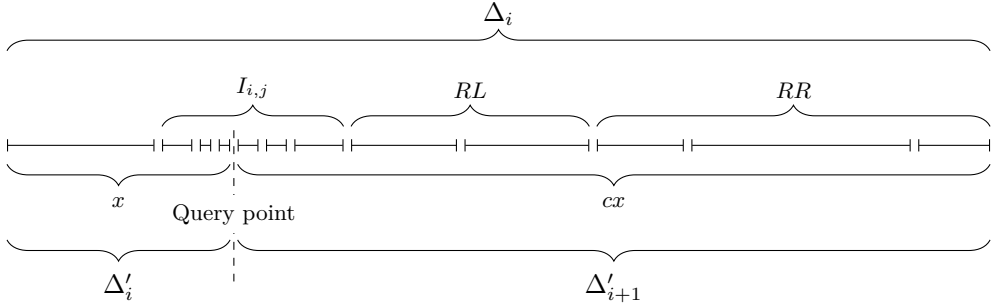
Case 2: $I_{i,j}$ is a right interval. As there are more elements to the left of $I_{i,j}$ than to the right in Δ_i , there are at most x elements remaining to the right of $I_{i,j}$. As there are no new intervals introduced, the potential function on the right intervals is therefore bounded by the number of elements, and thus $\Delta\Phi_R \leq \frac{1}{B}x$.

A.3.2 Gap Δ_i is 0-sided.

As there have been no queries to the left or right of the gap, then there can only be a single interval in Δ_i , which is $I_{i,j}$. The bottom of Figure 6 therefore illustrates this case. Upon splitting this interval, two 1-sided gaps are created. All elements are moved into these gaps from Δ_i , and therefore N_{01} is unchanged, and can be disregarded for $\Delta\Phi$. For both gaps, the queried side is at the query point. Therefore, the outside of each interval in Δ'_i and Δ'_{i+1}



■ **Figure 8** The categories of intervals upon splitting a 1-sided gap, with left queried side.



■ **Figure 9** The categories of intervals upon splitting a 1-sided gap, with right queried side.

is towards the middle. Following the analysis of $\Delta\Phi_I$ in the 2-sided case, it holds that the new potential is $\frac{1}{4} \frac{1}{B} |I_{i,j}|$, as the outside of the outermost intervals is towards the middle, and therefore $\Delta\Phi = 5 - \frac{3}{4} \frac{1}{B} |I_{i,j}|$.

Gap Δ_i is 1-sided.

As before, we assume for the current query, that the left resulting new gap has size x . If in Δ_i , the queried side was the left side, then gap Δ'_i becomes a 2-sided gap, and Δ'_{i+1} remains a 1-sided gap, with the queried side on the left. This case is illustrated on Figure 8. The outside of $I_{i,j}$ in Δ_i remains bounded by x , and the analysis on the change in potential of $I_{i,j}$ remains the same as in the 2-sided gap analysis, which bounds the potential increase by $\mathcal{O}(\frac{1}{B}x)$. For the intervals on the left, the potential may be at most $\frac{1}{B}x$, as in the 2-sided case. However, N_{01} decreases by x , which bounds the increase in potential on the left to at most 0. For the intervals on the right, it must hold that they were all left intervals in Δ_i and that they remain left intervals in Δ'_{i+1} . Following the argument for LL from the 2-sided gap case above, the increase in potential is bounded by $\mathcal{O}(\frac{1}{B}x \log_2 c)$.

The case when the queried side in Δ_i is to the right is illustrated on Figure 9. In this case, the outside of $I_{i,j}$ is no longer bounded by x , but by cx . However, all elements on the right leave the 1-sided gap Δ_i , and enter the 2-sided gap Δ'_{i+1} ; therefore N_{01} decreases by cx , and so the potential decreases by $\frac{1}{B}cx$. The intervals on the right must have been right intervals in Δ_i , and may be both left or right intervals in Δ'_{i+1} . Using the same analysis as in the 2-sided case for the RL and RR intervals, it holds that the potential on the right interval increase by at most $\frac{3}{B}x$. The intervals to the left spans at most x elements, and the

28 Lazy B-Trees

potential on these are at most $\frac{1}{B}x$, as in the 2-sided case. In total

$\Delta\Phi \leq 5$	The number of intervals created vs removed
$-\frac{1}{B} \max\{ I_{i,j} - o(I_{i,j}), 0\}$	Removing $I_{i,j}$
$+\frac{3}{4} \frac{1}{B} I_{i,j} $	Introducing new intervals
$-\frac{1}{B} cx$	Elements removed from 1-sided gap
$+\frac{3}{B}x + \frac{1}{B}x$	Potential change on the remaining intervals

Recall that $\mathbf{o}(I_{i,j}) \leq cx$. We then case analysis on the size of $I_{i,j}$. If $|I_{i,j}| \leq cx$, then $\Delta\Phi \leq 5 - \frac{1}{4} \frac{1}{B} cx + \frac{4}{B}x$. As splitting the interval uses $\mathcal{O}(\frac{1}{B} |I_{i,j}|)$ I/Os, the amortized cost of this case is $\mathcal{O}(\frac{1}{B}x)$ I/Os. Otherwise, if $|I_{i,j}| > cx$, then $\Delta\Phi \leq 5 - \frac{1}{4} \frac{1}{B} |I_{i,j}| + \frac{4}{B}x$, and the amortized cost of this case is therefore also $\mathcal{O}(\frac{1}{B}x)$ I/Os.

A.3.3 Total I/O Cost

In total, the increase in potential on the intervals, and therefore also the amortized I/O cost of the split, is bounded by $\mathcal{O}(\frac{1}{B}x \log_2 c)$. The INTERVALSPLIT operation, which includes searching for the query point and the INTERVALSMERGE operations, therefore uses amortized $\mathcal{O}(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B}x \log_2 c)$ I/Os in total.

This concludes the proof of Theorem 6.

B Supporting Lemmas

► **Lemma 10** (Bounding the Sum of Double Exponential Function). *Let $a, b \geq 2$. Then*

$$\sum_{i=0}^n a^{b^i} = \Theta(a^{b^n}).$$

Proof. The proof proceeds by showing the lower and upper bound separately. For the lower bound, as all terms of the sum is positive, then the sum is bounded below by the last term of the sum. To show the upper bound, it is shown by induction in n that

$$\sum_{i=0}^n a^{b^i} \leq 2a^{b^n}.$$

For $n = 0$ the inequality holds, as there is only one term which is equal to a .

$$\sum_{i=0}^n a^{b^i} = \sum_{i=0}^0 a^{b^i} = a^{b^0} = a \leq 2a = 2a^{b^0} = 2a^{b^n}.$$

For the induction step, let $n = n' + 1$, and let the inequality hold by induction for n' . Then

$$\sum_{i=0}^n a^{b^i} = \left(\sum_{i=0}^{n'} a^{b^i} \right) + a^{b^{n'+1}} \leq 2a^{b^{n'}} + (a^{b^{n'}})^b$$

From here, the goal bound is twice that of the last term in the sum. It can therefore be shown by proving that the second term bounds the first. For simplicity, let $k = a^{b^{n'}}$. Then it holds that

$$2k \leq k^b \iff \log_k(2k) \leq b \iff \log_k(2) + 1 \leq b.$$

As $a, b \geq 2$ and $n' \geq 0$, then $\log_k(2) \leq 1$, and the above inequality holds. It therefore holds that

$$2a^{b^{n'}} + \left(a^{b^{n'}}\right)^b \leq 2\left(a^{b^{n'}}\right)^b = 2a^{b^n} ,$$

Concluding the proof. ◀



Random Access on Narrow Decision Diagrams in External Memory

Steffan Christ Sølvsten^(), Casper Moldrup Rysgaard^(),
and Jaco Van de Pol^()

Aarhus University, Aarhus, Denmark
{soelvsten, rysgaard, jaco}@cs.au.dk

Abstract. The external memory BDD package Adiar can manipulate Binary Decision Diagrams (BDDs) larger than the RAM of the machine. To do so, it uses one or more priority queues to defer processing each recursion until the relevant nodes are encountered in a sequential scan. We outline how to improve the performance of Adiar’s algorithms if the BDD width of one of its inputs is small enough to fit into main memory. In this case, one of the algorithms’ priority queues can entirely be replaced with (levelised) random access to the nodes of the narrow BDD. This preserves the I/O efficiency of the original algorithm, is applicable to other types of decision diagrams, and significantly improves performance for many larger BDD computations.

Keywords: Binary Decision Diagrams · External Memory Algorithms

1 Introduction

Based on the work of Lars Arge [4, 5], Adiar¹ [24] is an implementation of Binary Decision Diagrams (BDD) [7] capable of handling BDDs larger than the machine’s random access memory (RAM). To achieve this, it uses time-forward processing [3, 8, 15] to replace the conventional depth-first recursion stack with one (or more) priority queue(s) that are synchronised with a sequential iteration through the input BDD(s).

The high performance of conventional BDD implementations is the result of several decades of research. Especially the unique node table and its layout has been vital [12, 14, 16, 19]. Yet, these and other ideas are not applicable to time-forward processing. Hence, new ideas are needed to make Adiar achieve a satisfactory performance. This has motivated the introduction of its levelised priority queue [23], its equality checking algorithm [24], and the concept of levelised cuts [21]. Common to all these optimisations is the use of some meta information about the BDD graph to substantially improve performance.

Adiar’s performance was evaluated [22, 24] on various combinatorial benchmarks. Each of these benchmarks accumulates a set of constraints, each of which

¹ github.com/ssoelvsten/adiar.

138 S. C. Sølvesten et al.

is a very narrow BDD, into one BDD whose size quickly grows large. Certain instances of symbolic model checking or symbolic SCC computation are quite similar. Here, the BDDs that represent transition relations in deterministic finite automata [9, 11] or in asynchronous models of concurrency [10], e.g. Petri Nets [13, 18], are narrow while the one for the accumulated state space is large.

1.1 Contributions

In the same vein as the prior optimisations in [21, 24], we show in Sect. 3 how Adiar can exploit the width of the input BDDs (defined in Sect. 2). In this case, the product construction algorithm in [24] can omit the use of one of its priority queues in favour of per-level using random access directly on the narrow BDD. Our experiments in Sect. 4 show that this considerably improves performance for the larger instances of both the motivating use case, i.e. when computing on at least one narrow BDD, and also average use cases.

1.2 Related Work

Prior to this work, levelised cuts [21] improves Adiar’s performance by soundly upper bounding the size of its priority queues. If it is smaller than main memory, then the external memory priority queue is replaced by a simpler and faster priority queue that only works in internal memory. This has been vital for Adiar’s performance on BDDs that do fit into the RAM. In this work, we instead improve Adiar’s performance by changing the algorithms’ logic. Hence, in contrast to the prior work, this optimisation targets the entire spectrum of BDDs. It especially is of benefit to some larger instances.

CAL [20] (based on [6, 17]) is to the best of our knowledge the only other BDD package to compute on BDDs that exceed main memory. To do so, it stores all BDD nodes in a unique node table and uses queues to execute its algorithms in the breadth-first manner. These node tables and queues can be offloaded to the disk via the operating system’s swap memory. Yet, this is only efficient, if every level fits into memory. In general, CAL is not I/O-efficient [5], whereas Adiar is [24].

2 Preliminaries

2.1 I/O Model

Aggarwal and Vitter [1] designed the I/O-model to analyse the data transfers between two levels of a memory hierarchy. Here, the internal memory, e.g. the RAM, has a finite size of M and data exceeding its capacity needs to be transferred in blocks of size B to/from the external memory, e.g. the Disk.

The number of block data transfers (I/Os) needed to sequentially read and write N amounts of data is $\text{scan}(N) \triangleq N/B$. To sort N amounts of data one

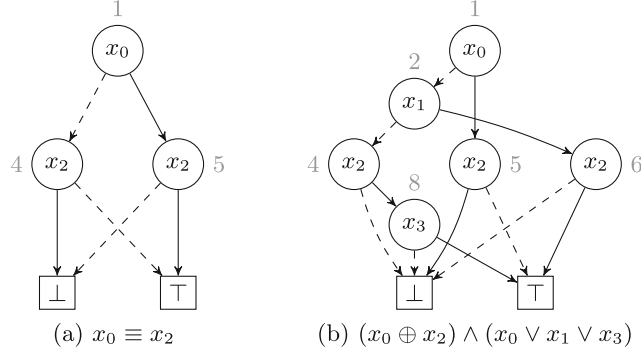


Fig. 1. Examples of Reduced and Ordered Binary Decision Diagrams. Terminals are drawn as boxes while internal nodes are drawn as circles with its decision variable. The *then* and *else* edges are respectively drawn solid and dashed.

needs to use $\text{sort}(N) \triangleq N/B \cdot \log_{M/B}(N/B)$ I/Os. Furthermore, one can also design a priority queue capable of inserting and extracting N elements in the optimal $\Theta(\text{sort}(N))$ number of I/Os [3]. For all realistic values of N , M , and B , both $\text{scan}(N)$ and $\text{sort}(N)$ are several magnitudes smaller than N itself.

2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDD) [7] provide a concise representation of Boolean functions $\mathbb{B}^n \rightarrow \mathbb{B}$ as a singly-rooted directed acyclic graph (DAG). As shown in Fig. 1, a BDD has two terminals with the Boolean values $\mathbb{B} = \{\perp, \top\}$ as the function's output whereas each internal BDD node provides an if-then-else decision on one of the n input variables, x_i .

What are colloquially referred to as BDDs are in fact *Reduced* and *Ordered* BDDs (ROBDDs). A BDD is ordered if the decision variables only occur once on each path from the root to a terminal and always following the same order. This induces a *levelisation* with level x_i only containing nodes with the said variable. The *width* of a BDD is the size of its largest level. A BDD is reduced, if there are (1) no *duplicate* nodes and (2) no *redundant* nodes. A node is a duplicate if it represents the same if-then-else. In conventional BDDs, a node is redundant if it has two identical children.

Fundamental to BDDs is the *Apply* operation, which, given BDDs f and g and a binary operator $\odot$, constructs the BDD for $f \odot g$. This is done via a product construction of both input BDDs and applying $\odot$ when arriving at a pair of terminals. As an example, Fig. 2 shows the product of Fig. 1a and 1b.

Here, we only provide a high-level description of Adiar's Apply algorithm that includes the details needed for Sect. 3; we refer to [24] for a detailed explanation. To make it I/O-efficient, Adiar imposes a total order on its BDD nodes such that the BDD is sorted level by level. Specifically, each node is associated with a numeric *time point* that they are encountered in the input (grey indices in

140 S. C. Sølvesten et al.

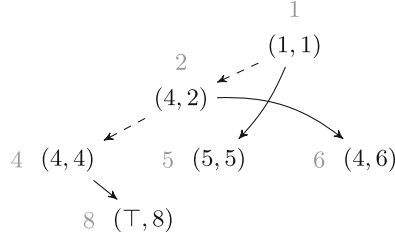


Fig. 2. Product Graph for BDDs in Fig. 1a and Fig. 1b.

Fig. 1). This ensures that each BDD node will always come after its parent during a sequential scan. To identify the pair of children of $(n_f, n_g) \in f \times g$ in a product construction, the ordering implies one needs the first to-be seen node, $\min(n_f, n_g)$, and possibly also the second one, $\max(n_f, n_g)$. Hence, a priority queue can make the recursion to target (n_f, n_g) match the sequential scan of the inputs by sorting it on the time point $\min(n_f, n_g)$. If $\max(n_f, n_g)$ is also needed then the BDD node $\min(n_f, n_g)$ is further forwarded to $\max(n_f, n_g)$ with a second priority queue. To do so, the second priority queue sorts its elements on the time point $\max(n_f, n_g)$. To guarantee a polynomial running time, recursions to the same target are grouped together. This is done by resolving ties in both priority queues' ordering via a lexicographical ordering of the recursion targets. For example, the product graph of the BDDs in Fig. 1a and Fig. 1b is resolved in [24] in the order depicted in Fig. 2.

This Apply algorithm only uses $\mathcal{O}(\text{scan}(N_f) + \text{scan}(N_g) + \text{sort}(T))$ I/Os unlike the $\mathcal{O}(T)$ I/Os used by conventional recursive implementations [5, 12], where N_f, N_g are the number of internal BDD nodes in f and g , respectively, and T is the number of BDD nodes in the output.

3 Using Random Access for Narrow Decision Diagrams

Without loss of generality, assume that the second input, g , to the Apply operation is *narrow*, i.e. the width of g is smaller than some threshold $\theta < M/2$. In this case, we can completely omit the second priority queue.

To do so, when processing level x_i , we load all BDD nodes of g at level x_i from external memory. This provides immediate random access to the entire level x_i of g . Hence, the second priority queue can be omitted if the ordering of the first priority queue is changed accordingly. Specifically, the first priority queue now solely has to respect the levels of the recursive calls and synchronise them with the sequential scan through f . Hence, the recursion target (n_f, n_g) should first be sorted on its level to not miss any requests where the level of n_f is below the one of n_g . Secondly, it is sorted lexicographically to respect the sequential scan through f . Furthermore, lexicographical sorting also groups recursive calls for the same target together, which preserves the polynomial running time and I/Os.

Doing so only affects the order in which all recursions are resolved; the output is still isomorphic to what is produced by the algorithm in [24]. For example in Fig. 2, if random access is used on the BDD from Fig. 1b then $(4, 6)$ is resolved prior to $(5, 5)$. In the previous algorithm [24], both the node at time point 4 in Fig. 1a and the one at 6 in Fig. 1b had to be visited in-order to resolve the product $(4, 6)$; hence, this product was resolved after $(5, 5)$. Instead, with random access the node at time point 6 in Fig. 1b is immediately available when reading the one at 4 in Fig. 1a; hence, $(4, 6)$ is resolved before $(5, 5)$.

Proposition 1. *The Apply algorithm with random access on a narrow BDD saves $\mathcal{O}(\text{sort}(T))$ I/Os in comparison to the prior algorithm from [24].*

Proof. Since the input BDDs are already sorted based on their level, loading the levels of g top-down is possible in a single sequential scan. Yet, the algorithm in [24] also needs to scan through g . Hence, loading the nodes of g for random access does not cost any additional I/Os. Yet, it completely removes the $\mathcal{O}(\text{sort}(T))$ I/Os incurred by the second priority queue.

Note that this is only a constant improvement over the algorithm in [24]. Furthermore, it is only an $\mathcal{O}(\text{sort}(T))$ rather than a $\Theta(\text{sort}(T))$ improvement, since recursion requests do not necessarily need to be moved into the second priority queue.

4 Experimental Evaluation

We have implemented the modified algorithm of Sect. 3 and run the benchmarks from [22, 24] with threshold $\theta = 0$, B (2 MiB), and ∞ . Using $\theta = 0$ essentially turns the random access optimisation off and provides a baseline. On the other hand, using $\theta = \infty$ entirely replaces the previous Apply algorithm. Finally, $\theta = B$ provides a small value which covers the motivating use cases while also leaving more of the internal memory to the remaining priority queue.

The benchmarks of [22, 24] consist of two categories. First, the *Combinatorial Counting* problems, e.g. the Queens problem, primarily involve the accumulation of lots of narrow decision diagrams. On the other hand, the *EPFL* [2] *Circuit Verification* provides a typical use case for decision diagrams.

As in [21, 22, 24], we have run all experiments on the CSCAA *Grendel* cluster where Adiar is initialised with $M = 300$ GiB. For each value of θ and each benchmark instance, the running time has been measured between 3 and 15 times (11.2 times on average) depending on its expected running time. We consider a measurement to be significant if the difference between the mean running time of $\theta = 0$ and $\theta = B, \infty$ is larger than twice their largest standard deviation. Figure 3 shows the speed-up in the mean running time for all 113 benchmark instances. Table 1 provides a summary of all significant instances.

Both $\theta = B$ and $\theta = \infty$ provide a significant performance increase in performance for the larger combinatorial benchmarks compared to the $\theta = 0$ baseline. Of the 64 combinatorial instances, 14 (21.9% of all of these instances) had

142 S. C. Sølvesten et al.

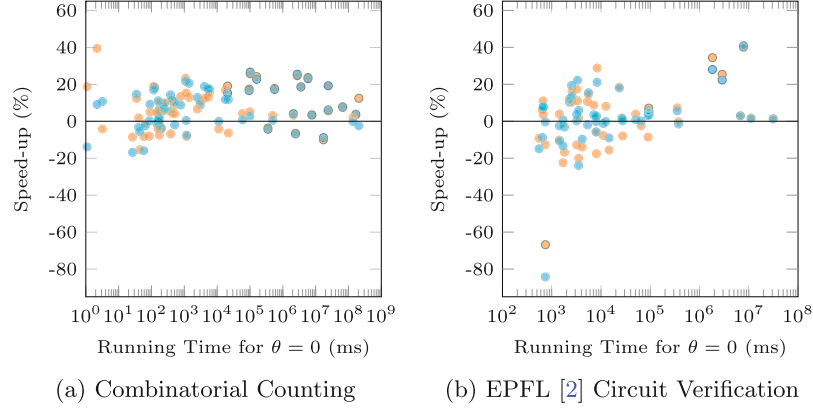


Fig. 3. Speed-up in running time with $\bullet$ $\theta = B$ and $\bullet$ $\theta = \infty$ relative to $\theta = 0$ (higher is better). Statistically significant measurements have a black border.

a significant improvement of 15.4% on average for $\theta = B$ and 16.3% for $\theta = \infty$. These 14 instances all require 10 s or more to solve with $\theta = 0$. In total, 26 out of the total 64 instances required this amount of time to solve. That is, performance improved for 53.8% of these larger instances.

Similarly to the combinatorial benchmarks, EPFL Verification also gains significant improvements for many of its larger instances. Only one circuit, `int2float`, requires significantly more time to verify. Yet, while a decrease in performance of 67% seems worrisome, it is only an increase in the computation time from 0.74 s to 1.12 s.

Finally, the optimisation presented in this work further closes the gap between Adiar and conventional BDD packages. For example, CUDD [26] can solve up to the 15-Queens problem with BDDs. In [24] the gap between Adiar and CUDD for this problem's instance was a factor of 1.42. In [21], this was improved to 1.26. With $\theta = B, \infty$, the gap is now further decreased down to 1.07. Simultaneously, this also improves performance for instances not solvable with CUDD. For example, it improves the solving time of 16-Queens by 19.2%.

Table 1. Speed-up (Δ) and slowdown (∇) of $\theta = B$ and $\theta = \infty$ for Combinatorial Counting (CC) and EPFL [2] Circuit Verification (EPFL) benchmarks.

			# Significant Instances				Average Relative Difference	
			Δ		∇		Δ	∇
CC	$\bullet$ $\theta = B$	14	(21.9%)	3	(4.7%)		15.4%	-6.7%
	$\bullet$ $\theta = \infty$	14	(21.9%)	3	(4.7%)		16.3%	-6.7%
EPFL	$\bullet$ $\theta = B$	6	(12.2%)	0	(0.0%)		25.1%	-
	$\bullet$ $\theta = \infty$	6	(12.2%)	1	(2.0%)		26.9%	-66.8%

5 Conclusion

Let a BDD be *narrow* if its width is smaller than some threshold $\theta < M/2$, where M is the amount of internal memory. Specifically, each individual level of a narrow BDD fits into internal memory. Hence, one can load each of its levels in their entirety and do random access on it. If some input to the Apply algorithm in [24] is narrow then one does not need to synchronise its computation with the sequential order of the narrow one. In this work, we have sketched how this algorithm can be adapted to this case, to save on computation time and I/Os.

For $\theta = B$ (B is the block transfer size), our experiments show a significant improvement in performance for larger instances. In the motivating use case with lots of narrow BDDs, performance increased significantly by 11.8% on average. In the average use case, performance even increased significantly by 25.1%.

Relative to $\theta = B$, our results with an unlimited θ further improves performance significantly for 3 larger combinatorial instances. No instances slowed down significantly. Hence, we have implemented levelised random access as part of Adiar 2.0 and based on the observation above, we use a large θ of $M/8$.

5.1 Future Work

While using levelised cuts [21] improves Adiar's performance for algorithms whose priority queues fit into RAM, it still leaves a gap between the running time of Adiar and conventional BDD packages for the smallest problems.

To efficiently solve the smallest of BDD instances (15 MiB or smaller), the BDD package CAL [20] switches from its breadth-first approach to the conventional depth-first algorithms.

The work presented in this paper can be extended to compute on input BDDs that are stored in an (internal memory) unique node table. Its (unreduced) output, which may exceed main memory, is still stored on the disk. Conversely, one can change Adiar's algorithms to place the final (reduced) BDD back in the node table. Hence, this work provides the basis for an efficient and seamless transition between a conventional depth-first approach and Adiar's external memory algorithms. This will be the final step to make Adiar competitive across the entire spectrum of BDDs.

Furthermore, this work applies to all of Adiar's product construction operations, such as variable quantification. Hence, this work, together with [21], is vital for the design of an I/O-efficient relational product that is usable in practice.

Acknowledgements. Thanks to the Centre for Scientific Computing, Aarhus, (phys.au.dk/forskning/cscaa/) for access to the Grendel cluster.

Data Availability Statement. The data presented in Sect. 4 is available at [25] while the code to run the benchmarks can be found at [27].

144 S. C. Sølvesten et al.

References

1. Aggarwal, A., Vitter, Jeffrey, S.: The input/output complexity of sorting and related problems. *Commun. ACM* **31**(9), 1116–1127 (1988). <https://doi.org/10.1145/48529.48535>
2. Amarí, L., Gaillardon, P.E., De Micheli, G.: The EPFL combinational benchmark suite. In: 24th International Workshop on Logic and Synthesis (2015)
3. Arge, L.: The buffer tree: a new technique for optimal I/O-algorithms. In: Workshop on Algorithms and Data Structures (WADS). LNCS, vol. 955, pp. 334–345. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60220-8_74
4. Arge, L.: The I/O-complexity of ordered binary-decision diagram manipulation. In: 6th International Symposium on Algorithms and Computations (ISAAC). LNCS, vol. 1004, pp. 82–91 (1995). <https://doi.org/10.1007/BFb0015411>
5. Arge, L.: The I/O-complexity of ordered binary-decision diagram. In: BRICS RS Preprint Series, vol. 29. Department of Computer Science, University of Aarhus (1996). <https://doi.org/10.7146/brics.v3i29.20010>
6. Ashar, P., Cheong, M.: Efficient breadth-first manipulation of binary decision diagrams. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 622–627. IEEE Computer Society Press (1994). <https://doi.org/10.1109/ICCAD.1994.629886>
7. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.* **C-35**(8), 677–691 (1986). <https://doi.org/10.1109/TC.1986.1676819>
8. Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1995), pp. 139–149. Society for Industrial and Applied Mathematics (1995)
9. Elgaard, J., Klarlund, N., Møller, A.: MONA 1.x: new techniques for WS1S and WS2S. In: Proceedings of the 10th International Conference on Computer-Aided Verification, CAV 1998. LNCS, vol. 1427, pp. 516–520. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-61648-9_56
10. Kant, G., Laarman, A., Meijer, J., Van de Pol, J., Blom, S., Van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
11. Klarlund, N.: Mona & Fido: the logic-automaton connection in practice. In: Computer Science Logic. LNCS, vol. 1414, pp. 311–326. Springer, Cham (1998). <https://doi.org/10.1007/BFb0028022>
12. Klarlund, N., Rauhe, T.: BDD algorithms and cache misses. In: BRICS Report Series, vol. 26 (1996). <https://doi.org/10.7146/brics.v3i26.20007>
13. Larsen, C.A., Schmidt, S.M., Steensgaard, J., Jakobsen, A.B., van de Pol, J., Pavlogiannis, A.: A truly symbolic linear-time algorithm for SCC decomposition. In: Tools and Algorithms for the Construction and Analysis of Systems (2). LNCS, vol. 13994, pp. 353–371. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-30820-8_22
14. Long, D.E.: The design of a cache-friendly BDD library. In: Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 639–645. Association for Computing Machinery (1998)
15. Meyer, U., Sanders, P., Sibeyn, J.: Algorithms for Memory Hierarchies: Advanced Lectures. Springer, Heidelberg (2003). <https://doi.org/10.1007/3-540-36574-5>

16. Minato, S.I., Ishiura, N., Yajima, S.: Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In: 27th Design Automation Conference (DAC), pp. 52–57. Association for Computing Machinery (1990). <https://doi.org/10.1145/123186.123225>
17. Ochi, H., Yasuoka, K., Yajima, S.: Breadth-first manipulation of very large binary-decision diagrams. In: International Conference on Computer Aided Design (ICCAD), pp. 48–55. IEEE Computer Society Press (1993). <https://doi.org/10.1109/ICCAD.1993.580030>
18. Pastor, E., Roig, O., Cortadella, J., Badia, R.M.: Petri net analysis using boolean manipulation. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 416–435. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58152-9_23
19. Pastva, S., Henzinger, T.: Binary decision diagrams on modern hardware. In: Conference on Formal Methods in Computer-Aided Design, pp. 122–131 (2023)
20. Sanghavi, J.V., Ranjan, R.K., Brayton, R.K., Sangiovanni-Vincentelli, A.: High performance BDD package by exploiting memory hierarchy. In: 33rd Design Automation Conference (DAC), pp. 635–640. Association for Computing Machinery (1996). <https://doi.org/10.1145/240518.240638>
21. Sølvsten, S.C., Van de Pol, J.: Predicting memory demands of BDD operations using maximum graph cuts. In: André, É., Sun, J. (eds.) Automated Technology for Verification and Analysis. LNCS, vol. 14216, pp. 72–92. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-45332-8_4
22. Sølvsten, S.C., Van de Pol, J.: Adiar 1.1: zero-suppressed decision diagrams in external memory. In: NASA Formal Methods Symposium, LNCS, vol. 13903, Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-33170-1_28
23. Sølvsten, S.C., Van de Pol, J., Jakobsen, A.B., Thomasen, M.W.B.: Efficient binary decision diagram manipulation in external memory. arXiv preprint [arXiv:2104.12101](https://arxiv.org/abs/2104.12101) (2021)
24. Sølvsten, S.C., Van de Pol, J., Jakobsen, A.B., Thomasen, M.W.B.: Adiar: binary decision diagrams in external memory. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 13244, pp. 295–313. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-030-99527-0_16
25. Sølvsten, S.C., Rysgaard, C.M., van de Pol, J.: Adiar 2.0.0-beta.3 : Experiment Data (2024). <https://doi.org/10.5281/zenodo.10493770>
26. Somenzi, F.: CUDD: CU decision diagram package, 3.0. Tech. rep., University of Colorado at Boulder (2015)
27. Sølvsten, S.C.: BDD Benchmark. Zenodo (2024). <https://doi.org/10.5281/zenodo.10803154>

The Contiguous Art Gallery Problem is Solvable in Polynomial Time

Magnus Christian Ring Merrild ✉ 

Department of Computer Science, Aarhus University, Denmark

Casper Moldrup Rysgaard ✉ 

Department of Computer Science, Aarhus University, Denmark

Jens Kristian Refsgaard Schou ✉ 

Department of Computer Science, Aarhus University, Denmark

Rolf Svenning ✉ 

Department of Computer Science, Aarhus University, Denmark

Abstract

In this paper, we study the *Contiguous Art Gallery Problem*, introduced by Thomas C. Shermer at the 2024 Canadian Conference on Computational Geometry, a variant of the classical art gallery problem from 1973 by Victor Klee. In the contiguous variant, the input is a simple polygon P , and the goal is to partition the boundary into a minimum number of polygonal chains such that each chain is visible to a guard. We present a polynomial-time RAM algorithm, which solves the contiguous art gallery problem. Our algorithm is simple and practical, and we make a C++ implementation available.

In contrast, many variations of the art gallery problem are at least NP-hard, making the contiguous variant stand out. These include the classical art gallery problem and the *edge-covering problem*, both of which being proven to be $\exists\mathbb{R}$ -complete recently by Abrahamsen, Adamaszek, and Miltzow [J. ACM 2022] and Stade [SoCG 2025], respectively. Our algorithm is a greedy algorithm that repeatedly traverses the polygon's boundary. To find an optimal solution, we show that it is sufficient to traverse the polygon polynomially many times, resulting in a runtime of $\mathcal{O}(n^6 \log n)$ arithmetic operations. We further bound the bit complexity of the computed values, showing that problem is in P. Additionally, we provide algorithms for the restricted settings, where either the endpoints of the polygonal chains or the guards must coincide with the vertices of the polygon.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Computational geometry

Keywords and phrases Art Gallery, Computational Geometry, Combinatorics, Discrete Algorithms

Funding Supported by Independent Research Fund Denmark (DFF), grants 9131-00113B and 10.46540/3103-00334B

Acknowledgements We thank Joseph O'Rourke for organizing the open problem session [4] at the Canadian Conference on Computational Geometry 2024 (CCCG24) at Brock University, Canada, where Thomas C. Shermer proposed the contiguous art gallery problem. The authors also thank the participants of CCCG24 for their lively discussions of the problem at the conference, especially Frederick Stock. This paper was submitted to the Symposium on Computational Geometry 2025 conference and accepted as a merge with [28] and [8] into [7]. We thank our collaborators for the engaging discussions that inspired optimizations to our work.

1 Introduction

The *art gallery problem*, introduced by Victor Klee in 1976, is a classical computational geometry problem where the goal is to find a minimum set of guards (points) in the interior of an input polygon P that sees every other point in P of the polygon. There are numerous variations of this problem, many of which are at least NP-hard or require complicated algorithms with a high polynomial running time. This is particularly the case for unrestricted

■ **Table 1** Summary of the work related to the contiguous art gallery problem, for polygons with n vertices where k^* is the size of the minimal decomposition. We abbreviate Art Gallery as AG, $\dagger$ refers to the guard location being restricted to vertices, and $\ddagger$ refers to the vertices of the pieces (the guarded area) being restricted to the vertices of the input polygon. Notice how the difficulty of the four problems increases along the diagonal from the bottom left vertex to the top right.

Problem	Vertex restricted	Unrestricted	With holes
Standard AG	NP-hard $\dagger$ [22]	$\exists\mathbb{R}$ -complete [1]	$\exists\mathbb{R}$ -complete [1]
Edge-covering AG	NP-hard $\dagger$ [19]	$\exists\mathbb{R}$ -complete [31]	$\exists\mathbb{R}$ -complete [31]
Star-shaped partition	$\mathcal{O}(n^7 \log n)^{\ddagger}$ [17]	$\mathcal{O}(n^{105})$ [2]	NP-hard [26]
Contiguous AG	$\mathcal{O}(n^2 \log^2 n)^{\ddagger}$ <i>new</i> $\mathcal{O}(n^2 \log n)^{\dagger}$ <i>new</i>	$\mathcal{O}(k^* n^5 \log n)$ <i>new</i>	<i>Open</i>

variants, where guard locations and the part of the polygon they cover are not constrained to the vertices of the input polygon. In this paper, we study the *contiguous art gallery problem* where the boundary of P should be partitioned into a minimum number of contiguous intervals, i.e., polygonal chains, such that each chain is visible to a guard in the interior of P . Neither the guards nor the endpoints of the chains are restricted to the vertices of P . The problem was introduced by Thomas C. Shermer at the Canadian Conference on Computational Geometry 2024. We resolve it by providing a polynomial-time real RAM algorithm.

► **Theorem 1.** *The contiguous art gallery problem for a simple polygon with n vertices is solvable in $\mathcal{O}(k^* n^5 \log n)$ arithmetic operations, where k^* is the size of an optimal solution.*

In section 6 we show that the bit complexity of the arithmetic operations are bounded polynomially in the number of bits used to encode the input polygon, therefore showing the aforementioned algorithm is in fact a polynomial-time RAM algorithm.

► **Theorem 2.** *The contiguous art gallery problem for a simple polygon encoded in N bits is a member of the complexity class P .*

In contrast to many other art gallery variants, it is a surprising and positive result that this variant allows for an efficient and simple algorithm. We demonstrate this by implementing the algorithm in C++ using CGAL [11, 12, 16, 32, 36]. Our code is available online [25].

1.1 Related work

The literature contains a variety of variations of the classical art gallery formulation [26, 30, 33], and we summarize the most important ones in Table 1. Most art gallery variants can be framed as *decomposition problems* [18, 23], where the goal is to decompose an input polygon P into less complicated components whose union is P . If the pieces may overlap, it is *covering problem*, and if not it is a *partition problem*. In the art gallery setting each piece must be guarded. The variants can further be categorized as *restricted* or *unrestricted*. In a restricted version of the problem, the guards and/or the vertices of each piece must coincide with the vertices of P . Conversely, an unrestricted version imposes no such constraints on the placement of guards or the vertices of the pieces. Variants also differ based on the complexity of P . We consider the two important cases of simple polygons with and without holes, with

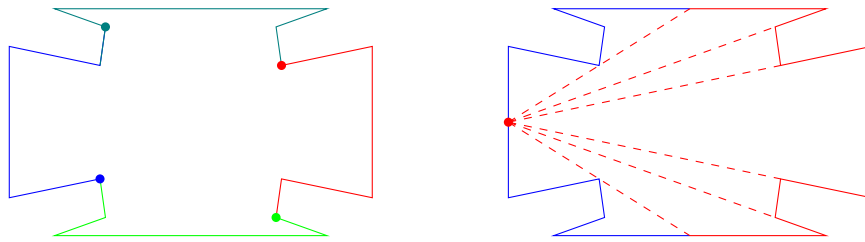
the latter typically being significantly more complicated. In Table 1 summarizes the difficulty of three fundamental art gallery variants and our contiguous variant.

The *classical art gallery problem* can be viewed as a covering problem in which the input polygon is decomposed into a minimal number of *star-shaped* polygons. A polygon is star-shaped exactly when it is possible to place a guard that sees all other points in the polygon. The decision problem “Can this polygon be guarded by k guards?” was recently shown by Abrahamsen, Adamaszek, and Miltzow [1] to be $\exists\mathbb{R}$ -complete [29] for any simple polygon with or without holes. If the guards are restricted to the vertices of the polygon, then Lee and Lin [21] showed that the problem is NP-hard.

The *edge-covering problem* is a variant of the art gallery problem, where the aim is to only cover the edges of a polygon, motivated by protecting valuable art on the walls of the gallery. The edge-covering variant was shown by Laurentini to be NP-hard [19] for guards restricted to vertices and $\exists\mathbb{R}$ -hard for unrestricted guards and polygons with holes. As seen in Figure 3, covering the edges does not imply covering the interior of P .

The *minimum star-shaped partition problem* was shown by Keil [17] to be solvable in $\mathcal{O}(n^7 \log n)$ arithmetic operations, when the star-shaped regions start and end at vertices. Without vertex restriction, Abrahamsen, Blikstad, Nusse, and Zhang [2] introduced a breakthrough algorithm solving this harder variant using $\mathcal{O}(n^{105})$ arithmetic operations. With holes, computing the minimum star-shaped partition is NP-hard due to O’Rourke [26].

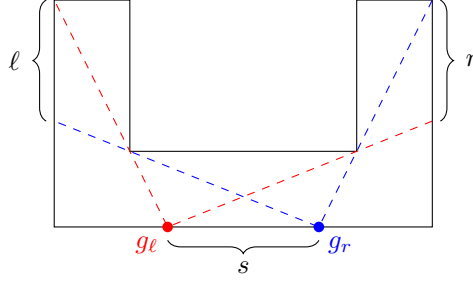
The *contiguous art gallery problem* was introduced at the 2024 Canadian Conference on Computational Geometry, by Thomas C. Shermer and this variant is the study of this paper. Framed as a decomposition problem, the goal is to partition the boundary ∂P into a minimal number of polygonal chains such that each can be seen by a guard interior to P . We mainly focus on the unrestricted version for a simple polygon without holes, but we also describe algorithms for restricted versions. Algorithm 1 does not generalize to polygons with holes, see Appendix C.2, and determining the hardness of this variant is an interesting open problem. The placement of guards in optimal solutions may differ across all variants, as demonstrated in Figures 3 and 4.



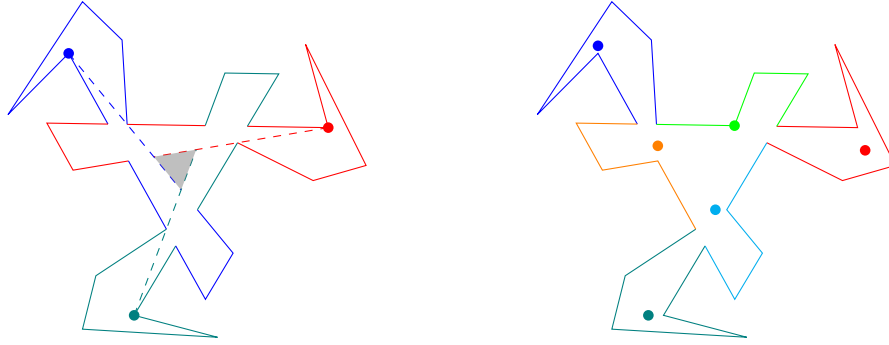
■ **Figure 1** Left, an optimal solution to the vertex restricted guards contiguous art gallery problem, discovered by greedily maximizing in one direction. Right, an optimal unrestricted contiguous art gallery solution requires 2 guards whose guarded pieces start and end at non-vertex points on the boundary, and these boundary points are not directly defined by segments of the polygon.

1.2 Limitations of Existing Approaches

We first sketch why the restricted version of the contiguous art gallery problem is relatively simpler to solve in polynomial time than the unrestricted variant, which is our focus. When partitioning the boundary of a simple polygon, it is natural to view it as a circle and the polygonal chains as arches or intervals; see Figure 12. By doing so, finding a minimal partition is similar to finding a *minimal circle-cover* among a set of intervals $\mathcal{C}$. This problem



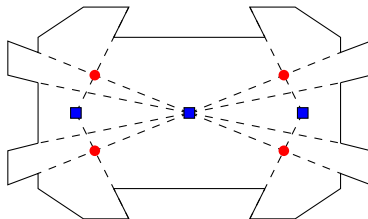
■ **Figure 2** A polygon demonstrating where the blue guard g_r sees all of r and none of ℓ , and the red guard g_ℓ is symmetric. Any guard placed along s will yield a trade-off between how much of ℓ and r it sees. This shows that it is non-trivial with unrestricted guards to find a finite set of polygonal chains that include an optimal solution as there may be infinite maximal intervals.



■ **Figure 3** Left, 3 optimal edge-covering guards where the inner gray triangle is not visible. Right, 6 guards that optimally contiguously cover the boundary. Guards are visualized as points whose color matches the part of the boundary they guard.

was studied by Lee and Lee [20] for finite cardinality C , giving an algorithm that runs in $\mathcal{O}(|C| \log |C|)$ time. Thus, given a finite set C of polygonal chains of the boundary of P such that an optimal solution is contained in C , then the contiguous art gallery problem can be solved in $\mathcal{O}(|C| \log |C|)$ time by simply viewing each polygonal chain as an arc and running the minimal circle-covering algorithm. In the restricted contiguous art gallery problem, that is, restricting guard positions to vertices of P or restricting the endpoints of the polygonal chains to coincide with vertices of P , leads to a set of polygonal chains C that includes an optimal solution of size $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$, respectively. The set C must also be computed, and in Appendix A we describe how to solve these two problems in $\mathcal{O}(n^2 \log n)$ and $\mathcal{O}(n^2 \log^2 n)$ time, respectively.

The unrestricted contiguous art gallery problem is our main focus, where guards may be placed anywhere interior to P , and polygonal chains of the boundary of P may start and end anywhere. In this setting, it is non-trivial to generate a polynomial-sized set of intervals that contains an optimal solution, which would allow the use of the algorithm for the circle-cover minimization problem. An approach that does not work is to generate all the different candidate intervals that are maximal, as there may be infinitely many of these, as shown in Figure 2. Another approach that we were unable to rule out is based on showing that an optimal solution coincides with a point of low *degree* [2] either with a guard or the endpoint of a polygonal chain. Points of degree 0 are the vertices of P , and points of degree $i > 0$ are points formed by intersections of lines formed by pairs of points of degree $(i - 1)$.



■ **Figure 4** A polygon, where the red guards (circles) represent an optimal solution to the contiguous art gallery problem and the blue guards (squares) are an optimal solution to the classical art gallery problem.

Letting D_i be the number of points of degree i , then D grows as $\Theta(n^{4^i})$ so even if this approach is feasible, it leads to high polynomial running time even for low degree points. For the minimum star-shaped partition problem the authors showed that first-degree points suffice as candidates [2].

1.3 Our Greedy Optimal Solution

Our solution takes a different and remarkably simple approach. At the core of our method is the GREEDYINTERVAL algorithm, which, given a point x on the polygon's boundary, finds the furthest point y along the boundary in the clockwise direction such that the polygonal chain from x to y can be guarded by an interior point of the polygon. We denote these as *greedy intervals*. Repeatedly taking these greedy steps until the boundary has been covered gives a solution of size $k \leq k^* + 1$. Each traversal of the boundary we denote by a *revolution*, and continuing this process performing $\mathcal{O}(kn^3)$ revolutions ultimately yields an optimal solution. Performing a single revolution by repeatedly using the GREEDYINTERVAL algorithm takes $\mathcal{O}(n^2 \log n)$ arithmetic operations, leading to $\mathcal{O}(k^* n^5 \log n)$ total arithmetic operations. The model of computation used here is the real RAM model [9] where arithmetic operations on real numbers take unit time. Here, the size of the input n is the number of real-valued inputs, that is, the vertices of a circular list of vertices of P . The main result of our paper is Theorem 1 that the contiguous art gallery problem is solvable in polynomial time in the real RAM model.

We also consider the problem in the RAM model, where the polygon is represented as an array of points, each point having two coordinates consisting of fractions with integer numerator and denominator. We consider the case where N bits are used to describe the entire input and show that $\text{poly}(N)$ operations is sufficient to find an optimal guarding. This is the contents of Section 6.

When the input polygon has holes our approach does not work and it is an intriguing open problem to determine whether this variant of the contiguous art gallery problem is solvable in polynomial time.

1.4 Concurrent work

This work is concurrent with [28] and [8] that both solve the contiguous art gallery problem using different techniques, the merge of this paper [7] was published at SoCG25. A notable difference is that this paper traverses the polygon clockwise, while the three others are counter-clockwise.

In [28] it is explored how GREEDYINTERVAL can be expressed as a piecewise rational linear function and show that repeatedly composing GREEDYINTERVAL with itself leads to a

piecewise rational linear function representing *all* optimal solutions.

In [8] it is explored how to generate $\mathcal{O}(n^4)$ candidate solutions by starting Algorithm 1 from a carefully selected set of points, such that at least one of these will be optimal.

1.5 Organization

In Section 2 we present our algorithm for solving the contiguous art gallery problem, and in Section 3 we cover details of how our algorithm can be implemented using basic computational geometry operations. In Sections 4 and 5, we derive an upper bound on the number of iterations of our greedy algorithm based on the geometric and combinatorial properties of our algorithm. In Section 6 we then use this bound to prove that the contiguous art gallery problem is in the complexity class P. Finally, in Section 7 we state related open problems.

In Appendix A we describe how to solve the contiguous art gallery problem under vertex restrictions. In Appendix B we prove geometric supporting lemmas. In Appendix C we give two examples that illustrate interesting behavior of Algorithm 1.

2 The Repeated Greedy Algorithm

Formally, let P be a simple polygon with n vertices $v_0, \dots, v_{n-1}$ and edges e_i connecting v_i and v_{i+1} . The boundary of P we denote as ∂P . The goal of the contiguous art gallery problem is to find the minimum k^* such that one can partition the boundary ∂P into k^* contiguous *visible* polygonal chains $I_1, I_2, I_3, \dots, I_{k^*}$ whose union is ∂P . A chain I_i is visible if there exists a guard g_i such that, for all points x on I_i , the line segment $\overline{xg_i}$ is contained in P . Conceptually, each I_i corresponds to an interval of the boundary between two points a and b , denoted as $[a, b]$. We allow guards to be colocated and consider them non-blocking, i.e., they can see through each other. Furthermore, for convenience, we always index the edges and vertices of P modulo n , meaning that $e_n = e_0$ and $v_n = v_0$.

Algorithm 1

Input: A simple polygon P with vertices $v_0, v_1, v_2, \dots, v_n$
Output: An explicit solution to the contiguous art gallery problem on P

```

1  $x_0 \leftarrow v_0$ 
2 for  $i = 1$  to  $T$  do
3    $x_i, g_i \leftarrow \text{GreedyInterval}(x_{i-1}, P)$ 
4    $j \leftarrow \max\{j \leq T - 2 \mid x_j \in (x_{T-1}, x_T)\}$ 
5 return  $\{(x_{i-1}, x_i, g_i) \mid j < i \leq T\}$  // The last segments/guards covering  $\partial P$ 

```

To solve the contiguous art gallery problem, we propose Algorithm 1 which is a conceptually simple greedy algorithm that starts at any point x_0 in ∂P (Line 1). Then repeatedly finds *greedy* intervals, that is, the longest visible interval of ∂P from a given starting point of ∂P , starting from where the previous segment ended (Lines 2 - 3). The greedy interval from any point $x \in P$ can be found in $\mathcal{O}(\text{poly}(n))$ time by combining basic computational geometry operations to calculate visibility polygons [5, 10], intersections of polygons [35] and common tangents between disjoint polygons [3]. The GREEDYINTERVAL algorithm is described in details in Section 3 as Algorithm 2. After $T \geq ck^2n^3$ iterations, where c is a sufficiently large constant and k is the number of iterations in the first revolution, the algorithm returns the start and endpoint of the last greedy intervals that form a partition of the boundary and the corresponding guards that see the segments (Lines 4 - 5).

3 The GREEDYINTERVAL algorithm

In this section, we present and analyze the GREEDYINTERVAL algorithm for computing greedy intervals, i.e. the longest visible interval I from a given point $x \in P$, along with a corresponding guard g that sees I . The algorithm uses only basic computational geometry operations. We assume for simplicity that P is not star-shaped, as otherwise $I = \partial P$. The idea is to extend I from one vertex of P to the next (clockwise around ∂P), starting from the edge containing x , while maintaining the feasible region of I . The feasible region of I is the set of points of P that can see all of I , i.e. the region of P where a guard for I can be placed. This happens on lines 2 - 3 where $VP(x, P)$ is a subroutine to compute the *visibility polygon* [10] of point $x \in P$ and $\cap$ computes the intersection between two simple polygons [35]. The correctness of extending I in discrete steps around ∂P follows by contrapositive of Lemma 55, which is that any point $u \in P$ that can see two points a, b on some edge e_j , can see all points on e_j between a and b . Finally, we extend I along edge e_{i-1} towards the last vertex v_i that causes the feasible region to be empty. This happens on line 4 where $lastVisiblePoint(P, F_{i-1}, e_{i-1})$ is a subroutine for computing the point furthest along e_{i-1} visible from F_{i-1} . To do so simply and efficiently we prove Lemma 5 which shows that it suffices to consider the vertices of the last feasible region.

Algorithm 2 GREEDYINTERVAL

Input: Point x on edge $e_i = (v_i, v_{i+1})$ of a simple and a non-star-shaped polygon P

Output: The endpoint of the longest visible interval I from x and a guard g that can see I

```

1  $F_i \leftarrow VP(P, x)$ 
2 while  $F \neq \emptyset$  do
3    $i \leftarrow i + 1$ 
4    $F_i \leftarrow F_{i-1} \cap VP(P, v_i)$ 
5 return  $lastVisiblePoint(P, F_{i-1}, e_{i-1})$ 

```

The main concern is to show that the running time of GREEDYINTERVAL is $\mathcal{O}(\text{poly}(n))$. Theorem 3 gives the precise output-sensitive running time, depending on the number e of edges of P that are intersected by I .

► **Theorem 3.** *The running time of the GREEDYINTERVAL algorithm for a polygon P with n vertices where the greedy interval that is output intersects e edges of P is $\mathcal{O}(en \log n)$.*

In the following three sections, we clarify how to perform the necessary computational geometry primitives and their efficiency to establish Theorem 3.

3.1 Visibility polygons

For a simple polygon P with n vertices, the *visibility polygon* [10] $VP(P, x)$ of point $x \in P$ is a polygon containing all points of P visible from x . A point that guards x must be placed in $VP(P, x)$, which we also call the feasible region of x . The visibility polygon $VP(P, x)$ has $\mathcal{O}(n)$ vertices and can be computed in $\mathcal{O}(n)$ time using the algorithm by Gindy and Avis [10].

3.2 Intersecting polygons

Computing the intersection of two polygons P and Q with n combined vertices is known as *polygon clipping*, and is a classical problem in computer graphics and computational geometry.

The running time of many of these algorithms, and in particular the output, depends on the number of intersections h between P and Q . An algorithm by Martínez, Rueda, and Feito [24] shows that the problem of polygon clipping can be solved in $\mathcal{O}((n + h) \log n)$ time. Their approach is similar to the classical sweep line algorithm by Bentley and Ottmann [6] for computing the intersections between a set of segments. There are also earlier algorithms for polygon clipping, but they are less efficient or do not handle degenerate cases [14, 34, 35].

In general, the intersection of two polygons of sizes n and m may have size $h = \Omega(nm)$, which could cause the repeated intersections of GREEDYINTERVAL to grow to size $\Omega(n^n)$. However, we prove that any feasible region will have at most $\mathcal{O}(n)$ vertices:

► **Lemma 4.** *The feasible region of a set of points S in a simple polygon P with n vertices has $\mathcal{O}(n)$ vertices.*

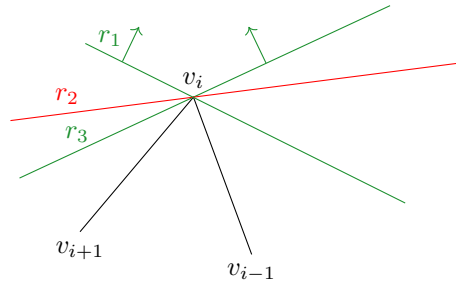
Proof. Let $v_1, v_2, \dots, v_t$ be the vertices of S . Let $F := VP(P, S)$ be the intersection $\bigcap_{i=1}^t VP(P, v_i)$. The boundaries of $VP(P, v_i)$ consists of sub line segments of ∂P and sub line segments of rays from v_i to reflex vertices of P . Hence, the boundary of F consists of the same sub line segments.

Any edge of ∂P can only contribute to one edge of ∂F (since $F \cap C$ is convex for every convex subset of P , Lemma 39), so at most n edges of ∂F come from these. Thus, we only need to restrict the number of edges coming from rays.

Consider a reflex vertex v_i and the rays stemming from visibility polygons defined by v_i . We now consider three rays, r_1 , r_2 and r_3 , which all pass through v_i . We will show, that at most two of these rays will contain sub line segments that are part of ∂F .

Order the rays, by the angle they make with edge $v_{i-1}v_i$ and assume that r_1 and r_3 contain sub line segments part of ∂F . Thus F will be contained in the half planes consisting of everything away from the path $v_{i-1}v_i v_{i+1}$ on the other side of r_1 and r_3 extended to lines. However, the only intersection of r_2 with these two half planes is v_i , hence it can not contribute an edge to ∂F . Doing this repeatedly, we get that at most two rays through v_i can contribute edges to the feasible region, see Figure 5.

By Lemma 39 it holds that $F \cap C$ is convex for each convex subset C of P , and we get that each ray can contribute at most one edge to the feasible region. Thus each reflex vertex can contribute at most two edges to F , meaning that the complexity of F is at most $3n = \mathcal{O}(n)$.



■ **Figure 5** r_1, r_2 and r_3 are rays passing through v_i . If both r_1 and r_3 contribute to the edges of F , then F lies in the quarter plane shown with arrows. Now r_2 will not be able to contribute to edges of F .

Combining Lemma 4 with the polygon clipping algorithm, we get a running time of $\mathcal{O}(n \log n)$ to compute the intersection between the previous feasible region and a visibility polygon.

3.3 The last visible point

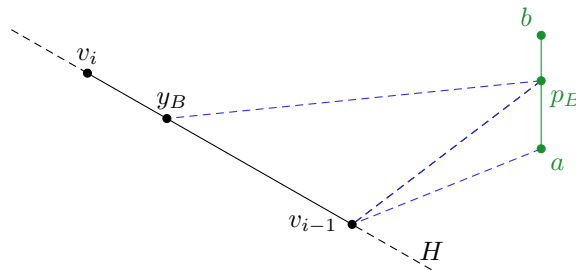
When computing $\text{lastVisiblePoint}(P, F_{i-1}, e_{i-1})$, the input is a simple polygon P with n vertices, an edge $e_{i-1} = (v_{i-1}, v_i)$, and a feasible region F_{i-1} with f vertices where v_{i-1} is visible from all points of F_{i-1} and v_i is not visible from any point of F_{i-1} . The output is the point y on e_{i-1} closest to v_i visible from F_{i-1} and the guard $g \in F_{i-1}$ that can see y . We first show that it suffices to consider only the vertices of F_{i-1} .

► **Lemma 5.** *Let P , F_{i-1} , and e_{i-1} be defined as above. Then the vertices of F_{i-1} can see at least as far along e_{i-1} as all of F_{i-1} .*

Proof. We prove the lemma by showing that points in the interior of F_{i-1} can see no further than points on the boundary ∂F_{i-1} which again can see no further than the vertices of F_{i-1} .

First, let y_I be a point on e_{i-1} visible from a point p_I in the interior. Clearly, the point of $\overline{p_I y_I} \cap F_{i-1}$ with minimal distance to y_I is a point on the boundary of F_{i-1} that can see y_I .

Second, let y_B be a point along e_{i-1} visible from a point p_B on edge (a, b) of the boundary of F_{i-1} that can see y_B . Note F_{i-1} cannot intersect the line $H := L(v_{i-1}, v_i)$ defined by e_{i-1} , since that would imply a point in F_{i-1} that can see v_i contradicting the input assumption that e_{i-1} cannot be seen from F_{i-1} . Recall that F_{i-1} is connected by Lemma 39 which further implies that it must lie exclusively on one side of H . If F_{i-1} is below H then all points of F_{i-1} can see no further than v_{i-1} . Otherwise, F_{i-1} is above H , and in particular edge (a, b) of F_{i-1} is above H . Let a (symmetrically b) be the vertex on the same side of the half-plane defined by $L(p_B, y_B)$ as v_{i-1} . By the contrapositive of Lemma 55 the triangle $\triangle p_B y_B v_{i-1}$ is inside P . Thus, if a is in $\triangle y_B p_B v_{i-1}$ it can also see y_B . Otherwise, a and v_i lie on opposite sides of the half-plane defined by (p_B, v_{i-1}) . See Figure 6 for an example of this case. Since a can see v_{i-1} , the triangle $\triangle a p_B v_{i-1}$ is also inside P by the contrapositive of Lemma 55. Thus, the segment from a to y_B is fully contained in triangles $\triangle a p_B v_{i-1}$ and $\triangle y_B p_B v_{i-1}$, which implies that a can see y_B . ◀

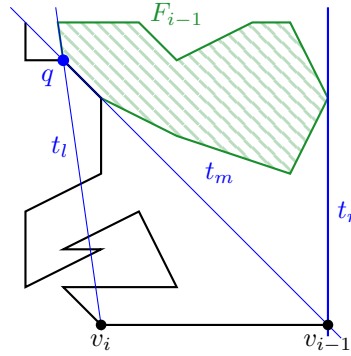


■ **Figure 6** The case in the proof of Lemma 5 where vertex a from the feasible region F_{i-1} satisfies the following conditions: (1) it lies above the half-plane H (2) it is on the same side of the half-plane defined by (p_B, y_B) as v_{i-1} (3) it is on the opposite side of the half-plane defined by (p_B, v_{i-1}) from v_i .

► **Remark 6.** The proof of Lemma 5 also shows that if multiple guard placements are optimal, they must be collinear with the endpoint $(y_B \text{ above})$.

Thus we want to find the vertex of F_{i-1} , which sees most of e_{i-1} . We do this by computing the common tangents with a part of ∂P and F_{i-1} . First, we need some terminology:

► **Definition 7** (Free area and congested area). Consider $e_{i-1} = (v_{i-1}, v_i)$ and F_{i-1} . Let t_r and t_m be the right and left tangent to F_{i-1} going through v_{i-1} . The region between these two tangents bounded by F_{i-1} will be without any part of ∂P (since F_{i-1} can see v_{i-1}) and we will therefor call it the free area. Let the left tangent to F_{i-1} through v_i be t_ℓ . The region bounded by t_m, t_ℓ and e_{i-1} will need to contain parts of ∂P and will be called the congested area. Let the intersection point between t_ℓ and t_m be q , see Figure 7.



■ **Figure 7** The last feasible region marked in green and tangents t_r, t_m and t_ℓ marked in blue.

The only parts of ∂P , that can block F_{i-1} from seeing v_i is the parts in the congested area. Thus, we construct a polygon from the ∂P in the congested area:

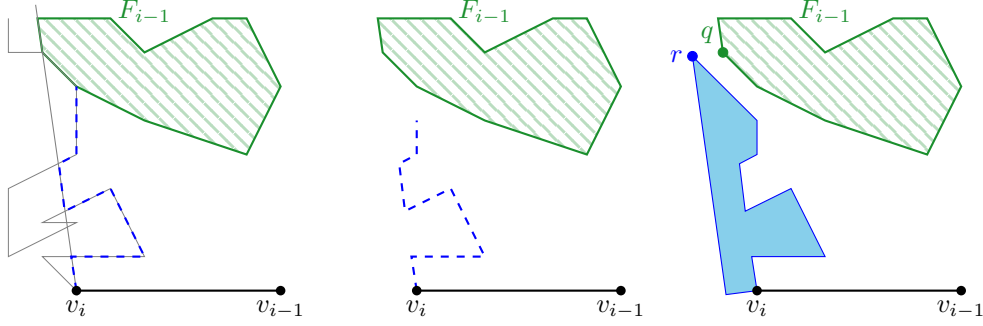
► **Definition 8** (Blocking polygon). We define the blocking polygon B by considering the following path: Starting from v_i , follow t_ℓ until you hit ∂P again. Then follow ∂P into the congested region until you hit t_ℓ again. Continue this until you hit q or F_{i-1} . If one hits q , one moves a little to the left to a point r that lies below t_m and then finishes by moving parallel to t_ℓ until you can move directly left to v_i .

If you hit F_{i-1} before q , then this must occur along t_m . By Lemma 39, F_{i-1} will only touch t_m along one edge of ∂F_{i-1} , so this edge is the one our path will touch. When the path hits this edge, we back up a small bit (small here depends on how close the rest of the path previously has gone to t_m and the distance from the point of contact to q) and then head directly to r (defined just as above) and then finish as above. By choosing the distance of the backtrack and distance between q and r small enough, it can be made so that the path does not intersect itself, as the path could not have hit t_m before the intersection point with F_{i-1} . This is illustrated on Figure 8. This path will be ∂B and B is the region bounded by this path.

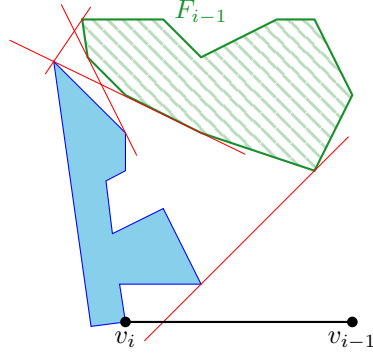
Now the claim is the following: F_{i-1} and B are disjoint polygons and one of the common tangents between F_{i-1} and B will be the line of sight of the guard that sees the most of e_{i-1} , see Figure 9. Thus, to find $\text{lastVisiblePoint}(P, F_{i-1}, e_{i-1})$ we can run the `COMMONTANGENTS` algorithm of Abrahamsen and Walczak [3] to find all common tangents (of which there are at most four) and manually check them to find the guard position that sees the most of e_{i-1} . The claims will be proven in the following.

► **Lemma 9** (Disjoint). B and F_{i-1} are disjoint.

Proof. F_{i-1} is contained in the free area and B is contained below t_m . Thus, the only intersection points will be along t_m . By backtracking on the path of ∂B before an intersection



■ **Figure 8** Following t_ℓ and ∂P into the congested area, one traces the boundary of B , marked in blue. Once we hit F_{i-1} we backtrack and head directly for r .



■ **Figure 9** Common tangents drawn. One of which sees the line of sight from the optimal guard to the farthest point on e_{i-1} , which is visible from F_{i-1}

with F_{i-1} and then going directly to r , we no longer touch the free area for the rest of the path, so ∂B does not touch F_{i-1} and must B not either. ◀

Let g be the vertex of F_{i-1} that sees the most of e_{i-1} and let c be the first vertex of ∂P blocking g from seeing any more of e_{i-1} (when moving along ∂P from v_i clockwise). We want to show that $L(g, c)$ is a common tangent of F_{i-1} and B . First, we show that c is a vertex of B .

► **Lemma 10** (c is present). c is a vertex of B .

Proof. For c to be the blocking vertex, it must be in the congested area. Many vertices of P , which are in the congested area, are vertices of B . The only ones that get skipped are ones that are hidden by other parts of ∂P like vertex h in Figure 10, and vertices that would be skipped, because we hit F_{i-1} and head directly for r .

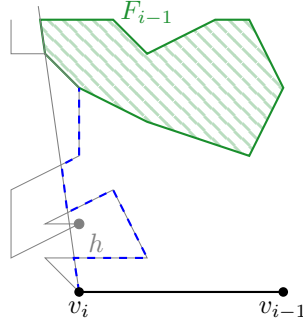
Vertices such as h cannot be the blocking vertex, since to draw a line from F_{i-1} to h , one must cross other edges of ∂P , so we exclude these.

Assume now that the path followed in Definition 8 where to intersect F_{i-1} , and the first time this happens is at point s . If s itself is a vertex of ∂P , it cannot be c , as by placing a guard at s (which is in F_{i-1}), one sees past s and sees more of e_{i-1} than g would, contradicting that s blocked the guard that sees the farthest.

Now assume that c is a vertex of ∂P in the congested area, which would have been hit after s by following ∂P and t_ℓ as in Definition 8 if we did not skip past them after hitting

s . Since the part of ∂P , in the congested region that contains s must enter and leave the congested area via t_ℓ , the path after s must continue left towards t_ℓ or stay away from t_ℓ . So c would be stuck to the left of the part of ∂B until s . So we again have a problem, since drawing a line from F_{i-1} to s , will hit ∂B in the congested area, i.e., hit ∂P and c cannot be a blocking vertex for g .

Since none of the above cases for c are possible, it must be that c is a vertex of B .



■ **Figure 10** h is hidden behind other parts of ∂P , so h cannot be a blocking vertex.

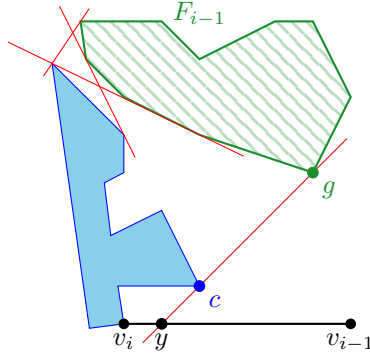
◀

Finally, we show that $L(g, c)$ is a common tangent.

► **Lemma 11** (Sight line is a common tangent). *$L(g, c)$ is a common tangent of F_{i-1} and B*

Proof. Consider the area below $L(g, c) =: \ell$ let ℓ intersect e_{i-1} at y , see Figure 11. If a point g' of F_{i-1} lies below ℓ , g' will see farther along e_{i-1} than y : The line segment $\overline{g'y}$ will now not touch B , as it is strictly below ℓ . Thus there will be some point y' further along e_{i-1} which also is not blocked from view of g' . This contradicts the optimality of g 's sight, so this cannot happen.

If a point b of B lies below ℓ , it must be a point along ∂P , and this will break the line of sight from g to y , since b must be connected to the rest of B , which lies above ℓ . Thus ℓ is a tangent of F_{i-1} and B .



■ **Figure 11** The line of sight, is included in one of the common tangents.

▶

► **Proposition 12** (Runtime). *$\text{lastVisiblePoint}(P, F_{i-1}, e_{i-1})$ runs in time $\mathcal{O}(n \log n)$*

Proof. $\text{lastVisiblePoint}(P, F_{i-1}, e_{i-1})$ has three steps:

1. Compute B .
2. Compute common tangents.
3. Check which tangent is $L(g, c)$ and compute y .

Step 2 is simply running the COMMONTANGENTS algorithm of Abrahamsen and Walczak [3], which takes $\mathcal{O}(n)$ time, and Step 3 can also be done in linear time, since there are at most four tangents to check. Thus, we need to show that B can be computed in $\mathcal{O}(n \log n)$ time.

First, we compute t_m and t_ℓ , which can be done in linear time. Next, we compute which segments of ∂P intersect t_ℓ and sort them along t_ℓ . This takes $\mathcal{O}(n \log n)$ time. Finally, determine the vertex or edge of F_{i-1} that lies on t_m . This can be done in linear time.

Now we can trace the path from Definition 8, keeping track of the minimal distance from the vertices considered to t_m . If ∂B does not hit F_{i-1} before reaching q , finish directly. Otherwise, compute a suitable backtrack distance and distance to r , so as not to overlap ∂B . This can be done in constant time, when the distance from ∂B to t_m is known. Then connect to r and finish as before. This takes $\mathcal{O}(n)$ time.

In total, it takes $\mathcal{O}(n \log n)$ time to compute B which is the bottleneck of the algorithm. ◀

3.4 Proving the runtime of GREEDYINTERVAL

We now have the tools necessary to prove Theorem 3, as restated below.

► **Theorem 3.** *The running time of the GREEDYINTERVAL algorithm for a polygon P with n vertices where the greedy interval that is output intersects e edges of P is $\mathcal{O}(en \log n)$.*

Proof. Computing the initial feasible region on line 2 takes $\mathcal{O}(n)$ time per Section 3.1. The while loop runs for $e+1 > 2$ iterations and in each iteration, a visibility polygon of complexity $\mathcal{O}(n)$ is computed and the intersection of two polygons of complexity $\mathcal{O}(n)$ is computed, resulting in a new complexity $\mathcal{O}(n)$ polygon (by Lemma 4 the feasible region has complexity $\mathcal{O}(n)$), taking $\mathcal{O}(n \log n)$ operations, according to Section 3.2. Thus, the while loop takes $\mathcal{O}(en \log n)$ operations.

Computing how far the greedy interval extends on the last edge takes $\mathcal{O}(n \log n)$ time per Section 3.3. In total, it takes $\mathcal{O}(en \log n)$ operations to run GREEDYINTERVAL. ◀

The above also implies a straightforward bound on the time for Algorithm 1 to perform a revolution of P .

► **Corollary 13.** *Repeatedly using the GREEDYINTERVAL algorithm to perform a revolution of P takes $\mathcal{O}(n^2 \log n)$ time.*

Proof. Let k be the number of iterations of the GREEDYINTERVAL algorithm to perform a revolution of P . The bound on the running time follows by applying Theorem 3 to each of the k iterations and using that the number of edges intersected by the greedy intervals sum to at most $2n$. ◀

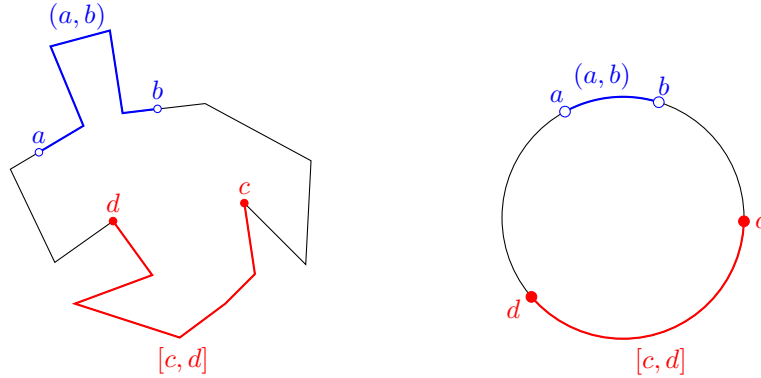
With the time complexity of GREEDYINTERVAL being polynomially bound, we only need to show that a polynomial number of revolutions is needed to find an optimal solution. This will be the content of Sections 4 and 5.

4 Algorithm 1, a combinatorial viewpoint

Summing up what we have done so far: Given any starting point x , the GREEDYINTERVAL algorithm finds the point on the boundary y that is furthest from x in the clockwise direction, such that there is a guard that can see the entire polygon chain between x and y . We use the notation $[x, y]$ for this polygonal chain, and call it a *greedy interval*. Generally, for any $a, b \in \partial P$ we let $[a, b]$ denote the polygon chain of ∂P from a to b including a and b . Likewise, we define (a, b) to be the polygon chain of ∂P from a to b excluding a and b . The notations $(a, b]$ and $[a, b)$ are defined analogously, with the inclusion of b or a , respectively. In this section, we focus solely on the combinatorial properties of greedy intervals, i.e. their relative positions to one another. As such, we visualize ∂P as a circle, as shown in Figure 12. In this way, we cast the problem as a circle-cover minimization problem with infinitely many circular arches. The setting with finitely many arches was studied by Lee and Lee [20]. Naturally, some concepts from their work are related, and we will highlight these connections where relevant.

► **Definition 14** (Visible intervals). *We denote a polygonal chain of the boundary between vertices a and b as the interval $[a, b] \subseteq \partial P$. If one can place a guard in P , that can see all of $[a, b]$ we call it a visible interval. The set of all visible intervals is denoted $\mathcal{V}$.*

► **Remark 15.** To simplify the notation and figures, we write G instead of GREEDYINTERVAL and let the input polygon and the output guard position be implicit, i.e. G takes a point c on the boundary and returns the farthest clockwise boundary point d such that $[c, d]$ is a visible interval.



■ **Figure 12** A simple polygon visualized as a circle such that contiguous polygonal chains of the boundary overlap in the polygon if and only if the corresponding intervals overlap in the circle representation. The interval (a, b) is not a greedy interval since b can be moved further around the polygon (clockwise) and remain visible to a guard. The interval $[c, d]$ is a greedy interval since d can be moved no further, i.e. $G(c) = d$.

► **Remark 16.** We will from now on always assume that P is not star-shaped, as otherwise when we run the GREEDYINTERVAL algorithm it will find that the entire boundary can be covered by a single guard, which is clearly optimal. Thus, $G(x) \neq x$ for all $x \in \partial P$. Furthermore, the greedy interval $[x, G(x)]$ from any point $x \in \partial P$ always contains at least one edge, since a guard can be placed at the first vertex encountered from x .

► **Lemma 17** (Properties of visible intervals).

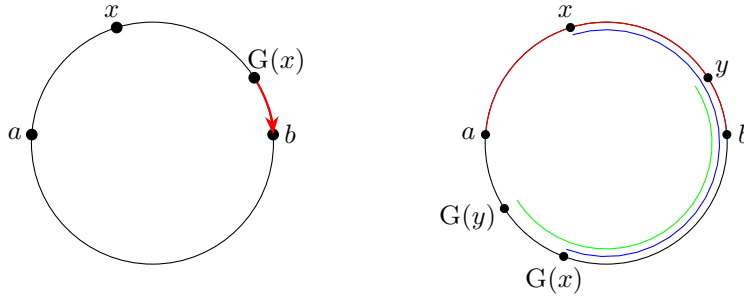
1. If $[a, b] \subseteq [c, d]$ and $[c, d] \in \mathcal{V}$ then $[a, b] \in \mathcal{V}$.
2. $[x, G(x)] \in \mathcal{V}$.
3. $[x, y] \subseteq [x, G(x)]$ if and only if $[x, y] \in \mathcal{V}$.

Proof. 1. and 2. follow by definition. 3. The only if implication follows directly from 1. and 2. For the if implication let $[x, y] \in \mathcal{V}$ and assume for contradiction that $[x, y] \not\subseteq [x, G(x)]$. Then $G(x)$ must be strictly before y , contradicting the maximality of GREEDYINTERVAL . $\blacktriangleleft$

Lemma 17 together with Remark 16 acts as combinatorial axioms for our problem. Thus, for the rest of the section, we will use these to obtain properties of GREEDYINTERVAL .

► **Lemma 18.** Let $[a, b] \in \mathcal{V}$ and $x, y \in \partial P$, then

1. $[x, G(x)] \subseteq [a, b]$ implies $G(x) = b$.
2. $x \in [a, b]$ implies $b \in [x, G(x)]$.
3. If $[x, y] \subseteq [a, b]$, then $G(x) \in [b, G(y)]$.



■ **Figure 13** Left, in Lemma 18.1 $G(x)$ has to reach b or further. Right, in Lemma 18.3 the intervals $[a, b]$, $[x, G(x)]$ and $[y, G(y)]$ and their relative positions are marked.

Proof. 1. If $[x, G(x)] \subseteq [a, b]$ then $[x, G(x)] \subseteq [x, b] \in \mathcal{V}$, since $[x, b] \subseteq [a, b]$ and $[a, b] \in \mathcal{V}$. Using Lemma 17.3. we get $[x, b] \subseteq [x, G(x)]$ thus $G(x) = b$.

2. If $G(x) \notin [x, b]$, then x, b and $G(x)$ appear in that order on ∂P , thus $b \in [x, G(x)]$. If $G(x) \in [x, b]$ then $G(x) = b$ by Lemma 18.1.

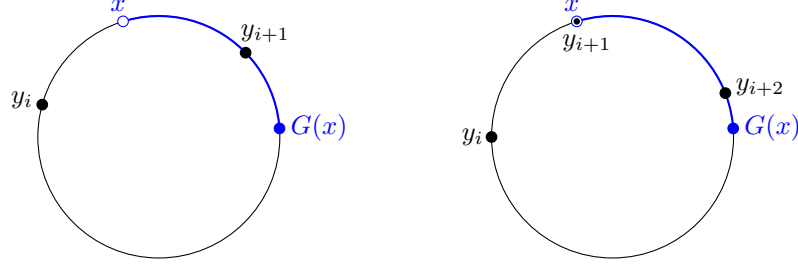
3. Since $x \in [x, y] \subseteq [a, b] \in \mathcal{V}$ we have by Lemma 18.2. that $y \in [x, G(x)]$ and using Lemma 18.2 again $G(x) \in [y, G(y)]$. If $G(x) \notin [y, b]$, we would contradict Lemma 18.1, so $G(x) \in [b, G(y)]$. $\blacktriangleleft$

An *optimal solution* refers to a set of points $y_0, \dots, y_{k^*-1}$ on the boundary such that they form a partition of ∂P into k^* visible intervals, i.e. $[y_i, y_{i+1}] \in \mathcal{V}$ and $\bigcup_{i=0}^{k^*-1} [y_i, y_{i+1}] = \partial P$ and k^* is minimal with this property. Next, we show that any *greedy step*, $(x, G(x))$, always contains at least one point from every optimal solution (similar to Lemma 2.3 in [20]).

► **Corollary 19** (Greedy steps contain optimal endpoints). Let $y_0, y_1, \dots, y_{k^*-1}$ be an optimal solution and $x \in \partial P$. Then $(x, G(x))$ contains at least one y_i .

Proof. Let $x \in [y_i, y_{i+1}]$, then $y_{i+1} \in [x, G(x)]$ by Lemma 18.2. If $y_{i+1} \neq x$ we are done (Figure 14 left). If $y_{i+1} = x$ we must have $[x, y_{i+2}] \in \mathcal{V}$ which by Lemma 17.3 implies that $[x, y_{i+2}] \subseteq [x, G(x)]$, and since $y_{i+1} \neq y_{i+2}$, we have $y_{i+2} \in (x, G(x))$ (Figure 14 right). $\blacktriangleleft$

Iteratively computing greedy steps is exactly Algorithm 1, leading to the following definition.



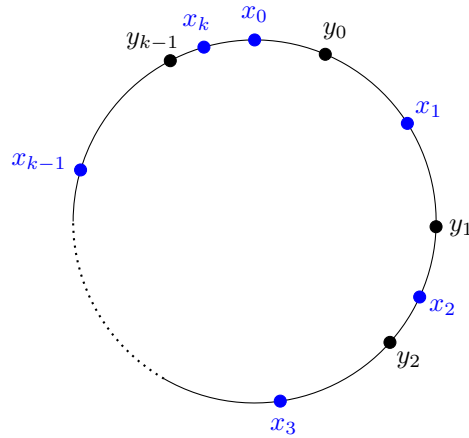
■ **Figure 14** All greedy steps contain a point from every optimal solution.

► **Definition 20** (Greedy Sequence). *Given a point x on ∂P , we denote the sequence $(x_i)_{i=0}^{\infty}$ with $x_0 = x$ and $x_{i+1} = G(x_i)$ as the greedy sequence starting at x . This is a sequence of endpoints of intervals. If none of the endpoints belong to an optimal solution then we call it a non-optimal greedy sequence. Otherwise, it is an optimal greedy sequence.*

Note that the greedy sequence $(x_i)_{i=0}^{\infty}$ is infinite, whereas Algorithm 1 only computes a prefix of this sequence. The goal of the rest of the paper is to show that for $T = \Omega(\text{poly}(n))$, then the sequence $(x_i)_{i=0}^T$ is optimal. Denote by *revolution* a single traversal around ∂P in $k+1$ steps, where k is the minimal number of greedy steps such that $G(x_k) = x_{k+1} \in [x_0, x_1]$.

Next, we show that the solution found after the first revolution is at most one interval longer than an optimal solution of size k^* (similar to Theorem 2.4 in [20]).

► **Theorem 21** (One revolution is at most one-off optimal). *Running Algorithm 1 from any point $x \in \partial P$ and stopping once $x_{k+1} \in [x_0, x_1]$ guarantees a solution of size $k \leq k^* + 1$.*



■ **Figure 15** Each greedy interval $(x_i, x_{i+1}]$ for $i = 0, 1, 2, \dots, k-1$ is disjoint and contains at least one point from any optimal solution by Corollary 19. This is not guaranteed for the last interval, i.e. the interval $(x_k, x_0]$ at the start of the next revolution which may not be a greedy interval.

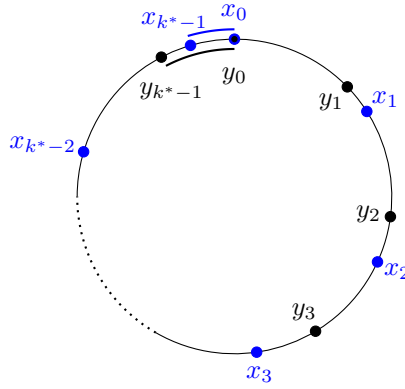
Proof. Let $(x_i)_{i=0}^{\infty}$ be the greedy sequence starting at x . Assume that $k > 1$ is minimal such that $x_{k+1} \in [x_0, x_1]$. Thus this candidate solution uses $k+1$ guards.

Let $y_0, \dots, y_{k^*-1}$ be an optimal solution. From Corollary 19 we know that for each $i = 0, 1, \dots, k^*-1$: $y_i \in (x_i, x_{i+1}]$, up to renumbering of the indices; see Figure 15. All these intervals are disjoint; hence an optimal solution contains at least $k-1$ endpoints, i.e. at least $k-1$ guards. Thus the greedy sequence yields a solution of size at most $k^* + 1$ ◀

Theorem 21 is very powerful since we now only need to distinguish between the case where Algorithm 1 is optimal and the case where it uses one extra guard. We now consider how the greedy sequence behaves in different settings, first if x_0 is an endpoint of an optimal solution:

► **Lemma 22.** *Let $x \in \partial P$ be a point in some optimal solution. Then Algorithm 1 returns an optimal solution in one revolution starting from x .*

Proof. Let $y_0, y_1, \dots, y_{k^*-1}$ be an optimal solution and $x = y_0$. Consider the first k^* terms of the greedy sequence starting at x : $x_0, x_1, \dots, x_{k^*-1}$, see Figure 16.



■ **Figure 16** An optimal solution $y_0, y_1, y_2, \dots, y_{k^*-1}$ and the greedy sequence $x_0, x_1, x_2, \dots, x_{k^*-1}$ starting from $x_0 = y_0$, that is also optimal since $[x_{k^*-1}, x_0] \subseteq [y_{k^*-1}, y_0]$ and $[y_{k^*-1}, y_0] \in \mathcal{V}$.

Since we know the length is optimal and the intervals $[x_0, x_1], \dots, [x_{k^*-2}, x_{k^*-1}]$ are visible, it is sufficient to show that $[x_{k^*-1}, x_0]$ is visible.

We have $x_0 = y_0$ by assumption and using Corollary 19 we know that each of the intervals $(x_{i-1}, x_i]$ contains a y_j for each $i = 1, 2, \dots, k^* - 2$. All these intervals are disjoint, and hence $y_i \in (x_{i-1}, x_i]$ for $i = 1, 2, \dots, k^* - 2$. Since $y_0, \dots, y_{k^*-1}$ is an optimal solution we have $[y_{k^*-1}, y_0] \in \mathcal{V}$. But since $y_{k^*-1} \in (x_{k^*-2}, x_{k^*-1}]$ we must have $x_{k^*-1} \in [y_{k^*-1}, y_0]$ and $[x_{k^*-1}, x_0] \subseteq [y_{k^*-1}, y_0] \in \mathcal{V}$, implying that $[x_{k^*-1}, x_0] \in \mathcal{V}$ and $x_0, \dots, x_{k^*-1}$ is an optimal solution. ◀

It follows immediately from Lemma 22 that once a greedy sequence reaches a point from an optimal solution, it will remain optimal from that point on.

► **Corollary 23** (Once optimal, always optimal). *Let $x \in \partial P$ and consider the greedy sequence starting at x . Assuming that there is some i such that $x_i, x_{i+1}, \dots, x_{i+k^*-1}$ is an optimal solution then $x_{i+j}, \dots, x_{i+k^*-1+j}$ will be an optimal solution for all $j \geq 0$.*

Proof. The proof follows by induction in j . For $j = 0$ the claim is trivial.

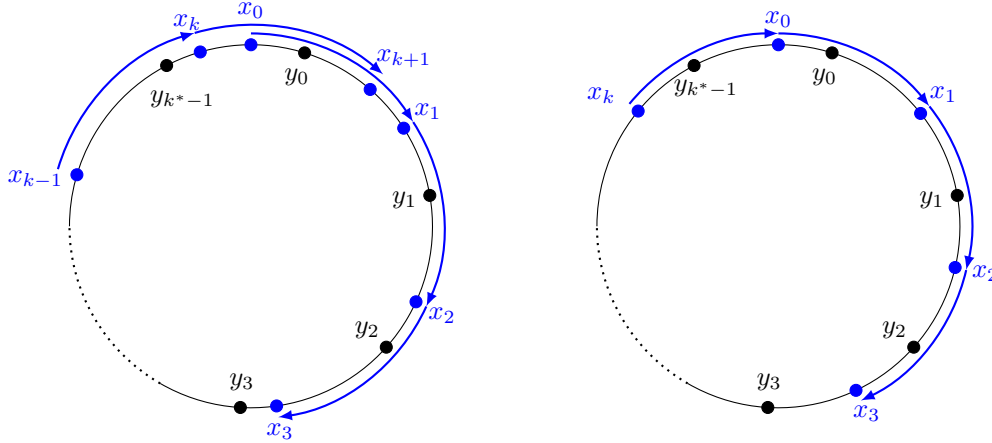
For the induction step, we assume $x_{i+j}, \dots, x_{i+k^*-1+j}$ is optimal. Thus x_{i+j+1} is an endpoint for an optimal solution, thus by Lemma 22 the greedy solution starting at x_{i+j+1} is optimal, which shows that the solution $x_{i+j+1}, \dots, x_{i+k^*-1+j+1}$ is optimal. ◀

We are now ready to determine when a solution is optimal. The first condition we establish is if the greedy sequence repeats in the first revolution:

► **Lemma 24** (Exact cover is optimal). *Let $x \in \partial P$, then if $x_i, x_{i+1}, \dots, x_{i+k}, x_i$ appear in the greedy sequence during a single revolution, then $x_i, \dots, x_{i+k}$ is an optimal solution.*

Proof. Let $y_0, y_1, \dots, y_{k^*-1}$ be an optimal solution. Each of the intervals $(x_{i+j}, x_{i+j+1}]$ (for $j = 0, \dots, k-1$) and $(x_{i+k}, x_i]$ must contain an endpoint from an optimal solution by Corollary 19. Thus, $k^* = k+1$ and $x_i, x_{i+1}, \dots, x_{i+k}$ is optimal. $\blacktriangleleft$

► **Remark 25.** The first $(x_0, x_1]$ and last step $(x_k, x_{k+1}]$ in a revolution of ∂P can overlap, hence an optimal point y_0 can satisfy the condition of Corollary 19 for both resulting in an $k^* + 1$ approximation. This is impossible when the greedy sequence repeats, see Figure 17.



■ **Figure 17** Left, a greedy sequence without repetitions where y_0 appears in both $(x_0, x_1]$ and $(x_{k-1}, x_k]$ which explains how a greedy revolution might not be optimal. Right, a greedy sequence that repeats after each revolution. Greedy intervals are visualized as directed circle arches.

We now know that if the greedy sequence repeats after a single revolution of the polygon, we have an optimal solution. However, to determine that a solution is optimal, we will need weaker conditions than repeating after just one revolution. To do this, we investigate how a non-optimal greedy sequence traverses ∂P , in relation to an optimal solution, see Figure 18.

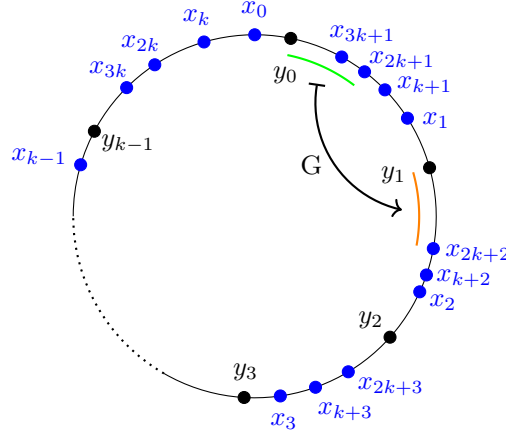
► **Proposition 26** (Behavior of non-optimal greedy sequences). *Let $x \in \partial P$ and consider the non-optimal greedy sequence $(x_i)_{i=0}^{k \cdot N}$ starting at x . Let $y_0, y_1, \dots, y_{k^*-1}$ be an optimal solution with $y_0 \in (x_0, x_1]$. Then $x_{ik+m} \in [y_{m-1}, x_{(i-1)k+m}]$ for all $m \in \{1, \dots, k\}$ and $i \in \{1, 2, \dots, N-1\}$.*

Proof. Like in the proof of Theorem 21, we distribute the points of an optimal solution y_i in the first k greedy steps. We can do this since the greedy sequence is not optimal implying that the first k greedy steps do not complete the first revolution. Theorem 21 implies that $x_{k+1} \in [x_0, x_1]$. We have $y_i \in (x_i, x_{i+1}]$ for $i = 0, \dots, k-1$. Now $x_{k+1} = G(x_k)$ is the first element of the greedy sequence to lie in $[x_0, x_1]$. More specifically, we must have $x_{k+1} \in [y_0, x_1]$, because if not then $x_{k+1} \in [x_0, y_0)$ and $[y_{k-1}, y_0] \not\subseteq (x_{k-1}, x_k]$ and since $[y_{k-1}, y_0]$ is visible, we contradict Lemma 17.3.

We now show $x_{ik+m} \in [y_{m-1}, x_{(i-1)k+m}]$ for $i \leq N$ and $m \in \{1, \dots, k\}$ by induction in $ik+m$. The induction start is exactly $x_{k+1} \in [y_0, x_1]$. For the induction step, assume that $x_{ik+m} \in [y_{m-1}, x_{(i-1)k+m}]$. If $m = k$ we will have a carry when we take a step (i.e. increment i by 1 and set $m = 1$), otherwise, we will simply add one to m . We assume we are in the second case to ease notation, but the same argument holds in the carry case.

It is clear, that $x_{(i-1)k+m} \in [y_{m-1}, y_m]$, combining this with $x_{ik+m} \in [y_{m-1}, x_{(i-1)k+m}]$ and Lemma 18.3 we get $G(x_{ik+m}) \in [y_m, G(x_{(i-1)k+m})]$, i.e. $x_{ik+m+1} \in [y_m, x_{(i-1)k+m+1}]$, see Figure 18.

Finally, since we assumed that none of the candidate solutions are optimal within kN greedy steps, the sequence will not repeat in one revolution as this would contradict Lemma 24, thus $x_{ki+m} \in [y_{m-1}, x_{k(i-1)+m})$ for all relevant $i < N$ and $m \in \{1, 2, \dots, k\}$. ◀



■ **Figure 18** For a non-optimal greedy sequence G maps points from $[y_0, x_{2k+1}]$ (green) to $[y_1, x_{2k+2}]$ (orange), and in the next revolution from $[y_0, x_{3k+1}]$ to $[y_1, x_{3k+2}]$.

► **Remark 27.** It is clear that $x_{ik+m} \in [y_{m-1}, x_{(i-1)k+m})$ implies $x_{ik+m} \in [x_{m-1}, x_{(i-1)k+m})$. This is relevant in practice, as we do not know the location of the y_i 's. Thus, the result of Proposition 26 can be restated as $x_{ik+m} \in [x_{m-1}, x_{(i-1)k+m})$ (these intervals are similar to the zones defined in [20]).

Proposition 26 shows that there is an important structure in the greedy sequence when viewed locally, which we capture in the following definitions:

► **Definition 28 (Local Greedy Sequence).** Let $x \in \partial P$, consider the greedy sequence $(x_i)_{i=0}^\infty$ starting at x . Then $(x_i^m)_{i=0}^\infty = (x_{ik+m})_{i=0}^\infty$ for $m = 1, \dots, k$ are local greedy sequences.

For non-optimal greedy sequences, its local greedy sequences move counterclockwise around ∂P by Lemma 26. We make this precise in the following definition.

► **Definition 29 (Fingerprint).** Let $(x_i^m)_{i=0}^\infty$ be the m 'th local greedy sequence. For $i \geq 1$ we say that x_i^m has a negative fingerprint if $x_i^m \in [x_{m-1}, x_{i-1}^m)$. Otherwise, we say that x_i^m has a positive fingerprint.

With the notion of fingerprints, we introduce *combinatorial optimality conditions*, which if satisfied by a greedy sequence guarantees that it is optimal, that is, it contains an optimal solution. The contrapositive of Proposition 26 gives the first such condition: A local greedy sequence element with a positive fingerprint is optimal.

► **Corollary 30 (Positive fingerprint implies optimality).** Let $x \in \partial P$. Assume that x_i^m has a positive fingerprint. Then $x_i^m, x_i^{m+1}, x_i^{m+2}, \dots, x_{i+1}^{m-1}$ is an optimal solution.

Proof. Proposition 26 states that if the sequence $x_0, \dots, x_{ik+m}$ has no subsequence that constitutes an optimal solution, then all individual points have negative fingerprints. The contrapositive of this statement is that if a point x_j in the sequence has a positive fingerprint, then it is a point in an optimal solution, which in turn, by Lemma 22 implies that $x_i^m, \dots, x_{i+1}^{m-1}$ is an optimal solution. ◀

The second combinatorial optimality condition states that if the local greedy sequence has moved out of the interval of the first revolution, then it is optimal.

► **Corollary 31** (Escaping an interval implies optimality). *Let $x \in \partial P$. If an element x_i^m of a local greedy sequence is not in $[x_{m-1}, x_m)$, then x_i^m is the starting point of an optimal solution.*

Proof. As $[x_{m-1}, x_m) \subseteq [x_{m-1}, x_{ik+m})$ for all i , this is a direct consequence of Remark 27. ◀

The third combinatorial optimality condition is perhaps the strongest; it states that if the greedy sequence has a periodic subsequence of any length, i.e., contains a repetition, then it is optimal.

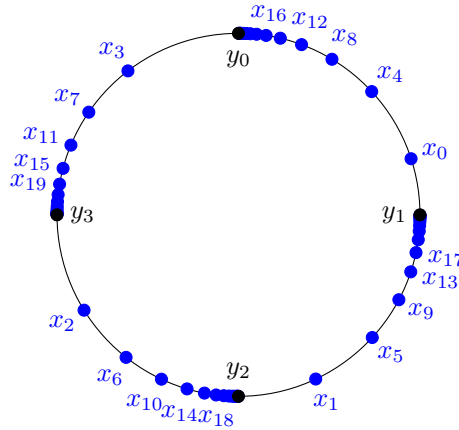
► **Corollary 32** (Periodicity implies optimality). *Let $x \in \partial P$ and $(x_i)_{i=0}^\infty$ be the greedy sequence starting at x . Assume there exists two indices $i < j$, such that $x_i = x_j$. Then $x_j, x_{j+1}, \dots, x_{j+k^*-1}$ is an optimal solution.*

Proof. Assume for contradiction that none of $x_0, \dots, x_j$ are optimal. Let for each $\ell > 0$

$$V_\ell = \bigcup_{m=1}^k [y_{m-1}, x_{\ell-1}^m).$$

After ℓk greedy steps, all future points in the greedy sequence will be contained in V_ℓ by Proposition 26, and since $V_\ell \cap \{x_t \mid t = 0, \dots, \ell k\} = \emptyset$, a point in the greedy sequence cannot be repeated. This is a contradiction, so one of the x_t must be an endpoint of an optimal solution. So by Corollary 23 $x_j, x_{j+1}, \dots, x_{j+k^*-1}$ is an optimal solution. ◀

We have established conditions in Corollary 30, 31 and 32 that characterize when we can guarantee that the greedy sequence is optimal. However, using only combinatorial insights, we still cannot distinguish between optimal and non-optimal greedy sequences (see Example 58 in Appendix C.1) nor guarantee that running Algorithm 1 long enough yields an optimal solution (See Figure 19). Thus, we return to the geometric setting with Corollary 30, 31 and 32 as optimality conditions.



■ **Figure 19** Visualization of a greedy sequence, that never reaches a combinatorial optimality condition since the local greedy sequences converge to their respective y 's but never reach them.

5 Algorithm 1, a geometric viewpoint

In this section, we explore the behavior of Algorithm 1 in the geometric setting. We observe the behavior of the (local) greedy sequences by adapting the combinatorial optimality conditions Corollaries 30, 31 and 32 to include geometric observations and finally show that the sequence contains an optimal solution within a polynomial number of revolutions.

First, Corollary 30 and 31 can be related in the geometric to when local greedy sequences move onto new edges:

► **Definition 33** (Edge jumps). *Let $(x_i^m)_{i=0}^\infty$ be a local greedy sequence. We say that $(x_i^m)_{i=0}^\infty$ is an edge jump at i if x_i^m is a vertex of P or x_i^m and x_{i+1}^m are in different edges of P and neither is a vertex of P .*

► **Lemma 34** (Maximal edge jumping). *Let $(x_i^1)_{i=0}^\infty, \dots, (x_i^k)_{i=0}^\infty$ be the local greedy sequences. If there are at least $n + 1$ edge jumps across all the local greedy sequences then some local greedy sequence $(x_i^m)_{i=0}^\infty$ will lie outside $[x_{m-1}, x_m]$ after these edge jumps.*

Proof. Let n_m be the number of vertices of P in (x_{m-1}, x_m) and n_{k+1} the number of vertices in (x_k, x_0) . Then $\sum_{m=1}^k n_m \leq \sum_{m=1}^{k+1} n_m \leq n$. If the local greedy sequence $(x_i^m)_{i=0}^\infty$ jumps $n_m + 1$ times it will at some point have passed x_{m-1} and thus one of the points must lie in $[x_{m-2}, x_{m-1})$. The lemma follows by the pigeonhole principle. ◀

This leads to a new progress condition based on edge jumps:

► **Corollary 35** (Many edge jumps implies optimal). *Let $(x_i)_{i=0}^N$ be a greedy sequence whose local greedy subsequences $n + 1$ edge jumps. Then the revolution after x_N will be optimal.*

Proof. The proof is immediate from Lemma 34 and Corollary 31. ◀

► **Remark 36** (Geometric progress conditions). We establish three progress conditions on the basis of our combinatoric optimality conditions and extend them with geometric results. If one of these are satisfied, we discover an optimal solution or make progress towards one.

1. If we see a positive finger print, the greedy sequence is optimal (Corollary 30).
2. If we see a repetition in the greedy sequence, the sequence is optimal (Corollary 32).
3. If we see $n + 1$ edge jumps, the greedy sequence is optimal (Corollary 35).

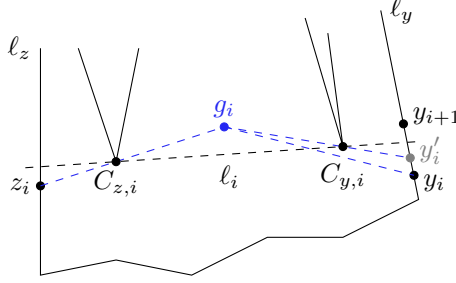
The goal of the remainder of this section is to prove that we eventually make progress:

► **Theorem 37** (Algorithm 1 will reach a progress condition). *Running GREEDYINTERVAL for $\mathcal{O}(kn^2)$ revolutions guarantees the occurrence of a progress condition.*

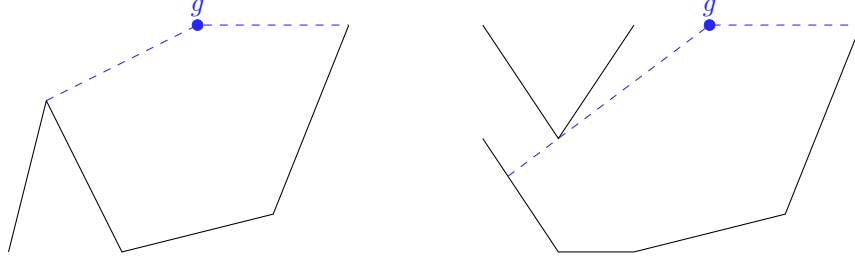
We will prove Theorem 37 by considering the local behavior of GREEDYINTERVAL when applied to one local greedy sequence. We will need terminology to describe this; see Figure 20.

We let $(y_i)_{i=0}^\infty$ and $(z_i)_{i=0}^\infty$ be local greedy sequences with $z_i = G(y_i)$. We let g_i be the guard found by GREEDYINTERVAL, that sees $[y_i, z_i]$. If $(y_i)_{i=0}^\infty$ or $(z_i)_{i=0}^\infty$ jumps edges, we will have a progress condition, so we assume that this does not happen. Let $(y_i)_{i=0}^\infty$ be contained on ℓ_y and $(z_i)_{i=0}^\infty$ be contained on ℓ_z . We also know, that $(y_i)_{i=0}^\infty$ will move up along ℓ_y and $(z_i)_{i=0}^\infty$ will move down along ℓ_z , as we otherwise would get a positive finger print. Of all the z_i it is only possible for z_0 to be a vertex of P . This is important, as we now know that all other z_i will be defined from a *blockage*, i.e. g_i can see no further than z_i , because there is some vertex of P in the way (see Figure 21).

So we have at least one blockage between z_i and g_i , the closest to z_i we denote $C_{z,i}$. If y_{i+1} could see g_i , then g_i could see $[y_{i+1}, z_i]$, thus we would repeat in the greedy sequence



■ **Figure 20** A greedy step from y_i on edge ℓ_y to $z_i := G(y_i)$ on edge ℓ_z guarded by g_i , however g_i can see from y'_i . The pivot point $C_{z,i}$ blocks g_i from seeing further than z_i and the other pivot $C_{y,i}$ blocks g_i in the other direction, so g_i , $C_{y,i}$ and y'_i lie on a line. The relative position of the line $\ell_i := L(C_{z,i}, C_{y,i})$ between the pivot points to the guard g_i will be vital for our analysis.



■ **Figure 21** Types of end points with (left) a horizon and (right) a blockage end point

(as $z_i = z_{i+1}$). Assuming this does not happen, there must be something blocking g_i from y_{i+1} . We let y'_i be the point on ℓ_y farthest up, which is still visible from g_i . As it is farther up, there must be some blockage between y'_i and g_i , the closest to y'_i we denote $C_{z,i}$.

We will refer to the points $C_{y,i}$ and $C_{z,i}$ as *pivot points* and $\ell_i := L(C_{y,i}, C_{z,i})$ the *pivot line* for a given i . It is clear that $C_{y,i} \neq C_{z,i}$ so the pivot line is well defined. The pivot points are all vertices of the polygon, thus there are at most $\mathcal{O}(n^2)$ possible pivot lines. In the following, we will look only at one pivot line at a time, and then run enough revolutions to see the same pivot line multiple times by the pigeonhole principle.

Finally, we track where we can place guards. For this, we define feasible regions:

► **Definition 38 (Feasible region).** Let $y, z \in \partial P$ and $[y, z]$ the interval of P from y to z . Then the feasible region $F([y, z])$ is the subset of P of all possible guard placements that see $[y, z]$. Specifically, $F([y, z]) \neq \emptyset$ if and only if $[y, z] \in \mathcal{V}$.

In the following, for a greedy step $G(y_i) = z_i$ where y_i and z_i are on edges ℓ_y and ℓ_z , we define $F := F([y, z])$, where y and z are the endpoints of edges ℓ_y and ℓ_z contained in $[y_i, z_i]$.

Strategy

We briefly outline our approach, which is based on the relationship between the feasible region and the pivot line in a greedy step, $G(y_i) = z_i$. In Section 5.2, we show that the case where F is entirely above ℓ_i will lead to a progress condition. In Section 5.3, we show that if the feasible region F contains points below ℓ_i , then within the next revolution a different pivot line $\hat{\ell}_i$ is visited, with corresponding feasible region $\hat{F}$ entirely above $\hat{\ell}_i$.

5.1 Supporting lemmas

Throughout this section, we will need the following two properties:

► **Lemma 39.** *For any $a, b \in \partial P$, the feasible region $F([a, b])$ is connected. Furthermore, for any convex subset $C \subset P$, $C \cap F([a, b])$ is convex.*

Proof Sketch. The proof follows by induction in the number of vertices in $[a, b]$. In the base case, the feasible region is a single visibility polygon, hence it is connected. In the induction step, we intersect the previous feasible region with a visibility polygon from a vertex v_{i+1} . If this new feasible region F were to be disconnected, we find two points S, T in different components of F , where $\overline{ST}$ is contained in a triangle completely visible to v_{i+1} , i.e. completely contained inside P . This will contradict Lemma 55 in Appendix B.1 and thus F will be connected. A similar argument shows in convexity claim.

For the detailed proof, see Appendix B.1. ◀

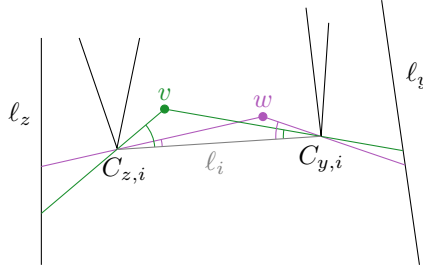
► **Lemma 40** (Blockings happen outside visible area). *Let $y \in \partial P$ and $z = G(y)$ and assume that neither y nor z is a vertex of P . Let g be a guard seeing $[y, z]$ and let c be a vertex blocking the view from g to z . Then $c \notin [y, z]$.*

Proof Sketch. One considers what g can see around C . It can be shown, that g cannot see everything in the vicinity of C (Lemma 57 in Appendix B.2). Assume for contradiction that $C \in [y, z]$, then either some part of $[y, z]$ is not visible from g or z can be moved to a visible vertex, yielding a contradiction.

For the detailed proof, see Appendix B.2. ◀

5.2 Feasible region strictly above pivot line

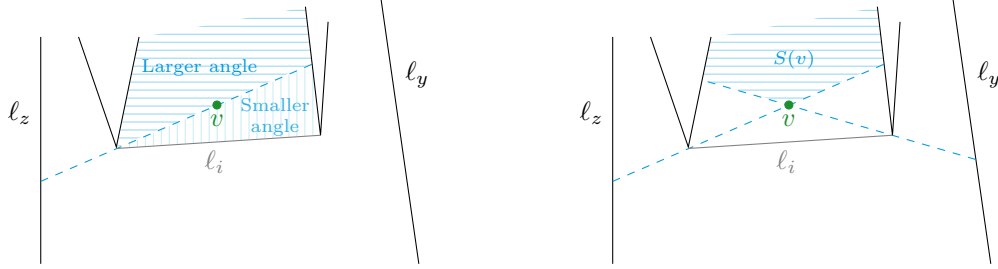
In this section, we analyze configurations in which the feasible region lies strictly above a pivot line. The main result of this section is to show that the guard will have a unique optimal position in F . This will imply a progress condition under the assumption that the same pivot line is chosen twice. In Figure 22 we see that the angles between the pivot points and the points above the pivot line are a good measure of the quality of prospective guard positions. In Figure 23 we use this insight to define the *shadow* of a point, which is used in Lemma 43 to show that the feasible region is contained in the shadow of one of its unique point closest to ℓ_i . Finally, in Proposition 44 we show that whenever the feasible region is above the pivot line, a progress condition will be satisfied.



■ **Figure 22** In the setup of Figure 20, a guard above the pivot line ℓ_i placed in point v can see more of the ℓ_y edge than if it was placed in w , and vice-versa, analogously to Figure 2.

Fixing v , it is clear that the area where a guard will have a smaller angle than v with respect to ℓ_i is below $L(v, C_{z,i})$, and a larger angle above $L(v, C_{z,i})$ see Figure 23.

The intersection of the half-planes above $L(a, C_{z,i})$ and $L(v, C_{y,i})$ is a quarter plane where every guard placement is worse than v , both with respect to ℓ_y and ℓ_z . For a point $v \in F$ we denote this quarter plane $S(v)$ and call it the *shadow* of v ; see Figure 23 (right).

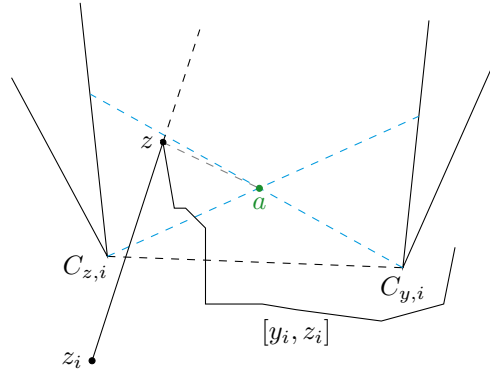


■ **Figure 23** Left, points with smaller angle than v with ℓ_i see more of ℓ_z and are better guards. Right, placing a guard strictly in the shadow $S(v)$ of v leads to a strictly worse guard.

► **Remark 41** (z above ℓ_i). Let z be the vertex of ℓ_z in $[y_i, z_i]$ and a is the vertex of F closest to ℓ_i . If z lies above $L(a, C_{z,i})$, as illustrated in Figure 24, we see some weird behavior:

For $C_{z,i}$ to be the pivot, z_i has to lie below ℓ_i , thus a cannot see any part of ℓ_z except for z . Thus it makes sense to force the points in F to see a small part of ℓ_y and ℓ_z . This can be done by setting $F = F([y - \varepsilon, z + \varepsilon])$, where $y - \varepsilon$ and $z + \varepsilon$ refers to points on ℓ_y respectively ℓ_z very close to y respectively z .

Making this change will not impact the other proofs other than fixing some special cases in this subsection. By doing this, z cannot lie above $L(a, C_{z,i})$, which is important for future proofs. The analogous behavior holds true if y lies above $L(C_{y,i}, a)$.



■ **Figure 24** a can see nothing on ℓ_i but z . Thus picking a as a guard would lead to edge jumps immediately

► **Lemma 42** (Successor and predecessor edges to $a \in F$ lie in $S(a)$). Assume F lies strictly above ℓ_i and let a be a vertex of F closest to ℓ_i . Let e and f denote the edges of F connected to a . Then $e, f \subseteq S(a)$.

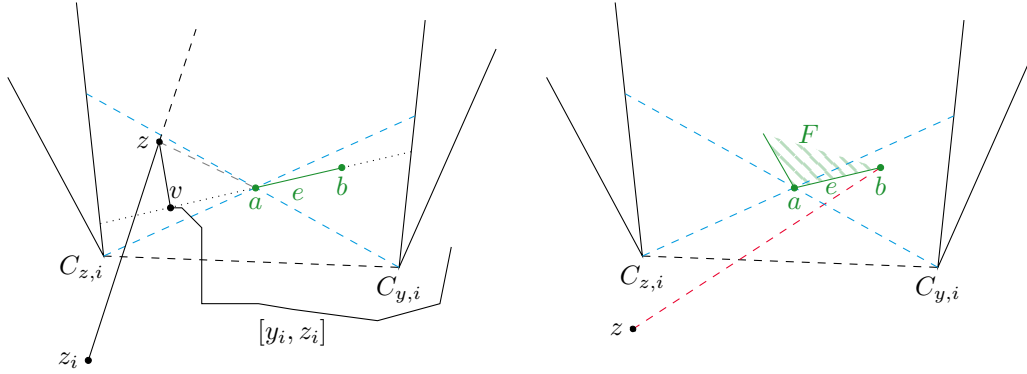
Proof. We will start by determining what characterizes edges in F . Since F is the intersection of the visibility polygons of vertices of ∂P , an edge of F has to be an part of an edge of the visibility polygon to a vertex v .

The edges of the visibility polygon from v come in two ways: Either they come from shooting rays from v through other vertices of P or as edges of P .

Assume for contradiction that e or f is not contained in $S(a)$. As the proof is symmetric, we assume $e \not\subseteq S(a)$, e 's other endpoint is b and b is closer to $C_{y,i}$ than $C_{z,i}$ (again by symmetry). Furthermore let z be the last vertex of $[y_i, z_i]$, thus z can see F and hence also points a and b . Now e must either be a segment contained in a ray from a vertex $v \in [y_i, z_i]$ or e is a part of ∂P .

▷ **Case 1.** If $e \subseteq \overrightarrow{v\bar{a}}$ we must have $v \in L(b, a)$. For z to see a , z must lie above $L(b, a)$ as $[v, z]$ would block it otherwise. However now we would need z above $L(a, C_{z,i})$, which we disallow by Remark 41. Thus we arrive at a contradiction.

▷ **Case 2.** If e is a part of ∂P , we can assume F lies above $L(b, a)$, since a is a lowest point of F ; see Figure 25 (right). However, now z cannot see b as the space directly below $\overline{ab}$ is not inside P and Remark 41 forcing z below $L(b, a)$, leading to a contradiction. ◀



■ **Figure 25** Left, if z sees a , z is above $L(b, a)$ and z_i is below ℓ_i for $C_{z,i}$ to be a pivot. Right, if nothing blocks visibility between z and b then a is not closest to $\ell_i = \overline{C_{y,i}C_{z,i}}$.

► **Lemma 43** (The feasible region is contained in the shadow). Assume that F lies above ℓ_i and let a be a vertex of F closest to ℓ_i . Then $F \subseteq S(a)$.

Proof. Assume for contradiction there are points in F outside of $S(a)$. Let b be a point in F such that $\angle baC_{y,i}$ or $\angle C_{z,i}ab$ is minimal (depending on whether b is closer to $C_{y,i}$ or $C_{z,i}$). Assume w.l.o.g. b is closer to $C_{y,i}$ than $C_{z,i}$. Because of the minimality of $\angle baC_{y,i}$, we now know all of F is above $L(b, a)$.

By Lemma 42 we know b is not the successor or predecessor to a , thus not all of $\overline{ab}$ will be in F . So for at least one vertex v , a and b will be visible, but not all of $\overline{ab}$. Let d be a point on $\overline{ab}$ not visible to v .

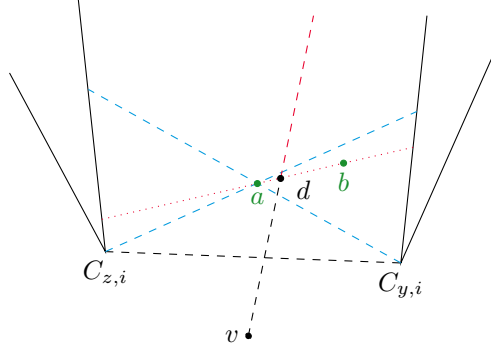
It must now hold that F cannot intersect the ray $\overrightarrow{v\bar{d}}$. Furthermore, F cannot cross $L(b, a)$ from the minimality of the angle (see Figure 26). Thus a and b must be in disconnected components of F . By Lemma 39 F must be connected, leading to a contradiction. ◀

We can now show that when F lies strictly above ℓ_i , we get a progress condition:

► **Proposition 44** (Feasible region over repeated pivot line implies a progress condition). If $\ell_i = \ell_j$ for some $i < j$ and F lies above ℓ_i , then we get a progress condition.

Proof. Let v be a vertex of F closest to ℓ_i . By Lemma 43 we know $F \subseteq S(v)$.

Consider y_i . If it can see any part of F , it can also see v , since $F \subseteq S(v)$, and any point in F can see no farther than v on ℓ_z , again because $F \subseteq S(a)$. Thus, z_i is the farthest point



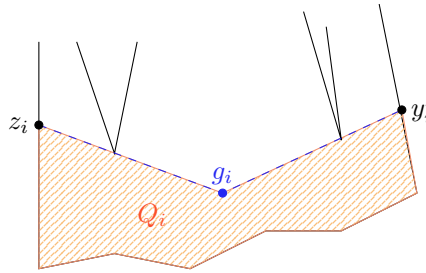
■ **Figure 26** Ray $\overrightarrow{vd}$ and $L(b, a)$ divides the feasible region F into disconnected components

on ℓ_z visible from v . If y_i could not see any point in F , then z_i could not lie on ℓ_z which is an edge jump, hence a progress condition.

Now consider y_j . We get the exact same considerations as above, so z_j is the farthest point on ℓ_z visible from v and $z_i = z_j$, a repetition, or $z_j \notin \ell_z$, and edge jump, both are progress conditions, as wanted. ◀

5.3 Feasible region below pivot line

In this section, we analyze what happens when F is not strictly above ℓ_i . Contrary to what we have seen in Section 5.2, when g_i can lie below ℓ_i , it does not hold that there is a unique optimal guard placement. Thus we will show that when F is on or below ℓ_i , we will be able to find some other guard, which is found by GREEDYINTERVAL, that lies strictly above its pivot line, thus forcing a unique optimal guard placement at some other point in the polygon. We denote the polygon constructed by the edges of $[y'_i, z_i]$, $\overline{z_i g_i}$ and $\overline{g_i y'_i}$ by Q_i , see Figure 27. Recall that y'_i is defined considering the last point on ℓ_y visible to g_i .



■ **Figure 27** The polygon Q_i defined by $[y'_i, z_i]$, $\overline{z_i g_i}$ and $\overline{g_i y'_i}$.

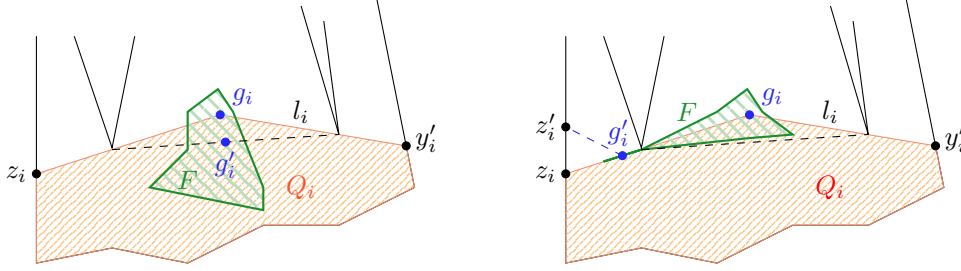
► **Lemma 45.** $[z_i, y'_i]$ does not cross the boundary of Q_i .

Proof. The guard g_i must be able to see y'_i and z_i , so $\overline{g_i y'_i}$ and $\overline{g_i z_i}$ must not be obstructed. Since all the boundary segments in $[y'_i, z_i]$ cannot be obstructed either as P is simple, no part of $[z_i, y'_i]$ (i.e. the interval not guarded by g_i) can cross the boundary of Q_i . ◀

If F lies above and below ℓ_i we could potentially have guards below and above. However, the following lemma shows that guards are below the pivot line if possible.

► **Lemma 46** (Guards are placed on or below the pivot lines if possible). Let ℓ_i^- be the closed half-plane below the i th pivot line. If $F \cap \ell_i^- \neq \emptyset$ then g_i will lie on or below ℓ_i .

Proof. Assume for contradiction that g_i lies strictly above ℓ_i . It holds that F is connected by Lemma 39. Combining this with the $F \cap \ell_i^- \neq \emptyset$ condition, then there must be a point on $\overline{C_{y,i} C_{z,i}} \subset \ell_i$ also contained in F (ℓ_i is the extended line, where $\overline{C_{y,i} C_{z,i}}$ is only the line segment). Since $\overline{C_{y,i} C_{z,i}} \subseteq Q_i$ we know that $F \cap \ell_i^- \cap Q_i \neq \emptyset$. Let $g'_i \in F \cap \ell_i^- \cap Q_i$.

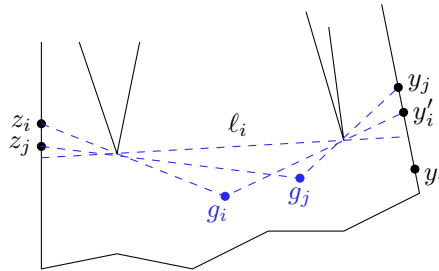


■ **Figure 28** Left, the setup of Lemma 46 with feasible region F , guard g_i that can see all of Q_i along with placement of g'_i . Right, g_i cannot be optimal as g'_i can see further than g_i .

As $g'_i \in F$, g'_i can see every vertex in $[y_i, z_i]$, and since $g'_i \in Q_i$, g'_i can also see y_i and z_i by Lemma 45 and Lemma 40, see Figure 28 left. Thus g'_i must also see all of $[y_i, z_i]$. However, we know from analyzing GREEDYINTERVAL (in Remark 6), that multiple optimal guards must be collinear with z_i , so $g'_i \in L(z_i, g_i) = L(z_i, C_{z,i})$ and since $g'_i \in \ell_i^-$, g'_i must lie in $\overline{z_i C_{z,i}}$. But now no vertex of P is between g'_i and z_i (since $C_{z,i}$ is the vertex closest to z_i between z_i and g_i), so g'_i must be able to see more than z_i making g_i not optimal, leading to a contradiction. ◀

Now we know that the placement of the guard must be below ℓ_i , if possible. We next prove that the points y_i and z_i will both be above ℓ_i if we see ℓ_i again:

► **Lemma 47** (y_j and z_j will lie above ℓ_j). *If $F \cap \ell_i^- \neq \emptyset$ and $\ell_i = \ell_j$ with $i < j$. Then y_j and z_j will both lie above ℓ_i or we reach a progress condition.*



■ **Figure 29** If the pivot line ℓ_i is reused at step $i < j$ then both y_j and z_j are above it.

Proof. We also have that $C_{y,i}$ lies on $\overline{g_i y'_i}$, thus y'_i lies above ℓ_i , and since $i < j$, either y_j will also lie above ℓ_i or we get a positive fingerprint.

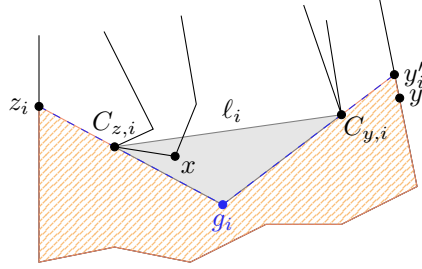
Analogously, $C_{y,i}$ lies on $\overline{g_i y'_i}$, thus y'_i lies above ℓ_i , and since $i < j$ (and we assume negative fingerprint), y_j will also lie above ℓ_i . ◀

We are now ready to give the main result of the subsection, essential in proving Theorem 37.

► **Proposition 48** (Guard placed below pivot implies other guard placed strictly above pivot). *Let g_i lies below the pivot line and y_i, z_i lie above the pivot line, then one of the following will occur:*

1. In the next revolution, we will see a progress condition.
2. z_{i+1} will lie below the pivot line ℓ_i .
3. There is some guard $\hat{g}$ found in the next revolution, which is below its pivot line $\hat{\ell}$.

Proof. Consider the triangle $\triangle gC_{z,i}C_{y,i}$. Let x be a point on $\partial P \cap \triangle gC_{z,i}C_{y,i}$ which is furthest below ℓ_i (it may be that $x = C_{z,i}$ or $x = C_{y,i}$). During the next $k+1$ greedy steps, we will at some point have a guard, that sees x as part of its interval. If x is the endpoint of such an interval, we have an edge jump. So we assume x is not an endpoint. Next, notice that $x \in [C_{z,i}, C_{y,i}]$, since if $x \in [y'_i, z_i]$, g_i would not be able to see y'_i and z_i and if $x \in [z_i, C_{z,i}]$ or $[C_{y,i}, y'_i]$ $x \in [z_i, C_{z,i}]$ or $x \in [C_{y,i}, y'_i]$, that piece of the boundary would need to enter Q_i contradicting Lemma 45.



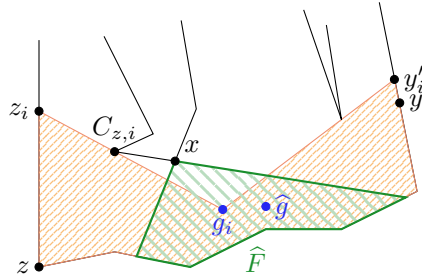
■ **Figure 30** A point $x \in \partial P$ furthest below ℓ_i must lie in the triangle $\triangle g_i C_{z,i} C_{y,i}$.

Let $\hat{g}$ be the guard that sees x . We introduce the same terminology around $\hat{g}$ as for g_i : $\hat{g}$ guards $[\hat{y}, \hat{z}]$ and $\hat{g}$ is blocked by $\hat{C}_z$ and $\hat{C}_y$ with $\hat{y}'$ being the furthest $\hat{g}$ can see when going backwards. Let $\hat{\ell} = L(\hat{C}_y, \hat{C}_z)$ be the pivot line associated with $\hat{g}$.

In the following, we will consider where $\hat{g}$, $\hat{C}_z$ and $\hat{C}_y$ can be placed in relation to each other. We will see that in all these cases one of the three criteria in the proposition will be satisfied.

Let $\hat{F}$ be the feasible region for the vertex x and the edges of P connected to x . $\hat{F}$ is obtained by extending the edges connected to x as in Figure 31.

Finally, let z be the vertex on $[y_i, z_i]$ just before z_i .



■ **Figure 31** The guard $\hat{g}$ that guards x must lie in the feasible region $\hat{F}$ of the two edges that share x as an endpoint.

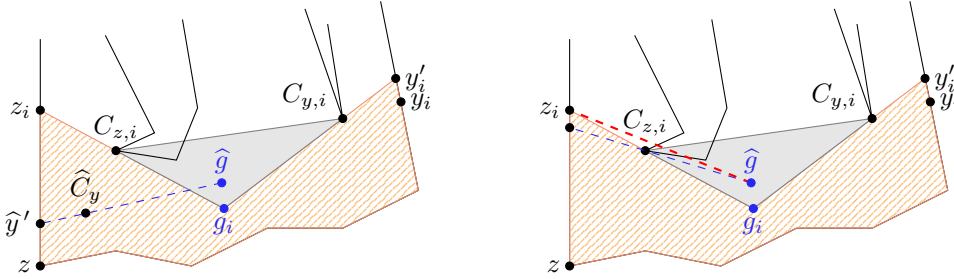
Since x is chosen lowest in $\triangle C_{z,i}C_{y,i}g_i \cap \partial P$, both edges in ∂P connected to x will point downwards (where ℓ_i is horizontal). Thus $\hat{F}$, whose boundary is the continuation of these edges, will be contained below ℓ_i .

We now look at the case, where $\hat{g}$ is not in Q_i :

If $\hat{g}$ is not in Q_i , it lies above $L(y'_i, g_i)$ and $L(g_i, z_i)$, as they are extensions of borders of Q_i . From the considerations above, we know that $\hat{g}$ must also lie below ℓ_i , thus $\hat{g} \in \triangle g_i C_{z,i} C_{y,i}$.

Furthermore, we know that x is the lowest point of ∂P inside $\triangle C_{z,i}C_{y,i}g_i$, and $\hat{g}$ is below x , thus below $\hat{g}$ there is no part of ∂P inside $\triangle gC_{z,i}C_{y,i}$.

We now assume for contradiction, that on of the pivots for $\hat{g}$, $\hat{C}_y$ or $\hat{C}_z$, lies below $\hat{g}$. Then $\hat{y}'$ or $\hat{z} \in [y'_i, z_i]$, since $\hat{y}'$, respectively $\hat{z}$, lie on the ray $\overrightarrow{\hat{g}\hat{C}_y}$, respectively the ray $\overrightarrow{\hat{g}\hat{C}_z}$, and these rays will be contained in the region of $\triangle g_iC_{z,i}C_{y,i}$ below $\hat{g}$ and Q_i (until they hit ∂P), where no part of $[z_i, y'_i]$ can enter by Lemma 45 (see Figure 32 left).



■ **Figure 32** Assuming $\hat{g}$ is not in Q_i and $\hat{C}_y$ below $\hat{g}$, we must have $\hat{y}$ in $[y'_i, z_i]$ (left). However $C_{y,i}$ blocks the view for $\hat{g}$ to see z_i (right), so $[\hat{y}, x]$ cannot both be visible from $\hat{g}$.

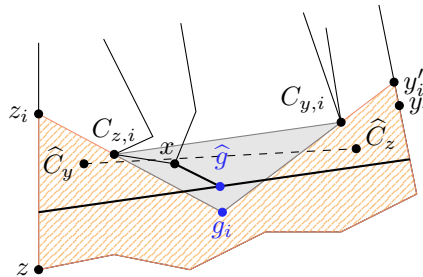
This implies that we would need $\hat{g}$ to be able to see all of $[z_i, x]$ or $[x, y'_i]$. However this is not possible, as $\hat{g}$ is strictly above the lines $L(C_{y,i}, g_i)$ and $L(g_i, C_{z,i})$ and thus the pivots $C_{y,i}$ and $C_{z,i}$ will block $\hat{g}$ from seeing y'_i and z_i . This yields a contradiction.

Now both $\hat{C}_y$ and $\hat{C}_z$ are above $\hat{g}$. From the assumption that $\hat{g}$ sees x , we know that $x \in [\hat{y}', \hat{z}]$, thus $\hat{C}_y$ will lie to the left of $\overrightarrow{\hat{g}x}$ and $\hat{C}_z$ to the right. So now $\hat{\ell}$ must lie below $\hat{g}$ and we have showed what we wanted (see Figure 33).

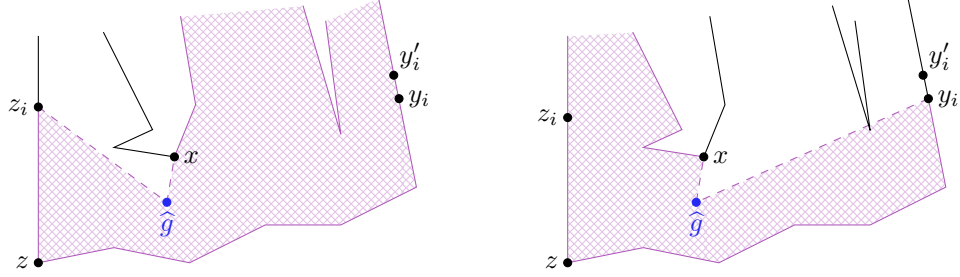
Now we assume that $\hat{g}$ lies in Q_i , and again we consider, where $\hat{C}_y$ and $\hat{C}_z$ can be located. Since x is in $[\hat{y}', \hat{z}]$, we cannot have $\hat{z} \in (z_i, x)$, as this would contradict the maximality of greedy step from y_i to z_i . Likewise we cannot have $\hat{y}' \in (x, y_i)$, again due to maximality. The possible placements of $\hat{z}$ and $\hat{y}$, along with the possible placements of $\hat{C}_z$ and $\hat{C}_y$ are marked on Figure 34.

▷ **Case 1.** If $\hat{z} \in (y'_i, y_i]$, then $\hat{z}$ will be in the same local greedy sequence as y_i , hence $\hat{z} = y_{i+1}$, but now g_i sees $[y_{i+1}, z_i]$, thus we must repeat endpoints and the greedy sequence repeats, which is one of the desired conditions.

▷ **Case 2.** If $\hat{z} \in (y_i, z_i]$, then $\hat{z}$ must be in the same local greedy sequence as z_i , i.e. $\hat{z} = z_{i+1}$, implying that $\hat{y}$ must be in the same local greedy sequence as y_i , so $\hat{y} = y_{i+1}$, but $\hat{y}$ is in $[y'_i, C_{z,i}]$, so y_{i+1} either jumps edges, has a positive fingerprint or $z_{i+1} = z_i$, a repetition.

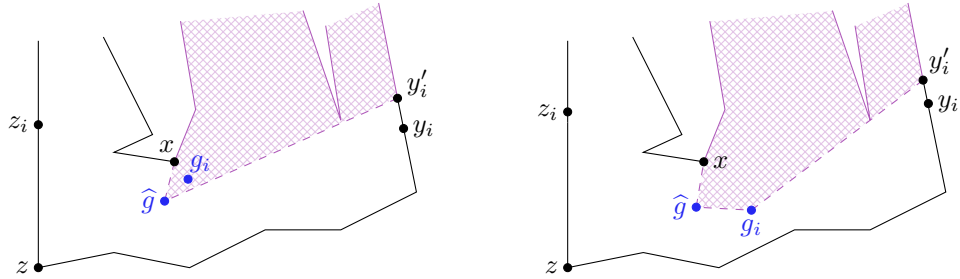


■ **Figure 33** If $\hat{C}_y$ and $\hat{C}_z$ are above $\hat{g}$, and $x \in [\hat{y}', \hat{z}]$ then we must have $\hat{g}$ above $\hat{\ell}$.



■ **Figure 34** Left, $\hat{z}$ must lie in $[x, z_i]$ and $\hat{C}_z$ in the polygon defined by $[x, z_i]$, $\overline{z_i \hat{g}}$ and $\overline{\hat{g} x}$. Right, analogously for $\hat{g}$ and $\hat{C}_y$.

▷ **Case 3.** Now $\hat{z} \in [x, y'_i]$. Consider the subcases where g_i lies above or below $L(y'_i, \hat{g})$: If g_i lies above, $\hat{C}_z$ must lie above $L(y'_i, \hat{g})$, since we otherwise would have the ray $\overrightarrow{\hat{g} \hat{C}_z}$ contained in Q_i (until it hits ∂P), since $\hat{g} \in Q_i$, making $\hat{z} \in (y'_i, z_i]$. If g_i lies below $L(y'_i, \hat{g})$, we must have $\hat{C}_z$ above $L(y'_i, g_i)$ and $L(g_i, \hat{g})$ for the same reason (see Figure 35).

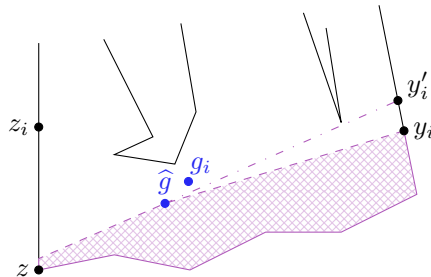


■ **Figure 35** Further restrictions to Figure 34. On the left, the possible placement of $\hat{z}$ and $\hat{C}_z$ are marked in red, when g_i lies above $L(y'_i, \hat{g})$. On the right, the possible placements when g_i lies below $L(y'_i, \hat{g})$.

We now look at where we can place $\hat{C}_y$ so that $\hat{g}$ lies below $\hat{\ell}$ in both of these cases:

▷ **Subcase 3.1** (g_i above $L(y'_i, \hat{g})$). If g_i lies above $L(y'_i, \hat{g})$, we must place $\hat{C}_y$ below $L(y'_i, \hat{g})$ for $\hat{\ell}$ to lie above $\hat{g}$. Combined with the previous restrictions, we see on Figure 36, the possible placements for $\hat{C}_y$.

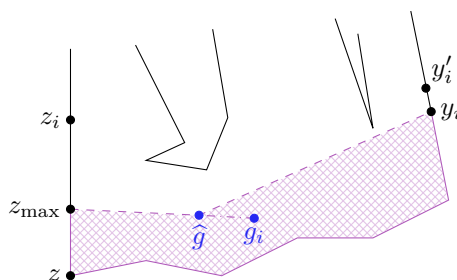
Now the ray $\overrightarrow{\hat{g} \hat{C}_y}$ will be contained in Q_i , so $\hat{y}' \in [y_i, z_i]$. This means, that $\hat{y}$ will be in the same local greedy sequence as z_i , thus $\hat{y} = z_i$. If $\hat{y}' \in [y_i, z]$, either $z_{i+1} \in [y_i, \hat{y}']$ and



■ **Figure 36** When g_i is over $L(y'_i, \hat{g})$ then $\hat{y}$ and $\hat{C}_y$ are restricted by y_i and $L(\hat{g}, y'_i)$

We have y'_i above ℓ_i and $\hat{g}$ below ℓ_i . Thus $\hat{y}'$ will also lie below ℓ_i . If $z_{i+1} \in [\hat{y}', z_i]$ we will again get a repetition, so we need $z_{i+1} \in [z, \hat{y}]$, and z_{i+1} is below ℓ_i , which is the third terminal condition of the proposition.

▷ **Subcase 3.2** (g_i below $L(y'_i, \hat{g})$). If g_i lies below $L(y'_i, \hat{g})$ we again consider where to place $\hat{C}_y$. To the right of $\hat{g}$, it must be below $L(y_i, \hat{g})$ by the previous arguments. To the left of $\hat{g}$ it must be below $L(g_i, \hat{g})$ for $\hat{g}$ to lie below $\hat{\ell}$ (see Figure 37).



Since $\hat{g}$ lies in Q_i , we must have that the ray $\overrightarrow{\hat{g}g_i}$ is contained in Q_i . Let the intersection of this ray with l_z be $z_{\max}$. Thus $\hat{y}' \in [y_i, z_{\max}]$ and we again have $\hat{y} = z_i$, likewise, if $\hat{y}' \in [y_i, z]$ we will get an edge jump or repetition as in case 3.1. So we assume $\hat{y} \in (z, z_i]$.

Considering all the above cases, we show that when $\hat{g}$ lies below $\hat{\ell}$ it implies either an edge jump, a repetition, a positive finger print or z_{i+1} being below ℓ_i , as wanted. $\blacktriangleleft$

► **Theorem 37** (Algorithm 1 will reach a progress condition). *Running GREEDYINTERVAL for $\mathcal{O}(kn^2)$ revolutions guarantees the occurrence of a progress condition.*

Proof. Assume for contradiction that we see no progress conditions in the first $\mathcal{O}(kn^2)$ revolutions of Algorithm 1. Let $(x_i^m)_{i=0}^\infty$ for $m = 1, \dots, k$ be the local greedy sequences. All these are contained on their respective edges (since we see no edge jumps). Let F^m be the feasible region for the vertices of ∂P in the interval $[x_{m-1}, x_m]$. Furthermore let ℓ_i^m be the pivot line for the pair (x_i^{m-1}, x_i^m) . Consider the choice of ℓ_i^1 for $i = 1, \dots, \mathcal{O}(kn^2)$ when Algorithm 1 makes $\mathcal{O}(kn^2)$ revolutions. We assume for contradiction that we see no progress conditions and consider the following possible cases:

▷ **Case 1.** If F^1 lies strictly above ℓ_i^1 , we cannot repeat ℓ_i^1 , as this leads to a progress condition (Proposition 44). Thus this can occur at most $\mathcal{O}(n^2)$ times (i.e. once for each possible pivot line).

▷ **Case 2.** If F^1 lies at or below ℓ_i^1 we consider whether it is the first time we have seen this pivot line or not:

▷ **Subcase 2.1.** If it is the first time ℓ_i^1 is a pivot line (i.e. $\ell_i^1 \neq \ell_j^1$ for all $j < i$), we cannot deduce anything. However, this can happen at most $\mathcal{O}(n^2)$ times (i.e. once for each possible pivot line).

▷ **Subcase 2.2.** If it is not the first time, Lemma 47 states that x_{i-1}^k and x_i^1 lie above ℓ_i^1 . Now the conditions of Proposition 48 are satisfied, so we either obtain a progress condition within one revolution, x_{i+1}^1 lies below ℓ_i^1 , or some guard g_i^m that lies above ℓ_i^m . We consider the last two cases.

▷ **Subsubcase 2.2.1.** If x_{i+1}^1 lies below ℓ_i^1 , we cannot use ℓ_i^1 again as this would break Lemma 47. Thus this case can happen at most $\mathcal{O}(n^2)$ times.

We have shown, that at most $\mathcal{O}(n^2)$ of the $\mathcal{O}(kn^2)$ i 's can fall into the above cases, hence we still have $\mathcal{O}(kn^2)$ i 's left for this final case:

▷ **Subsubcase 2.2.2.** Finally some other g_i^m lies above its pivot line ℓ_i^m . Since there are $k - 1$ other feasible regions, and each of these has $\mathcal{O}(n^2)$ pivot lines, at some point, we will have chosen the same feasible region with the same pivot line pair twice, by the pigeonhole principle. Since the guard is above this pivot line, the entire feasible region will also be above by contraposition of Lemma 46. Now we have seen the same pivot line twice with the same feasible region strictly above it, so Proposition 44 yields a progress condition. ◀

5.4 Proof of main theorem

Finally, we prove Theorem 1:

► **Theorem 1.** *The contiguous art gallery problem for a simple polygon with n vertices is solvable in $\mathcal{O}(k^*n^5 \log n)$ arithmetic operations, where k^* is the size of an optimal solution.*

Proof. Running Algorithm 1 for $\mathcal{O}(kn^2) = \mathcal{O}(k^*n^2)$ revolutions guarantees a progress condition i.e. a positive fingerprint, a repetition, or an edge jump by Theorem 37. Thus, repeating $n + 1$ times guarantees at least one positive fingerprint, at least one repetition, or $n + 1$ edge jumps. A positive fingerprint or repetition implies optimality by Corollary 30 and 32. If we have $n + 1$ edge jumps, we are optimal by Corollary 35. Now after $\mathcal{O}(k^*n^3)$ revolutions Algorithm 1 will find an optimal endpoint, and by Corollary 23 output an optimal solution. The algorithm makes a revolution in $\mathcal{O}(n^2 \log n)$ arithmetic operations by Corollary 13, the total number of arithmetic operations is $\mathcal{O}(k^*n^5 \log n)$. ◀

6 Bit Complexity of Algorithm 1

In this section we show how to bound the bit complexity of the points calculated by the GREEDYINTERVAL algorithm, to then bound the overall bit complexity of the arithmetic operations of both GREEDYINTERVAL and Algorithm 1. We define the bit complexity as follows, following the definition of Grötschel, Lovász and Schrijver [15].

► **Definition 49** (Bit complexity). *Let $\langle v \rangle$ denote the bit complexity of value v .*

- *If v is an integer, then $\langle v \rangle = 1 + \lceil \log_2(|v| + 1) \rceil$, i.e. the number of bits used to encode v , including the sign bit.*
- *If v is a fraction $\frac{v^n}{v^d}$, then $\langle v \rangle = \langle v^n \rangle + \langle v^d \rangle$.*
- *If v is a point (v_x, v_y) , then $\langle v \rangle = \langle v_x \rangle + \langle v_y \rangle$.*

Note that $\langle v \rangle$ denotes the number of bits used to represent the value v up to a constant factor, as it does not include bits used to separate, i.e., the numerator and denominator of a fraction. The notation is stronger than $\mathcal{O}$ -notation, as it does not allow to hide a possible constant, to ensure that the hidden constant in the $\mathcal{O}$ -notation truly is a constant. In the following lemmas, we will need the following rewriting properties of bit complexity on integers.

► **Lemma 50** (Properties of bit complexity). *Let a and b be integers. Then the following holds:*

- $\langle a \cdot b \rangle \leq \langle a \rangle + \langle b \rangle$.
- $\langle a + b \rangle \leq 1 + \max\{\langle a \rangle, \langle b \rangle\}$.

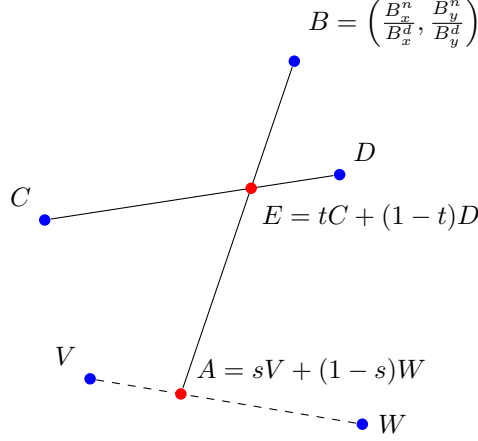
Proof. Follows from Definition 49 and logarithm rules. ◀

In this section we shall assume, that the input polygon P is encoded in N bits as a list of points, i.e., $4n$ integers encoded in binary. The polygon may be encoded using fewer bits by a different clever encoding, however, as the main goal is to show that contiguous art gallery problem is in the complexity class P, this simple encoding will suffice.

To bound the complexity of the points produced by GREEDYINTERVAL, we need the notion of a *scalar point*, which is defined as a point using two vertices of the input polygon and a fractional scalar, which denotes at what fractional point along the line defined by the two vertices the scalar point is located. Note that the scalar point does not need to be on the segment between the two points that defines it. Crucially, the intersection between two lines defined from three input vertices and a scalar point is a scalar point, as shown in the following lemma, which also bounds the bit complexity of the new scalar point as a function of the old.

► **Lemma 51** (Bit complexity of segment intersection). *Let N be the number of bits used to represent the input polygon. Let V, W, B, C , and D be input vertices, with $V = \left(\frac{V_x^n}{V_x^d}, \frac{V_y^n}{V_y^d}\right)$, and similarly for the other input vertices, where all values $v_x^n, v_x^d, v_y^n, v_y^d$ of the points are integers. Let s be a scalar $\frac{s^n}{s^d}$, with s^n and s^d being integers, where $\langle s \rangle$ may be of arbitrary complexity. Let A be the scalar point $sV + (1 - s)W$. Assume $L(A, B)$ and $L(C, D)$ are non-parallel lines and let the point E be their intersection. Then there exists a scalar $t = \frac{t^n}{t^d}$, such that the intersection E is a scalar point of the form $tC + (1 - t)D$, where $\max\{\langle t^n \rangle, \langle t^d \rangle\} \leq \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$.*

Proof. The setup is illustrated in Figure 39. Using a formula for the intersection of the lines defined by the segments $\overline{AB}$ and $\overline{CD}$ as stated by Goldman [13], the values of t^n and t^d



■ **Figure 39** Setup of the intersection. Blue points denote input vertices and the red points denote scalar points. Note that it is not a requirement that the segments $\overline{AB}$ and $\overline{CD}$ intersect, only that their lines intersect.

can be expressed as follows:

$$\begin{aligned}
 t^n = & \quad s^n (+ B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^d W_y^d + B_x^d B_y^d C_x^d C_y^d D_x^n D_y^d V_x^d V_y^d W_x^d W_y^n \\
 & + B_x^d B_y^n C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^n W_x^d W_y^d \\
 & - B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^d W_x^n W_y^d - B_x^d B_y^n C_x^d C_y^d D_x^n D_y^d V_x^d V_y^n W_x^d W_y^d \\
 & - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^d W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d) \\
 & + s^d (+ B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^d W_x^n W_y^d + B_x^d B_y^n C_x^d C_y^d D_x^n D_y^d V_x^d V_y^d W_x^d W_y^d \\
 & + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d - B_x^d B_y^d C_x^d C_y^d D_x^n D_y^n V_x^d V_y^d W_x^n W_y^d \\
 & - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d) \\
 \\
 t^d = & \quad s^n (+ B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^d W_y^d + B_x^d B_y^d C_x^d C_y^d D_x^n D_y^d V_x^d V_y^d W_x^d W_y^n \\
 & + B_x^d B_y^n C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^n W_x^d W_y^d \\
 & - B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^d W_x^n W_y^d - B_x^d B_y^n C_x^d C_y^d D_x^n D_y^d V_x^d V_y^n W_x^d W_y^d \\
 & - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^d W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d) \\
 & + s^d (+ B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^d W_x^n W_y^d + B_x^d B_y^n C_x^d C_y^d D_x^n D_y^d V_x^d V_y^d W_x^d W_y^n \\
 & + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^d W_y^d \\
 & - B_x^d B_y^d C_x^d C_y^d D_x^n D_y^n V_x^d V_y^d W_x^n W_y^d - B_x^d B_y^n C_x^d C_y^d D_x^n D_y^d V_x^d V_y^n W_x^d W_y^d \\
 & - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^n W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d)
 \end{aligned}$$

Following Lemma 50 it holds that $\langle t^n \rangle$ and $\langle t^d \rangle$ both are bounded by $\max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$, as the point values of the input vertices are bounded trivially by N , and therefore $\max\{\langle t^n \rangle, \langle t^d \rangle\} \leq \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$. ◀

Further, the bit complexity of a scalar point, when computed into a point, is bounded.

► **Lemma 52** (Bit complexity of scalar point). *Let N be the number of bits used to represent the input polygon. Let V and W be input vertices, with $V = \left(\frac{V_x^n}{V_x^d}, \frac{V_y^n}{V_y^d}\right)$, and similarly for W , where each component of the points are integers. Let s be a scalar $\frac{s^n}{s^d}$ of arbitrary*

complexity, with s^n and s^d being integers, and let A be the scalar point $sV + (1-s)W$. Then $\langle A \rangle \leq 4 \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$.

Proof. Let $A = (A_x, A_y)$. It then holds that

$$A_x = \frac{V_x^d W_x^n s^d - V_x^d W_x^n s^n + V_x^n W_x^d s^n}{V_x^d W_x^d s^d},$$

and similarly for A_y . Applying Definition 49 and Lemma 50 immediately concludes the proof. $\blacktriangleleft$

Using the lemmas on scalar points and intersections, we can show that when the input point to GREEDYINTERVAL is a scalar point, then the output point is a scalar point, which scalar complexity is bounded by the scalar complexity of the input point.

► **Lemma 53** (Bound output bit complexity of GREEDYINTERVAL). *Let P be a simple polygon encoded in N bits and let x be a scalar point on ∂P . Let V and W be vertices of P and s be a scalar $s = \frac{s^n}{s^d}$, s.t. $x = sV + (1-s)W$. Then there exists vertices V' and W' of P and a scalar $s' = \frac{s'^n}{s'^d}$, s.t. $G(x) = s'V' + (1-s')W'$, and it holds that $\max\{\langle s'^n \rangle, \langle s'^d \rangle\} \leq \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$.*

Proof. GREEDYINTERVAL proceeds in two phases. First, the feasible region F of x and a maximal number of contiguous edges from x is computed. Next, the guard seeing the furthest along the next edge of the boundary is then a corner of F , and $G(x)$ is found. We therefore need to bound the complexity of the corners of F to then bound the final output point.

The feasible region F is found by the intersection of the visibility polygons from x and some of the input vertices. It must therefore hold that each vertex of F is the intersection of lines from the visibility polygons. Each line of a visibility polygon is either from a segment of P , which is therefore a line from an input vertex to an input vertex, a line from a reflex vertex to an input vertex, which is an input vertex to an input vertex line, or from the point x to a reflex vertex. Two lines both containing x must intersect in x , and such point is by definition a scalar point between input vertices. Any other intersection is then either between four input vertices, or three input vertices and the point x . By Lemma 51, it holds that each corner of the feasibility region F is a scalar point on the form $s''V'' + (1-s'')W''$, for some input vertices V'' and W'' , and scalar $s'' = \frac{s''^n}{s''^d}$, and it holds that $\max\{\langle s''^n \rangle, \langle s''^d \rangle\} \leq \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$.

Next, the point $G(x)$ is found as the intersection of the line segment of some edge of P , and the line between a guard placement and a reflex vertex of P . As the guard placement is a corner of F , then the intersection is between three input vertices and a scalar point. Let the guard point be $s''V'' + (1-s'')W''$. By Lemma 51, then there are vertices V' and W' , which is the endpoints of the edge $G(x)$ is on, and a scalar $s' = \frac{s'^n}{s'^d}$, s.t. $G(x) = s'V' + (1-s')W'$, and it holds that $\max\{\langle s'^n \rangle, \langle s'^d \rangle\} \leq \max\{\langle s''^n \rangle, \langle s''^d \rangle\} + \mathcal{O}(N) \leq \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$. This therefore concludes the lemma. $\blacktriangleleft$

Similarly, it can be shown that the internal arithmetic operations GREEDYINTERVAL performs are bounded by the complexity of the input point.

► **Lemma 54** (Bound internal bit complexity of GREEDYINTERVAL). *Let P be a simple polygon encoded in N bits and let x be a point on ∂P . Then the internal arithmetic operations of GREEDYINTERVAL with input x is polynomially bounded in complexity by N and $\langle x \rangle$.*

Proof. Each internal arithmetic computation of GREEDYINTERVAL are computed by a constant size straight line program, which therefore is computeable in time linear in N and $\langle x \rangle$. By Theorem 3 GREEDYINTERVAL performs polynomially many such computations in N , which concludes the lemma. $\blacktriangleleft$

We now have the building blocks necessary to show the main theorem.

► **Theorem 2.** *The contiguous art gallery problem for a simple polygon encoded in N bits is a member of the complexity class P .*

Proof. Algorithm 1 repeatably applies GREEDYINTERVAL to make revolutions around the polygon. By Lemma 54 the internal arithmetic operations of GREEDYINTERVAL are bounded by the complexity of the input point and the polygon. It therefore suffices to argue for the bit complexity of each intermediate point on ∂P where GREEDYINTERVAL is applied to.

Algorithm 1 starts at some point on ∂P . Let this starting point be some vertex V of P . Let W be any other vertex of P . It then holds that the starting point is on the form $sV + (1-s)W$ for scalar $s = 1/1$. Let $s_t = \frac{s_t^n}{s_t^d}$ be the scalar after t applications of GREEDYINTERVAL. By induction and Lemma 53 it holds that $\max\{\langle s_t^n \rangle, \langle s_t^d \rangle\} \leq \max\{\langle s_0^n \rangle, \langle s_0^d \rangle\} + t \cdot \mathcal{O}(N) = \mathcal{O}(tN)$. By Theorem 1 the number of applications of GREEDYINTERVAL is $\mathcal{O}(k^*2n^3)$, which bounds t . As each scalar s_t corresponds to a scalar point is between some input vertices of P , then by Lemma 52 it therefore holds that the bit complexity of any point computed by GREEDYINTERVAL on the boundary ∂P is bounded by $\mathcal{O}(k^*2n^3 \cdot N)$. This, by Lemma 54, concludes the proof. $\blacktriangleleft$

7 Open problems

We showed that the contiguous art gallery problem is solvable in $\mathcal{O}(k^*n^5 \log n)$ time by bounding the number of revolutions before Algorithm 1 finds an optimal solution to $\mathcal{O}(k^*n^3)$ and showing that the bit complexity is bounded. We conjecture that $\mathcal{O}(1)$ revolutions are sufficient. To provide evidence for this we simulated more than 2.000.000 random art galleries using the provided C++ implementation, and in all instances, it found an optimal solution (a repetition even) within 4 revolutions. If this conjecture is true, the complexity of Algorithm 1 becomes $\mathcal{O}(n^2 \log n)$. Analyzing the behavior of Algorithm 1 on axis-aligned input polygons appears to be a good step towards proving this conjecture.

We leave it as an intriguing open problem to decide whether the contiguous art gallery problem with holes is polynomial-time solvable, in contrast to the other art gallery variants we considered (see Table 1).

If an unrestricted guard may cover h intervals ($h > 1$), we believe the problem is NP-hard. This is the case for $h = n$ since this is exactly the edge-covering art gallery problem.

References

- 1 Mikkel Abrahamsen, Anna Adamaszek, and Tillmann Miltzow. The art gallery problem is $\exists\mathbb{R}$ -complete. *J. ACM*, 69(1), December 2021. doi:10.1145/3486220.
- 2 Mikkel Abrahamsen, Joakim Blikstad, André Nusser, and Hanwen Zhang. Minimum star partitions of simple polygons in polynomial time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC 2024, page 904–910, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3618260.3649756.
- 3 Mikkel Abrahamsen and Bartosz Walczak. Common tangents of two disjoint polygons in linear time and constant workspace. *ACM Trans. Algorithms*, 15(1), December 2018. doi:10.1145/3284355.

- 4 Reymond Akpanya, Bastien Rivier, and Frederick B. Stock. Open problems CCCG 2024. In *Proceedings of the 36th Canadian Conference on Computational Geometry*, pages 167–170, 2024.
- 5 Avis and Toussaint. An optimal algorithm for determining the visibility of a polygon from an edge. *IEEE Transactions on Computers*, C-30(12):910–914, 1981. doi:10.1109/TC.1981.1675729.
- 6 Bentley and Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979. doi:10.1109/TC.1979.1675432.
- 7 Ahmad Biniaz, Anil Maheshwari, Magnus Christian Ring Merrild, Joseph S. B. Mitchell, Saeed Odak, Valentin Polishchuk, Eliot W. Robson, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, Thomas Shermer, Jack Spalding-Jamieson, Rolf Svenning, and Da Wei Zheng. Polynomial-Time Algorithms for Contiguous Art Gallery and Related Problems. In Oswin Aichholzer and Haitao Wang, editors, *41st International Symposium on Computational Geometry (SoCG 2025)*, volume 332 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:21, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SocG.2025.20>, doi:10.4230/LIPIcs.SocG.2025.20.
- 8 Ahmad Biniaz, Anil Maheshwari, Joseph S. B. Mitchell, Saeed Odak, Valentin Polishchuk, and Thomas Shermer. Contiguous boundary guarding, 2024. URL: <https://arxiv.org/abs/2412.15053>, arXiv:2412.15053.
- 9 Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and Real Computation*. Springer New York, 1998. doi:10.1007/978-1-4612-0701-6.
- 10 H El Gindy and D Avis. A linear algorithm for computing the visibility polygon from a point. *Journal of Algorithms*, 2(2):186–197, 1981. doi:10.1016/0196-6774(81)90019-5.
- 11 Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of cgal a computational geometry algorithms library. *Software: Practice and Experience*, 30(11):1167–1202, 2000.
- 12 Efi Fogel, Ophir Setter, Ron Wein, Guy Zucker, Baruch Zukerman, and Dan Halperin. 2D regularized boolean set-operations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 edition, 2024. URL: <https://doc.cgal.org/6.0.1/Manual/packages.html#PkgBooleanSetOperations2>.
- 13 Ronald Goldman. Intersection of two lines in three-space. In ANDREW S. GLASSNER, editor, *Graphics Gems*, page 304. Morgan Kaufmann, San Diego, 1990. doi:10.1016/B978-0-08-050753-8.50064-4.
- 14 Günther Greiner and Kai Hormann. Efficient clipping of arbitrary polygons. *ACM Trans. Graph.*, 17(2):71–83, April 1998. doi:10.1145/274363.274364.
- 15 Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988. doi:10.1007/978-3-642-97881-4.
- 16 Michael Hemmer, Kan Huang, Francisc Bungiu, and Ning Xu. 2D visibility computation. In *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 edition, 2024. URL: <https://doc.cgal.org/6.0.1/Manual/packages.html#PkgVisibility2>.
- 17 J. Mark Keil. Decomposing a polygon into simpler components. *SIAM Journal on Computing*, 14(4):799–817, 1985. doi:10.1137/0214056.
- 18 J. Mark Keil and Jorg-R. Sack. Minimum decompositions of polygonal objects. In Godfried T. TOUSSAINT, editor, *Computational Geometry*, volume 2 of *Machine Intelligence and Pattern Recognition*, pages 197–216. North-Holland, 1985. doi:10.1016/B978-0-444-87806-9.50012-8.
- 19 Aldo Laurentini. Guarding the walls of an art gallery. *The Visual Computer*, 15(6):265–278, 1999. doi:10.1007/S003710050177.
- 20 C.C. Lee and D.T. Lee. On a circle-cover minimization problem. *Information Processing Letters*, 18(2):109–115, 1984. doi:10.1016/0020-0190(84)90033-4.

- 21 D. Lee and A. Lin. Computational complexity of art gallery problems. *IEEE Transactions on Information Theory*, 32(2):276–282, 1986. doi:10.1109/TIT.1986.1057165.
- 22 D. T. Lee and Arthur K. Lin. *Computational Complexity of Art Gallery Problems*, pages 303–309. Springer New York, New York, NY, 1990. doi:10.1007/978-1-4613-8997-2_23.
- 23 J. Mark Keil. Chapter 11 - polygon decomposition. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 491–518. North-Holland, Amsterdam, 2000. doi:10.1016/B978-044482537-7/50012-7.
- 24 Francisco Martínez, Antonio Jesús Rueda, and Francisco Ramón Feito. A new algorithm for computing boolean operations on polygons. *Computers & Geosciences*, 35(6):1177–1185, 2009. doi:10.1016/j.cageo.2008.08.009.
- 25 Magnus C. R. Merrild, Casper M. Rysgaard, Jens K. R. Schou, and Rolf Svenning. An Algorithm for the Contiguous Art Gallery Problem in C++. <https://github.com/RolfSvenning/ContiguousArtGallery>, 2024.
- 26 Joseph O’Rourke. *Art gallery theorems and algorithms*. Oxford University Press, Inc., USA, 1987.
- 27 Mark H Overmars. Dynamization of order decomposable set problems. *Journal of Algorithms*, 2(3):245–260, 1981. doi:10.1016/0196-6774(81)90025-0.
- 28 Eliot W. Robson, Jack Spalding-Jamieson, and Da Wei Zheng. The analytic arc cover problem and its applications to contiguous art gallery, polygon separation, and shape carving, 2024. URL: <https://arxiv.org/abs/2412.15567>, arXiv:2412.15567.
- 29 Marcus Schaefer and Daniel ŹTefankoviř. Fixed points, nash equilibria, and the existential theory of the reals. *Theor. Comp. Sys.*, 60(2):172–193, February 2017. doi:10.1007/s00224-015-9662-0.
- 30 Thomas C. Shermer. Recent results in art galleries (geometry). *Proc. IEEE*, 80(9):1384–1399, 1992. doi:10.1109/5.163407.
- 31 Jack Stade. The Point-Boundary Art Gallery Problem Is $\exists\mathbb{R}$ -Hard. In Oswin Aichholzer and Haitao Wang, editors, *41st International Symposium on Computational Geometry (SoCG 2025)*, volume 332 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 74:1–74:23, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SoCG.2025.74>, doi:10.4230/LIPIcs.SoCG.2025.74.
- 32 The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 edition, 2024. URL: <https://doc.cgal.org/6.0.1/Manual/packages.html>.
- 33 Jorge Urrutia. Art gallery and illumination problems. *Handbook of Computational Geometry*, 12 2000. doi:10.1016/B978-044482537-7/50023-1.
- 34 Bala R. Vatti. A generic solution to polygon clipping. *Commun. ACM*, 35(7):56–63, July 1992. doi:10.1145/129902.129906.
- 35 Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. *SIGGRAPH Comput. Graph.*, 11(2):214–222, July 1977. doi:10.1145/965141.563896.
- 36 Ron Wein, Eric Berberich, Efi Fogel, Dan Halperin, Michael Hemmer, Oren Salzman, and Baruch Zukerman. 2D arrangements. In *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 edition, 2024. URL: <https://doc.cgal.org/6.0.1/Manual/packages.html#PkgArrangementOnSurface2>.

A Vertex restricted contiguous art gallery

In the following, we show how to efficiently compute an optimal solution to the contiguous art gallery problem, when the problem is vertex restricted, as mentioned in Section 1.2 and Table 1. There are two variants of this restriction, one is when the *intervals are restricted to vertices* and the other is when *guards are restricted to vertices*, which we cover in Appendix A.1 and A.2, respectively. We let for the following the input be a simple polygon P containing n vertices.

A.1 Intervals are restricted to vertices

If the contiguous interval seen by a guard is restricted to start and end at a vertex, the problem can be solved as follows. In a polygon of n vertices, there are $\Theta(n^2)$ contiguous intervals of the boundary, that is, from any vertex to any other. However, not all of these may be valid as there may not exist a guard that can see the whole interval. When all valid intervals have been found, the algorithm of Lee and Lee [20] can be used to find an optimal solution. Note that if a valid interval is contained entirely in another valid interval, then it can be pruned by Lemma 17.1, without removing optimality. For the valid intervals generated, we report only the intervals starting at a vertex and being maximal clockwise.

This is exactly equal to computing GREEDYINTERVAL from a starting vertex and restricting the final point to the last vertex on the computed interval. An iteration of GREEDYINTERVAL, when the interval contains e edges, runs in $\mathcal{O}(en \log n) = \mathcal{O}(n^2 \log n)$ time (Theorem 3), leading to total $\mathcal{O}(n^3 \log n)$ time used to compute the intervals. Computing the optimal solution of these intervals takes $\mathcal{O}(n)$ time, as the intervals are generated in sorted order, which leads to $\mathcal{O}(n^3 \log n)$ time in total.

Note that if the interval $[v_i, v_j]$ is computed from vertex v_i , then the interval starting at v_{i+1} must be able to reach at least v_j . Recomputing GREEDYINTERVAL for each vertex does not use this fact, and leads to the following optimization. If the visibility polygon of a vertex can be efficiently removed from the feasible polygon GREEDYINTERVAL used to determine the guard location and therefore also to decide whether an interval is maximal, then recomputation time can be optimized. The intersection algorithm is capable of intersecting both visibility polygons and feasible regions. Computing the feasible polygon of a set of visibility regions is, therefore, solvable by a simple divide-and-conquer algorithm.

The result of Overmars [27] show how to convert such static divide-and-conquer algorithms into a dynamic version, allowing for both insertion and deletion of visibility polygons from the set. The intersection algorithm takes two objects and computes their intersection. Let m be the total number of underlying visibility polygons on which the intersection is computed, with m_1 and m_2 visibility polygons contained in the two objects to intersect. The intersection is computed in $\mathcal{O}((n + h) \log n)$ time, with h describing the number of intersections between the two objects, due to an algorithm by Martínez, Rueda, and Feito [24], see Section 3.2. Note that intersections must be either some vertex of an input object intersecting the edge of the other input object, or a proper intersection between edges of the input objects. Every such proper intersection must lead to a vertex in the output object. The complexity of the output object is $\mathcal{O}(n)$ (Lemma 4), and as there similarly is $\mathcal{O}(n)$ and $\mathcal{O}(n)$ vertices in the input objects, then the number of total intersections $h \leq \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$. The running time of the intersection is therefore $\mathcal{O}(n \log n)$. This, by Overmars [27, Theorem 3.4], yields a data structure over m visibility polygons, allowing for both insertions and deletions of visibility polygons in $\mathcal{O}(n \log n \log m)$ time, while maintaining the feasible polygon over all current visibility polygons.

The optimized algorithm is then as follows. Start at any vertex v_i , and insert the visibility polygon of this vertex in the data structure that maintains the feasible region polygon. Then repeatedly insert the visibility polygon of the next vertex until the feasible polygon becomes empty. Let this lastly inserted vertex be v_{j+1} . Then there must exist a guard that can see the interval $[v_i, v_j]$, which is outputted. Note that on an insertion, the visibility polygon of the vertex must be computed in $\mathcal{O}(n)$ time, using the algorithm by Gindy and Avis [10], see Section 3.1, to then be inserted. To compute the interval starting at v_{i+1} , then v_i is deleted from the data structure, and until the feasible polygon again becomes empty, vertices from v_{j+2} and onward are inserted. This procedure then generates the interval starting at v_{i+1} . This process continues to compute the maximal interval starting at every vertex. During execution of this procedure, the number of vertices in the data structure is at most n , and therefore $m \leq n$. Every vertex is inserted and deleted $\mathcal{O}(1)$ times. It takes $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$ total time to compute the visibility polygons of every vertex. The overall running time is therefore $\mathcal{O}(n^2) + n \cdot \mathcal{O}(1) \cdot \mathcal{O}(n \log n \log m) = \mathcal{O}(n^2 \log^2 n)$. This outputs n maximal intervals. Computing an optimal solution from the intervals takes $\mathcal{O}(n)$ time, as the intervals are generated in sorted order, leading to an overall $\mathcal{O}(n^2 \log^2 n)$ time to compute an optimal solution.

A.2 Guards are restricted to vertices

If the guard locations are restricted to the vertices, the problem can be solved as follows. If the polygon P has n vertices, then there is only n possible placements of a guard. Note that the visibility polygon from a point v yields exactly the area that can be seen from v . By intersecting the visibility polygon with the boundary ∂P , the contiguous intervals of ∂P can be computed. The visibility polygon from a vertex v_i can be computed in $\mathcal{O}(n)$ time by the algorithm of Gindy and Avis [10], see Section 3.1. The intersection with ∂P is computeable in $\mathcal{O}(n)$ time. Each guard may see multiple contiguous intervals of ∂P , however, there is at most $\mathcal{O}(n)$ intervals for each guard. In total, all contiguous intervals visible to a guard located in any vertex of P are computeable in $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$ time yielding $\mathcal{O}(n^2)$ intervals. The intervals are not computed in sorted order, so the Lee and Lee [20] algorithm computes an optimal solution in $\mathcal{O}(n^2 \log n)$.

B Proof of supporting lemmas

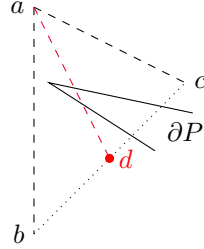
B.1 Feasible regions are connected

An important property of the feasible region F is that it is connected. To prove this, we show a generalization of Avis and Toussain [5, Lemma 1]:

► **Lemma 55** (Convex viewing lemma). *Let a, b and c be points in the interior of P (or on ∂P) such that a sees both b and c and that there is some point $d \in P$ on $\overline{bc}$, which is not visible from a . Then ∂P must intersect $\overline{bc}$.*

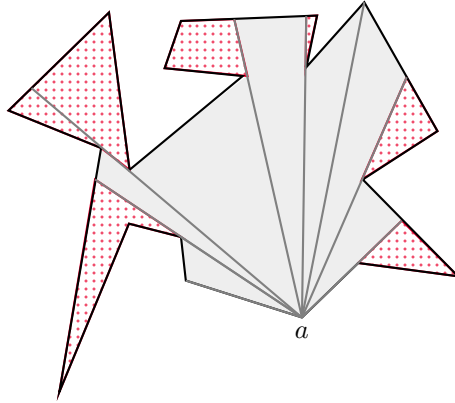
Proof. Since a can see both b and c , $\overline{ac}$ and $\overline{bc}$ are unobstructed. However, since a cannot see d , ∂P must intersect $\overline{ad}$. Since a, b and c are all inside P then ∂P must intersect triangle abc to enter, so it can block d from a , however, this can only happen by intersecting $\overline{bc}$ (see Figure 40). ◀

The feasible regions are constructed as the intersection between *visibility polygons* for single points on ∂P . We cover in detail how the GREEDYINTERVAL algorithm computes these in Section 3.



■ **Figure 40** When a can see b and c , but not d , some part of ∂P intersects $\overline{bc}$.

A visibility polygon from a point $a \in \partial P$ can be decomposed into *visible triangles* by drawing lines from a to all the vertices of ∂P visible from a (see Figure 41).



■ **Figure 41** The grey region is a visibility polygon $VP(P, a)$ from point a in polygon P . It is further subdivided into visible triangles and the red non-visible areas are pockets that connect to $VP(P, a)$ at windows.

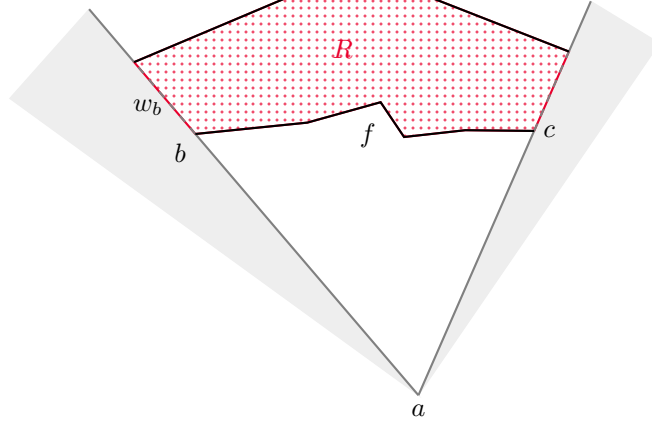
Note that these visible triangles may be degenerate (i.e. just a line). The areas marked in red, which are not visible to a , we denote *pockets*. Each pocket is connected to the visibility polygon through line segments shared with one visible triangle. We denote these shared line segments as *windows* (similar to Gindy and Avis [10]). In Figure 41 each pocket has exactly one window. The following lemma shows that this is always true:

► **Lemma 56** (Pockets have exactly one window). *Let $a \in \partial P$ and let R be a pocket of $VP(P, a)$. Then there is exactly one window for R .*

Proof. We show the lemma by contradiction. It is clear, that R must be connected to at least one visible triangle by an edge, as R otherwise would be disconnected from P and hence not a part of P .

Thus we assume there are at least 2 windows for R . Consider the connected components of $\partial P \cap \partial R$, i.e. the boundary of R without the windows. As parts of ∂P they are connected in a cycle with other parts of ∂P and as parts of ∂R they are connected in a cycle by windows. We consider a point b , which is a vertex of R , where one edge connected to b is in a component of $\partial P \cap \partial R$, which we denote f and the other is a window w_b . Furthermore, we assume B is the closer point to a , of the endpoints of w_b . Let c be the other endpoint f . See Figure 42.

We now consider where ∂P can continue from b . We consider four cases:



■ **Figure 42** When a pocket is connected by multiple windows, the polygon cannot be simple.

- ▷ **Case 1.** ∂P can continue along w_b . However, this will contradict the assumption that b is connected to the window.
- ▷ **Case 2.** ∂P can continue into the visible triangle, but then the triangle will no longer be completely visible, a contradiction.
- ▷ **Case 3.** ∂P can continue into R , however R is completely contained in P , and if an edge of ∂P lies in R , there will be some area on one side of the edge, which is not contained in P . Thus, we have a contradiction.
- ▷ **Case 4.** ∂P can continue along $\overline{ab}$ or the white region below R (see Figure 42). Once it enters here, it will become stuck inside the area bounded by the two visible triangles and f . It can exit through C , but this will create a loop in ∂P not including the top of R , which would introduce a hole in P .

It could also enter a . However, we could do the same case analysis for C , and it also need to enter into a , thus we again have a loop, and we are done. ◀

With this, we are now ready to prove the main lemma of this subsection:

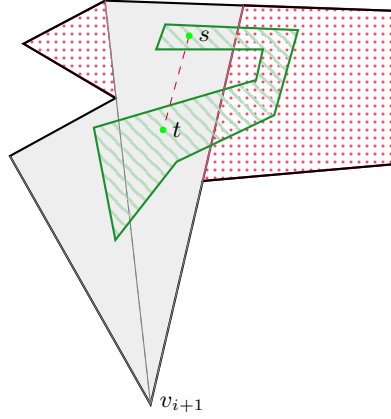
► **Lemma 39.** *For any $a, b \in \partial P$, the feasible region $F([a, b])$ is connected. Furthermore, for any convex subset $C \subset P$, $C \cap F([a, b])$ is convex.*

Proof. The proof of connectivity follows by induction in the number of visible polygons we intersect in the GREEDYINTERVAL algorithm (see Section 3).

For the base case, i.e. $F([a, a])$ where $a \in \partial P$, the feasible region is the visibility polygon $VP(P, a)$. Visibility polygons are star-shaped, hence connected.

For the induction step, we assume the feasible region seeing $[a, v_i]$ is connected and let v_{i+1} be the next vertex along ∂P or the point b if no such vertex exists. We assume for contradiction $F([a, v_{i+1}])$ is not connected. For this to happen, the feasible region $F([a, v_i])$ has to enter a pocket of the visibility polygon somewhere and exit elsewhere, so that the visibility polygon contains two disconnected components of $F([a, v_i])$ (see Figure 43). Since each pocket only has one window by Lemma 56, the two components will intersect the same visible triangle. Let points s and t from different components of $F([a, v_i])$ in the same visible triangle.

Since the previous feasible region is the intersection of visibility polygons for points $[a, v_i]$, there must be some point for which s and t are visible while some point on $\overline{st}$ is not. By



■ **Figure 43** In the induction step, we intersect $VP(P, v_{i+1})$ (in gray) with the feasible region $F([a, v_i])$ (in green) and assume for contradiction that this disconnects the feasible region. Taking points s and t in different components, we now use convex viewing lemma (Lemma 55) to get a contradiction

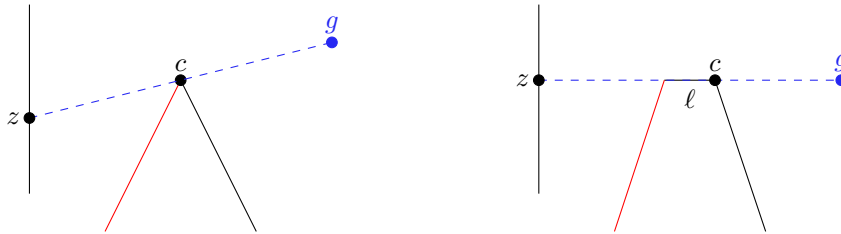
Lemma 55, there must be some part of ∂P that intersects $\overline{st}$, however, $\overline{st}$ is contained in a visible triangle, where no part of ∂P can lie. Hence, the new intersected feasible region will continue to be connected, finishing the induction.

The same considerations about s and t show the convexity claim. ◀

B.2 Visible intervals cannot block their endpoints

Since we only look at blockage (Remark 21), it will be important to look at the point that blocks parts of edges from guards:

► **Lemma 57** (What g sees around blocking vertices). *Let g be a guard seeing $[y, z]$ and assume that c is a vertex of P that blocks g from seeing more than z . Then either g cannot see both edges connected to c , or one of these edges ℓ is a segment of $L(z, g)$ and g cannot see both of the edges connected to ℓ .*

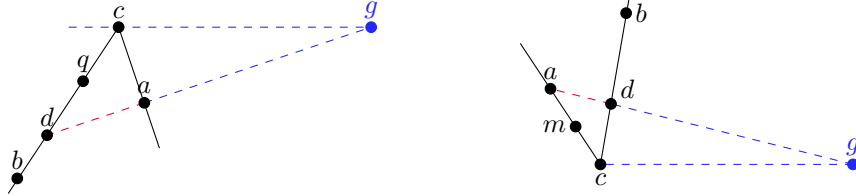


■ **Figure 44** Left, c blocks g and g sees only one edge connected to c . Right, g sees both edges, but one is a segment of $L(g, c)$ and the edge after is not visible.

Proof. Assume g can see both edges connected to c . Since c blocks g from seeing any more than z , we know z, g and c are collinear. Let a and b be points on the edge after, respectively before c , which are visible from c . We now show, that either a and b lie on different sides of $L(g, c)$, or one of the points lies on $L(g, c)$.

So assume for contradiction a and b lie on the same side of $L(g, c)$. Assume w.l.o.g. $\angle cga \leq \angle cgb$.

Thus the ray $\overrightarrow{ga}$ intersects $\overline{bc}$, at some point which we denote d . Let m be the midpoint of $\overline{ac}$ and q the midpoint of $\overline{cd}$. We split into two cases: Either a lies on $\overline{gd}$ or d lies on $\overline{ga}$ (see Figure 45).



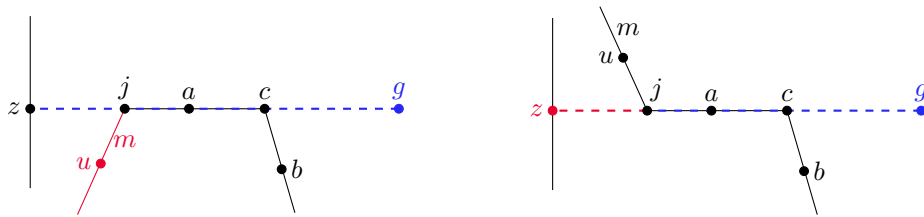
■ **Figure 45** a and b on the same side of $L(g, c)$.

▷ **Case 1.** If a lies on $\overline{gd}$ then g cannot see q (Figure 45 (left)).

▷ **Case 2.** If d lies on $\overline{ga}$ then g cannot see m (Figure 45 (right)).

Since we assumed that g sees a , b , and c , g must also be able to see $\overline{ac}$ and $\overline{bc}$, which is impossible as either q or m cannot be seen, thus a and b must either lie on different sides of $L(g, c)$ or one must lie on $L(g, c)$ (we cannot have both lie on $L(g, c)$ since c is a vertex).

Assuming a and b are on either side of $L(g, c)$, now the interval $[a, b]$ will block the view from g to z completely, thus we can discard this case. We assume w.l.o.g. a lies on $L(g, c)$. Let the endpoint of the edge containing a different from c be j and let m be the other edge connected to j . If g can see no more of m than j , we are done, so assume g can see some point u on m . u cannot lie on $L(g, c) = L(g, j)$ since j is a vertex. If u and b lie on the same side of $L(g, c)$ we have the same problem as for a and b above (Figure 46 left). If k and b lie on different sides of $L(g, c)$ then $[b, u]$ will block the view from g to z (Figure 46). ◀



■ **Figure 46** Left, u lies on same side as b and is not visible from g . Right, u lies on different side, but now z is not visible from g .

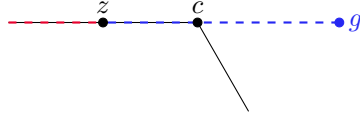
This can now be used to show where the blocking points are located in the figure:

► **Lemma 40** (Blockings happen outside visible area). *Let $y \in \partial P$ and $z = G(y)$ and assume that neither y nor z is a vertex of P . Let g be a guard seeing $[y, z]$ and let c be a vertex blocking the view from g to z . Then $c \notin [y, z]$.*

Proof. Assume for contradiction that $C \in [y, z]$. Since c is a vertex of P , c is neither y nor z , thus y and z must lie on different sides of c along $[y, z]$. From Lemma 57 we have two cases for what g can see close to c along ∂P .

▷ **Case 1.** If g cannot see both edges connected to c , either $[y, c]$ or $[c, z]$ is not visible to g contradicting the fact that g sees $[y, z]$.

▷ **Case 2.** If g is able to see both edges connected to c , then we know one is contained in $L(g, z)$ and the next edge is not visible from g , hence z must lie on the edge connected to c , which is contained in $L(g, z)$, for $[c, z]$ to be visible. However now z can be moved until the endpoint of the edge containing z , hence z is a vertex, which is contradicting the assumption (see Figure 47). ◀



■ **Figure 47** If c is to act as blockage between g and z , then g should see no more than z . However, when g, c and z are collinear g can see the entire edge containing z .

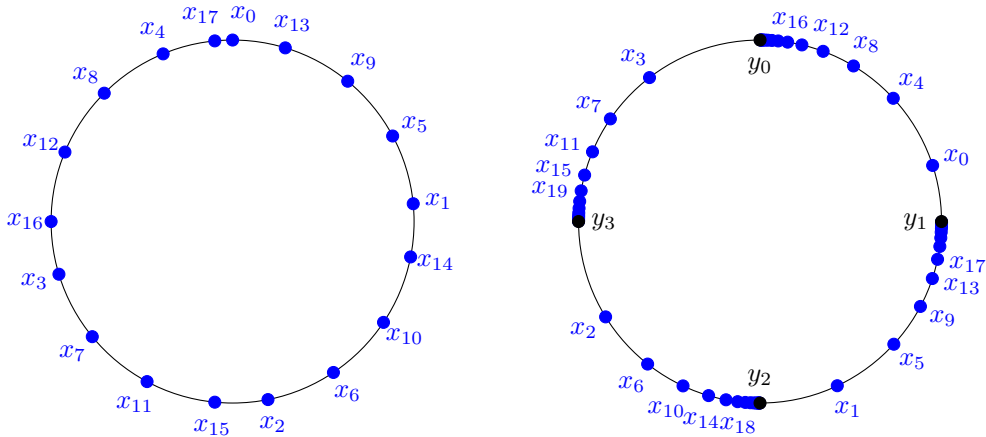
C Insightful examples

C.1 Example of two combinatorially indistinguishable functions

► **Example 58.** An examples of functions, which combinatorially look identically, but behave very different is given below:

For this, we represent points on the circle by numbers in $[0, 1)$. We then consider the two functions G_1 and G_2 . The first (see Figure 48 (left)) is defined as $G_1(x) = \{x + \frac{1}{k} - \frac{\varepsilon}{k}\}$ where ε is some fixed small number and $\{a\}$ is the decimal part of a (i.e. $a = \lfloor a \rfloor + \{a\}$). This will require $k + 1$ guards, but ε^{-1} steps of GREEDYINTERVAL will be required to satisfy one of the optimality conditions (in this case we satisfy Corollary 31).

Secondly, we define $G_2(x) = \{x + \frac{1}{k} - \alpha \frac{\{kx\}}{k}\}$, where α is some number in $(0, 1)$ (see Figure 48 (right)). It has an optimal solution at $y_i = i/k$ with k intervals. However starting anywhere other than in an optimal solution and running Algorithm 1 will never yield an optimal solution, since $\frac{\{xk\}}{k}$ is the distance x needs to move backwards to hit the optimal solution, however we only move α times that distance each step.



■ **Figure 48** Left, G_1 with $k = 4$ is used to generate a greedy sequence, where x_{17} shows, that we are optimal. Right, G_2 with $k = 4$ and $\alpha = 0.1$ is used. Starting at $x_0 = 0.2$, we never see an optimal solution.

Combinatorially, these two functions are identical (until ε^{-1} rounds have passed), so we have no guarantee, that the algorithm terminates with a solution in polynomial time.

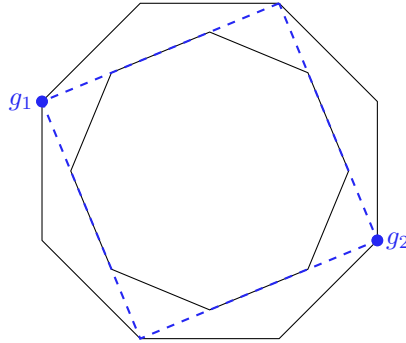
C.2 Polygons with holes

One natural generalization of the contiguous art gallery problem is the contiguous art gallery problem with holes. In this section, we study the variant in which the goal is to guard the external boundary of the polygon.

Many parts of the geometric analysis break down when we introduce holes. The most glaring is the fact that the entire strategy of Proposition 48 does not work, as we are using the fact that the pivot points (and edges close to them) have to be guarded by a guard at some point and now these pivot points could be on a hole.

Furthermore, the algorithm given for GREEDYINTERVAL in Section 3 does not work as a point in P can now see two vertices of P without seeing the entire edge between them. This issue can be fixed, but even if it is, we show that Algorithm 1 can run in superpolynomial time in the number of vertices when P has holes.

► **Example 59 (Octagon with hole).** Consider a regular octagon with a rotated regular octagon inside like shown on Figure 49, where the inner octagons edges line up exactly with the dashed line segments. Placing guards g_1 and g_2 will guard the entire boundary contiguously.



■ **Figure 49** P is an octagon with an octagonal hole in the center. Guards at g_1 and g_2 will guard the entire boundary contiguously.

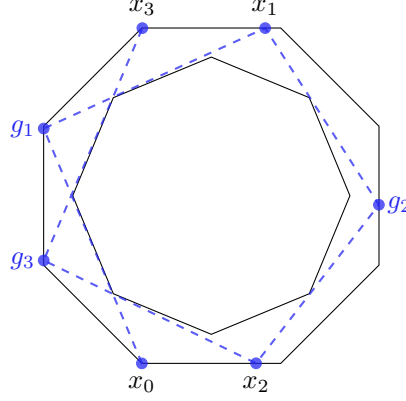
Now we enlarge the inner polygon by a tiny ε , which will make this solution invalid. When we run Algorithm 1 in this slightly different polygon, it is evident, that the best guard is placed on the outer boundary of P as far as the starting point can see (see Figure 50) and the greedy interval is found by taking the furthest point which this guard can see.

As long as the found endpoint/guard is closer to the next vertex than the previous vertex (along the outer boundary of P), it is the same vertex of the inner polygon which will block the view to the next guard/endpoint (the one marked in Figure 51).

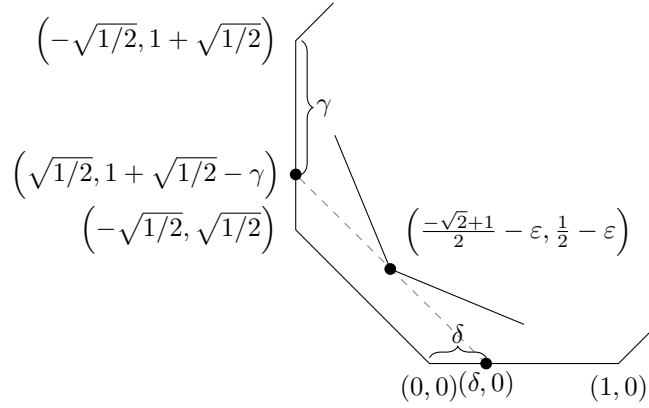
We consider how far a point on the lower edge of P can see on the left vertical edge of P . To do this, we embed P into a coordinate system with the lower edge having endpoints in $(0, 0)$ and $(1, 0)$. The relevant coordinates are drawn on Figure 51.

Calculating, we get that the equation of the dashed line is:

$$y = \left(\frac{\sqrt{2} + 2\delta}{\sqrt{2} - 1 + 2\varepsilon + 2\delta} - 1 \right) (\delta - x)$$



■ **Figure 50** We have here enlarged the inner polygon with $\varepsilon = 0.001$, we now start Algorithm 1 at x_0 . The optimal guard for x_0 is found by finding the furthest point along the outer boundary which x_0 can see. This point is denoted g_1 which can see until x_1 and so on.



■ **Figure 51** P embedded into a coordinate system with coordinates marked. A guard placed in $(\delta, 0)$ can see no longer than $(-\sqrt{1/2}, 1 + \sqrt{1/2} - \gamma)$. Here γ is the distance between the new point and the next vertex of P .

Inserting $x = -\sqrt{1/2}$ in this equation will yield the y-coordinate, y' , of the intersection with the dashed line and the left vertical edge:

$$\begin{aligned} y' &= \left(\frac{\sqrt{2} + 2\delta}{\sqrt{2} - 1 + 2\varepsilon + 2\delta} - 1 \right) \left(\delta + \sqrt{\frac{1}{2}} \right) \\ &= \frac{1 + 2\sqrt{2}\delta + 2\delta^2}{\sqrt{2} - 1 + 2\varepsilon + 2\delta} - \sqrt{\frac{1}{2}} - \delta \end{aligned}$$

And now γ is calculated as $\gamma = 1 + \sqrt{\frac{1}{2}} - y'$:

$$\begin{aligned} \gamma &= \delta + \frac{2\delta - 2\delta^2 + (2 + 2\sqrt{2})\varepsilon}{\sqrt{2} - 1 + 2\varepsilon + 2\delta} \\ &\in (\delta, 7\delta + 12\varepsilon) \end{aligned}$$

We now choose $\varepsilon = \frac{1}{2 \cdot 12 \cdot 8^{2N}}$ where N is some large number. Consider the sequence $(\gamma_i)_{i=0}^{2N}$ of distances from the furthest visible point and the next vertex when taking such

steps. Note that in Figure 51 we have, say, $\delta = \gamma_i$ and $\gamma = \gamma_{i+1}$, i.e. the δ is the member of the of the γ_i sequence that precedes γ . The above bound then becomes $\gamma_{i+1} \in (\gamma_i, 7\gamma_i + 12\varepsilon)$ and especially $\gamma_i < \gamma_{i+1}$ hence the associated local greedy sequences will have negative fingerprints and no repetitions.

Furthermore we show by induction that $\gamma_i < \frac{1}{2 \cdot 8^{2N-i}}$ as $\gamma_0 = 0$ and if $\gamma_i < \frac{1}{2 \cdot 8^{2N-i}}$, we get:

$$\begin{aligned} \gamma_{i+1} &< 7\gamma_i + 12\varepsilon \\ &= \frac{7}{2 \cdot 8^{2N-i}} + \frac{1}{2 \cdot 8^{2N}} \\ &\leq \frac{1}{2 \cdot 8^{2N-(i+1)}} \end{aligned}$$

And we thus have $\gamma_i \in [0, 1)$ for all $\gamma = 0, 1, \dots, 2N$, thus for all the guards will be on the same two edges, i.e. we get no edge jumps. Thus in the first N greedy steps we do not reach a geometric progress condition.

Since N is independent of n the runtime of Algorithm 1 is unbounded in the real RAM model.

On Finding Longest Palindromic Subsequences Using Longest Common Subsequences

Gerth Stølting Brodal  

Aarhus University, Denmark

Rolf Fagerberg  

University of Southern Denmark, Odense, Denmark

Casper Moldrup Rysgaard  

Aarhus University, Denmark

Abstract

Two standard textbook problems illustrating dynamic programming are to find the longest common subsequence (LCS) between two strings and to find the longest palindromic subsequence (LPS) of a single string. A popular claim is that the longest palindromic subsequence in a string can be computed as the longest common subsequence between the string and the reversed string. We prove that the correctness of this claim depends on how the longest common subsequence is computed. In particular, we prove that the classical dynamic programming solution by Wagner and Fischer [JACM 1974] for finding an LCS in fact does find an LPS, while a slightly different LCS backtracking strategy makes the algorithm fail to always report a palindrome.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Palindromic subsequence, longest common subsequence, dynamic programming

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.35

Funding Gerth Stølting Brodal: Independent Research Fund Denmark, grant 9131-00113B.

Rolf Fagerberg: Independent Research Fund Denmark, grant 0135-00420B. EU Horizon Europe, MSC grant 101072930.

Casper Moldrup Rysgaard: Independent Research Fund Denmark, grant 9131-00113B.

1 Introduction

This paper addresses the frequently appearing statement that the longest palindromic subsequence of a string can be computed as the longest common subsequence between the string and the reversed string – a statement also given by AI assistants (see Appendix A). We show that the correctness of this statement depends on how the longest common subsequence is computed.

The statement appears in the context of dynamic programming, a standard algorithmic design technique taught in most undergraduate courses on algorithms. In textbooks, the longest common subsequence (LCS) problem on two input strings is the prevailing example of a problem solvable by dynamic programming. The LCS problem asks to find a longest possible string that is a subsequence of two given input strings, or equivalently, to delete the fewest possible symbols from the two strings, such that they become equal. Figure 1 (left) illustrates that the two strings `abcaabcabc` and `bcabbbac` have `bcabbc` as an LCS. The LCS is in general not unique – for instance, `bcabac` is also an LCS for these two strings. Solving the LCS problem on two strings of length m and n , respectively, takes $O(mn)$ time using dynamic programming. In the standard textbook exposition, the lengths of optimal solutions for subproblems are tabulated first, and an actual string solution is found as a postprocessing step backtracking through the tabulated results.



© Gerth Stølting Brodal, Rolf Fagerberg, and Casper Moldrup Rysgaard;
licensed under Creative Commons License CC-BY 4.0

32nd Annual European Symposium on Algorithms (ESA 2024).

Editors: Timothy Chan, Johannes Fischer, John Iacono, and Grzegorz Herman; Article No. 35; pp. 35:1–35:16



Leibniz International Proceedings in Informatics

LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The longest palindromic subsequence (LPS) problem on a single input string is sometimes presented (e.g., in an exercise) as another example of a problem solvable by dynamic programming. The LPS problem asks to find a longest subsequence of a string that is a palindrome, i.e., the subsequence should read the same forwards and backwards. Solving the LPS problem on a string of length n using textbook dynamic programming takes time $O(n^2)$.

The arguments behind the recurrences for the two problems have some analogies, but the resulting recurrences are distinct. For instance, the recurrence for LCS fills out a table of rectangular shape, while the recurrence for LPS uses a triangular tabular area. Still, there is a strong connection: the *length* of LPS solutions on a string s is the same as the length of LCS solutions on s and the reversed string s^R . This statement can be found repeated several places on the web, but usually without a proof. Indeed, a proof is complicated by the fact that the *set* of LPS solutions on a string s is *not* the same as the set of LCS solutions on s and the reversed string s^R : members of the first set are easily seen to be members of the second set, but the reverse inclusion is not always true. One example of this is $s = \text{abccab}$, where accb is an LCS of s and s^R , but accb is not a palindrome. Figure 1 (right) contains another example.

So, a natural question is, how to arrive at an LPS solution via the LCS route. As mentioned, textbook explanations of LCS often include a method for producing an actual solution. The method consists of backtracking over the LCS table, following optimal choices in the LCS recurrence. The recurrence for LCS can actually take on a few variations, but almost all textbooks (including, e.g., [11, pages 397–398]) use the same variant and hence the same backtracking algorithm for producing a solution, namely the one from the classical paper by Wagner and Fischer [28].

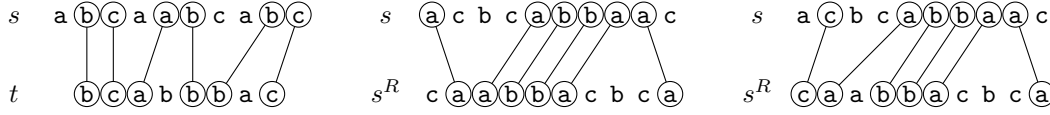
The starting point of this paper is the empirical observation that the above backtracking algorithm *always* finds a palindromic solution to LCS on s and s^R , while some other variants of it do not. In this paper, we set out to find out why.

1.1 Previous work on LCS

The LCS problem and its applications have been studied intensely in the literature. Here, we briefly review some of the main results and refer the reader to the surveys by Bergroth, Hakonen, and Raita [4] and Navarro [25] for further references on the LCS problem.

Wagner and Fischer [28] presented the classical $O(mn)$ time dynamic programming algorithm for finding an LCS between two strings s and t of length m and n , respectively. Their solution requires space $O(mn)$. The same time bound is also achieved by Sankoff [27] and the bound is mentioned by Chvatal, Karner and Knuth [10, Problem 35], who ask if one can do better. Navarro [25] gives an overview of several concurrent inventions of the algorithm. Hirschberg [17] improved the space usage to $O(m + n)$. Masek and Paterson [23] improved the running time to $O(n \cdot \max(1, m/\log n))$, when $m \leq n$ and the alphabet Σ for the strings is finite, by applying the “the four Russians” tabulation technique to the algorithm by Wagner and Fischer. Subsequently, Bille and Farach-Colton [5] and Grabowski [14] improved and generalized the result to arbitrary alphabet sizes achieving $O(n + mn(\log \log n)^2 / \log^2 n)$ time and $O(mn \log \log n / \log^2 n)$ time, respectively.

Hunt and Szymanski [19] presented an algorithm with running time $O((r + n) \log n)$ for two strings of length n , where $0 \leq r \leq n^2$ is the number of pairs of positions at which the two strings match, i.e., the running time is between $O(n \log n)$ and $O(n^2 \log n)$, depending on r . For $m \leq n$ and k denoting the length of an LCS, Hirschberg [18] presented two algorithms with running time $O(kn + n \log n)$ and $O(k(m + 1 - k) \log n + m \log n)$, respectively, and Nakatsu, Kambayashi and Yajima [24] presented an algorithm with running time $O((m + 1 - k)n)$.



■ **Figure 1** (left) the LCS **bcabbc** between two strings s and t ; (center) the LCS **aabbaa** between s and s^R is a palindrome; (right) the LCS **cabbba** between s and s^R is *not* a palindrome; (center) and (right) are obtained by the backtracking strategies $\langle \nwarrow, \uparrow, \leftarrow \rangle$ and $\langle \uparrow, \leftarrow, \nwarrow \rangle$, respectively (see Section 3 for the definition of a backtracking strategy).

Aho, Hirschberg and Ullman [2] considered decision tree lower bounds for the LCS problem. Abboud, Backurs and Williams [1], and independently Bringmann and Künnemann [7], showed that the LCS problem cannot be solved in time $O(n^{2-\varepsilon})$, for any $\varepsilon > 0$, under the *strong exponential time hypothesis* (SETH). This explains the lack of essential improvements over the $O(n^2)$ bound from Wagner and Fischer. Due to the SETH lower bound, recent work on the LCS problem has been in the direction of approximation algorithms, see, e.g., recent work by Bringmann, Cohen-Addad and Das [6].

1.2 Previous work on LPS

Algorithms for finding palindromes in strings is discussed in many textbooks on string algorithms, like Gusfield [15] and Crochemore and Rytter [12], although the focus is on finding longest palindromic substrings instead of subsequences, and variations like approximate palindromes [26] and gapped palindromes [21]. Whereas the LCS problem is covered in detail in textbooks as a canonical example of dynamic programming, the LPS problem appears more rarely, and if so often as an exercise – for instance, Cormen, Leiserson, Rivest and Stein et al. [11, Problem 14-2] ask for an efficient algorithm for the LPS problem and Dasgupta, Papadimitriou and Vazirani [13, Problem 6.7] ask explicitly for an $O(n^2)$ algorithm.

Manacher [22] described a linear time algorithm to identify all palindromic prefixes of a string. As observed by, e.g., Apostolico, Breslauer and Galil [3], Manacher’s algorithm also identifies for each position in the string the longest palindromic substring centered at this position, i.e., solves the longest palindromic substring problem in linear time. Chowdhury, Hasan, Iqbal and Rahman [8] initiated the study of finding the longest common palindromic subsequence in two strings (without reference to the simpler LPS problem for a single string).

Hasan, Islam, Rahman and Sen [16] described an automaton (denoted a palindromic subsequence automaton) that is a compact representation of all palindromic subsequences of a string of length n . Their construction uses space $O(n^2)$, improving upon the exponential space used by Chuang, Lee and Hunag [9]. As an application, Hasan et al. showed how to use their automaton to compute the longest common palindromic subsequence for two strings of length n in worst-case time $O(n^4|\Sigma|)$, but the algorithm can be faster depending on the size of the automata for the two strings. Their paper makes crucial use of the following claim [16, page 221] (rephrased to use our notation): “*Intuitively, we can always obtain a longest palindromic subsequence of s by first taking the longest common subsequence (LCS) t of s and s^R and then “reflecting” the first half of the result onto the second half; that is, if t has k symbols, then we replace the last $\lfloor k/2 \rfloor$ symbols of t by the reverse of the first $\lfloor k/2 \rfloor$ symbols of t .*” In Section 2 we provide a proof of this statement and the implication that the length of an LCS for s and s^R equals the length of an LPS for s .

The LCS and LPS problems are equally hard. This follows from the facts that we can use an LCS algorithm to compute an LPS (Lemma 3) and that an LCS between two strings s and t of length n can be computed by finding an LPS of $s^n t^R$ (see Figure 2). Inenaga and



■ **Figure 2** The reduction of $\text{LCS}(s, t)$ to $\text{LPS}(s^n t^R)$. The circled symbols are an LPS of $s^n t^R$, and the circled symbols of s are the corresponding LCS of s and t .

■ **Table 1** The six different strategies to deterministically backtrack when computing $\text{LCS}(s, s^R)$. E.g., $\langle \nwarrow, \uparrow, \leftarrow \rangle$ denotes that the prioritized backtracking order of moving from (i, j) is to $(i-1, j-1)$, $(i-1, j)$, $(j-1, j)$ (this corresponds to the Wagner and Fischer algorithm [28]). We prove in this paper that the four strategies on the left always report a palindrome, whereas the two strategies on the right might fail to report a palindrome.

LCS(s, s^R) is always a palindrome				LCS(s, s^R) can be non-palindromic	
$\langle \nwarrow, \uparrow, \leftarrow \rangle$	$\langle \nwarrow, \leftarrow, \uparrow \rangle$	$\langle \uparrow, \nwarrow, \leftarrow \rangle$	$\langle \leftarrow, \nwarrow, \uparrow \rangle$	$\langle \uparrow, \leftarrow, \nwarrow \rangle$	$\langle \leftarrow, \uparrow, \nwarrow \rangle$

Hyrrö [20, Section 3] gave a generalized version of this reduction for the case of finding the longest common subsequence among four strings to finding a longest common palindromic subsequence in two strings. Note that while a longest palindromic substring can be found in $O(n)$ time [22], finding a longest palindromic subsequence requires worst-case time $\Omega(n^{2-\varepsilon})$, for any $\varepsilon > 0$, under the strong exponential time hypothesis (SETH), by the above reduction and the conditional lower bound for the LCS problem [1, 7].

1.3 Results of this paper

In this paper we perform a detailed study of how to find an LPS of a string s by computing an LCS of s and s^R .

In Section 2, we provide proofs for the following two facts. These facts are folklore, but proofs appear to be scarce in the literature. For completeness, we fill this gap.

1. The length an LCS equals the length of an LPS (Lemma 2).
2. Reflecting an LCS in the middle (as described by Hasan et al. [16]) gives an LPS (Lemma 3).

Interestingly, one does not always need to reflect the found LCS to obtain an LPS. In particular, our starting point for this paper was the empirical observation that the Wagner and Fischer algorithm [28] seems to *always* find a palindrome. In Section 3, we prove the following new results:

3. The Wagner and Fischer algorithm [28] always reports a palindrome (Theorem 14).
4. The backtracking in a dynamic programming based solution for LCS, like the Wagner and Fischer algorithm [28], can be ordered in six different ways. Four of these always return a palindrome (Theorem 14), whereas two fail to do so (Lemma 15). See Table 1.

The proof of our main result (Theorem 14) is composed of the following steps: We first prove that the standard textbook algorithm [28] for constructing an LCS path always produces the same solution – although not necessarily via the same path in the LCS table – as the algorithm whose path is the “right-most extreme path”. We then prove that for the LCS of s and s^R , the set of edges from all paths in the table must be symmetric around a table diagonal. From this follows that the right-most extreme path itself must be symmetric, which implies that it spells out a palindrome. Hence, the textbook algorithm must always return a palindrome.

1.4 Preliminaries

We let $s = s_1 s_2 \cdots s_n$ denote a string of length $|s| = n$ with symbols s_i from an alphabet Σ . The reversed string is denoted $s^R = s_n s_{n-1} \cdots s_2 s_1$. For $1 \leq i \leq j \leq n$, $s[i..j] = s_i s_{i+1} \cdots s_j$ denotes a substring of s of length $j - i + 1$. For $i > j$, $s[i..j]$ denotes the empty string. A *subsequence* of s is a string $s_{i_1} s_{i_2} \cdots s_{i_k}$, where $0 \leq k \leq n$ and $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. For strings s and t , $\text{LCS}(s, t)$ denotes the length of a *longest common subsequence* of the two strings, i.e., the maximal length over all strings which are subsequences of both s and t . For two strings s and t of length m and n , respectively, and $0 \leq i \leq m$ and $0 \leq j \leq n$, we let $\text{LCS}(i, j) = \text{LCS}(s[1..i], t[1..j])$, when s and t are obvious from the context. A string s is a *palindrome* if $s = s^R$. We let $\text{LPS}(s)$ denote the length of a *longest palindromic subsequence* of s .

The six LCS dynamic programming algorithms that we study are all captured by the following recurrence. Let s and t be two strings of length m and n , respectively, and let $\text{lcs}_{i,j}$ denote an LCS of the first i and j symbols in s and t , respectively. Finding an LCS of s and t is then to find $\text{lcs}_{m,n}$. One way to compute $\text{lcs}_{i,j}$ is by the recurrence

$$\text{lcs}_{i,j} = \begin{cases} \varepsilon & \text{if } i = 0 \vee j = 0 \\ \text{longest}(\text{lcs}_{i-1,j}, \text{lcs}_{i,j-1}) & \text{if } i > 0 \wedge j > 0 \wedge s_i \neq t_j \\ \text{longest}(\text{lcs}_{i-1,j}, \text{lcs}_{i,j-1}, \text{lcs}_{i-1,j-1} \cdot s_i) & \text{if } i > 0 \wedge j > 0 \wedge s_i = t_j \end{cases} \quad (1)$$

where ε denotes the empty string, $\cdot$ denotes string concatenation, and $\text{longest}(\dots)$ returns a longest string among a list of strings. What LCS is identified as $\text{lcs}_{m,n}$ only depends on what string $\text{longest}(\dots)$ returns in case of multiple strings of equal length. These choices correspond to the choices one has to do when backtracking a dynamic programming table to produce an LCS string. Typical dynamic programming solutions observe that the last case in the recurrence can be simplified to $\text{lcs}_{i-1,j-1} \cdot s_i$, limiting the possible LCSs that can be produced.

2 Relationship between LCS and LPS

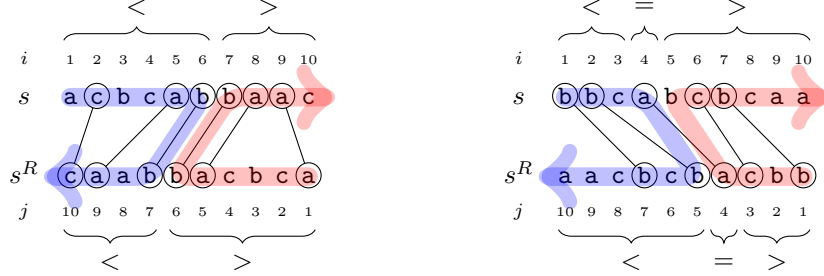
In this section, we furnish proofs of two core statements on the relationship between $\text{LCS}(s, s^R)$ and $\text{LPS}(s)$, namely Lemma 2 and Lemma 3. Potential novelty in this section lies in the exposition, not in the results stated.¹

► **Lemma 1.** *Let t be a common subsequence of s and s^R , where $|t| = k$. Then there exist two (possibly identical) palindromic subsequences of s of combined length $2k$.*

Proof. A symbol of t is a match between a symbol s_i in s and a symbol s_j in s^R . In this proof, we let both indices i and j refer to positions in s (see Figure 3), i.e., the match is between the i th *first* symbol in s and the j th *last* symbol in s^R . Let “<”, “=”, and “>” designate the sets of matchings in t for which $i < j$, $i = j$, and $i > j$, respectively.

If the set “=” is empty, consider the subsequence of symbols in s identified by the matchings of the set “<”, as pointed out by the blue arrow in Figure 3 (left). More precisely, consider the symbols in circles passed by the blue arrow, taken in the order of that arrow.

¹ We do note, though, that we have so far been unable to find solid arguments for these statements in the literature. The only close contender found is [29] (which does not consider the length of the underlying string when reversing the first half of the LCS, hence the reversing may fail – the proof can be completed by arguing that the reversing can only fail in one of the directions).



■ **Figure 3** Proof of Lemma 1: A common subsequence between s and s^R (shown as circled symbols) and its partition into two palindromic subsequences (blue and red), depending on whether an index is matched to itself (right, index 4) or not (left).

These symbols constitute a subsequence of s (as their indices in s are increasing) and constitute a palindrome of even length by the matchings of t . The same holds for the subsequence of symbols in s identified by the matchings of the set “>”, as pointed out in a similar manner by the red arrow. Combined, these two subsequences cover all symbols of s and s^R which are matched by t , hence the two subsequences have a combined length of $2k$.

If the set “=” is non-empty, it must have size one, as indices in s of the matchings of t increase in the top row and decrease in the bottom row of Figure 3. The matchings of the two sets “<” and “>”, each together with one end of the single matching from the set “=”, give two palindromic subsequences with the same properties (except for having odd lengths), as pointed out by the blue and red arrows in Figure 3 (right). ◀

From Lemma 1 we can derive that the length of an LPS of a string s equals the length of an LCS of s and s^R .

► **Lemma 2.** For all strings s , we have $\text{LCS}(s, s^R) = \text{LPS}(s)$.

Proof. Consider a string s and a palindrome p that is a subsequence of s . Then p^R is a subsequence of s^R . Since $p^R = p$ we have that p is a common subsequence of s and s^R . This shows $\text{LCS}(s, s^R) \geq \text{LPS}(s)$.

Conversely, consider a common subsequence t between s and s^R of length k . By Lemma 1, we can find at least one palindromic subsequence of s of length at least k , which shows $\text{LCS}(s, s^R) \leq \text{LPS}(s)$. Combining the two inequalities, we get $\text{LCS}(s, s^R) = \text{LPS}(s)$. ◀

Given an LCS solution, an LPS solution can be produced using Lemma 1 by reflecting the LCS string in the middle.

► **Lemma 3.** Let t be an LCS of s and s^R , where $|t| = k$. Then both $p_1 = t[1..\lceil \frac{k}{2} \rceil] \cdot t[\lceil \frac{k}{2} \rceil..k]^R$ and $p_2 = t[\lfloor \frac{k+2}{2} \rfloor..k]^R \cdot t[\lceil \frac{k+2}{2} \rceil..k]$ are LPS of s .

Proof. Since t is an LCS of s and s^R , the two palindromic subsequences from Lemma 1 must both have length k , as otherwise maximality of t would be contradicted. In that situation, these two palindromic subsequences, i.e., the two subsequences in blue and red in Figure 3 (left or right, depending on whether k is even or odd), are exactly p_1 and p_2 . ◀

As an example, Lemma 3 applied to the LCS accb of $s = \text{abccab}$ and s^R gives the two palindromic LCSs acca and bccb .

3 Properties of LCS algorithms

In this section we consider different backtracking based LCS algorithms and prove that two of the algorithms, ALGMATCH and ALGCONTOUR, report identical subsequences, while a third algorithm, ALGSKIP, not always reports the same LCS subsequences as the first two.

The standard textbook recurrence for calculating the length of an LCS between two strings s and t of length m and n , respectively, is the following (see, e.g., [11, page 396])

$$\text{LCS}(i, j) = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ \max\{\text{LCS}(i-1, j), \text{LCS}(i, j-1)\} & \text{if } i > 0 \wedge j > 0 \wedge s_i \neq t_j \\ 1 + \text{LCS}(i-1, j-1) & \text{if } i > 0 \wedge j > 0 \wedge s_i = t_j, \end{cases} \quad (2)$$

where $0 \leq i \leq m$, $0 \leq j \leq n$, and $\text{LCS}(i, j)$ is the length of an LCS of $s[1..i]$ and $t[1..j]$. By tabulating the formula over all indexes i and j , a table of the LCS for all combinations of prefixes of the input strings is obtained. See Figure 4 for an example of such a table.

LCS strings can be produced by backtracking a path in the table starting in the lower-right corner cell (m, n) . At each cell (i, j) , one has the option to go up ($\uparrow$) to cell $(i-1, j)$ if $i > 0$ and $\text{LCS}(i-1, j) = \text{LCS}(i, j)$, go left ($\leftarrow$) to cell $(i, j-1)$ if $j > 0$ and $\text{LCS}(i, j-1) = \text{LCS}(i, j)$, and diagonal up-left ($\nwarrow$) to cell $(i-1, j-1)$ if $i > 0$ and $j > 0$ and $s_i = t_j$. When going diagonal up, the symbol s_i is reported. These options correspond exactly to the possible choices in the recurrence of Equation 1 (in Section 1.4).

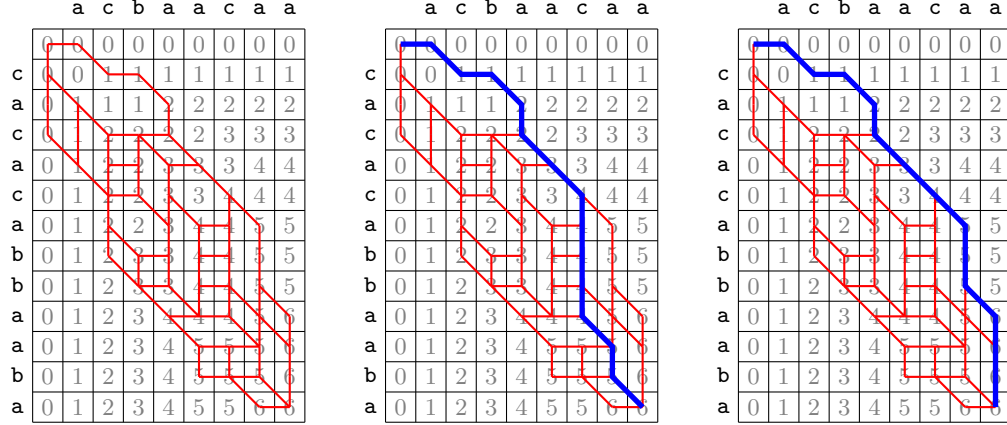
In cells where there is a choice between multiple options, different prioritization among them gives rise to different backtracking algorithms. These will follow different paths, but each will produce an LCS. Indeed, the set of oriented paths from cell (m, n) to cell $(0, 0)$ which only contain edges corresponding to the options above exactly captures all LCSs, hence we call such paths *LCS paths*.

► **Lemma 4.** *Any LCS path will produce an LCS, and for any LCS there is an LCS path producing it.*

Proof. Any LCS path starts in cell (m, n) with cell value $\text{LCS}(m, n)$ and ends in cell $(0, 0)$ with cell value zero. Cell values along the path change only when traversing $\nwarrow$ edges, and this change is a decrease by one in the cell value. Hence, such edges are traversed exactly $\text{LCS}(m, n)$ times, which will produce a common subsequence (in backwards order) of this length.

Conversely, let ℓ be any LCS of s and t . Then ℓ is a sequence of matchings of symbols of s and t . An LCS path producing ℓ can be created by having a $\nwarrow$ edge for all indices (i, j) where s_i and t_j is a matching in ℓ , and then connecting these by edges of the types $\leftarrow$'s and $\uparrow$. We now prove that the latter is possible. Let the k th matching of ℓ be between s_i and t_j , and let the $(k+1)$ th matching in ℓ be between $s_{i'}$ and $t_{j'}$, where $i < i'$ and $j < j'$. We have $\text{LCS}(i, j) = k$, since the first k matchings of ℓ constitute a common subsequence of $s[1..i]$ and $t[1..j]$ and must be optimal for those strings (or ℓ would not be optimal). Similarly, $\text{LCS}(i', j') = k+1$. By the third line of Equation 2 we have $\text{LCS}(i'-1, j'-1) = k$. As it follows from Equation 2 that cell values are non-increasing when moving up or left, we can conclude that the rectangular subset of the table having cell (i, j) as the upper left corner and cell $(i'-1, j'-1)$ as the lower right corner all will have cell values equal to k . Thus, this rectangle can be traversed by any combination of edges of the types $\leftarrow$ and $\uparrow$, which was what we wanted to prove. ◀

We denote the union of all LCS paths the *profile* of the LCS table. See Figure 4a for an illustration of a profile (directions of edges omitted for clarity).



(a) Profile of the LCS table in red.

(b) LCS path produced by ALGMATCH in blue.

(c) Contour of the profile in blue. This is also the LCS path produced by ALGCONTOUR.

Figure 4 LCS table of the strings $s = \text{cacacabbaaba}$ and $t = \text{acbaacaa}$, with the LCS values in gray and the profile overlayed.

► **Definition 5** (profile). *The profile is the union of all edges of all LCS paths.*

The following lemma shows that the profile contains no further paths than the LCS paths present by construction.

► **Lemma 6.** L is an LCS path $\iff L$ is a path from cell (m, n) to cell $(0, 0)$ in the profile.

Proof. The $\Rightarrow$ direction follows from the definition of the profile from Definition 5. For the $\Leftarrow$ direction, each edge of the profile exists in some LCS path, and must therefore fulfill the conditions on cell contents and string symbols associated with the edge. Hence, a path from (m, n) to $(0, 0)$, using only edges in the profile, is an LCS path. ◀

To produce an LCS solution, a natural method is to fix a deterministic choice in case of ties in the recurrence (implying nodes in the profile with outdegree more than one). For instance, based on the recursive LCS definition in Equation 2, a very common choice (including in [11]) is the one of ALGMATCH² in Figure 5 (left). The particular algorithm chooses to match symbols whenever possible, i.e., following the edge $\nearrow$ whenever it can. If it cannot follow this edge, then it chooses to follow $\uparrow$ and then $\leftarrow$. This can be summarized as the *strategy* of ALGMATCH being $\langle \nearrow, \uparrow, \leftarrow \rangle$. See Figure 4b for an example of the path chosen by ALGMATCH.

Any strategy for the order on the edges of the profile to follow produces an LCS by Lemma 6. The strategy $\langle \uparrow, \nearrow, \leftarrow \rangle$ follows the upper right most part of the profile, as choosing to go up as long as possible and waiting with going left as long as possible leaves no other LCS path above or to the right of it. This strategy is defined as the ALGCONTOUR algorithm, as it follows the *contour* of the profile. The contour is defined in Definition 7. See Figure 5 (right) for an implementation of ALGCONTOUR and Figure 4c for an illustration of the contour.

► **Definition 7** (Contour). *The contour of a profile is the upper right most path of the profile.*

² Note this is the same algorithm as that of Wagner and Fischer [28]. The renaming in this paper is to clearly show that this is the backtracking strategy which prioritizes to *match* symbols first.

ALGMATCH(s, t) <pre> 1 Compute table L by LCS 2 $i \leftarrow m, j \leftarrow n$ 3 $lcs \leftarrow \varepsilon$ 4 while $i > 0$ and $j > 0$ 5 if $s_i = t_i$ 6 // Follow $\nearrow$ 7 $lcs \leftarrow s_i \cdot lcs$ 8 $i \leftarrow i - 1, j \leftarrow j - 1$ 9 elseif $L(i, j) = L(i - 1, j)$ 10 // Follow $\uparrow$ 11 $i \leftarrow i - 1$ 12 else 13 // Follow $\leftarrow$ 14 $j \leftarrow j - 1$ 15 return lcs</pre>	ALGCONTOUR(s, t) <pre> 1 Compute table L by LCS 2 $i \leftarrow m, j \leftarrow n$ 3 $lcs \leftarrow \varepsilon$ 4 while $i > 0$ and $j > 0$ 5 if $L(i, j) = L(i - 1, j)$ 6 // Follow $\uparrow$ 7 $i \leftarrow i - 1$ 8 elseif $s_i = t_i$ 9 // Follow $\nearrow$ 10 $lcs \leftarrow s_i \cdot lcs$ 11 $i \leftarrow i - 1, j \leftarrow j - 1$ 12 else 13 // Follow $\leftarrow$ 14 $j \leftarrow j - 1$ 15 return lcs</pre>

■ **Figure 5** Pseudocode implementations of (left) ALGMATCH, which is the backtracking of the LCS table using strategy $\langle \nearrow, \uparrow, \leftarrow \rangle$ and (right) ALGCONTOUR using strategy $\langle \uparrow, \nearrow, \leftarrow \rangle$.

► **Corollary 8.** *The contour of a profile is an LCS path.*

Proof. The contour is a path in the profile, which by Lemma 6 must be an LCS path. ◀

We are now ready to prove that the two algorithms ALGMATCH and ALGCONTOUR compute the same output LCS strings.

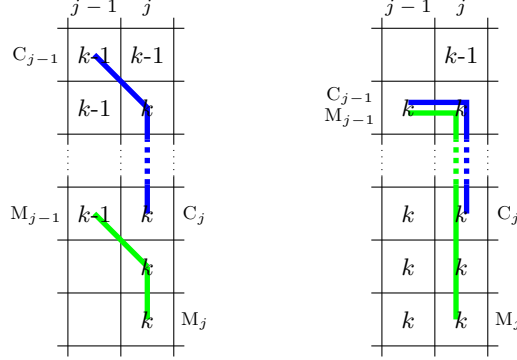
► **Lemma 9.** *For all pairs of input strings s and t , algorithms ALGMATCH and ALGCONTOUR produce the same LCS string.*

Proof. Recall that the strategies of ALGMATCH and ALGCONTOUR are $\langle \nearrow, \uparrow, \leftarrow \rangle$ and $\langle \uparrow, \nearrow, \leftarrow \rangle$, respectively, cf. the pseudocode in Figure 5. Any LCS path must cross exactly n columns. This happens via the $\nearrow$ and $\leftarrow$ steps of the path. A path outputs symbols of the LCS exactly when moving by a $\nearrow$ step, hence paths can only output symbols when changing column. Define M_j to be the row where the LCS path of ALGMATCH first enters column j and C_j to be the row where the LCS path of ALGCONTOUR first enters column j . Clearly, $M_j \geq C_j$ for all columns j , as ALGCONTOUR is the upper right most LCS path of all LCS paths. Also note that in any LCS table, it follows from Equation 2 that cell values are non-increasing when moving up or left.

The key element of the proof is the following *claim*: If the table values $LCS(M_j, j)$ and $LCS(C_j, j)$ are equal, then

1. The LCS path of ALGMATCH must leave column j by the same type of step ($\nearrow$ or $\leftarrow$) as the LCS path of ALGCONTOUR leaves column j by.
2. The table values $LCS(M_{j-1}, j - 1)$ and $LCS(C_{j-1}, j - 1)$ are also equal.

Since both LCS paths start in the same table cell (m, n) , the condition of the claim trivially holds in the beginning. Hence, by induction on (decreasing values of) j , the two paths leave exactly the same columns by $\nearrow$ steps, hence they produce the same LCS string, assuming the claim is true.



■ **Figure 6** Behavior of ALGCONTOUR (in blue, starting in row C_j) and ALGMATCH (in green, starting in row M_j), when (left) ALGCONTOUR leave column j by $\nwarrow$, and (right) ALGCONTOUR leave by $\leftarrow$.

To prove the claim, we make a case analysis on the step type by which ALGCONTOUR leaves column j . Let k denote the shared table value $k = \text{LCS}(M_j, j) = \text{LCS}(C_j, j)$.

Case 1. ALGCONTOUR leaves column j by a $\nwarrow$ step. The situation is depicted in Figure 6 (left). Before this step, ALGCONTOUR takes ℓ steps of type $\uparrow$, for $\ell \geq 0$. All ℓ table cells passed must also have the value k , by the condition for making $\uparrow$ steps. Hence, the cell $(C_{j-1}, j-1)$ arrived at in column $j-1$ by the $\nwarrow$ step must have value $k-1$, by the condition for making $\nwarrow$ steps. By the precondition $k = \text{LCS}(M_j, j) = \text{LCS}(C_j, j)$ of the claim and the vertical monotonicity of cell values, the cells in column j between cell (M_j, j) and the cell $(C_j - \ell, j)$ where the LCS path of ALGCONTOUR made the $\nwarrow$ step must all have the value k . By the strategy of ALGMATCH, this algorithm will never take a $\leftarrow$ step if $\uparrow$ steps are possible, which is the case in column j by the vertical repetition of the cell value k . Hence, the LCS path of ALGMATCH can only leave column j by a $\nwarrow$ step. This must happen either before it reaches the cell $(C_j - \ell, j)$ where the LCS path of ALGCONTOUR made the $\nwarrow$ step, or at the latest when it reaches that cell, since the LCS path of ALGCONTOUR is the upper right most path. By the conditions for making a $\nwarrow$ step, the cell arrived at in column $j-1$ by the LCS path of ALGMATCH must have value $k-1$. This establishes the claim in this case.

Case 2. ALGCONTOUR leaves column j by a $\leftarrow$ step. The situation is depicted in Figure 6 (right). Again, ALGCONTOUR may first take ℓ steps of type $\uparrow$, for $\ell \geq 0$, and all cells passed during these must also have the value k . By the condition for making $\leftarrow$ steps, the cell $(C_j - \ell, j)$ arrived at in column $j-1$ by the $\leftarrow$ step must have value k . By monotonicity, the cells in column $j-1$ from cell $(C_{j-1}, j-1)$ and down to cell $(M_j, j-1)$ must all have value k . As the LCS path of ALGMATCH must leave column j before it reaches the cell $(C_j - \ell, j)$ where the LCS path of ALGCONTOUR made the $\leftarrow$ step, or at the latest when it reaches that cell, it cannot leave by a $\nwarrow$ step, as this would need to end up in a cell in column $j-1$ of value $k-1$, by the conditions for such steps. Hence, it must leave column j by a $\leftarrow$ step, and must end up in a cell of value k . As the strategy of ALGMATCH priorities $\uparrow$ over $\leftarrow$, it actually must leave column j from the same cell as ALGCONTOUR. This establishes the claim in the second case. ◀

Another strategy is $\langle \uparrow, \leftarrow, \nwarrow \rangle$, i.e., to choose to skip symbols over matching them. We denote the algorithm using this strategy as ALGSKIP, see Figure 7 for the pseudocode for this backtracking algorithm. Using this strategy may seen at a first glance like a non-intuitive

way of implementing a traversal of the LCS table. However, in favor of this algorithm is the observation that it only performs comparisons of values in the LCS table, while comparisons of string symbols are not needed. Thus, if only the LCS indexes in s and t are to be reported, access to the strings is never needed.

```

ALGSKIP( $s, t$ )
1  Compute table  $L$  by LCS
2   $i \leftarrow m, j \leftarrow n$ 
3   $lcs \leftarrow \varepsilon$ 
4  while  $i > 0$  and  $j > 0$ 
5      if  $L(i, j) = L(i - 1, j)$ 
6          // Follow  $\uparrow$ 
7           $i \leftarrow i - 1$ 
8      elseif  $L(i, j) = L(i, j - 1)$ 
9          // Follow  $\leftarrow$ 
10          $j \leftarrow j - 1$ 
11      else
12          // Follow  $\nwarrow$ 
13           $lcs \leftarrow s_i \cdot lcs$ 
14           $i \leftarrow i - 1, j \leftarrow j - 1$ 
15  return  $lcs$ 

```

■ **Figure 7** Pseudocode implementation ALGSKIP, which is the backtracking of the LCS table using strategy $\langle \uparrow, \leftarrow, \nwarrow \rangle$.

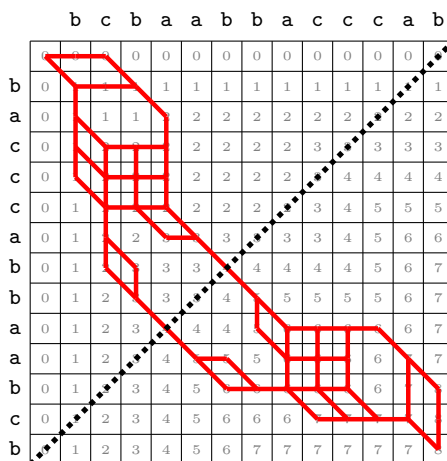
The ALGSKIP algorithm is not guaranteed to output the same LCS string as ALGCONTOUR. Trying to prove Lemma 9 breaks for ALGSKIP in the case where ALGSKIP and ALGCONTOUR start on the same cell, and ALGSKIP is allowed to leave the column by $\leftarrow$, with ALGCONTOUR leaving by $\nwarrow$. Concretely, on input strings $s = \text{cacacabbaaba}$ and $t = \text{acbaacaa}$ from Figure 4, ALGSKIP produces the LCS string acacaa , while ALGCONTOUR produces the string caacaa .

This covers the three basic strategies. Three other strategies can be obtained by switching the order of $\uparrow$ and $\leftarrow$ in the three presented algorithms. Switching $\uparrow$ and $\leftarrow$ is equal to switching the input strings s and t in the LCS table. Therefore, the switched ALGCONTOUR algorithm, strategy $\langle \leftarrow, \nwarrow, \uparrow \rangle$, instead follows the lower left contour of the profile. The switched ALGMATCH algorithm, strategy $\langle \nwarrow, \leftarrow, \uparrow \rangle$, outputs the same indexes in s as the switched ALGCONTOUR algorithm, leading it to agree with a contour of the profile. The switched ALGSKIP algorithm, strategy $\langle \leftarrow, \uparrow, \nwarrow \rangle$, similarly has no guarantee to output the same string as the switched ALGCONTOUR.

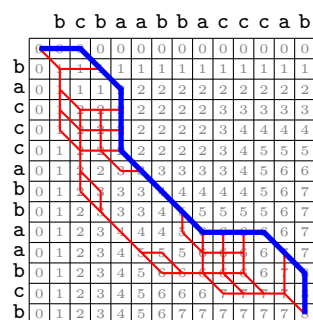
4 Computing an LPS using an LCS algorithm

In Section 3 it was shown that ALGMATCH outputs the same string as ALGCONTOUR. In this section we show that ALGCONTOUR(s, s^R) outputs an LPS of a string s , which implies that ALGMATCH(s, s^R) outputs an LPS of s .

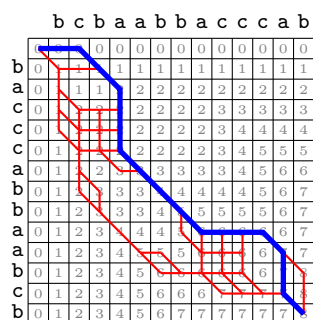
To argue that ALGCONTOUR outputs a palindrome, we observe that the profile is *symmetric* when the LCS is over some string s and the reversed string s^R . In Figure 8a, the profile is its own mirror over the dotted diagonal from the top-right to the bottom-left corner,



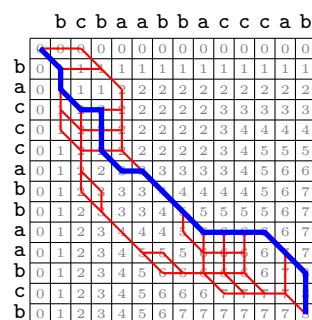
(a) Symmetry of the profile over the top-right to the bottom-left diagonal (dotted).



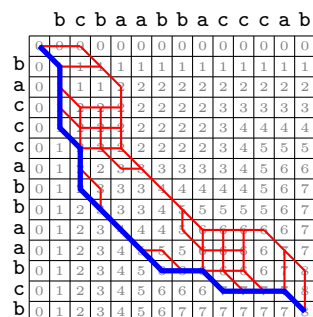
(b) Path using $\langle \uparrow, \swarrow, \leftarrow \rangle$, output string: baabbaab.



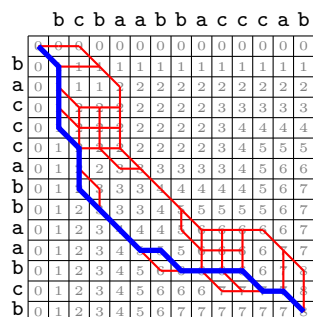
(c) Path using $\langle \swarrow, \uparrow, \leftarrow \rangle$, output string: baabbaab.



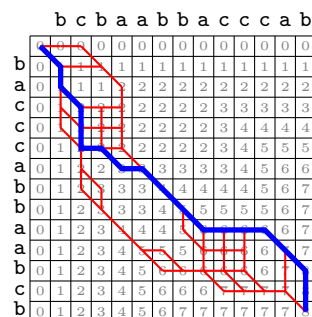
(d) Path using $\langle \uparrow, \leftarrow, \swarrow \rangle$, output string: bcabbaab.



(e) Path using $\langle \leftarrow, \swarrow, \uparrow \rangle$, output string: bcbaabcb.



(f) Path using $\langle \swarrow, \leftarrow, \uparrow \rangle$, output string: bcbaabcb.



(g) Path using $\langle \leftarrow, \uparrow, \swarrow \rangle$, output string: bcabbaab.

■ **Figure 8** LCS table of the string baccabbaabcb and the reversed string, including the profile (in red) with (top) highlighted mirror line and (bottom) different LCS paths (in blue) dependent on strategy.

when not considering the underlying orientation of the edges. In this mirroring, a cell (i, j) is *mirrored* to cell $(n - j, n - i)$. The individual edges are only of type $\nwarrow$, $\uparrow$, and $\leftarrow$, and under the mirroring new types of edges should not be created. An edge $(i, j) \rightarrow (i', j')$ is therefore mirrored to $(n - j', n - i') \rightarrow (n - j, n - i)$, i.e., the endpoints of the edge are mirrored and then the orientation is reversed, to preserve the set of types of edges. This results in $\nwarrow$ mirrored to $\nwarrow$, $\uparrow$ mirrored to $\leftarrow$, and $\leftarrow$ mirrored to $\uparrow$. Note that a $\nwarrow$ originating from cell (i, j) in the mirror becomes a $\nwarrow$ originating from cell $(n + 1 - j, n + 1 - i)$. In the following, define for an LCS path L the *mirror* L^R , as the edges of L mirrored and their order reversed.

► **Definition 10 (Symmetric).** *A profile is defined to be symmetric when the set of edges in the profile is equal to the set of mirrored edges. An LCS path L is defined to be symmetric when $L = L^R$, i.e., it is equal to its own mirror.*

► **Lemma 11.** *The profile of the LCS table of s and s^R is symmetric.*

Proof. Any edge in the profile must be included in some LCS path L . By showing that L^R is in the profile, it shows that the mirrored edge is in the profile.

If a cell (i, j) contains a $\nwarrow$, then it must hold that $s_i = s_j^R$. By definition of reversing a string it must hold that $s_i = s_{n+1-i}^R$, i.e., that the i 'th symbol of s is equal to the i 'th last symbol of s^R . Therefore it also holds that $s_{n+1-j} = s_{n+1-i}^R$, which results in cell $(n + 1 - j, n + 1 - i)$ also contains a $\nwarrow$.

All $\nwarrow$ in L become $\nwarrow$ in L^R , and by the above argument, all $\nwarrow$ of L^R exist in the table. All other steps of L^R are $\uparrow$ or $\leftarrow$, which must exist as subpaths between pairs of $\nwarrow$. As L^R contains the same number of $\nwarrow$ as L , then using a similar argument as in Lemma 4, these subpaths exist, and therefore L^R is an LCS path, concluding that L^R is in the profile. ◀

► **Lemma 12.** *Any symmetric LCS path L outputs a palindromic string.*

Proof. The proof goes by induction on the number of $\nwarrow$ in L . If there is at most one $\nwarrow$ in L , then the output string contains at most one symbol, and is therefore a palindrome.

Assume that L contains two or more $\nwarrow$. Traverse L until the first $\nwarrow$ is reached in cell (i, j) . By symmetry of L , then the last $\nwarrow$ of L is in cell $(n + 1 - j, n + 1 - i)$. The first symbol L outputs is s_i and the last symbol is s_{n+1-i}^R , which, by definition of s^R , are equal. By induction on the remaining of L , the output string is a palindrome. ◀

► **Lemma 13.** *The contour of the profile of the LCS table of s and s^R is symmetric.*

Proof. Assume for contradiction that the contour is not symmetric. Define the contour as C . By Lemma 11 it holds that the mirror C^R of the contour C is contained in the profile. As C is not symmetric, then there must be some part of C and C^R which is non overlapping. By the mirroring, this part exists with C^R above and/or to the right of C . Therefore C is not the top right most part of the profile, leading to the contradiction that C is not the contour. Therefore, the contour is symmetric. ◀

► **Theorem 14.** *The string produced by ALGMATCH of s and s^R is a palindrome.*

Proof. By Lemma 9, the string produced by ALGMATCH is equal to the string produced by ALGCONTOUR on any pairs of input strings, and therefore also on input s and s^R . By Lemma 13 and Lemma 12 the contour outputs a palindromic string. Therefore, the string produced by ALGMATCH is a palindrome. ◀

This implies that the first four strategies from Table 1 always outputs a palindromic string, as the first two strategies, $\langle \nwarrow, \uparrow, \leftarrow \rangle$ and $\langle \nwarrow, \leftarrow, \uparrow \rangle$, corresponds to the ALGMATCH and the switched ALGMATCH algorithms, with the next two strategies, $\langle \uparrow, \nwarrow, \leftarrow \rangle$ and $\langle \leftarrow, \nwarrow, \uparrow \rangle$, being the ALGCONTOUR and the switched ALGCONTOUR algorithms. To show that the last two strategies can output non-palindromic strings, Lemma 15 below shows this by example, using the string from Figure 8.

► **Lemma 15.** *The string produced by ALGSKIP can be non-palindromic.*

Proof. ALGSKIP follows the strategy $\langle \uparrow, \leftarrow, \nwarrow \rangle$. On input string $s = \text{bacccabbaabcb}$ and the reversed string s^R the output LCS string is bcabbaab , which is non-palindromic. See Figure 8d. ◀

References

- 1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.14.
- 2 Alfred V. Aho, Daniel S. Hirschberg, and Jeffrey D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, 23(1):1–12, 1976. doi:10.1145/321921.321922.
- 3 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141(1&2):163–173, 1995. doi:10.1016/0304-3975(94)00083-U.
- 4 Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In Pablo de la Fuente, editor, *Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000, A Coruña, Spain, September 27-29, 2000*, pages 39–48. IEEE Computer Society, 2000. doi:10.1109/SPIRE.2000.878178.
- 5 Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theoretical Computer Science*, 409(3):486–496, 2008. doi:10.1016/J.TCS.2008.08.042.
- 6 Karl Bringmann, Vincent Cohen-Addad, and Debarati Das. A linear-time $n^{0.4}$ -approximation for longest common subsequence. *ACM Transactions on Algorithms*, 19(1):9:1–9:24, 2023. doi:10.1145/3568398.
- 7 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 79–97. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.15.
- 8 Shihabur Rahman Chowdhury, Md. Mahbubul Hasan, Sumaiya Iqbal, and M. Sohel Rahman. Computing a longest common palindromic subsequence. In S. Arumugam and W. F. Smyth, editors, *Combinatorial Algorithms, 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19-21, 2012, Revised Selected Papers*, volume 7643 of *Lecture Notes in Computer Science*, pages 219–223. Springer, 2012. doi:10.1007/978-3-642-35926-2_24.
- 9 K. Chuang, R. Lee, and C. Huang. Finding all palindrome subsequences in a string. In *The 24th Workshop on Combinatorial Mathematics and Computation Theory*, 2007.
- 10 V. Chvatal, D. A. Karner, and D.E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, June 1972. URL: <http://i.stanford.edu/pub/ctr/reports/cs/tr/72/292/CS-TR-72-292.pdf>.
- 11 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition*. MIT Press, 2022. URL: <https://mitpress.mit.edu/9780262046305>.
- 12 Maxime Crochemore and Wojciech Rytter. *Jewels of stringology*. World Scientific, 2002. doi:10.1142/4838.

- 13 Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2006. URL: <https://www.mheducation.com/highered/product/M9780073523408.html>.
- 14 Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Applied Mathematics*, 212:96–103, 2016. doi:10.1016/J.DAM.2015.10.040.
- 15 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CB09780511574931.
- 16 Md. Mahbubul Hasan, A. S. M. Sohidull Islam, M. Sohel Rahman, and Ayon Sen. Palindromic subsequence automata and longest common palindromic subsequence. *Mathematics in Computer Science*, 11(2):219–232, 2017. doi:10.1007/S11786-016-0288-7.
- 17 Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. doi:10.1145/360825.360861.
- 18 Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675, 1977. doi:10.1145/322033.322044.
- 19 James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest subsequences. *Communications of the ACM*, 20(5):350–353, 1977. doi:10.1145/359581.359603.
- 20 Shunsuke Inenaga and Heikki Hyvrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Information Processing Letters*, 129:11–15, 2018. doi:10.1016/J.IPL.2017.08.006.
- 21 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theoretical Computer Science*, 410(51):5365–5373, 2009. doi:10.1016/J.TCS.2009.09.013.
- 22 Glenn K. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.
- 23 William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.
- 24 Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982. doi:10.1007/BF00264437.
- 25 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001. doi:10.1145/375360.375365.
- 26 Alexandre H. L. Porto and Valmir Carneiro Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 35(11):2581–2591, 2002. doi:10.1016/S0031-3203(01)00179-0.
- 27 David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences of the United States of America*, 69(1):4–6, 1972. URL: <http://www.jstor.org/stable/60967>.
- 28 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.
- 29 Woburn Collegiate Institute Programming Enrichment Group (PEG) Wiki. Longest palindromic subsequence. http://wcipeg.com/wiki/Longest_palindromic_subsequence?#LCS-based_approach. Accessed: 2024.04.20.

A AI assistants

A frequent statement on the internet³ is that an LPS of a string can be computed as an LCS of the string and the reversed string^{4,5}, which is reflected in the suggestions from AI assistants like GitHub Copilot⁶, Google Gemini⁷, and Open AI ChatGPT⁸. The Python 3.12 code in Figure 9 was given to the free versions of GitHub Copilot, Google Gemini, and OpenAI ChatGPT 3.5, without the implementation for `palindrome` but including its docstring. All three AI assistants suggested `return lcs(x, x[::-1])` as the completion of the code, i.e., to return the LCS of the string and the reversed string. This suggestion would be correct if `lcs(x, y)` was implemented using Wagner and Fischer [28], but in the code provided to the AI assistants, the `lcs` function uses a dynamic programming implementation with strategy $\langle \uparrow, \leftarrow, \nearrow \rangle$ as the backtracking prioritization, and the reported subsequence is not a palindrome.

Identical suggestions were obtained on `lmsys.org`⁹ for ChatGPT 4 (gpt-4o-2024-05-13), Meta Llama 3 (llama-3-70b-instruct), Claude 3.5 Sonnet (version claude-3-5-sonnet-20240620), Databricks DBRX (dbrx-next), Microsoft Phi-3 mini (phi-3-mini-4k-instruct), NVIDIA Nemotron-4 340B (nemotron-4-340b), Alibaba Cloud Qwen2 (qwen2-72b-instruct) and others (deepseek-coder-v2, glm-4-0520, reka-flash-preview-20240611, yi-large-preview).

```
from functools import cache

@cache
def lcs(x, y):
    '''Longest common subsequence of strings x and y.'''

    if not x or not y:
        return ''
    if x[-1] == y[-1]:
        return max(lcs(x[:-1], y), lcs(x, y[:-1]),
                    lcs(x[:-1], y[:-1]) + x[-1], key=len)
    return max(lcs(x[:-1], y), lcs(x, y[:-1]), key=len)

def palindrome(x):
    '''Longest palindromic subsequence of string x.'''

    # Suggestion by GitHub Copilot, Google Gemini, OpenAI ChatGPT
    return lcs(x, x[::-1])

print(f'{palindrome('acbcabbaac')} = }')
```

Output

```
palindrome('acbcabbaac') = 'cabbaa'
```

Figure 9 Implementation of `palindrome` according to GitHub Copilot, Google Gemini, and OpenAI ChatGPT 3.5.

³ All links in this section were accessed June 25, 2024.

⁴ <https://www.geeksforgeeks.org/print-longest-palindromic-subsequence/>

⁵ <https://www.educative.io/answers/longest-palindromic-subsequence-algorithm>

⁶ <https://github.com/features/copilot/>

⁷ <https://gemini.google.com/>

⁸ <https://chat.openai.com/>

⁹ <https://chat.lmsys.org/>

Pure Binary Finger Search Trees*

Gerth Stølting Brodal[†]

Casper Moldrup Rysgaard[‡]

Abstract

We present dynamic binary search trees where each node only stores a value and pointers to its parent and its children. We denote such binary search trees *pure* binary search trees. Our structure supports finger searches in worst-case $\mathcal{O}(\lg d)$ time, where d is the rank difference between the node given by the finger and the node found by the search. Inserting a new node with a successor or predecessor value for a node pointed to by a finger and deleting a node in the tree pointed to by a finger are supported in amortized $\mathcal{O}(1)$ time and worst-case $\mathcal{O}(\lg n)$ time, where n is the number of nodes in the tree. The temporary working space during the operations is $\mathcal{O}(1)$ words. The result is obtained by an alternative representation of the red-black trees by Guibas and Sedgwick [FOCS 1978] that encodes bits of information in the tree structure, generalizing the encoding of 2-3-trees by Brown [IPL 1979], and rearranging the nodes in a red-black tree (“folding” left and right paths) such that the predecessor and successor of a node can always be found in worst-case constant time. The same time bounds can easily be obtained by, say, red-black trees and AVL trees augmented with pointers to the predecessor and successor of each node. The novelty of our result is that we store no extra information than the binary tree structure. The structure can be represented by two pointers per value, i.e., the same representation as a doubly linked list.

Keywords: Binary search trees, finger searches, dynamic data structure, information encoding

1 Introduction

In this paper we present the first *pure* binary search trees supporting finger searches and finger updates in optimal time. By a pure binary search tree we denote a binary search tree (BST) where each node only stores a single value and pointers to the children and parent. No additional pointers or bits of information is allowed to be stored in a node. All previous finger search trees require more information to be stored for each node to achieve matching time bounds.

The overall goal of the presented research is to study how efficient operations one can achieve for BSTs if the representation should be as simple as a pure BST. The goal of the research presented in this paper became more general over time. Our final construction solves each of the following open problems. *i*) The initial open problem was the following: Some search trees, e.g., red-black trees [26], support the insertion of new values in a tree with n values in worst-case $\mathcal{O}(\lg n)$ time¹, but if the insertion point of the new value (empty leaf) is already known, the insertion can be performed in amortized constant time. But red-black trees store balance information at each node (a bit indication the color red or black). The first problem asked was if there exist balanced pure BSTs without any balance information at the nodes (like splay trees [41], scapegoat trees [3, 23], or encoded 2-3 trees [14, 15]) that could achieve matching time bounds for searching and adding a new leaf? *ii*) Adding a new leaf might be a canonical operation on a BST, but a more canonical black-box interface is to add a predecessor or successor value for an existing node in the search tree, like in a doubly linked list (see Figure 1). Can such insertions be supported in constant time in a balanced BST? In a balanced BST the initial step of an insertion (before some rebalancing is performed to ensure logarithmic height) is to insert the new value at the leftmost empty leaf in the right subtree of the node (see Figure 2). But finding this leaf can take worst-case $\Omega(\lg n)$ time, provided no shortcut pointers are maintained to the successor and predecessor of a node. *iii*) In addition to insertions, can deletions of arbitrary nodes be supported in constant time, like in a doubly linked list (see Figure 1)? In a balanced BST the deletion of a node with two children usually first swaps the node with its predecessor or successor, that takes worst-case $\Omega(\lg n)$ time to find, such that the deletion is reduced to the deletion of a node with at most one child. *iv*) Finally, can we in addition to $\mathcal{O}(\lg n)$ time top-down searches in a balanced pure BST

*Supported by Independent Research Fund Denmark, grant 9131-00113B.

[†]Aarhus University, Department of Computer Science, gerth@cs.au.dk. ORCID 0000-0001-9054-915X.

[‡]Aarhus University, Department of Computer Science, rysgaard@cs.au.dk. ORCID 0000-0002-3989-123X.

¹ $\lg n$ denotes the binary logarithm of n .

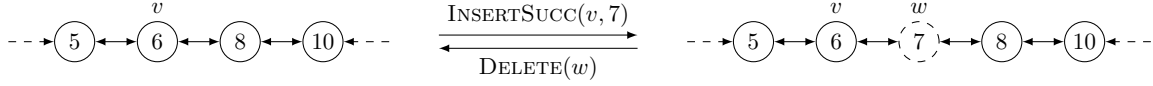


Figure 1: Inserting 7 as a new successor node w to an existing node v in a doubly linked list, and reversely the deletion of a node w from a doubly linked list.

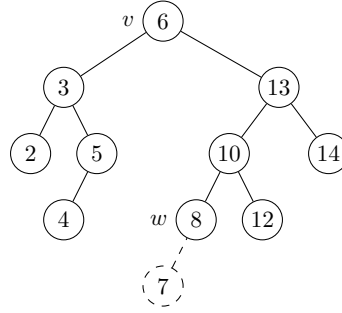


Figure 2: Performing $\text{INSERTSUCC}(v, 7)$ inserts 7 as a successor of 6 in a BST by inserting 7 as a predecessor (left child) of the successor w of v .

also support finger searches in $\mathcal{O}(\lg d)$ time (d being the rank difference between the start and end nodes of a search)? Previous finger search trees require additional pointers to achieve this goal, e.g., like [27, 12].

1.1 Search tree operations In this section we give a more formal definition of the BST operations we consider in this paper. We assume that a BST stores n ordered values $e_1 \leq \dots \leq e_n$ in nodes $v_1, \dots, v_n$, respectively. Note that we do not require the values stored to be distinct, i.e., the same value can be stored in multiple nodes. We assume that the n nodes form a BST, i.e., v_i is the i th node in an inorder traversal of the tree. We say that e_i and v_i have *rank* i , i.e., if there are multiple nodes with equal value, the rank of a value is defined by the index of the node it is stored in. We only consider “stable” BSTs, i.e., whenever a value is inserted into a node and a pointer to this node is returned, then the value resides in the node until deleted from the BST. See [7] for a discussion of stability in practical data structure libraries. The operations we consider supported are:

PRED(v) Takes a pointer v to a node v_i . Returns a pointer to v_{i-1} , if $i > 1$, and NIL otherwise.

SUCC(v) Takes a pointer v to a node v_i . Returns a pointer to v_{i+1} , if $i < n$, and NIL otherwise.

INSERTPRED(v, e) Takes a pointer to a node v and a value e . Creates a new node v' storing e . If $v = \text{NIL}$, v' is a new tree by itself. Otherwise assume v points to v_i . It is a precondition that $e_{i-1} \leq e \leq e_i$ (assuming $e_0 = -\infty$). Inserts v' in the tree, such that v' is after v_{i-1} (if v_{i-1} exists) and before v_i in the new inorder of the tree. Returns a pointer to v' .

INSERTSUCC(v, e) Takes a pointer to a node v and a value e . Creates a new node v' storing e . If $v = \text{NIL}$, v' is a new tree by itself. Otherwise assume v points to v_i . It is a precondition that $e_i \leq e \leq e_{i+1}$ (assuming $e_{n+1} = \infty$). Inserts v' in the tree, such that v' is after v_i and before v_{i+1} (if v_{i+1} exists) in the new inorder of the tree. Returns a pointer to v' .

DELETE(v) Takes a pointer to a node v . Removes the node v and its associated value from the tree containing v .

FINGERSEARCH(v, e) Takes a pointer v to a node v_i and a value e . If $e < e_1$, returns a pointer to v_1 . Otherwise, returns a pointer to the node v_j , such that $e_j \leq e < e_{j+1}$ (assuming $e_{n+1} = \infty$). For a finger search, we let $d = |j - i|$ denote the rank difference between the start node and the node returned by the search.

We denote the three operations INSERTPRED , INSERTSUCC and DELETE as *finger updates*.

1.2 Binary search trees BSTs are one of earliest data structures in computer science, see, e.g., Windley [47] from 1960 for an early description of BSTs. Knuth [31, page 453] gives an account of the early history of BSTs, and Andersson et al. [4] of later developments on balanced BSTs.

The classic AVL trees of Adelson-Velsky and Landis [1] support insertions, deletions and searches in $\mathcal{O}(\lg n)$ time. Mehlhorn and Tsakalidis [35] showed that the amortized restructuring cost of AVL trees after an insertion is amortized $\mathcal{O}(1)$ time, i.e., an AVL tree supports finger insertions in amortized constant time if the AVL tree is augmented with successor/predecessor pointers. The red-black trees of Guibas and Sedgwick [26] achieve matching $\mathcal{O}(\lg n)$ worst-case bounds. Tarjan [44] proved that the rebalancing required by both insertions and deletions in a red-black tree is amortized constant time, i.e., a red-black tree supports finger updates in amortized constant time provided the red-black tree is augmented with successor/predecessor pointers. Insertions and deletions perform at most two and three rotations, respectively [43, 44].

The splay trees of Sleator and Tarjan [41] are arguably the simplest BST representation: Each node stores a value and two pointers to the children. They support insertions, deletions and searches in amortized $\mathcal{O}(\lg n)$ time. Even when a finger insertion can be done in worst-case constant time in a splay tree, as shown in Figure 3, the insertions must be charged logarithmic potential increase to support amortized logarithmic queries, i.e., the amortized insertion time is logarithmic.

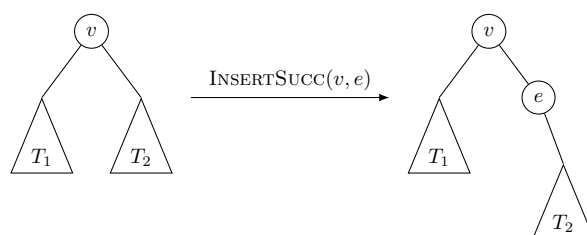


Figure 3: $\text{INSERTSUCC}(v, e)$ in a splay tree.

Levcopoulos and Overmarks [33] describe BSTs supporting finger updates in worst-case $\mathcal{O}(1)$ time and searches in $\mathcal{O}(\lg n)$ time, essentially by maintaining bags of $\Theta(\lg^2 n)$ values.

Seidel and Aragon [40] presented simple randomized search trees denoted *treaps*, where a random priority is assigned to each value and the BSTs are heap ordered with respect to the priorities. Treaps support updates and searches in expected $\mathcal{O}(\lg n)$ time, and store expected $\mathcal{O}(1)$ bits at each node, and with high probability at most $\mathcal{O}(\lg n)$ bits. Martínez and Roura [34] considered a variant of treaps where each node instead of a random priority only stores the subtree size. Subsequently, Seidel [39] presented a variant of treaps where nodes store no balance information, i.e., they are pure BSTs, but the update time increases to expected $\mathcal{O}(\lg^2 n)$ time. Gila, Goodrich and Tarjan [24] presented zip-zip trees, that similar to treaps store random bits at the nodes of a BST, but they reduce the high probability bound on the number of random bits to $\mathcal{O}(\lg \lg n)$.

Andersson [3] and independently Galperin and Rivest [23] presented the scapegoat trees, that are essentially pure BSTs, except for a global variable storing the tree size n . The invariant is that the tree has height $\mathcal{O}(\lg n)$, which is maintained by rebuilding subtrees whenever the height invariant is violated.

Brown [14, 15] presented a pure BST supporting updates and searches in $\mathcal{O}(\lg n)$ time, by encoding a 2-3 tree [2] in the structure of a pure BST, where pairs of nodes encode bits of balance information for a 2-3 tree node. It is crucial for the decoding that operations start at the root, i.e., finger operations cannot be supported.

1.3 Finger search data structures A finger search in a sorted array can be implemented by an exponential search as shown in Figure 4, consisting of a doubling phase followed by a binary search phase, in total performing $2 \lg d + \mathcal{O}(1)$ comparisons.

Bentley and Yao [8] studied the comparison complexity of finger searches. They proved that a finger search requires $\lg^{(1)} d + \lg^{(2)} d + \lg^{(3)} d + \dots + \lg^{(\lg^* d)} d \pm \Theta(\lg^* d)$ comparisons, where $\lg^{(1)} d = \lg d$, $\lg^{(i+1)} d = \lg(\lg^{(i)} d)$, and $\lg^* d = \min\{i \in \mathbb{N} \mid \lg^{(i)} d \leq 1\}$, i.e., they proved that the number of comparisons performed by exponential search can be improved by essentially a factor two.

A balanced BST, like an AVL tree or a red-black tree, can support a finger search in $\mathcal{O}(\lg d)$ if the finger is at the smallest or largest value in the tree: Move up along the left or right spine of the tree, respectively, until a

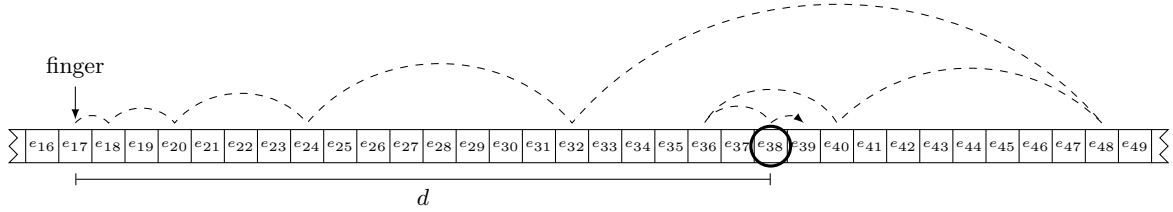


Figure 4: Exponential search for e in a sorted array, where $e_{38} \leq e < e_{39}$.

top-down search can be performed on a single subtree attached to the spine. Building on this observation, Guibas et al. [25] presented search trees based on B-trees supporting $\mathcal{O}(1)$ *moveable* fingers, finger updates in constant time, and finger searches and moving a finger in $\mathcal{O}(\lg d)$ time. Kosaraju [32] achieved a similar result based on 2-3-trees, Tsakalidis [46] a solution based on AVL trees, and Tarjan and van Wyk [45] a solution based on red-black trees. Huddleston and Mehlhorn [27] presented a solution based on *level-linked* (a, b) -trees, supporting finger updates in amortized constant time and finger searches in worst-case $\mathcal{O}(\lg d)$ time—without any restriction on the number of fingers. Their structure requires five pointers per node. Brodal et al. [12] presented an involved pointer based finger search structure supporting finger updates in worst-case constant time and finger searches in worst-case $\mathcal{O}(\lg d)$ time.

Treaps [40] support finger updates in expected constant time and finger searches in expected $\mathcal{O}(\lg d)$ time, essentially since the random structure of treaps implies that a finger is expected to be close to a leaf. Splay trees were analyzed in the context of finger searches by Cole et al. [16, 17], who proved that splay trees support searches, insertions, and deletions in amortized $\mathcal{O}(\lg d)$ time, where d is the rank distance from the last accessed node.

Deviating from pointer based structures, Anderson and Thorup [5] presented optimal finger search trees in the RAM model supporting finger updates in constant time and finger searches in $\mathcal{O}(\sqrt{\lg d / \lg \lg d})$ time. Kaporis et al. [30] considered interpolation based finger search structures on the RAM model achieving $\mathcal{O}(\lg \lg d)$ expected search time for a large class of input distributions.

For more results on finger search data structures, see the survey by Brodal [10]. Table 1 summarizes the finger update bounds for various BSTs supporting an arbitrary number of fingers and the amount of balance information they store in each node.

1.4 Result Our main result is essentially an alternative representation of red-black trees achieving:

THEOREM 1.1. *There exist pure binary finger search trees supporting PRED and SUCC in worst-case $\mathcal{O}(1)$ time, INSERTPRED, INSERTSUCC, and DELETE in amortized $\mathcal{O}(1)$ time and worst-case $\mathcal{O}(\lg n)$ time, and FINGERSEARCH in worst-case $\mathcal{O}(\lg d)$ time, where n is the number of nodes in the tree before the operation and d is the rank difference between the node given by the finger and the node found by the search. The tree can be represented by n records, where each node stores a value and two pointers (a pointer to the left child, and a pointer to either the right sibling or the parent). The temporary working space during the operations is $\mathcal{O}(1)$ words.*

The result is achieved by combining four simple ideas: *i)* Red-black trees support finger searches in asymptotic optimal time if the successor and predecessor of a node can be accessed in constant time; *ii)* rearranging the nodes in a red-black tree by “folding” left and right paths (adding a single bit to the nodes but no additional pointers), successor and predecessor nodes can be located in $\mathcal{O}(1)$ time, while still allowing access to the parent and children of a red-black tree node in $\mathcal{O}(1)$ time; *iii)* by partitioning the nodes of a pure BST into “metanodes”, $\mathcal{O}(1)$ bits of information can be encoded by the tree structure of each metanode; and *iv)* by ensuring all unary nodes have a left child, two pointers are sufficient to represent each node.

1.5 Outline of paper The paper is organized top-down gradually introducing details to keep a focus of the contribution of the individual ideas to our construction. In Section 2 we first recall the definition and operations on red-black tree. In Section 3 we describe how red-black trees support optimal finger searches, provided that

Reference	Finger updates	Information stored
AVL trees [1]	$\mathcal{O}(\lg n)$	2 bits per node
Red-black trees [26]	$\mathcal{O}(\lg n)$	1 bit per node
Red-black trees + predecessor/successor links [44]	$\mathcal{O}_A(1)$	$\mathcal{O}(\lg n)$ bits per node
Scapegoat trees [3, 23]	$\mathcal{O}_A(\lg n)$	global tree size
Level-linked (a, b) -trees [27]	$\mathcal{O}_A(1)$	$\mathcal{O}(\lg n)$ bits per node
BST with $\mathcal{O}(\lg^2 n)$ sized bags [33]	$\mathcal{O}(1)$	$\mathcal{O}(\lg n)$ bits per node
Treaps [40]	$\mathcal{O}_E(1)$	$\mathcal{O}_E(1)$ bits per node
Seidel random BST [39]	$\mathcal{O}_E(\lg^2 n)$	none
Splay trees [41]	$\mathcal{O}_A(\lg n)$	none
Encoded 2-3 trees [14, 15]	$\mathcal{O}(\lg n)$	none
<i>This paper</i>	$\mathcal{O}_A(1)$	none

Table 1: Results for selected balanced BSTs. $\mathcal{O}_A$ and $\mathcal{O}_E$ denote amortized and expected bounds, respectively. All search trees have logarithmic height, and support searches in logarithmic time, except [39, 40] are randomized and bounds are expected, and bounds for [41] are amortized, with no bound on the tree height.

red-black trees are augmented to support finding the successor and predecessor of a node in worst-case constant time. In Section 4 we describe how paths in red-black trees can be rearranged such that the resulting tree is still a BST with two bits of information at each node, but no additional pointers, where we can navigate and update as in a red-black tree plus have access to the predecessor and successor of a node in constant time. In Section 5 we describe how we can group nodes of a pure BST into *metanodes*, where the metanodes form binary subtrees and encode bits of information through the tree structure, and in Section 6 we describe how to store the resulting pure BST with only two pointers per node. In Section 7 we combine the constructions of the preceding sections to achieve our main result (Theorem 1.1). In Section 8 we argue that it is not possible to achieve worst-case constant time finger updates and logarithmic searches if the structure is a pure BST, but in Section 9 we show that better bounds are possible for general pointer structures where each node only stores two pointers. Finally, in Section 10 we give some concluding remarks and mention some open problems.

2 Red-black trees

In this section we briefly recall the red-black trees of Guibas and Sedgewick [26], and how they support the operations described in Section 1.1. The amortized restructuring analysis of red-black trees is due to Tarjan [44]. The rebalancing operations in Figure 6 and 7 are from Cormen et al. [18].

Each node stores a value, pointers to its left child, right child, and parent, and a single bit indicating if the node is red or black. Red-black trees satisfy the invariants:

1. A red node has a black parent, and
2. the number of black nodes on all root-to-external-leaf paths are the same.

If k denotes the *black height* of the tree, i.e., the number of black nodes on any root-to-leaf path, then the tree contains between $2^k - 1$ and $2^{2k} - 1$ nodes, implying that the height of a red-black tree is $\mathcal{O}(\lg n)$. Figure 5 shows a red-black tree with black height 3.

A shortcoming of red-black trees, and most BSTs, is that the successor and predecessor of a node are not necessarily adjacent to the node (e.g., in Figure 5 the nodes storing values 6 and 7 are not adjacent). In general there can be $\Theta(\lg n)$ nodes on the paths between a node and its successor and/or predecessor. In Section 3 we describe how a red-black tree can support finger searches in $\mathcal{O}(\lg d)$ time, provided the successor and predecessor of a node can be accessed in $\mathcal{O}(1)$ time, e.g., by adding additional links. In Section 4 we describe how to rearrange the nodes in the representation of a red-black tree such that we can both navigate and update the red-black tree, as if it was stored using the normal representation, and simultaneously have access to the successor and predecessor of a node in $\mathcal{O}(1)$ time, without additional links (but one more bit per node).

A $\text{Succ}(v)$ query is supported in $\mathcal{O}(\lg n)$ time as follows: If v has a right child, returns the node found by moving to the right child of v and then repeatedly moving to the left child until a node with no left child is reached. If v does not have a right child, repeatedly moves up to the parent until v is in the left subtree. If such

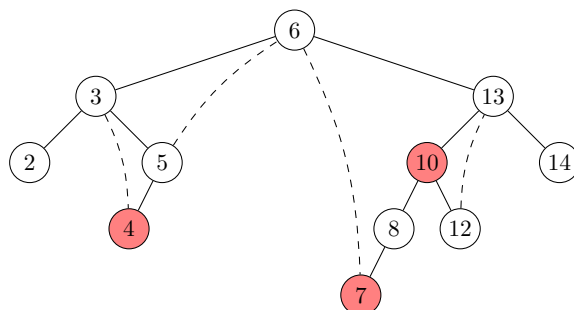


Figure 5: A Red-black tree (red nodes are shaded red; the remaining nodes are black). Dashed lines show where the successor or predecessor of a node is not an adjacent node.

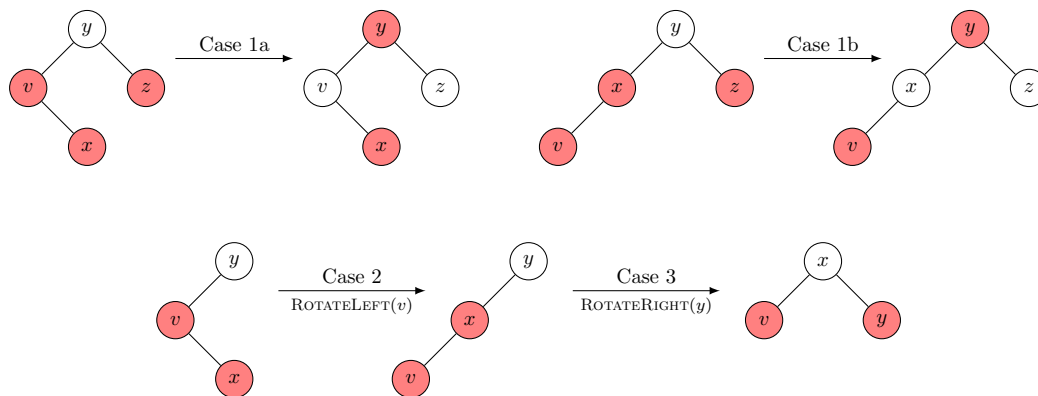


Figure 6: Insertion rebalancing cases on red-black trees, when the two adjacent red nodes are on the left. Omitted subtrees have a black root or are empty. Cases 1a and 1b perform recoloring and continues from node y , while Cases 2 and 3 perform rotations in addition to recoloring, before terminating.

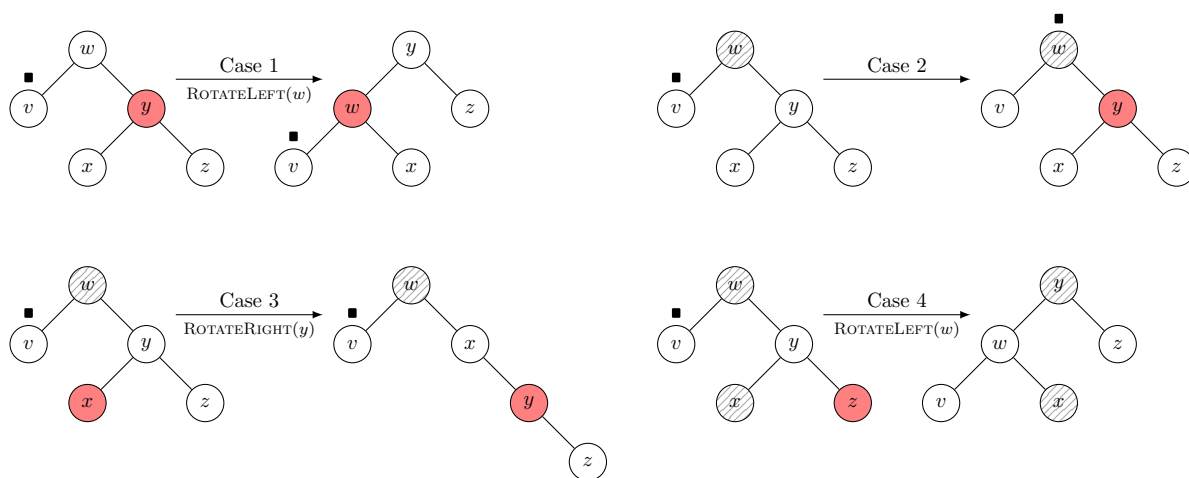


Figure 7: Deletion rebalancing cases on red-black trees, when the double black node (marked ■) is on the left. The dashed nodes denote a node which color can be either red or black. Omitted subtrees can be empty or have a black or red root. The node v can possibly be an empty subtree. Case 2 performs recoloring and continues from node w , while Cases 1, 3, and 4 perform rotations in addition to recoloring. In Case 4, the color of node y after rebalancing is the color of node w before rebalancing.

a node is found, it is returned. Otherwise, v is the rightmost node in the tree and NIL is returned. The query $\text{PRED}(v)$ is handled symmetrically.

The finger insertion $\text{INSERTSUCC}(v, e)$ creates a new node v' with value e . If v has no right child, v' becomes the right child of v . Otherwise, v' becomes the left child of $\text{SUCC}(v)$. In both cases v' becomes a red leaf and the black height invariant is maintained, but potentially v' might have a red parent. This is fixed in amortized $\mathcal{O}(1)$ time by performing amortized $\mathcal{O}(1)$ local transformations on the tree, involving amortized $\mathcal{O}(1)$ recoloring at most two rotations. The transformations are recalled in Figure 6. $\text{INSERTPRED}(v, e)$ is handled symmetrically. A finger deletion $\text{DELETE}(v)$ first checks if v has two children. If v has at most one child, we delete v from the tree, replacing it by its child if it had one. If v has two children, the values of v and $\text{SUCC}(v)$ are swapped, and we delete $\text{SUCC}(v)$, that has at most one child. If the deleted node was red, we are done. Otherwise the black height invariant might be violated, and a sequence of amortized $\mathcal{O}(1)$ transformations are applied where one node is treated having an extra layer of black color (initially the child that replaced the deleted node, possibly a NIL external leaf). The transformations are shown in Figure 7, where the “double black” node is marked with $\blacksquare$. If a red node becomes marked $\blacksquare$, it is made a black node, and the transformations terminate. The transformations perform amortized $\mathcal{O}(1)$ recoloring and at most three rotations. In total, the time for a finger update is amortized $\mathcal{O}(1)$ plus the time for a SUCC query (and worst-case $\mathcal{O}(\lg n)$ time).

3 Finger searches using predecessor and successor links

In this section we describe how a balanced binary tree, say a red-black tree or an AVL tree, can support a finger search in $\mathcal{O}(\lg d)$ time, if the predecessor $\text{PRED}(v)$ and successor $\text{SUCC}(v)$ of a node v can be found in constant time. Whereas $\text{PRED}(v)$ and $\text{SUCC}(v)$ can be supported in constant time trivially by augmenting a BST with these as pointers (such that the nodes are also maintained in a sorted doubly linked list), we in Section 4 show how the additional pointers can be avoided for the case of red-black trees by rearranging the nodes.

Let u denote the node to be returned by a finger search $\text{FINDERSEARCH}(v, e)$, i.e., the rightmost node in the tree storing a value $\leq e$. In a normal top-down search starting at the root, we would go left whenever a node stores a value larger than the query value e and right otherwise, until we reach an empty leaf. The node u to be returned is the last visited node on the search path with value $\leq e$, i.e., the last node where the search continued to the right child. Our finger search algorithm makes repeated use of such top-down searches on various subtrees.

Our finger search algorithm uses a parameter h , bounding how many levels we are allowed to move up in the tree. Initially h is $\mathcal{O}(1)$. If a search fails because h is too small, we double h , and restart the search. For each choice of h the algorithm will spend $\mathcal{O}(h)$ time, i.e., the total running time will be a geometric sum asymptotically bounded by the final choice of h , that will be $\mathcal{O}(\lg d)$.

In the following we assume the value at v is $\leq e$. The case where the query value e is smaller than the value at v follows by a symmetric argument, where calls to SUCC are replaced by calls to PRED . A search $\text{FINDERSEARCH}(v, e)$ proceeds as follows and as illustrated in Figure 8. If v has a right child, call $\text{SUCC}(v)$ to find the leftmost node v_s in the right subtree of v . If the value at v_s is $> e$, return v as the answer. If v does not have a right child, let $v_s = v$. From v_s move up the ancestor path h levels to reach node r_ℓ (or until the root is reached). Perform a top-down search in the subtree rooted at r_ℓ . If the top-down search from r_ℓ finds a node with value $> e$, we return the answer u found in the subtree rooted at r_ℓ . Otherwise, we have reached the rightmost node ℓ in the subtree rooted at r_ℓ , and the value at ℓ is $\leq e$. We find $c = \text{SUCC}(\ell)$. If c is NIL , then ℓ stores the largest value in the tree, and we return ℓ . Otherwise, c is an ancestor of both ℓ and r_ℓ , and ℓ is the rightmost node in the left subtree of c . If c stores a value $> e$, return ℓ . Otherwise, compute $r = \text{SUCC}(c)$. If r is NIL or stores a value $> e$, return c . Otherwise, from r move up the ancestor path h levels to reach node r_r (or until the root is reached), and perform a top-down search in the subtree rooted at r_r . If the top-down search from r_r finds a node with value $> e$, we return the answer u found in the subtree rooted at r_r . Otherwise, we declare the search to have failed, and restart with a larger h .

For the analysis, we assume all root-to-leaf paths in a subtree are guaranteed to contain the same asymptotic number of nodes. E.g., for red-black trees and AVL trees the difference between the longest and shortest paths is a factor two. For fixed h , the total search time is $\mathcal{O}(h)$ plus the time for the top-down searches from r_ℓ and r_r . For r_ℓ , observe that we reach r_ℓ along an ancestor path of length h from v or $\text{SUCC}(v)$, where either v has no right child or $\text{SUCC}(v)$ has no left child, respectively. It follows that the subtree rooted at r_ℓ has height $\mathcal{O}(h)$. For r_r , observe that r either has no left child (r is the leftmost node in the right subtree of c), or c has no right child and c is the left child of r . Since we reach r_r from r along an ancestor of length h , the subtree rooted at r_r

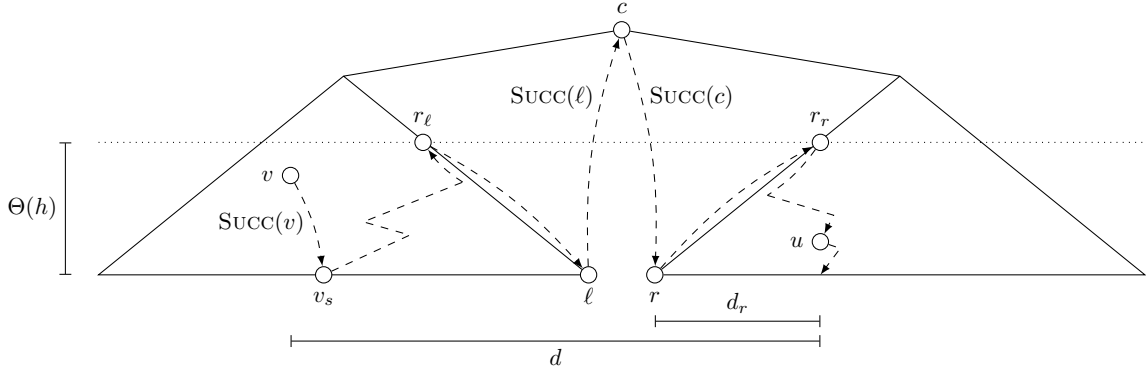


Figure 8: The steps of $\text{FINGERSEARCH}(v, e)$ for a fixed h in a BST with successor and predecessor links. The dashed path are the vertices visited and u the returned node.

has height $\mathcal{O}(h)$. For fixed h , this implies that the top-down searches at r_ℓ and r_r take $\mathcal{O}(h)$ time, i.e., $\mathcal{O}(h)$ total time. If a search fails, then the value at r_r is $\leq e$, and the final answer will be r_r or a node to the right of r_r . If r_r is in the right subtree of c , the rank distance d between v and the final node u is at least the rank difference d_r between r and r_r , and d_r equals the number of nodes in left subtree of r_r , where $d_r \geq 2^{\Omega(h)}$. If r_r is c or an ancestor of c (possibly to the left of v), and the search fails, then all $2^{\Omega(h)}$ nodes in the right subtree of c store values $\leq e$, i.e., $d \geq 2^{\Omega(h)}$. In both cases h is $\mathcal{O}(\lg d)$ when the search fails. Since h for the final successful search is twice the last failed h value, it follows that the total time for a finger search is $\mathcal{O}(\lg d)$.

4 Folded red-black trees

In this section we describe the *folded* representation of red-black trees, that we denote *folded red-black trees*, further abbreviated as *folded trees*. We let *unfolded red-black trees* denote regular red-black trees (Section 2), similarly abbreviated *unfolded trees*. The folded tree representation of an unfolded tree is a unique BST on the same set of nodes as the unfolded tree, but with an alternative arrangement. The goal of folded trees is to support access to the predecessor or successor of a node in constant time, access to the parent and children in the corresponding unfolded tree in constant time, and to support finger updates in amortized constant time (and worst-case logarithmic time), *without* adding additional pointers to the nodes of the tree (but with adding one additional bit to each node).

A folded tree consists of folding disjoint *left paths* and *right paths* in the unfolded tree. A left path (right path) is a maximal path of nodes that are left (right) children. The construction is applied recursively, such that the right subtrees of a left folded path are *right folded*, and the left subtrees of a right folded path are *left folded*. The initial fold is on the left path from the root.

Consider a left path in an unfolded tree (right paths are handled symmetrically). If the path contains only a single node, i.e., the node has no left child, the folded path is only this node and its right subtree recursively right folded. Otherwise, we split the unfolded left path in two parts; an *upper* part and a *lower* part. Denote nodes on the upper part as *upper nodes* and nodes on the lower part as *lower nodes*. The lower part is partitioned into connected *lower chunks*. The topmost lower chunk is denoted the *last lower chunk*. See Figure 9 (left) for an illustration of these definitions. The partitioning is created as follows to guarantee that *the number of upper nodes is equal to the number of lower chunks*: Initially, the topmost node of the path is the only upper node and all other nodes constitute the last lower chunk. Repeatedly, split off the lowest black node from the last lower chunk, and possibly its left red child, as a new lower chunk and split off the topmost node as a new upper node, provided the last lower chunk remains non-empty. Each iteration creates one new lower chunk and one new upper node, i.e., the final number of upper nodes equals the number of lower chunks. All lower chunks, except the last lower chunk, by definition consist of a lower black node, and additionally its left child if this node is red. Since in each iteration two or three nodes are removed from the last lower chunk, we can always split off an upper node and a lower chunk while the last lower chunk has four or more nodes, i.e., the final non-empty last lower chunk can at most have three nodes. Furthermore, if it contains three nodes, the lowest node must be red, since otherwise

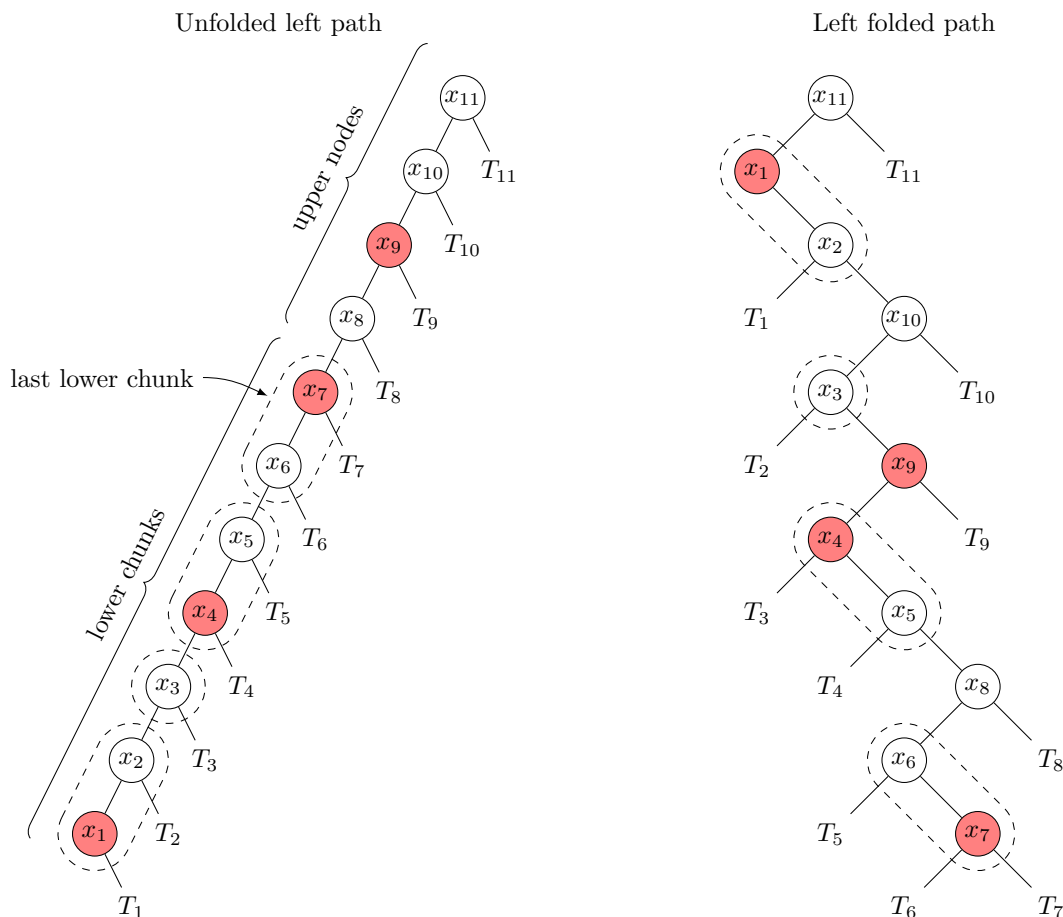


Figure 9: (left) unfolded left path and (right) the folded rearrangement of the path. The dashed areas denote the lower chunks, and the nodes in non-dashed areas denote the upper nodes, with (left) their unfolded positions and (right) their folded positions. T_i is (left) the right subtree of node x_i , and (right) the recursively folded subtree at the new location in the folded path.

we can split off an upper node and a single black node as a lower chunk. We conclude that the last lower chunk contains 1–3 nodes, and only three nodes if the lowest node in the last lower chunk is red. The possible last lower chunks are illustrated in Figure 10.

The left folded path is constructed by alternating upper nodes and lower chunks, starting with the top upper node and ending with the last lower chunk, such that lower chunks are left children of upper nodes, and upper nodes are right children of lower chunks. The top-down order of the lower nodes are reversed in the left folded path, such that if in an unfolded tree a lower chunk contains v and its left child $u = v.l$, then in the folded tree v is the right child of u . See Figure 9 for an illustration of an unfolded left path, and the folded representation of that path. Let $x_1, x_2, \dots$ denote the nodes bottom-up on the unfolded left path. Each node x_i on the left path in the unfolded tree has a right subtree T_i , possibly empty. For each upper node x_i in the folded tree, the right subtree is T_i . Similarly, the right subtree of the node at the end of the last lower chunk is placed as the right child (x_7 and T_7 in Figure 9). Otherwise, the right subtree T_i of a lower node x_i is placed as the left subtree of x_{i+1} (since T_i contains values between x_i and x_{i+1}). For each subtree T_i , we recursively fold the right path.

In the folded tree, when given a node, it cannot be detected locally if the node is on a right or left path. We therefore add one *additional bit* to each node, to mark the *orientation*, i.e., if the node is a left or right child in the unfolded. Using this information, it can be detected in constant time if a node is an upper or lower node: for a left folded path, only upper nodes have left children, which are also on the left path, i.e., have the same

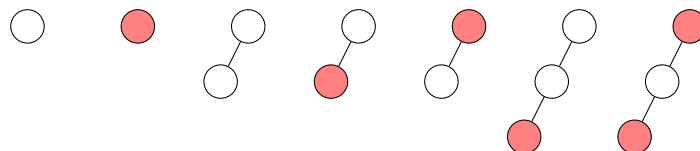


Figure 10: Possible colorings of the last lower chunk on an unfolded left path.

orientation, and symmetrically for the right folded path. If a node defines a path by itself, it can similarly be detected in constant time. Using the information left/right fold and upper/lower, the local lower chunks can be detected locally in constant time.

4.1 Properties of folded trees Having defined the folded representation of an unfolded tree, we summarize the properties of the folded representation and how it supports navigation in the unfolded tree:

- The inorder of the nodes of the unfolded and folded trees are identical,
- left and right paths in the unfolded tree become paths in the folded tree with identical topmost nodes,
- the topmost and bottommost nodes on a left (right) path in the unfolded tree are adjacent in the folded tree, i.e., given the topmost node in a left (right) path, the minimum (maximum) node in the subtree rooted at the node in the unfolded tree can be found in constant time,
- given a node in the folded tree, the left child, right child, and parent of the node in the unfolded tree can be found in constant time, and
- the folded tree is unique for a given unfolded tree,
- the height of the folded tree is logarithmic.

To see that the folded trees preserve inorder, observe the layout of the folded path in Figure 9 (right) satisfies inorder, and apply the argument recursively. By definition a left/right path becomes a folded path with the same topmost node and the topmost and bottommost nodes on left and right paths are adjacent.

Let v be a node in the unfolded tree, which is the left child of some node u , where u is either part of a left or right path. We will argue that the path between u and v in the folded tree contains at most two additional nodes, i.e., we can navigate from v to u and from u to v in constant time. If u is on a left path, then v is on the same left path, and will in the folded path be separated by at most two nodes (two upper nodes are separated by a lower chunk containing one or two nodes; two lower nodes can be adjacent or are separated by one upper node; and the adjacent upper node and lower node by the at most two other nodes in the last lower chunk). If u is on a right path, then v is the root of the left subtree of u and the topmost node on a left path, and will in the folded right path be separated by at most two nodes (a lower and an upper node; see the distance between x_i and the root of T_i in Figure 9 (right)). Symmetrically, this applies to right children.

The folding definition is deterministic. The root of the folded tree is the same node as the root of the unfolded tree. Given a node in the folded tree, the left and right children in the unfolded tree can be found. Therefore there exists a deterministic unfolding algorithm of a folded tree. The folded tree is therefore unique for a given unfolded tree.

To bound the height of a folded tree, let v be a deepest node in the folded tree. Since the root in the folded and unfolded trees are identical, the above argument implies that if the simple path in the unfolded tree from v to the root contains k nodes, there exists a path (not necessarily simple) from v to the root containing at most $k + 2(k - 1) = 3k - 2$ nodes. Therefore, the simple path in the folded tree from v to the root contains at most $3k - 2$ nodes. Since unfolded trees have logarithmic depth (Section 2), folded trees also have logarithmic depth.

4.2 Finding the predecessor and successor In this section we consider how to find the predecessor and successor of a node in a folded tree in constant time, even if they do not exist in the corresponding subtree in the unfolded tree, but as an ancestor. We only describe how to find the successor of a node. Finding the predecessor is symmetric.

Denote the starting node as v . If the right subtree of v in the unfolded tree is non-empty, then the successor is the minimum value in this subtree. Denote the right child of v as r , and the left subtree of r as T_r . Note that r must exist, as the right subtree of v is non-empty, and r can be found in constant time in the folded tree (Section 4.1). If T_r is empty, then the successor of v is r . Otherwise the successor is the minimum value in T_r . Independent if v is on a left or right folded path, T_r must be left folded, as illustrated on Figure 11. As T_r is left folded, the minimum in T_r can be accessed in constant time from v (Section 4.1). If the right subtree of v is empty, then the successor of v must be an ancestor u of v , where v is the maximum in the left subtree of u . Note that v is the predecessor of u . By symmetric argument from above for finding the predecessor in a non-empty left subtree, we from u can access v in constant time. As the pointers of the tree can be traversed in both directions, we from v can access its successor u in constant time. This concludes that the successor of any node can be found in constant time.

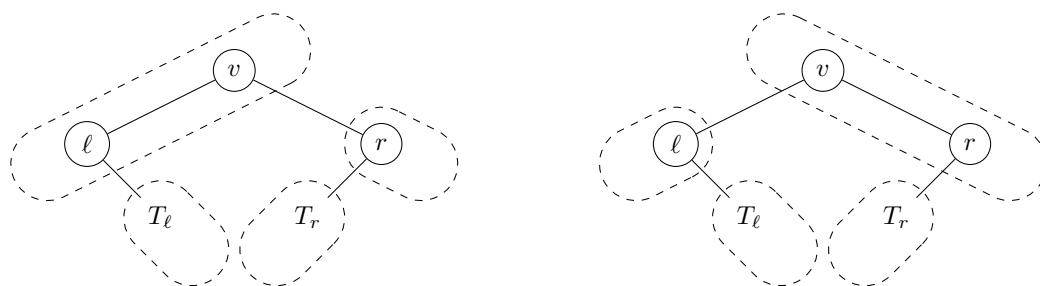


Figure 11: Folding orientations covering node v in an unfolded tree. The dashed areas show if the underlying path is left or right folded, with (left) the path containing v being left folded and (right) right folded. The subtrees T_ℓ and T_r may be empty.

4.3 Translating rotations from unfolded red-black trees An unfolded tree is maintained balanced by performing rotations. When a rotation occurs while rebalancing the unfolded tree, the folded tree must be updated accordingly. How to do so depends on the orientation of the paths in the unfolded tree. Figure 12 shows the four cases how rotations are performed in unfolded trees and how the orientation of the paths may be.

Consider the right rotation on Figure 12 (top). Define the *main path* to be the path containing the node rotated, in this case v , and the *height* of the rotation to be the height of the node rotated in the unfolded tree. In this case the main path loses node v . If v is an upper node or a node in the last lower chunk, then the path needs to be rebuilt, which takes time proportional to the height of the rotation, as the upper nodes are in the top half of the path. If v is a red lower node not in the last lower chunk, then the number of lower chunks is unaltered, and updating the main path takes constant time. The right path starting at T_4 gains a new upper node, the right path starting at w loses an upper node and the left path starting at T_2 gains an upper node, which all need to be rebuilt in time proportional to the height of the rotation. All other paths are unaltered.

Symmetric arguments exist for the other three cases of rotations. The total time spent to perform a rotation is proportional to the height of the rotation in the unfolded tree, provided that the number of lower chunks is unaltered in the case when the node rotated is a lower node.

4.4 Insertions in folded red-black trees In this section we describe how to perform INSERTSUCC on a folded tree in amortized constant time. Performing INSERTPRED is symmetric.

For an INSERTSUCC(v, e) operation, SUCC(v) can be found in $\mathcal{O}(1)$ time (Section 4.2). Remaining is to show that the time spent maintaining the folded tree to accommodate the new node in the folded tree is proportional to the time spent on rebalancing the unfolded tree. The rebalancing steps on the folded tree is identical to the rebalancing on unfolded trees (Section 2), utilizing that the local nodes needed in the rebalancing can be found in constant time (Section 4.1). First the successor leaf of v is located to create the new node containing e . The leaf is $v.r$ if $v.r = \text{NIL}$ and otherwise it is SUCC(v). ℓ . Initially the new node is colored red. If it is placed in an empty path, it becomes the root of the path. Otherwise, it is inserted into the top of the topmost lower chunk in the folded tree, i.e., the location of the lowest node on the unfolded path in the folded tree (see node x_1 on

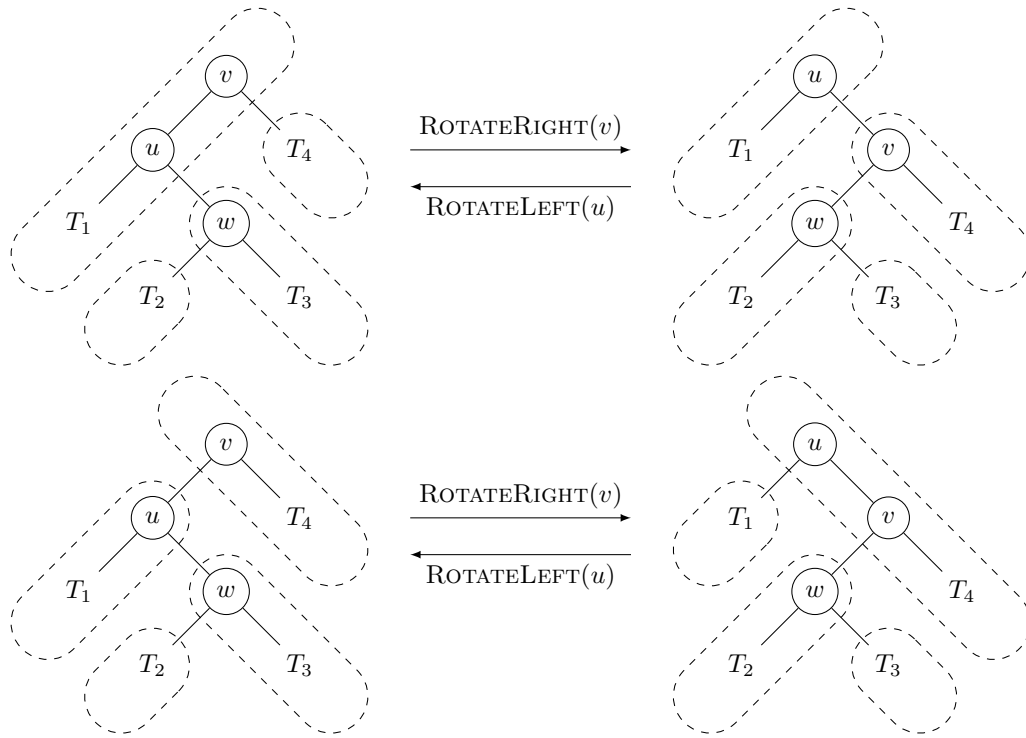


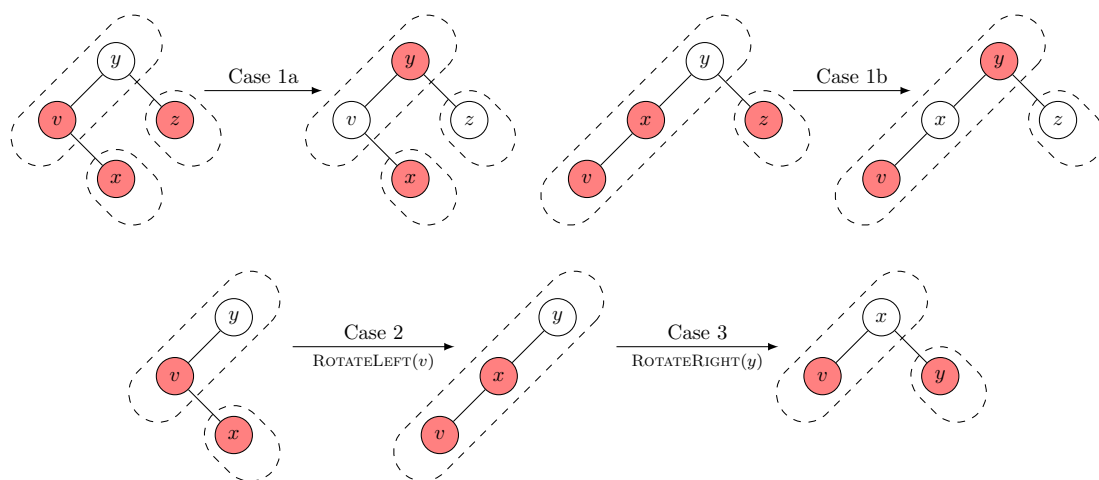
Figure 12: Rotations in an unfolded binary tree. The dashed areas show if the underlying path is left or right folded, with (top) the main path being left folded and (bottom) right folded.

Figure 9). The folded tree may during rebalancing contain a lower chunk with three nodes, with one black node at the bottom and two red nodes above it. It is an invariant during the rebalancing that each non-last lower chunk, contains exactly one black node and the red nodes below it from the unfolded tree. Since during insertions, at most one location in the unfolded tree contains two consecutive red nodes, all non-last lower chunks contain 1–3 nodes with exactly one black node, with at most one chunk being of size three. Further, in each non-last lower chunk the black node is at the bottom. Rebalancing proceeds by first performing a phase of recoloring nodes, and then terminating either at the root of the tree or by performing at most two rotations with associated recoloring of the rotated nodes. In Figure 6 these cases are shown for the unfolded tree, with Case 1a and 1b being the recoloring cases, and Case 2 and 3 being the terminating cases, which perform rotations. For the possible orientations of the paths, see Figure 13.

During the recoloring phase, three nodes are recolored in each iteration, and the rebalancing continues two nodes higher up in the unfolded tree. If an upper node is recolored, the tree remains correctly folded, as upper nodes may be of any color. If a lower node in a non-last lower chunk is recolored red, and it is the bottom node in the chunk in the folded tree, it is moved to the top of the lower chunk below in the folded tree, to ensure that the bottom node in the non-last lower chunk is black, which according to the recoloring cases it must, as the now bottom node has been recolored black. If this moves the node to the last lower chunk, it is further checked if the last lower chunk can split to introduce a new extra lower chunk and upper node. When a recoloring appears in the last lower chunk, this may similarly allow for splitting the last lower chunk. Both checks on the last lower chunk can be done similarly to how the folded tree was initially constructed. Note that during the recoloring, it is not possible that two nodes in the same lower chunk become black, and each lower chunk always contains exactly one black node, which therefore lead to only checks on nodes colored red in lower chunks being needed. Each of these operations correspond to a single recoloring step in the unfolded tree, and as each operation can be executed in constant time, the total time for the recoloring phase on the folded tree is asymptotically equal to the time for recoloring the unfolded tree.

If the recoloring phase does not terminate at the root of the tree, then at most two rotations are performed,

Left folded



Right folded

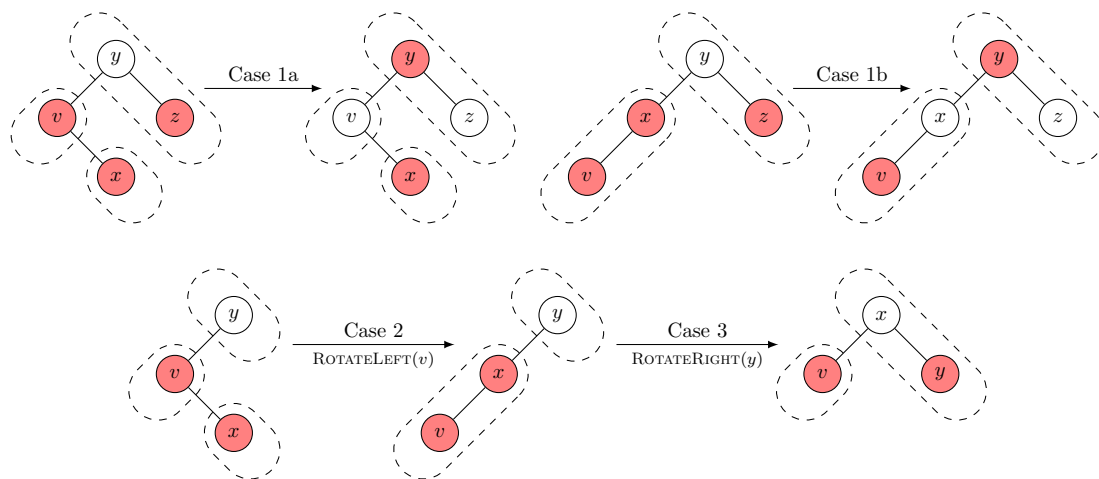


Figure 13: Insertion rebalancing cases on unfolded trees, when the two adjacent red nodes are on the left, with the underlying possible folding of the paths shown with dashed areas. Omitted subtrees have a black root or are empty. Cases 1a and 1b perform recoloring and continue from node y , while Cases 2 and 3 perform rotations in addition to recoloring, before terminating.

along with constant recoloring. These rotations are performed on the unfolded tree, and are therefore to be performed as in described in Section 4.3 on the folded tree. The rotations take time proportional to the height of the rotated node in the unfolded tree, assuming that the number of lower chunks in the main path remains the same if the rotated node is a lower node, without counting updates to the number of lower chunks from splitting or joining the last lower chunk. In each case of rotations performed in Case 2 and 3, the assumption that the number of lower chunks is unaltered is satisfied, utilizing in Case 3 that nodes are recolored, and therefore the rotations take time proportional to the height of the rotated node. As the height increases by a constant for each recoloring iteration, then the total time does not asymptotically exceed the time spent on the recoloring phase.

It therefore holds that the folded tree can be maintained under the rebalancing following an insertion, with the same asymptotic running time as the unfolded tree. The INSERTSUCC operation on an unfolded tree takes amortized constant time for the rebalancing plus the time for a SUCC operation (Section 2). The SUCC operation on folded trees take $\mathcal{O}(1)$ time, resulting in the INSERTSUCC and symmetrically INSERTPRED operations running in amortized constant time on the folded tree.

4.5 Deletions in folded red-black trees In this section we show how to perform the DELETE operation on a folded tree in amortized constant time. Similar to insertions, the procedure uses the same rebalancing rules as unfolded trees, and it must be shown that the time spent on maintaining the folded tree to remove the desired node is proportional to the time spent on rebalancing the unfolded tree.

When performing a deletion on an unfolded tree, if the node to be removed has two children, it is swapped with its successor, including swapping their colors, and then remove the node. This guarantees that the deleted node has at most one child (if it has a child, it must be a single red right child), and let the rebalancing operation proceed from this node. In the folded tree, the successor can be found in $\mathcal{O}(1)$ time (Section 4.2). If the removed node is red, then the unfolded tree remains a valid red-black tree. In the folded tree, this node can be removed in $\mathcal{O}(1)$ time, as it must either be a red lower node, which does not decrease the number of lower chunks, or it is an upper node, on a path consisting of only that node. Otherwise, the removed node is black and the number of black nodes on the path in the unfolded tree decreases, which breaks the black height invariant. A rebalancing procedure proceeds from the child replacing the removed node (possibly a NIL pointer).

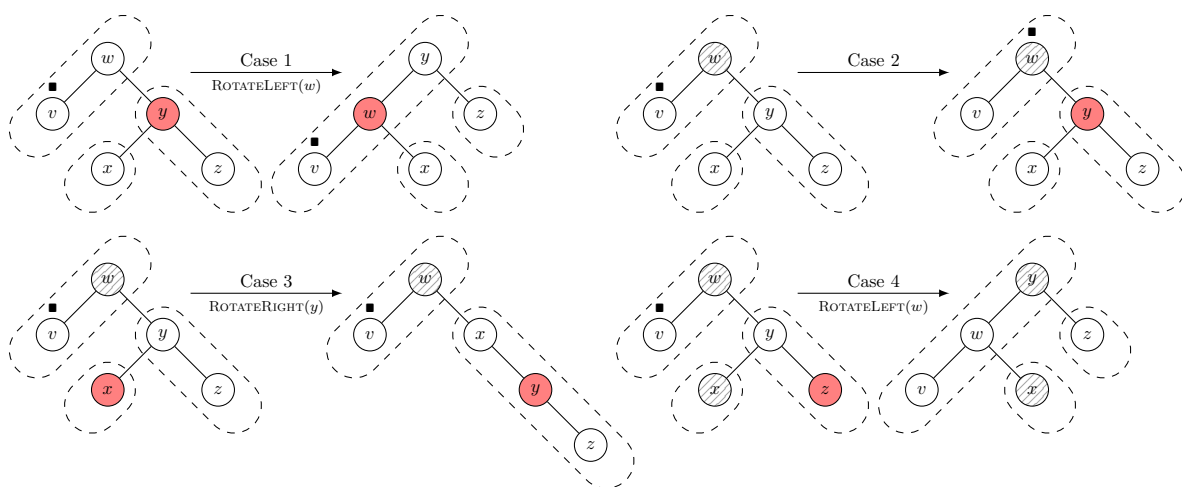
In the folded tree, if the double black node is NIL, it remains a leaf, and therefore does not give rise to any complications. However, in the folded tree this location may exist a non-leaf, and therefore the situation must be handled with care, e.g., by introducing a temporary black dummy node.

In the folded tree, if the removed black node is a lower node, then this removes a lower chunk. Therefore the node marked double black must count as the black node in an additional lower chunk. To handle this, during the rebalancing, if the node marked double black is a lower node, and there exists a lower chunk below it in the unfolded tree, then this non-last lower chunk is *broken*, meaning that it contains no nodes, or only a single red node. During rebalancing, this broken lower chunk must be fixed, when progressing up the unfolded tree, such that the node marked double black still counts as the needed black node in the neighboring lower chunk.

The cases of the rebalancing on unfolded trees can be seen in Figure 7, where the node the rebalancing is executed from is node v which is marked double black. For the possible orientations of the paths, see Figure 14. The procedure on unfolded trees runs in amortized $\mathcal{O}(1)$ time (Section 2). Only in Case 2 the rebalancing proceeds upwards by a single node in the unfolded tree. In all other cases, a constant number of rotations and recoloring is performed, before terminating. At the point of the rotations, the time spent on rebalancing is equal to the height of the double black node and therefore also asymptotically equal to the height of the rotations. Note that in all cases, the node marked double black can be assumed to be black, since if the marked node is red, it may freely be colored black, thereby fixing the black height invariant, and the procedure terminates. In the folded tree, there may be a broken lower chunk, which in this case of termination, by recoloring the red node marked double black into a black node, introduces a new black node without removing a lower chunk, and the broken chunk can be fixed in constant time.

In Case 2, the sibling y is colored red. If y is a lower node, then this breaks the lower chunk of y . However, in this case v must be the root of an unfolded path, and therefore it is an upper node in the folded path, and there is therefore no broken lower chunk below v , and as the procedure proceeds from node w , then y is exactly the allowed broken lower chunk below w . If v is the last node in the last lower chunk, then this chunk must be fixed when moving to the upper node w on the same path. This can be fixed by merging the last lower chunk with the neighboring upper node and lower chunk in the unfolded tree to create a new last lower chunk, which

Left folded



Right folded

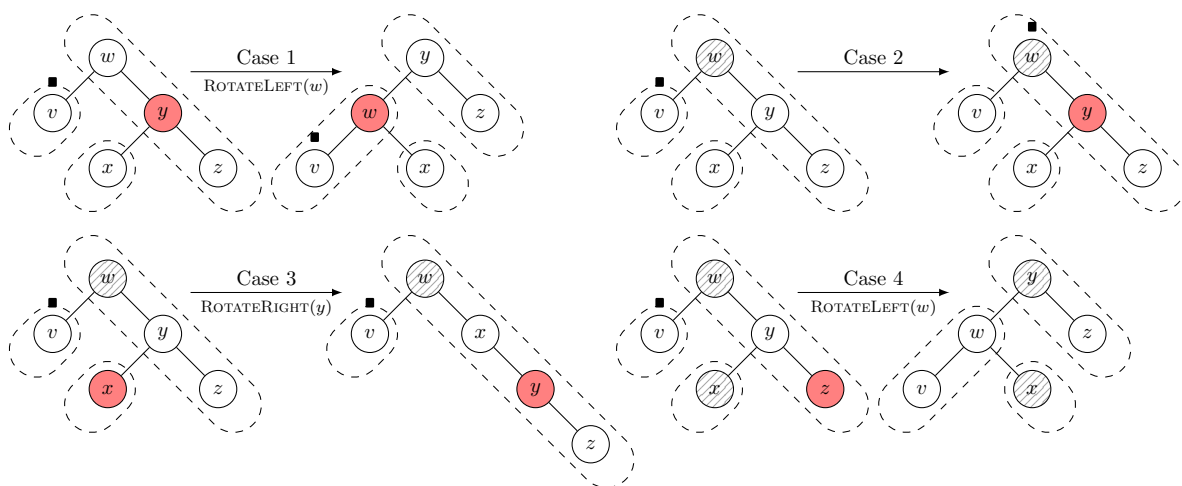


Figure 14: Deletion rebalancing cases on unfolded trees, when the double black node (marked $\blacksquare$) is on the left, with the underlying possible folding of the paths shown with dashed areas. The dashed nodes denote a node which color can be either red or black. Omitted subtrees can be empty or have a black or red root. The node v can possibly be an empty subtree. Case 2 performs recoloring and continues from node w , while Cases 1, 3, and 4 perform rotations in addition to recoloring. In Case 4, the color of node y after rebalancing is the color of node w before rebalancing.

then potentially splits. The broken lower chunk then ceases to exist. The total time for Case 2 on the folded tree is $\mathcal{O}(1)$.

In Case 1, a rotation is performed around w . According to Section 4.3, this rotation takes time proportional to the height of node w , if w is either an upper node, or if the number of lower chunks in the path of w is unaltered. If the path around w is right folded, then removing a red lower node does not alter the number of lower chunks. If the path is left folded, then a red node is added to the path, which does not alter the number of lower chunks. Both cases use that nodes are recolored alongside the rotation. The potentially broken lower chunk below v remains. The rebalancing then continues to Case 2, 3, or 4. If it continues in Case 2, then the procedure terminates afterwards, as w then is red.

In Case 3, a rotation is performed around y , and by a similar argument to above, this rotation runs in time proportional to the height of y , and the potentially broken lower chunk below v is preserved, with no new broken lower chunks being created. This runs in time proportional to the height of y . The rebalancing then continues to Case 4.

In Case 4, a rotation is performed around w . If the path of w is right folded, then the number of lower chunks is preserved. If the path is left folded and w is a lower node, then a new lower chunk consisting of a single black node is created. However, there must then be a broken lower chunk below v , which then can be filled, to allow the number of lower chunks to be preserved. It is used that nodes are recolored. The time is therefore proportional to the height of node w . The rebalancing then terminates.

In Case 2, the time for the rebalancing on the folded tree runs in $\mathcal{O}(1)$ time, which does not increase the asymptotic time, from the unfolded tree. In Case 1, 3, and 4, the time for the rebalancing on the folded tree runs in time proportional to the height of the rotation. Each of these operations can only be run once. As the height of the rotations increases by a constant for each iteration of Case 2, the folded tree can be maintained under rebalancing with the same asymptotic running time as the unfolded tree. The DELETE operation on an unfolded tree takes amortized $\mathcal{O}(1)$ time for the rebalancing plus the time for a SUCC operation (Section 2). The SUCC operation on folded trees take $\mathcal{O}(1)$ time, resulting in the DELETE operation running in amortized constant time on the folded tree.

5 Metanodes

In this section we describe how a BST T_{bits} , where each node stores a constant number of m bits, can be represented by a pure BST T with no information at the nodes and without increasing the running times by more than a constant factor. Each node in T_{bits} is represented by $\mathcal{O}(1)$ nodes in T . The basic idea is to partition a sorted sequence of n values into groups of $4 + 2m$ to $7 + 4m$ consecutive values, where each group is stored as a small pure BST, denoted a *metanode*. The metanodes become the binary nodes of a T_{bits} tree, where the tree structure of a metanode encodes m bits. For the special case where the tree contains less than $4 + 2m$ nodes, we have no metanode and the tree is just a single left path.

A metanode consists of $3 + 2m + s$ pure binary nodes, $1 \leq s \leq 4 + 2m$, each node storing a value, and in left-to-right inorder are the nodes

$$x, y, w_1, \dots, w_s, b_1^1, b_1^2, \dots, b_m^1, b_m^2, z.$$

The root of the metanode is y with left child x and right child z storing the smallest and largest value in the metanode, respectively. If all m bits $b_1, \dots, b_m$ are equal to zero, the remaining nodes are stored as a left path in the left subtree of z . The nodes $w_1, \dots, w_s$ are buffer nodes, and b_i^1 and b_i^2 encode bit b_i . If bit b_i is set to 1, node b_i^2 is rotated right to become the right child of b_i^1 . Flipping bit b_i from 0 to 1 is done by ROTATERIGHT(b_i^2), and reversely flipping bit b_i from 1 to 0 is done by ROTATELEFT(b_i^1). Finally the left child of x is either NIL or the root L of another metanode (storing values smaller than or equal to the value at x), and similarly the right child R of z is possibly a metanode (with values larger than or equal to the value at z). The “value” of a metanode is the *value interval* between the values in x and z . Figure 15 shows a metanode encoding two bits.

The resulting pure BST T will be composed of multiple metanodes, and the search tree operations should be performed on the tree T . It is crucial that given a node v , to determine if v is the root of a metanode, since then we can decode the bits and access the left and right metanode children L and R as $v.l.l$ and $v.r.r$, respectively. The following lemma states that a metanode root can be uniquely determined by computing (5.1).

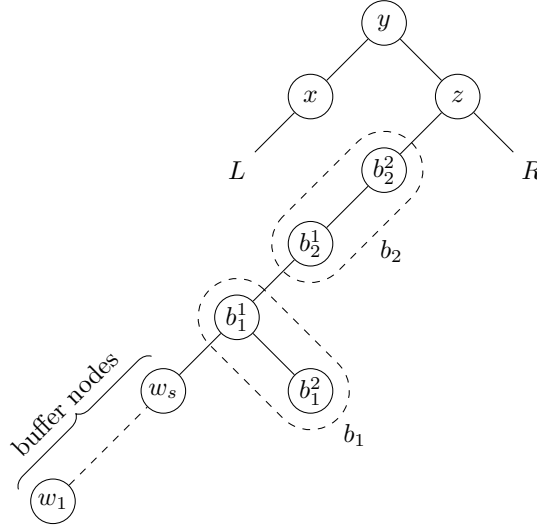


Figure 15: A metanode with root y , encoding two bits $b_1 = 1$ and $b_2 = 0$, and L and R are the roots of two metanodes (possibly NIL).

LEMMA 5.1. A node v is the root of a metanode if and only if

$$(5.1) \quad v.l \neq \text{NIL} \wedge v.r \neq \text{NIL} \wedge v.r.l \neq \text{NIL} \wedge (v.l.l = \text{NIL} \vee (v.l.l.r \neq \text{NIL} \wedge v.l.l.r.l \neq \text{NIL}))$$

Proof. If v is the root y of a metanode, it by definition has two children $v.l = x$ and $v.r = z$, and its right child $v.r$ has a left child $v.r.l$ that is either b_m^1 or b_m^2 . Either the metanode with root v has no left child metanode, i.e., $v.l.l = L = \text{NIL}$, or the left child metanode exists and $v.l.l.r.l$ exists and is a node in the encoding of the b_m bit of the metanode with root L . This completes the “ $\Rightarrow$ ” part of the proof. To prove “ $\Leftarrow$ ”, observe that the only nodes in a metanode with both a left and right child are the root y and possibly z and all b_i^1 , but only y and z can have a right child with a left child, i.e., only y and z can satisfy $v.l \neq \text{NIL} \wedge v.r \neq \text{NIL} \wedge v.r.l \neq \text{NIL}$. Finally, if $v.l.l = \text{NIL}$, then $v \neq z$, since $z.l.l$ is a b_i^j or w_s ($z.l.l = w_s$ only if $m = 1$ and bit $b_1 = 1$), i.e., $v = y$. Otherwise, $v.l.l \neq \text{NIL}$, and by (5.1) we must have $v.l.l.r \neq \text{NIL}$ and $v.l.l.r.l \neq \text{NIL}$. If $z.l.l.r \neq \text{NIL}$ then $z.l.l = b_{m-1}^1$, $z.l.l.r = b_{m-1}^2$, and $z.l.l.r.l = \text{NIL}$. We have $v \neq z$, and the only possible value for v is the metanode root y . $\square$

Note that all nodes with a right child (y and possibly z and all b_i^1) also have a left child. This fact is used in Section 6 to represent each node in a pure BST using two pointers only.

Given a T_{bits} binary finger search tree, a pure binary finger search tree is created as follows. The sorted list of n values is partitioned left-to-right in a sequence of metanodes. The metanodes become the nodes of a T_{bits} tree. To perform $\text{FINDERSEARCH}(v, e)$, where v is a node in a metanode, we find the root of the metanode by repeatedly moving up along the path until (5.1) is true or we reach the root of the tree. To verify that the tree actually contains sufficiently many nodes to encode the first metanode, we check if the tree contains at least $4 + 2m$ nodes, by traversing the tree until $4 + 2m$ nodes have been identified. If the tree is smaller than a single metanode, the answer can be computed by comparing e with the values at all nodes. Otherwise, the search is performed on T_{bits} using $v.l.l$, $v.r.r$ and $v.p.p$ to navigate to the left child, right child and parent, respectively. To compare e with a metanode, we compare e with the value e_x and e_z at the nodes x and z of the metanode. If $e_x \leq e < e_z$, the answer to the search is inside the metanode and we return the result of a top-down search in the metanode. Otherwise, $e < e_x$ or $e_z \leq e$ and we answer that e is less than or greater than or equal to the value of the metanode, respectively, and we continue the search in T_{bits} .

To perform a finger update at a node v in a metanode, we find the root of the metanode and identify left-to-right the remaining nodes of the metanode in sorted order. Insertions create a new node next to v in the list, whereas deletions remove v . If the resulting list contains between $4 + 2m$ and $7 + 4m$ nodes, the nodes are rearranged to represent a metanode representing the same bits, possibly updating pointers to changed x , y and z ,

i.e., the T_{bits} tree is unchanged. If the update is an insertion that causes the metanode to overflow, i.e., to contain $8 + 4m$ nodes, the metanode is split into two metanodes with $4 + 2m$ values each, and we perform INSERTSUCC on T_{bits} to insert the new metanode. If the update is a deletion that causes the metanode to underflow, i.e., contain $\leq 3 + 2m$ nodes, we try to transfer a node from the successor or predecessor metanode, provided one of them exists and would not underflow. If the successor or predecessor metanode would underflow (i.e., before the deletion stores $4 + 2m$ nodes), we remove the successor or predecessor metanode from T_{bits} by performing DELETE, and fusion the two metanodes to one metanode with $4m + 7$ nodes. Overall a finger search or update takes worst-case $\mathcal{O}(1)$ time plus the time for searching or updating T_{bits} with a constant factor overhead for decoding metanodes.

6 Representation of pure binary search trees

The canonical representation of a pure BST is to store each node as a record containing a value and three pointers ℓ , r and p to the left child, right child and parent of the node, respectively. See Figure 16 (left). The pure BSTs resulting from the metanode construction in Section 5 satisfy that *all nodes with at least one child have a left child*. Therefore, the nodes can be represented only using two pointers per node: a pointer ℓ to the left child and a pointer r to either the right sibling, if it exists, or the parent of the node. For the root, the r pointer is NIL. See Figure 16 (right).

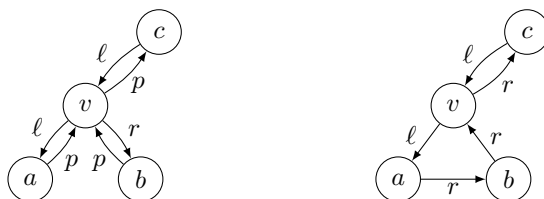


Figure 16: Representation of a node v in a pure BST with parent c and two children a and b and no right sibling: (left) left-right-parent representation using three pointers; (right) left-right sibling/parent representation using two pointers.

In the two pointer representation, the right child and parent of a node v can be determined as follows: The right child of v is $v.\ell.r$, if $v.\ell \neq \text{NIL}$ and $v.\ell.r \neq v$. Otherwise, v has no right child. If $v.r = \text{NIL}$, then v is the root and has no parent (and no sibling). The parent of v is $v.r.r$, if $v.r.r \neq \text{NIL}$ and $v.r.r.\ell = v$ (v is the left sibling of $v.r$). Otherwise, $v.r$ is the parent of v .

The leftmost child plus right sibling representation is traditionally used to represent multiway trees—essentially storing the children of a node as a singly linked list. Iverson [28] denoted this representation a filial-heir chain; Sussenguth [42] denoted it a doubly-chained tree; Knuth [31, page 477] denoted it the binary tree representation; and Fredman et al. [22] denote it the child, sibling representation. Fredman et al. [22] in their paper on pairing heaps discuss a representation of binary trees only using two pointers, storing a pointer to the leftmost child and a combined right sibling/parent pointer. They store a bit per node to distinguish if a node with only one child has a left or right child. We avoid this by our invariant that all nodes with at least one child have a left child. Note that if the tree is a single left path then the representation is exactly a doubly linked list.

7 Pure binary finger search trees

We now combine the results of Sections 2–6 to achieve our main result (Theorem 1.1): A pure BST supporting finger searches in $\mathcal{O}(\lg d)$ time and finger updates in amortized constant time. The tree can be represented by n records, each storing a value and two pointers.

The pure BST is simply a folded-red black tree (Section 4) represented using metanodes (Section 5). This is possible since the nodes in a folded red-black tree only store left child, right child, and parent pointers and two bits (color and orientation). It can be represented by records storing a value and two pointers (Section 6), since each node in a metanode cannot have a right child without a left child.

Since a folded red-black tree allows navigation like in a red-black tree plus predecessor and successor queries in worst-case constant time (Section 4), they allow finger searches in $\mathcal{O}(\lg d)$ time (Section 3), and the metanode encoding (Section 5) only causes a constant factor overhead. By the same argument (combining Sections 2, 4, 5),

17.8. WORST-CASE LOWER BOUND FOR PURE BINARY SEARCH TREES

finger updates will take amortized constant time, since red-black trees support amortized constant time finger updates, provided `SUCC` can be supported in constant time (Section 2).

In the preceding sections we did not discuss the temporary working space during the operations. But searches and updates in a red-black tree only requires constant temporary space to remember the current state during a search and rebalancing. Since our pure binary trees support both navigation to the parent and children of a node, we can with constant working space move between adjacent nodes in the pure binary tree, and therefore also in the folded and unfolded trees represented by the pure binary tree, using additional constant space to decode the local cases in the encoded metanodes and folded trees. In total $\mathcal{O}(1)$ temporary working space is sufficient for the operations listed in Section 1.1.

8 Worst-case lower bound for pure binary search trees

The finger update times achieved in Section 7 are inherently *amortized* constant. In this section we argue that this is a crucial requirement for *pure* BSTs, i.e., BSTs without any additional pointers than pointers to the parent and children of a node (the following arguments even apply if nodes can store any additional balance information). We show that *worst-case* constant finger updates in a pure BST can result in trees with asymptotic linear height, causing searches to take worst-case linear time.

LEMMA 8.1. *Assume that an `INSERTPRED` or `INSERTSUCC` operation can at most access (and possibly modify) a connected component of c nodes in a pure BST containing the node pointed to by the finger. Then there exists a sequence of n insertions causing the tree to have height at least $\lfloor n/2^c \rfloor$.*

Proof. We first make an observation, that we will use repeatedly: Consider a binary tree with at least 2^c nodes. In such a tree there must exist at least one node v with depth $c + 1$ (the root having depth one), since the first c levels of a binary tree can at most contain $\sum_{i=0}^{c-1} 2^i = 2^c - 1$ nodes. By performing `INSERTPRED` or `INSERTSUCC` at v , the changes caused by the insertion cannot reach the root of the tree, and the root remains unchanged.

After the first 2^c insertions, we consider the root v_1 fixed. We then repeatedly insert 2^c additional nodes as predecessors or successors to a deepest node (in the subtree rooted at v_1). None of these updates can reach v_1 . The subtree rooted at v_1 now contains 2^{c+1} nodes. The left or right subtree of v_1 must contain at least 2^c nodes (otherwise the subtree rooted at v_1 would contain at most $1 + 2 \cdot (2^c - 1) = 2^{c+1} - 1$ nodes). We let v_2 be a child of v_1 with a largest subtree. We now repeat inserting 2^c nodes in the subtree rooted at v_2 , and let v_3 be a child of v_2 with largest resulting subtree, etc. After $k \cdot 2^c$ insertions, we will have constructed a path $v_1, v_2, \dots, v_k$ starting at the root v_1 , i.e., the tree has height at least k . It follows that the tree after n insertions has height at least $\lfloor n/2^c \rfloor$. $\square$

From Lemma 8.1 we have the following two corollaries.

COROLLARY 8.1. *If `INSERTPRED` or `INSERTSUCC` takes worst-case $\mathcal{O}(c)$ time in a pure BST, then searches take worst-case $n/2^{\mathcal{O}(c)}$ time.*

COROLLARY 8.2. *Pure BSTs with height $\mathcal{O}(\lg n)$ can only be achieved if `INSERTSUCC` and `INSERTPRED` take worst-case $\Omega(\lg n)$ time.*

Proof. If the height of a pure BST is at most $\alpha \lg n$ after n `INSERTPRED` insertions, for some constant α , and the maximum number of nodes accessed by `INSERTPRED` is c , then by Lemma 8.1 the height can be at least $\lfloor n/2^c \rfloor$, i.e., we have the constraint $\lfloor n/2^c \rfloor \leq \alpha \lg n$, implying $c \geq \lg \frac{n}{1+\alpha \lg n} = \Omega(\lg n)$. Similarly for `INSERTSUCC`. $\square$

Interestingly, in the comparison model, if arbitrary insertions and deletions (not given by a finger) are allowed to make $\mathcal{O}(c)$ comparisons, queries are known to require $n/2^{\mathcal{O}(c)}$ comparisons in both the worst-case and amortized sense [9, 11, 29]. For finger updates, like studied in this paper, total order is provided for free to the algorithm. Only the $\Omega(\lg n)$ lower bound for binary searches applies together with the lower bound of Bentley and Yao for finger searches [8].

It is important to note that the above lower bounds do not rule out general pointer based finger search structures with worst-case constant time finger updates. Such a data structure was described by Brodal et al. [12]. It remains an open problem if a pointer based data structure exists, supporting worst-case $\mathcal{O}(\lg d)$ time finger searches and $\mathcal{O}(1)$ time finger updates, where each node only stores a value and two pointers. In the next section we describe a general transformation that potentially could lead to this result. In particular, it is an open problem if the transformation can be applied to the construction in [12].

9 Linked list metanodes

In the previous sections the underlying assumption is that the final finger search data structure should be a pure BST, where each node can be represented using two or three pointers. In this section we briefly deviate from the pure binary tree assumption and present alternative data structures, where each record only stores a single value and two pointers. The main goal is to show that the lower bounds on the worst-case update time in Section 8 do not apply under this relaxation.

Assume the goal is to store a sorted list of n values. The basic idea is to maintain the values in a sorted singly linked list, where each node stores a value and has a *next* pointer to the next node in the list, where the *next* pointer of the last node is NIL. Finally, each node has a secondary pointer *info*. The linked list is partitioned into *metanodes* consisting of $\Theta(1)$ consecutive nodes. Each metanode can represent a record in a pointer based structure storing t pointers to other records and m bits. The nodes of a metanode left-to-right are: a head marker, with *info* pointing to the node itself; a node for each of the t pointers, with *info* pointing to the head of the corresponding metanode; a node for each of the m bits, with *info* being NIL if the bit is zero, or *info* equal to *next* otherwise, i.e., non-NIL; s buffer nodes, where $0 \leq s \leq 1 + t + m$, all with *info* equal to NIL; and a terminating tail node with *info* pointing to the head of the metanode. The “value” of a metanode is the *value interval* between the values in head and tail. Figure 17 shows a metanode representing a red node in a red-black tree with parent, left child, right child pointers and links to the predecessor and successor metanodes, and a single color bit set to one. Note that the successor pointer in this representation points to the node following tail, and therefore could be omitted by only introducing a constant overhead to find the successor of a metanode.

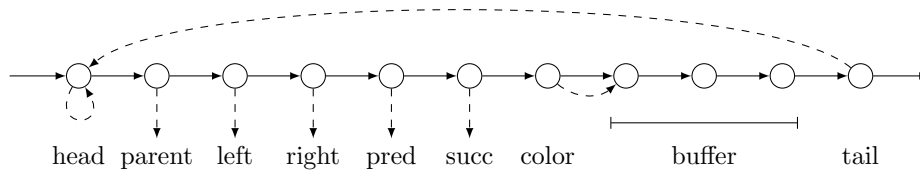


Figure 17: A linked-list metanode representing a node in a red-black tree with predecessor and successor links. Horizontal arrows are *next* pointers, and dashed arrows are *info* pointers.

Head nodes are uniquely determined by *info* pointing to the node itself. Tail nodes are uniquely determined by *next* being NIL or pointing to the head node of the next metanode in the linked list (with *info* pointing to itself). Given a finger to a node v in a metanode, we can find the head of the metanode by repeatedly following *next* pointers until we reach a tail node, where *info* points to the head.

We give two applications of the above metanode encodings. The first application is a simplified dynamic finger structure achieving time bounds matching those in Section 7. Consider a red-black tree with explicit successor and predecessor pointers. Such a tree supports amortized constant time finger updates (Section 2) and worst-case $\mathcal{O}(\lg d)$ finger searches (Section 3). Since the resulting data structure consists of records storing a value, five pointers (parent, left child, right child, predecessor and successor), and one bit (node color) we can apply the above metanode construction and achieve a finger search data structure where each node only stores a value and two pointers. To support $\text{FINGERSEARCH}(v, e)$, where v is a node inside a metanode, we first find the head of the metanode containing v in $\mathcal{O}(1)$ time. We then perform the search in the red-black tree of metanodes as described in (Section 3), with the following twist: When comparing e with a metanode we compare e with the values e_{head} and e_{tail} at head and tail. If $e_{\text{head}} \leq e < e_{\text{tail}}$, the answer to the finger search is inside the metanode and the query can be finalized by a scan through the metanode in $\mathcal{O}(1)$ time. Otherwise, $e < e_{\text{head}}$ or $e_{\text{tail}} \leq e$ and the search proceeds left or right in the red-black tree, respectively. To perform a finger insertion $\text{INSERTPRED}(v, e)$ or $\text{INSERTSUCC}(v, e)$ for a node v in a metanode, we add a new node v' storing the value e to the singly linked list before or after v , respectively, such that v' becomes a new node in the metanode containing v . The information in the *info* pointers of the metanode is updated to reflect the same metanode information (pointers to other metanodes and bits). Note that to insert e to the left of head we need to update the *next* pointer in a node in the predecessor metanode that can be accessed using the predecessor pointer. Also if head changes, then all pointers to this metanode should be updated, but since these are the parent and the children of the metanode in the red-black tree, this can be done in $\mathcal{O}(1)$ time. Finally, if the buffer overflows, i.e., contains $2 + t + m$ buffer nodes,

the metanode is split to create a new metanode containing the last $2 + t + m$ nodes, by updating the *info* pointers, where each of the two metanodes have empty buffers. The new metanode is then inserted as the successor of the old metanode in the red-black tree in amortized $\mathcal{O}(1)$ time (Section 2). A $\text{DELETE}(v)$ operation similarly deletes a node from a metanode, and requires similar updates. If a metanode underflows, i.e., has $1 + t + m$ nodes, the metanode either gets a node from the successor or predecessor metanode or fusions with either the predecessor or successor metanode with an empty buffer (like a deletion in a B-tree [6]) such that the resulting metanode has a buffer of size $1 + t + m$. This requires deleting a metanode from the red-black tree in amortized $\mathcal{O}(1)$ time (Section 2). It follows that finger updates can be performed in amortized constant time and finger searches in $\mathcal{O}(\lg d)$ time. The same result could also have been achieved by combining the metanode encoding idea with the level-linked (a, b) -trees of Huddleston and Mehlhorn [27].

Our second application is the existence of a search tree encoding supporting *worst-case* constant time insertions. Levcolopolous and Overmars [33] presented a BST supporting finger updates in worst-case constant time and searches in worst-case $\mathcal{O}(\lg n)$ time. We sketch below how this structure can be adapted such that we can apply the metanodes of this section, and support INSERTSUC in worst-case $\mathcal{O}(1)$ time and searches in $\mathcal{O}(\lg n)$ time, i.e., surpassing the lower bound for pure BSTs (Corollary 8.1). The structure in [33] is pointer based, but uses integers and records not storing values, two requirements falling outside the requirements for the metanode encoding. We next discuss how to circumvent these restrictions, such that we can apply the metanode encoding and achieve the same bounds for INSERTSUC and searches using only two pointers per node. Essentially, [33] partitions the sorted list of values into *bags* of $\mathcal{O}(\lg^2 n)$ values, and stores the bags as the nodes of a balanced BST, say a red-black tree. During a sequence of insertions, a bag of maximum size is split into two bags and the new bag is inserted into the search tree incrementally over the next $\mathcal{O}(\lg n)$ operations. This ensures that all bags will have size $\mathcal{O}(\lg^2 n)$ [33, Lemma 3.3]. Bags are represented as lists-of-lists, such that all lists have length $\mathcal{O}(\lg n)$. Bags of equal size are again stored in a linked list, and all these lists-of-bags are maintained in a linked list by increasing bag size. Finally, to achieve worst-case bounds bag splitting is performed incrementally while inserting into the bags, and incremental global rebuilding [38] is applied to handle changing values of $\lg n$. By letting a record storing a value contain multiple pointers for each of the involved linked lists, pointers to head of the lists etc., we can manage with one record for each value. To be able to store bag sizes, and compare bag sizes for equality, we can let the record for each value represent an integer $1, \dots, n$, and maintain these in increasing order. To store an integer we have a pointer to this list. Crucially, for this to work we disallow deletions, such that whenever a new value is inserted it can just be added as the next value $n + 1$ to this list. We omit further details, since the main goal of this section is just to highlight that the lower bounds in Section 8 only apply to pure BSTs.

We leave it as an open problem if the finger search data structure by Brodal et al. [12] supporting worst-case constant time finger updates and $\mathcal{O}(\lg d)$ finger searches can be encoded using linked-list metanodes.

10 Conclusion and open problems

We have presented optimal pure binary finger search trees. Whereas each of the ideas is conceptually simple, the resulting data structure has a quite large constant factor overhead. It would be interesting to find simpler pure BSTs with a smaller overhead.

In this paper we have only considered pointer based data structures. One could also consider the problem in the more general RAM model, e.g., allowing bit operations on words and aiming at more succinct representations. Recently, Bender et al. [7] considered a randomized scheme to encode BSTs in this model using *tiny pointers*. They showed how to encode dynamic balanced BST using only $\mathcal{O}(n)$ bits for storing the tree structure, while each tree modification takes $\mathcal{O}(\lg^* n)$ time. An open problem is what the best possible space and time bounds are for finger search trees in this model.

For the extreme case, where no additional space is allowed, Munro and Suwanda [37] introduced the notion of *implicit data structures*, where only an array of the values is stored, without any additional information. All additional information must be encoded in the permutation of the values. Munro and Suwanda presented an implicit dynamic dictionary with $\mathcal{O}(n^{1/3} \lg n)$ time bounds. Frederickson [21] improved the update time bounds to $\mathcal{O}(n^{\sqrt{2}/\lg n} \lg^{3/2} n)$ with $\mathcal{O}(\lg n)$ time searches. Munro [36] achieved $\mathcal{O}(\lg^2 n)$ time bounds for queries and updates by encoding AVL-trees, Franceschini et al. [20] achieved $\mathcal{O}(\lg^2 n / \lg \lg n)$ time bounds for queries and updates by encoding implicit B-trees. Finally, Franceschini and Grossi [19] achieved searches and updates in worst-

case optimal $\mathcal{O}(\lg n)$ time. Dictionaries supporting finger searches in this very restricted model were considered by Brodal, Nielsen and Truelsén [13], who proved trade-offs between the time for finger queries and the time to move a single finger.

Generally, in the RAM model, it is an interesting open problem how efficient dynamic finger search structures can be obtained with limited space in addition to the stored values, e.g., are there trade-offs between update time, query time and space usage? In particular, what are the best possible bounds for *stable* dictionaries, i.e., values are never moved once inserted? Stability appears crucial to be able to support an arbitrary number of fingers.

Acknowledgments The authors wish to thank the anonymous reviewers for detailed feedback on the presentation.

References

- [1] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)*, 146:263–266, 1962.
- [2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [3] Arne Andersson. Improving partial rebuilding by using simple balance criteria. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989. doi:10.1007/3-540-51542-9_33.
- [4] Arne Andersson, Rolf Fagerberg, and Kim S. Larsen. Balanced binary search trees. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 10. Chapman and Hall/CRC, 2004. doi:10.1201/9781420035179.CH10.
- [5] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):13, 2007. doi:10.1145/1236457.1236460.
- [6] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683.
- [7] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. Tiny pointers. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 477–508. SIAM, 2023. doi:10.1137/1.9781611977554.CH21.
- [8] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976. doi:10.1016/0020-0190(76)90071-5.
- [9] Allan Borodin, Leonidas J. Guibas, Nancy A. Lynch, and Andrew Chi-Chih Yao. Efficient searching using partial ordering. *Information Processing Letters*, 12(2):71–75, 1981. doi:10.1016/0020-0190(81)90005-3.
- [10] Gerth Stølting Brodal. Finger search trees. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004. doi:10.1201/9781420035179.CH11.
- [11] Gerth Stølting Brodal, Shiva Chaudhuri, and Jaikumar Radhakrishnan. The randomized complexity of maintaining the minimum. *Nordic Journal of Computing*, 3(4):337–351, 1996.
- [12] Gerth Stølting Brodal, George Lagogiannis, Christos Makris, Athanasios K. Tsakalidis, and Kostas Tsichlas. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences*, 67(2):381–418, 2003. doi:10.1016/S0022-0000(03)00013-8.
- [13] Gerth Stølting Brodal, Jesper Sindahl Nielsen, and Jakob Truelsén. Finger search in the implicit model. In Kun-Mao Chao, Tsan-sheng Hsu, and Der-Tsai Lee, editors, *Algorithms and Computation - 23rd International Symposium, ISAAC 2012, Taipei, Taiwan, December 19-21, 2012. Proceedings*, volume 7676 of *Lecture Notes in Computer Science*, pages 527–536. Springer, 2012. doi:10.1007/978-3-642-35261-4_55.
- [14] Mark R. Brown. A storage scheme for height-balanced trees. *Information Processing Letters*, 7(5):231–232, 1978. doi:10.1016/0020-0190(78)90005-4.
- [15] Mark R. Brown. Addendum to “a storage scheme for height-balanced trees”. *Information Processing Letters*, 8(3):154–156, 1979. doi:10.1016/0020-0190(79)90009-7.
- [16] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: the proof. *SIAM Journal of Computing*, 30(1):44–85, 2000. doi:10.1137/S009753979732699X.
- [17] Richard Cole, Bud Mishra, Jeanette P. Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. Part I: splay sorting $\log n$ -block sequences. *SIAM Journal of Computing*, 30(1):1–43, 2000. doi:10.1137/S0097539797326988.

- [18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. MIT Press, 2022.
- [19] Gianni Franceschini and Roberto Grossi. Optimal implicit dictionaries over unbounded universes. *Theory of Computing Systems*, 39(2):321–345, 2006. doi:10.1007/S00224-005-1167-9.
- [20] Gianni Franceschini, Roberto Grossi, J. Ian Munro, and Linda Pagli. Implicit B -trees: a new data structure for the dictionary problem. *Journal of Computer and System Sciences*, 68(4):788–807, 2004. doi:10.1016/J.JCSS.2003.11.003.
- [21] Greg N. Frederickson. Implicit data structures for the dictionary problem. *Journal of the ACM*, 30(1):80–94, 1983. doi:10.1145/322358.322364.
- [22] Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986. doi:10.1007/BF01840439.
- [23] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 165–174. ACM/SIAM, 1993. URL: <https://dl.acm.org/doi/10.5555/313559.313676>.
- [24] Ofek Gila, Michael T. Goodrich, and Robert E. Tarjan. Zip-zip trees: Making zip trees more balanced, biased, compact, or persistent. In Pat Morin and Subhash Suri, editors, *Algorithms and Data Structures - 18th International Symposium, WADS 2023, Montreal, QC, Canada, July 31 - August 2, 2023, Proceedings*, volume 14079 of *Lecture Notes in Computer Science*, pages 474–492. Springer, 2023. doi:10.1007/978-3-031-38906-1_31.
- [25] Leonidas J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 49–60. ACM, 1977. doi:10.1145/800105.803395.
- [26] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3.
- [27] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982. doi:10.1007/BF00288968.
- [28] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.
- [29] Haim Kaplan, Or Zamir, and Uri Zwick. The amortized cost of finding the minimum. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 757–768. SIAM, 2015. doi:10.1137/1.9781611973730.51.
- [30] Alexis C. Kaporis, Christos Makris, Spyros Sioutas, Athanasios K. Tsakalidis, Kostas Tsichlas, and Christos D. Zaroliagis. Improved bounds for finger search on a RAM. *Algorithmica*, 66(2):249–286, 2013. doi:10.1007/S00453-012-9636-4.
- [31] Donald Ervin Knuth. *The art of computer programming, Volume III: Sorting and Searching, 2nd Edition*. Addison-Wesley, 1998.
- [32] S. Rao Kosaraju. Localized search in sorted lists. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*, pages 62–69. ACM, 1981. doi:10.1145/800076.802458.
- [33] Christos Levkopoulos and Mark H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Informatica*, 26(3):269–277, 1988. doi:10.1007/BF00299635.
- [34] Conrado Martínez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998. doi:10.1145/274787.274812.
- [35] Kurt Mehlhorn and Athanasios K. Tsakalidis. An amortized analysis of insertions into AVL-trees. *SIAM Journal of Computing*, 15(1):22–33, 1986. doi:10.1137/0215002.
- [36] J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33(1):66–74, 1986. doi:10.1016/0022-0000(86)90043-7.
- [37] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21(2):236–250, 1980. doi:10.1016/0022-0000(80)90037-9.
- [38] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983. doi:10.1007/BFB0014927.
- [39] Raimund Seidel. Maintaining ideally distributed random search trees without extra space. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 134–142. Springer, 2009. doi:10.1007/978-3-642-03456-5_9.
- [40] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. doi:10.1007/BF01940876.
- [41] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.

- [42] Edward H. Sussenguth, Jr. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–279, 1963. doi:10.1145/366552.366600.
- [43] Robert Endre Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Inf. Process. Lett.*, 16(5):253–257, 1983. doi:10.1016/0020-0190(83)90099-6.
- [44] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. doi:10.1137/0606031.
- [45] Robert Endre Tarjan and Christopher J. Van Wyk. An $O(n \log \log n)$ -time algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, 17(1):143–178, 1988. doi:10.1137/0217010.
- [46] Athanasios K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67(1-3):173–194, 1985. doi:10.1016/S0019-9958(85)80034-6.
- [47] P. F. Windley. Trees, forests and rearranging. *The Computer Journal*, 3(2):84–88, 1960. doi:10.1093/COMJNL/3.2.84.

Part III

Extensions and Future Work

Chapter 18

Supporting Additional Operations for Buffered Partially-Persistent External-Memory Search Trees

In this chapter I show how the construction presented in the paper “Buffered Partially-Persistent External-Memory Search Trees” [44], may be altered to support further operations and use cases. These extensions were created in collaboration with Gerth Stølting Brodal and Rolf Svenning from Aarhus University, before the submission of the paper, but due to constraints was chosen not to be included. In the paper, it is discussed how to extend the SEARCH operation to predecessor, exact, as well as strict predecessor and successor queries. Further, the paper discusses how to use the construction to maintain partially persistent dictionaries, consisting of key-value pairs. For an overview of the paper see Chapter 4. In this extension, we shall consider altering the RANGE operation, as well as support semi offline updates, where the deletion time is known upon an insertion.

18.1 Partial RANGE Queries

The construction may be extended to further support the following operation.

- $\text{RANGE}(v, x, K)$ Reports the first K keys in $\mathcal{S}_v \cap [x, \infty]$ in increasing order.

In the amortized case, this extension is trivial, as the RANGE query may simply report from each rectangle the regular query visits one at a time, and then terminate after reporting the first K keys. In total, amortized $\mathcal{O}(\frac{1}{\varepsilon} \log_B N_v + K/B)$ I/Os are used. When $M = \Omega(B^{1-\varepsilon} \log_2(\max_v N_v))$, the operations is supported at the same I/O cost in worst-case.

With no bound on M , the worst-case result does not report from the rectangles one at a time, and the simple implementation cannot be used. Specifically, the worst-case result uses that the range of buffered updates to locate in the B^ε -tree T is known, to then merge the updates down layer by layer. An exponential search may

be employed over the range, reporting from the rectangles in increasing intervals of whole rectangles, and capping the final search. This, however, imposes a $\mathcal{O}(\log K)$ multiplicative overhead on the I/O cost.

Instead, we can use that each rectangle must contain $\Theta(R)$ segments, where $\Omega(R)$ are spanning. From the minimum spanning requirement, the result of the RANGE query must be contained within the first $\mathcal{O}(K/R)$ rectangles from the location of x . This rectangle may be found using $\mathcal{O}(K/R)$ I/Os by linearly traversing the B-tree P_v . This range contains at least the K keys contained in the queried version, however, more than K keys may be reported, as each rectangle may contain more spanning segments or other non-spanning segments overlapping with the query range. However, as each rectangle contains $\mathcal{O}(R)$ segments, at most $\mathcal{O}(K)$ keys may be reported in total. Using the regular RANGE query therefore uses $\mathcal{O}(\frac{1}{\varepsilon} \log_B N_v + \gamma + K/B)$ I/Os. The final output may then cap the resulting list of $\mathcal{O}(K)$ keys, to report exactly the first K keys, as the reported keys of the regular RANGE query are reported in order. In total, $\mathcal{O}(\frac{1}{\varepsilon} \log_B N_v + \gamma + K/B)$ I/Os are used in the worst-case, with no bounds on M .

This new RANGE query may therefore be supported in the same I/O bound, as the regular RANGE query, for both the amortized and worst-case setting.

18.2 Supporting Partially Ordered Sets

One classical use of partially persistent search trees, is to support *planar point location*. However, for allowing the planar point location result of Sarnak and Tarjan [115], the partially persistent search tree must support keys which are only *partially ordered*. Specifically it must hold that keys contained in the same version are comparable, but keys not contained in the same version need not be comparable. This is supported by the structure of Arge, Danner, and Teh, leading to an external-memory solution for planar point location [13], which uses $\mathcal{O}(N \log_B N)$ I/Os for preprocessing, which can answer point location queries using $\mathcal{O}(\log_B N)$ I/Os.

Our structure requires that the keys are *totally ordered*. Specifically, the splitting keys contained in the B^ε -tree T must be comparable to the buffered updates. These splitting keys are the sides of the open rectangles, and are therefore important to ensure that the buffered updates can locate the correct open rectangle containing it. When such a key is deleted, then the two open rectangles are no longer clearly separable, and must be closed. If this key is spanning in the rectangle, then it is not deleted before the rectangle normally is finalized, which leads to an efficient structure. However, as we do not know if and when a given key is deleted, we cannot ensure that the splitting keys are spanning. Therefore, we cannot ensure that each rectangle receives $\Omega(R)$ updates before it is finalized, and therefore the space usage is not bounded by $\mathcal{O}(N/B)$ blocks.

However, for planar point location, the problem is *offline*, as the order of all insertions and deletions is known in advance. This ensures that the updates to the construction are *semi offline*, as upon inserting a key, we know at what later version that key is deleted. We can therefore ensure that the splitting keys are spanning

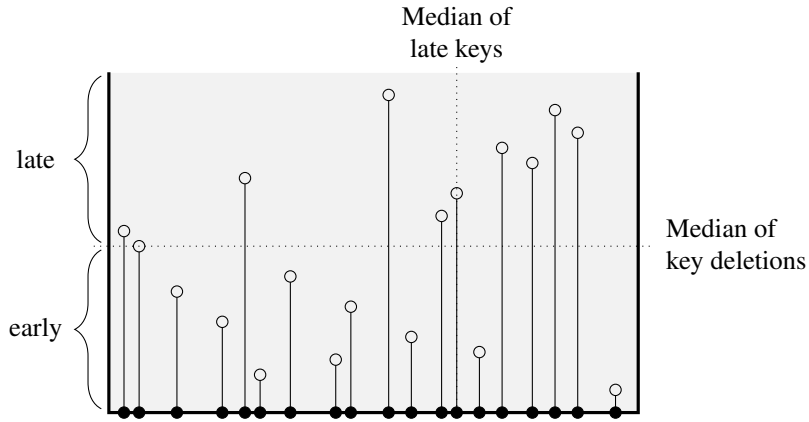


Figure 18.1: Splitting an open rectangle under semi offline updates. Each segment knows the deletion time, which therefore are not buffered. The keys are split at the median key deletion time into the (bottom) early keys and the (top) late keys. The rectangle is then split as the median key of the late keys.

segments in the open rectangles, as we know the deletion time of each segment. Additionally, this ensures that all version B-trees P_v only contain comparable keys, as the spanning splitters must all reside at version v to be included in P_v .

Similar to the regular construction, we finalize a rectangle after it receives $\Omega(R)$ updates. However, we shall also finalize the rectangle when the splitting key of either side reaches its deletion version. When splitting the key range a rectangle covers, we will first partition the keys by deletion time by the median deletion time using $\mathcal{O}(R/B)$ I/Os. This yields two sets of keys; the keys which are deleted before the median time denoted as the *early* keys, and symmetrically the *late* keys. An illustration of a rectangle under splitting with the early and late key sets is shown in Figure 18.1. As the partition is at the median deletion time, and the rectangle must contain $\Theta(R)$ initial keys at the bottom version, there must be $\Theta(R)$ early and late keys. Any key in the late set may therefore be spanning, as $\Omega(R)$ deletion updates from the early keys must happen before the deletion of any late key. The splitting key is then chosen as the median value among the late keys. This split, by the final median selection, still ensures that each of the two new rectangles contain $\Theta(R)$ initial keys, as half of the late keys ends in each rectangle.

Note that the median of medians selection may split the rectangle, where one side may contain up to $3/4^{\text{th}}$ of the initial keys of the split rectangle, with the other side may contain as few as $1/4^{\text{th}}$ of the keys. The constants of the construction must therefore be altered to support these new possible initial key count. Note that the minimum spanning requirement may easily be supported, by finalizing the rectangle when the median of the initial deletion times is reached and not the selected splitting element. This still ensures that all early keys support the finalization, where all late keys then must be spanning. As the late keys are split along the median, this ensures that both rectangles contains $1/4^{\text{th}}$ of the initial keys of the split rectangle as spanning

segments.

A finalization due to a splitting key reaching its deletion time must require $\Omega(R)$ unique deletion updates from the early keys to occur, which correspond to the early keys. A finalization due to a rectangle receiving too many updates must require $\Omega(R)$ unique insertion of new keys to occur. Each finalization therefore correspond to $\Omega(R)$ unique updates, and therefore the partition contains $\mathcal{O}(\bar{N}/R)$ rectangles, leading to the space usage being $\mathcal{O}(\bar{N}/B)$ blocks. This construction may then be used with global rebuilding, to achieve bounds dependent on N_v .

Note that the above construction needs to further maintain the invariant that any buffered update is comparable to all splitting keys in its subtree. Upon a finalization, it is therefore important to flush all updates of all buffers on the path, to ensure that the newly inserted splitting key does not break the invariant. Each of the $\leq H$ nodes on the path spend amortized $\mathcal{O}(\Delta)$ I/Os to flush all buffered updates, with the amortization due to possible cascades. As the emptying of the path buffers must occur due to $\Omega(R)$ unique updates in the leaf, the amortized cost of each update increase by $\mathcal{O}(\frac{H \cdot \Delta}{R}) = \mathcal{O}(\frac{1}{B^{1-\epsilon}})$ I/Os, which does not alter the asymptotic cost of updates.

In total, this leads to a construction supporting semi offline updates and regular queries in the same amortized bounds as the construction of the paper. This new construction may then be used to construct a planar point location structure over N segments in the plane, using $\mathcal{O}(\frac{N}{\epsilon B^{1-\epsilon}} \log_B N)$ I/Os, using $\mathcal{O}(N/B)$ blocks of space, which supports point location queries using $\mathcal{O}(\frac{1}{\epsilon} \log_B N)$ I/Os. This is therefore an improvement over the structure by Arge, Danner, and Teh [13], matching the improvement B^ϵ -trees obtain over regular B-trees. However, the construction is not an improvement over the state-of-the-art, which supports K batched queries on N segments using $\mathcal{O}\left(\left(\frac{N}{B} + \frac{K}{B}\right) \log_{M/B} \frac{N}{B}\right)$ I/Os [71].

Chapter 19

Extending Finding Longest Palindromic Subsequences Using Longest Common Subsequences to other Algorithms

In our paper “On Finding Longest Palindromic Subsequences Using Longest Common Subsequences” [35] we show how the longest common subsequence (LCS) algorithm of Wagner and Fischer [133] may be used to produce a longest palindromic subsequence (LPS) on a string s , by inputting to the algorithm the string s and its reverse s^R . For an overview of the paper see Chapter 8.

The LCS problem is most generally used to compute the *edit distance* of two strings, i.e., the number of changes required to turn the string s into the string t , with the LCS consisting of the largest portion of the strings which may be preserved under the edit. This edit distance is used in, e.g., the `diff` command from GNU and in `git`, in both cases on large files. However, when Wagner and Fischer is run on input string s and t of lengths m and n respectively, the algorithm uses $\mathcal{O}(m \cdot n)$ time and space. This runtime is far too great to run for any sufficiently large files.

By a reduction from the *strong exponential time hypothesis* (SETH), Abboud, Backurs and Williams [1], and separately Bringmann and Künnemann [27] showed that the quadratic runtime of Wagner and Fischer is required to solve the LCS problem in general. This bound only depends on the input lengths, and if instead considering other measures of the input than just length, algorithms with better runtimes may be constructed, as they circumvent this lower bound.

In this chapter I will discuss several other LCS algorithms, and prove if they produce an LPS when executed on a string s and its reverse s^R . Note that due to [35, Lemma 3], any LCS algorithm may be converted to produce an LPS with an additive $\mathcal{O}(k)$ time overhead, when the output LCS is of length k . I therefore only consider if the algorithms, without modification, already produce an LPS, as with the Wagner and Fischer algorithm. An overview of the results presented in this chapter is shown

Table 19.1: The six different new LCS algorithms analyzed in this chapter. We prove that the three algorithms on the left always report a palindrome, whereas the three algorithms on the right might fail to report a palindrome.

LCS(s, s^R) is always a palindrome	LCS(s, s^R) can be non-palindromic
HIRSCHBERG (Theorem 1)	HUNTSZYMANSKI (Theorem 2)
HUNTSZYMANSKIMOD (Theorem 3)	MYERS (Theorem 5)
KUOCROSS (Theorem 4)	MYERSINV (Theorem 6)

in Table 19.1.

19.1 Hirschberg

Hirschberg improved the space usage of Wagner and Fischer's algorithm to $O(m + n)$, while the runtime remained at $\mathcal{O}(m \cdot n)$ [75]. We shall denote this algorithm by HIRSCHBERG. The algorithm proceeds by computing the location the backtracking path cross the middle row, and then recurse on the two sub-problems. During this, only $\mathcal{O}(1)$ rows of the table needs to be remembered. The computation proceeds by dynamic programming, computing for each cell (i, j) both the LCS value of the prefixes, and which cell is visited in the middle row, along the path extending from cell (i, j) . How to choose which cell is reached, then dictates the backtracking. We can therefore show the following theorem.

Theorem 1. *The string produced by HIRSCHBERG of s and s^R is a palindrome.*

Proof. For each cell (i, j) , HIRSCHBERG may choose to obtain the crossing cell from any neighbor where the computed LCS value may originate from. By choosing this cell using the same backtracking strategy as Wagner and Fischer, then HIRSCHBERG produces the same path as Wagner and Fischer on any input, and therefore also the same output LCS string. By [35, Theorem 14], then HIRSCHBERG produces a palindrome when executed on s and s^R . $\square$

19.2 Hunt-Szymanski

To improve below the lower bound, Hunt and Szymanski designed an adaptive algorithm, which on input strings s and t of lengths m and n , with r matching symbols between s and t , has the runtime $\mathcal{O}((r + m) \log n)$ [78]. Note that $0 \leq r \leq m \cdot n$. For larger values of r , the Hunt and Szymanski algorithm is therefore slower than the Wagner and Fischer algorithm. The intuition is that for file systems, where the edit distance is computed line by line, there are not many lines matching, which therefore results in a faster algorithm. To improve checking for matches, Hunt and MacIlroy applied hashing to the lines for easy non-matching checks [79].

We shall denote the algorithm as HUNTSZYMANSKI for convenience. The algorithm builds on the following observation. In the table computed by Wagner and

Fischer, for any cells that does not include a match, the length of the LCS is unchanged. The important cells are therefore the *match cells*. In a match cell (i, j) , the $\nwarrow$ step is allowed, which by Wagner and Fischer, is always optimal to traverse. As $\uparrow$ and $\leftarrow$ steps are allowed from any cell, the match cell may therefore reach any cell (i', j') with $i' < i$ and $j' < j$. Therefore, the *value* of the match cell (i, j) is the largest value among the cells (i', j') plus 1 from the match. By applying a data structure allowing for easily computing the largest valued (i', j') in time $\mathcal{O}(\log n)$, the value of the match cell (i, j) may be computed efficiently. This thereby links cell (i, j) to (i', j') as a *jump*, which may be traversed in the backtracking.

Before stating the HUNTSZYMANSKI algorithm in details, we first show that all jumps from a match cell (i, j) to some target cell (i', j') , may always choose the target cell as a match cell. This allows HUNTSZYMANSKI to simply compute on the match cells. The structure of the proof follows the same order as [76, Lemma 1].

Lemma 1. *For any cell (i, j) of value k then either $k = 0$ or $\exists i', j'$ s.t. $i' \leq i, j' \leq j$, and (i', j') is a match cell of value k , i.e., any cell must have a match cell of equal value above and/or to the left.*

Proof. If cell (i, j) have value 0, the proof follows trivially. Let $k \geq 1$. The proof follows by induction over the rows and columns. In the base case (i, j) is a match cell. Let $i' = i$ and $j' = j$, showing their existence. In the inductive step cell (i, j) is a non-match cell. It must then hold that either cell $(i - 1, j)$ or $(i, j - 1)$ have value k . Let such a cell be denoted as (i'', j'') . By induction there is a match cell (i', j') , s.t. $i' \leq i''$ and $j' \leq j''$. As $i'' \leq i$ and $j'' \leq j$, then this is also a valid match cell for (i, j) . Note that as $k \geq 1$ and the first row and column of the LCS table is 0, then the base case must be reached, and such a match cell is found. $\square$

In the *longest increasing subsequence* (LIS) problem, the input is a list of n elements, and the goal is to compute a longest subsequence, s.t. the values of the subsequence are increasing [116]. Note that since HUNTSZYMANSKI may jump only between match cells, due to Lemma 1, HUNTSZYMANSKI can be seen as computing a LIS over the match cells, where the match cells form a partial order. This thereby ties the LIS problem into the LCS and LPS problems by reduction.

We are now ready to define the HUNTSZYMANSKI algorithm in details. The algorithm of Hunt and Szymanski is stated in [78, Algorithm 2], and restated in Figure 19.1 for convenience. First, the match cells are located by first computing *matchlists*, which computes for each symbol in s a list of the locations of that symbol in t . These matchlists are computed in time $\mathcal{O}(n \log n)$. This allows for traversing the match cells in order row by row. By processing the match cells in reversed column order, the data structure to locate the (i', j') when computing the value for match cell (i, j) may be a simple list, where binary search is applied to find the needed cell and update the list. This step thereby takes $\mathcal{O}(r \log n)$ time. Finally, an LCS path is backtracked to compute the output LCS.

Figure 19.2a shows the path chosen by HUNTSZYMANSKI when run on input $s = \text{bbcdabbbacbc}$ and $t = \text{abcadcbbc}$. The path chosen by ALGCONTOUR and

```

HUNTSZYMANSKI( $s, t$ )
1  integer array  $THRESH[0 : n]$ 
2  list array  $MATCHLIST[1 : m]$ 
3  pointer array  $LINK[1 : n]$ 
4  pointer  $PTR$ 
5  // Step 1: build linked lists
6  for  $i \leftarrow 1$  step 1 until  $m$ 
7       $MATCHLIST[i] \leftarrow \langle j_1, j_2, \dots, j_p \rangle$  such that
           $j_1 > j_2 > \dots > j$  and  $s[i] = t[j_q]$  for  $1 \leq q \leq p$ 
8  // Step 2: initialize the  $THRESH$  array
9   $THRESH[0] \leftarrow 0$ 
10 for  $i \leftarrow 1$  step 1 until  $n$ 
11      $THRESH[i] \leftarrow n + 1$ 
12  $LINK[0] \leftarrow \text{NULL}$ 
13 // Step 3: compute successive  $THRESH$  values
14 for  $i \leftarrow 1$  step 1 until  $m$ 
15     for  $j$  in  $MATCHLIST[i]$ 
16         find  $k$  such that  $THRESH[k - 1] < j \leq THRESH[k]$ 
17         if  $j < THRESH[k]$ 
18              $THRESH[k] \leftarrow j$ 
19              $LINK[k] \leftarrow \text{newnode}(i, j, LINK[k - 1])$ 
20 // Step 4: recover longest common subsequence in reverse order
21  $k \leftarrow$  largest  $k$  such that  $THRESH[k] \neq n + 1$ 
22  $PTR \leftarrow LINK[k]$ 
23 while  $PTR \neq \text{NULL}$ 
24     print  $(i, j)$  pair pointed to by  $PTR$ 
25     advance  $PTR$ 

```

Figure 19.1: Pseudocode implementations of the Hunt and Szymanski algorithm from [78, Algorithm 2], tailored to handle input strings s and t of possible different lengths m and n .

ALGLOWERCONTOUR, which traverse the upper and lower contour respectively, on the same input can be seen in Figure 19.2b and 19.2b. Here it can be observed that the path chosen by HUNTSZYMANSKI first follows the lower contour, where it then suddenly switches over the middle of the profile, to then follow the upper contour. This is also demonstrated by the output LCS, which consists of the first half of the upper contour, followed by the last half of the lower contour.

The switch between the contours is due to the *jump priority* chosen by HUNTSZYMANSKI, i.e., when multiple match cells (i', j') may be jumped to, which cell is then chosen. The priority is chosen by the order of the matchlists defined in line 7,

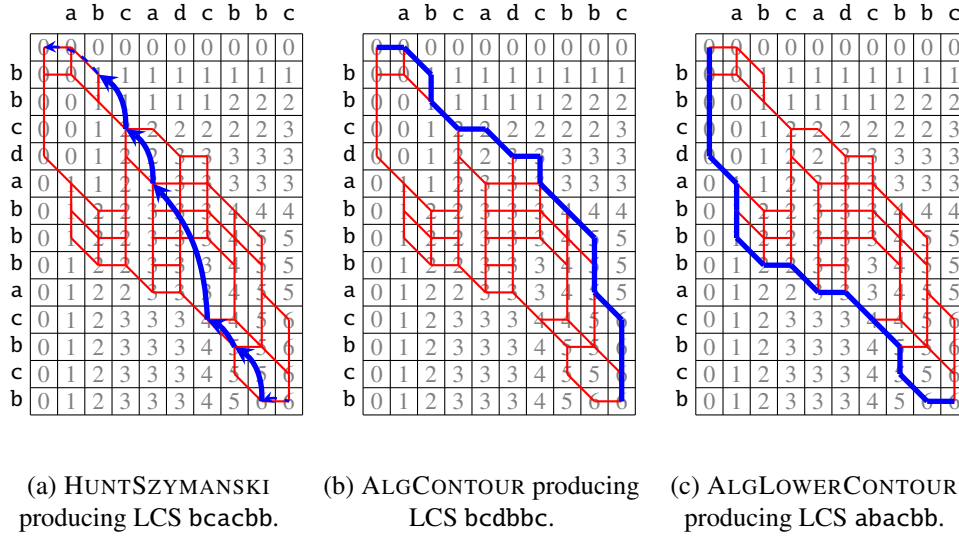


Figure 19.2: LCS table of the strings $s = \text{bbcdabbbacbcbb}$ and $t = \text{abcadcbbbc}$, with the LCS values in gray and the profile overlayed in red. The blue path, is the path taken by the given listed algorithm, and the LCS produced by each algorithm is listed below. For the HUNT SZYMANSKI algorithm in Figure 19.2a, the bent arrows are jumps between match cells, with the dashed arrows denoting the initial and final step, which does not contribute to the output LCS.

and the logic in lines 16-19 of the pseudocode in Figure 19.1. Here it can be derived that HUNT SZYMANSKI jumps from match cell (i, j) of value k to match cell (i', j') with value $k - 1$, s.t. cell (i', j') is the leftmost and then the topmost match cell of value $k - 1$, i.e. j' is minimal and tied by i' being minimal. This can also be seen in Figure 19.2a, when jumping from the cell with value 4, there are three reachable cells with value 3, with only the jump strategy of HUNT SZYMANSKI reaching the cell of value 3 the path jumps to, when the strategy must be choosing maximal or minimal in i and j in some order. This priority is valid to ensure that the output is an LCS. However, this is what allows HUNT SZYMANSKI switching from the lower contour to the upper contour, as it may effectively choose an $\uparrow$ instead of a $\nwarrow$ step, where the ALG LOWER CONTOUR follows $\langle \leftarrow, \nwarrow, \uparrow \rangle$, and therefore must choose the $\nwarrow$.

In general, HUNT SZYMANSKI may follow some arbitrary path in the middle of the profile after any number of jumps, and have no guarantee to follow either part of the upper or lower contour. This also has the consequence that HUNT SZYMANSKI may output non-palindromic LCS strings on s and s^R .

Theorem 2. *The string produced by HUNT SZYMANSKI on s and s^R can be non-palindromic.*

Proof. On input string $s = \text{acababcbacccbb}$ and the reversed string s^R the output LCS string of HUNT SZYMANSKI is cabcbab , which is non-palindromic. See Figure 19.3.

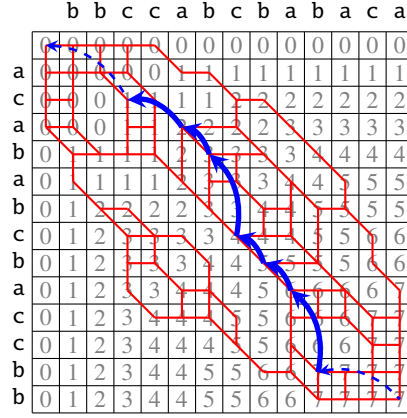


Figure 19.3: LCS table of $s = \text{acababcbacbb}$ and s^R , with the profile overlaid in red. The blue bent arrows are the jumps between match cells performed by HUNTSZYMANSKI, with the dashed arrows denoting the initial and final step. The output LCS of HUNTSZYMANSKI is cabcbab , which is non-palindromic.

□

In the example string of Theorem 2, the upper and lower contour strings are acabaca and bbccbb , respectively. Note that only the middle symbol of the lower contour matches in the HUNTSZYMANSKI output string, further demonstrating that HUNTSZYMANSKI have no guarantee to follow any parts of the contours.

Recall that the jump priority of HUNTSZYMANSKI is choosing the match cell (i', j') with value $k - 1$, s.t. cell (i', j') is the leftmost and then the topmost match cell of value $k - 1$. The criteria on being leftmost is thereby following in the direction $\leftarrow$ as much as possible. Then, when there are no further left match cells, the topmost match cell is chosen, which is following in the $\uparrow$ direction as much as possible. This thereby implicitly priorities the *skipping* steps over the *match* step. The algorithm ALGLOWERSKIP follows strategy $\langle \leftarrow, \uparrow, \nwarrow \rangle$, which resembles the prioritization of HUNTSZYMANSKI. This observation is proven in the lemma below.

Lemma 2. *For all pairs of input strings s and t , algorithms HUNTSZYMANSKI and ALGLOWERSKIP produce the same LCS string.*

Proof. Recall that the strategy of ALGLOWERSKIP is $\langle \leftarrow, \uparrow, \nwarrow \rangle$. The proof goes by induction over the number of jumps performed by HUNTSZYMANSKI.

Initially, HUNTSZYMANSKI starts at the leftmost tied by topmost match cell (i', j') , which has the same value k as the bottom rightmost cell (i, j) of the table. Note that ALGLOWERSKIP starts in cell (i, j) . If ALGLOWERSKIP traverses to exactly cell (i', j') and follows the $\nwarrow$, then after HUNTSZYMANSKI performs a jump, the same situation arises. Otherwise, the cell value of (i, j) is 0, and no more symbols are output to the LCS. We shall therefore show the claim that ALGLOWERSKIP traverses to the cell of HUNTSZYMANSKI.

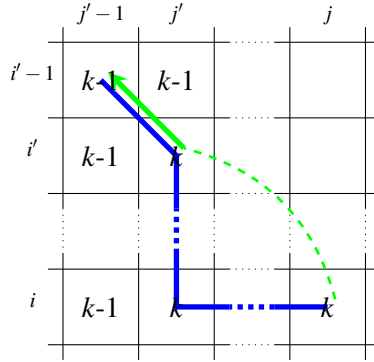


Figure 19.4: Behavior of ALGLOWERSKIP (in blue) starting in cell (i, j) and traversing to cell (i', j') where HUNTSZYMANSKI (in green) jumped to, where they both leave by a $\nwarrow$ step.

The setup is depicted in Figure 19.4. First note that both cells (i, j) and (i', j') have value k . By monotonicity, all cells inside the sub-table must therefore have value k , which allows all $\leftarrow$ and $\uparrow$ steps between the cells. As ALGLOWERSKIP prioritizes $\leftarrow$ steps, first it traverses to cell (i, j') , directly below HUNTSZYMANSKI. If then a $\leftarrow$ step is available from cell (i, j') , then by Lemma 1 there would then exist a match cell of value k strictly to the left of j' , which contradicts that (i', j') is a leftmost match cell of value k . The value of $(i, j' - 1)$ must therefore be $k - 1$. As cell (i', j') is a match cell, then cell $(i' - 1, j' - 1)$ must have value $k - 1$. By monotonicity of the LCS table, then all cells above $(i, j' - 1)$ must have a value $k - 1$, and therefore in all cells between (i, j') and (i', j') , there is no $\leftarrow$ steps possible. As the $\uparrow$ steps are possible, then by the strategy of ALGLOWERSKIP, the algorithm will traverse up to cell (i', j') . Using a symmetric argument, on the topmost property of the match cell (i', j') , there is then no $\uparrow$ step available further up. Therefore, the only step possible is the $\nwarrow$ step, which ALGLOWERSKIP follows.

This thereby concludes the proof. $\square$

As ALGLOWERSKIP is not guaranteed to produce palindromes on s and s^R , following a version of [35, Lemma 15], then Theorem 2 may also be proven, not by example, but by using Lemma 2, to relate HUNTSZYMANSKI to a non-palindrome producing algorithm.

19.2.1 Modifying Hunt-Szymanski

By altering the jump priority of HUNTSZYMANSKI to instead choose the cell (i', j') which is the leftmost and then the *downmost* match cell of value $k - 1$, i.e. j' is minimal and tied by i' being maximal, then the algorithm may not step over a $\nwarrow$ step when traversing up $\uparrow$ steps. We shall denote this modified algorithm as HUNTSZYMANSKI-MOD. Interestingly, this jump priority can be achieved with the pseudocode in Figure 19.1, by removing the if check on line 17, thereby always executing lines 18 and 19. Note that this does not alter the runtime of the algorithm. Further the

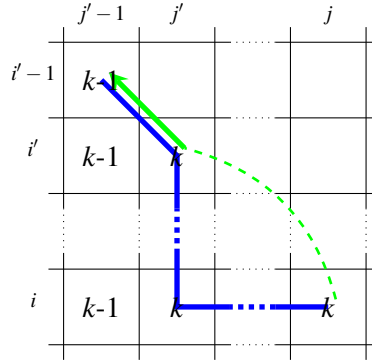


Figure 19.5: Behavior of ALGLOWERCONTOUR (in blue) starting in cell (i, j) and traversing to cell (i', j') where HUNTSZYMANSKIMOD (in green) jumped to, where they both leave by a $\nwarrow$ step.

correctness holds by observing that $j \leq \text{THRESH}[k]$ by definition of line 16, and the removal of the check allows the inner code to execute additionally only when $j = \text{THRESH}[k]$. This prompts line 18 to not alter the threshold array, thereby maintain the correctness. The only change is that the linking array containing the jumps is updated when a match in the same column, but further down, is found.

This modified algorithm does in fact follow the lower contour, which is proven below.

Lemma 3. *For all pairs of input strings s and t , algorithms HUNTSZYMANSKIMOD and ALGLOWERCONTOUR produce the same LCS string.*

Proof. The proof follows the structure of Lemma 2.

Recall that the strategy of ALGLOWERCONTOUR is $\langle \leftarrow, \nwarrow, \uparrow \rangle$. The proof goes by induction over the number of jumps performed by HUNTSZYMANSKIMOD.

Initially, HUNTSZYMANSKIMOD starts at the leftmost tied by downmost match cell (i', j') , which has the same value k as the bottom rightmost cell (i, j) of the table. Note that ALGLOWERCONTOUR starts in cell (i, j) . If ALGLOWERCONTOUR traverses to exactly cell (i', j') and follows the $\nwarrow$, then after HUNTSZYMANSKIMOD performs a jump, the same situation arises. Otherwise, the cell value of (i, j) is 0, and no more symbols are output to the LCS. We shall therefore show the claim that ALGLOWERCONTOUR traverses to the cell of HUNTSZYMANSKIMOD.

The setup is depicted in Figure 19.5. First note that both cells (i, j) and (i', j') have value k . By monotonicity, all cells inside the sub-table must therefore have value k , which allows all $\leftarrow$ and $\uparrow$ steps between the cells. As ALGLOWERCONTOUR prioritizes $\leftarrow$ over $\uparrow$ steps, first it traverses to cell (i, j') , directly below HUNTSZYMANSKIMOD. If then a $\leftarrow$ step is available from cell (i, j') , then by Lemma 1 there would then exist a match cell of value k strictly to the left of j' , which contradicts that (i', j') is a leftmost match cell of value k . The value of $(i, j'-1)$ must therefore be $k-1$. As cell (i', j') is a match cell, then cell $(i'-1, j'-1)$ must have value $k-1$. By monotonicity of the LCS table, then all cells above $(i, j'-1)$ must have a value $k-1$,

and therefore in all cells between (i, j') and (i', j') , there is no $\leftarrow$ steps possible. It holds that all cell between (i, j') and $(i' - 1, j')$ cannot contain a $\nwarrow$ step, as this would contradict that (i', j') is the downmost match cell of value k in column j' . Therefore `ALGLOWERCONTOUR` must use the $\uparrow$ steps to traverse to cell (i', j') . As there is not $\leftarrow$ step from this cell, but a $\nwarrow$ step is possible, then `ALGLOWERCONTOUR` follows the $\nwarrow$ step.

This thereby concludes the proof. $\square$

We may therefore conclude that `HUNT SZYMAN SKIMOD` outputs palindromes on s and s^R by the following theorem.

Theorem 3. *The string produced by `HUNT SZYMAN SKIMOD` of s and s^R is a palindrome.*

Proof. By Lemma 3 the string produced by `HUNT SZYMAN SKIMOD` is the same as `ALGLOWERCONTOUR` on any input strings. By following [35, Lemma 12 and 13], the lower contour outputs a palindrome. Therefore, `HUNT SZYMAN SKIMOD` outputs a palindrome on input s and s^R . $\square$

19.3 Kuo and Cross

Both versions of the Hunt and Szymanski algorithm, presented in Section 19.2, have the runtime $\mathcal{O}((r + m) \log n)$. As $0 \leq r \leq m \cdot n$, the runtime may therefore in the worst case be $\mathcal{O}(mn \log n)$, which is a factor $\mathcal{O}(\log n)$ slower than the algorithm of Wagner and Fischer. In the runtime, the $\mathcal{O}(\log n)$ stems from the binary search the algorithm of Hunt and Szymanski performs for each match cell. Kuo and Cross noted that when computing the threshold values for each cell, then within a row, the resulting threshold of the first match cells limits the range of possible threshold values for the next match cell [88]. They then altered the algorithm, by instead of performing a binary search, instead performed a linear sweep over the thresholds along with the match cells of each row. We shall denote this algorithm as `KUOCROSS`. If this sweep is performed from the end of the list, as needed by the reversed order of the match cells, this leads to a $\mathcal{O}(mn)$ algorithm, as the only initial threshold value for each row is n . By instead computing the match cells in increasing order for each row, the sweep cannot traverse further than the LCS length k . This improvement is shown in detailed pseudocode in [88, Figure 2]. The runtime of the algorithm then improves to $\mathcal{O}(r + m(k + \log n))$, which in the worst-case is not worse than the algorithm of Wagner and Fischer.

The altering of the `HUNT SZYMAN SKI` algorithm to obtain the `KUOCROSS` algorithm alters the jump priority of the algorithm. Specifically, following the details of the pseudocode, the threshold array is updated as soon as a new threshold location is found. As the sweep is from the left to the right, this ensures that the jumps prioritizes leftmost cells. As the threshold is allowed to be updated to the same value, then a lower row in the same column may update the link, and thereby the jumps are tied by choosing the downmost cell. I.e., `KUOCROSS` jumps from match cell (i, j) of value

k to match cell (i', j') with value $k - 1$, s.t. cell (i', j') is the leftmost and then the downmost match cell of value $k - 1$, i.e. j' is minimal and tied by i' being maximal. KUOCROSS thereby alter the jump priority of HUNT SZYMANSKI to instead follow the same priority as HUNT SZYMANSKIMOD.

We can therefore conclude the following theorem.

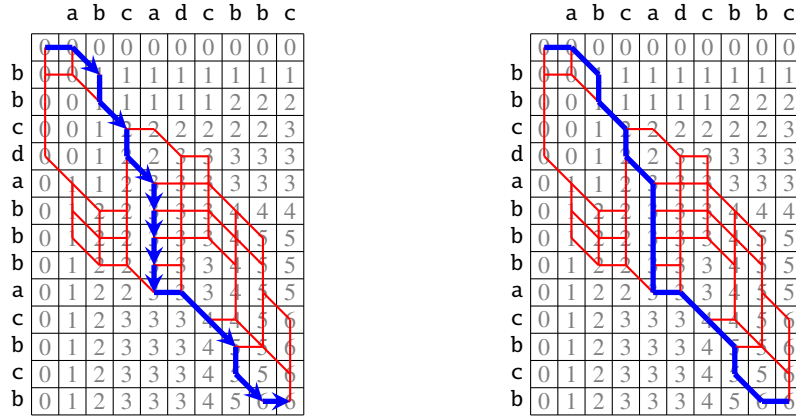
Theorem 4. *The string produced by KUOCROSS of s and s^R is a palindrome.*

Proof. As KUOCROSS follows the same jump priority as HUNT SZYMANSKIMOD, they both output the same LCS string. By Theorem 3, HUNT SZYMANSKIMOD outputs a palindrome on input s and s^R , and therefore KUOCROSS also outputs a palindrome. $\square$

19.4 Myers

As mentioned, LCS algorithms are classically used to compute the edit distance of two strings. The adaptive algorithms of HUNT SZYMANSKI and KUOCROSS are both fast, when the number of matches is few and the length of the output LCS short. However, when the LCS is short, then the edit distance is long. For checking differences between files, we usually assume that the distance is short, and therefore the algorithms are slow. The basic idea of HUNT SZYMANSKI is that for files, when comparing line by line, most lines in a file are different, and therefore the number of matches r is small. However, for, e.g., a file containing code, many lines are left blank for readability. Even if only 10% of the lines are blank, this still leads to $r \geq 0.01m \cdot n$, and the asymptotic runtime of both HUNT SZYMANSKI and KUOCROSS does not beat the runtime of the original Wagner and Fischer algorithm.

Instead, Myers created an algorithm which is fast, when the edit distance is small, i.e., faster for longer LCS string [102]. In the paper, different variations of the algorithm is described, but for the following we shall only focus on the simple greedy algorithm of [102, Section 3], which we shall denote as the MYERS algorithm. The idea of the algorithm is to compute for increasing values of edit distances D a set of *greedy* paths from the top left cell of the LCS table. Each such path aims at reaching the furthest down along the diagonal of the table. Note that the order of computation is inverted to the regular backtracking of the LCS table. At each step, the paths of distance $D - 1$ is extended by a $\leftarrow$ or $\uparrow$ step to reach a cell of edit distance D . Then $\nearrow$ steps are followed as far as possible along the reached diagonal. Myers denotes each set of reached cells for a value D as the *envelope* of D . See [102, Figure 3] for an illustration of the paths and envelopes. Crucially to MYERS is that only a single cell is needed to be computed for each pair of distance D and *diagonal* δ . Additionally, for each edit distance only every other diagonal is reachable. This allows for computing the envelopes in a single array. See [102, Figure 2] for the pseudocode. Effectively MYERS can be seen as a BFS over the edges of the LCS table, where $\nearrow$ steps have a cost of 0 and the $\leftarrow$ and $\uparrow$ steps have a cost of 1.



(a) MYERS producing LCS bcacbb.

(b) ALGLOWERSKIP producing LCS bcacbb.

Figure 19.6: LCS table of the strings $s = \text{bbcdabbbacbcbb}$ and $t = \text{abcadcbbc}$, with the LCS values in gray and the profile overlayed in red. The blue path, is the path taken by the given listed algorithm, and the LCS produced by each algorithm is listed below.

The runtime of MYERS on input strings of length m and n with edit distance D is $\mathcal{O}((m+n)D)$. The algorithm is therefore faster for smaller edit distances.

Figure 19.6a shows the path chosen by MYERS on input $s = \text{bbcdabbbacbcbb}$ and $t = \text{abcadcbbc}$. Note that this input is the same input as in Figure 19.2. The path chosen by MYERS can be seen following the same path as ALGLOWERSKIP, which is shown in Figure 19.6b. ALGLOWERSKIP follows the backtracking strategy $\langle \leftarrow, \uparrow, \nearrow \rangle$, which therefore chooses to skip as much as possible. MYERS is trying to greedily match as much as possible, like ALGMATCH, however, the greedy matching is from the top left corner and not the lower right corner. That MYERS and $\langle \leftarrow, \uparrow, \nearrow \rangle$ follows the same path on the example is not a coincidence, and in the following we shall prove that the algorithms always takes the same path.

First we need to show that MYERS greedy strategy ensures that the furthest cell in each diagonal for each edit distance is reached.

Lemma 4. *For each edit distance D and diagonal δ , then MYERS will visit the furthest cell on the diagonal.*

Proof. The proof goes by induction on the edit distance D . For $D = 0$ MYERS starts in cell $(0,0)$, in diagonal $\delta = 0$ and traverses $\nearrow$ steps as far as possible, reaching some cell c . By definition, the edit distance of this cell is 0. Any path to any cell further down the diagonal than c must traverse either a $\leftarrow$ or $\uparrow$ step, which would increase the edit distance to > 0 . The reached cell is therefore the furthest cell on the diagonal for $D = 0$.

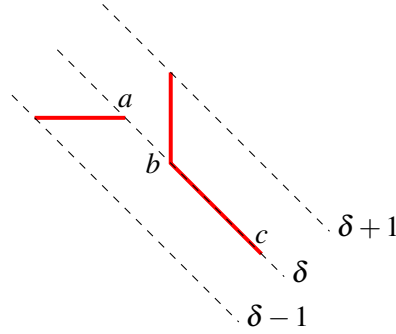


Figure 19.7: Induction step of MYERS reaching the furthest cell along diagonal δ for edit distance D . The cells a and b are the initial candidate cells obtained from the induction hypothesis, and c is the farthest reachable cell using only $\nwarrow$ steps from the furthest candidate of a or b .

Let $D \geq 1$, and the diagonal be δ . By induction, MYERS must have visited cells with edit distance $D - 1$ in diagonals $\delta - 1$ and $\delta + 1$. Note that it may have only visited one other diagonal, if the cells are close to the border of the table, but as $D \geq 1$, it must have visited at least one of the diagonals. By induction, the visited cells are the furthest on their respective diagonals. The setup is illustrated in Figure 19.7. MYERS then steps from these cells at distance $D - 1$ using a $\leftarrow$ or $\uparrow$ step, onto the diagonal δ , to gain up to two candidate cells a and b . It then picks the candidate which is the furthest on the diagonal. Then $\nwarrow$ steps are followed as far as possible, reaching some cell c . Any cell c' further down the diagonal than c , must therefore reach c' via either a $\leftarrow$ or $\uparrow$ step. This step must be taken from either the $\delta - 1$ or $\delta + 1$ diagonal, and must have distance $D - 1$. Such a cell c' would therefore contradict the induction hypothesis. The reached cell c is therefore the furthest cell on diagonal δ with edit distance D , concluding the proof. $\square$

We are now able to prove that MYERS and ALGLOWERSKIP always produce the same output LCS string.

Lemma 5. *For all pairs of input strings s and t , algorithms MYERS and ALGLOWERSKIP produce the same LCS string.*

Proof. The proof goes by induction over the steps performed by ALGLOWERSKIP, showing that MYERS performs the same steps. Initially, MYERS and ALGLOWERSKIP both start in the bottom right cell of the table. As an invariant, we show that the algorithms stay in the same cell.

Let the two algorithms be in cell (i, j) . Recall that the strategy of ALGLOWERSKIP is $\langle \leftarrow, \uparrow, \nwarrow \rangle$. If only a $\nwarrow$ step is possible, they must both take this step, ending in the same cell, and outputting the same symbol to the LCS. Otherwise, either a $\leftarrow$ or $\uparrow$ step is available. Let the current diagonal be δ and the edit distance be D . These steps are therefore to an edit distance $D - 1$ of a neighboring diagonal. As MYERS

have not left diagonal δ yet, then by Lemma 4, this neighboring cell must be the furthest along its diagonal. MYERS therefore take the step. Of only one of $\leftarrow$ or $\uparrow$ is available, then both ALGLOWERSKIP and MYERS chose the same step. Otherwise, the strategy of ALGLOWERSKIP chose the $\leftarrow$ step. MYERS break ties among the candidates when traversing into a new diagonal, by choosing the lower value diagonal, thereby choosing the $\leftarrow$ step.

In all cases both algorithms leave the cell using the same step, ending in the same cell, and therefore outputs the same LCS string. $\square$

As a consequence of following the path of ALGLOWERSKIP, the output of MYERS is not guaranteed to be an LPS.

Theorem 5. *The string produced by MYERS on s and s^R can be non-palindromic.*

Proof. By [35, Lemma 15], the string produced by ALGSKIP on input $s = \text{bacccabbaabcb}$ and s^R , the output LCS is bcabbaab . This same non-palindromic string is produced by ALGLOWERSKIP, when the order of the inputs is swapped, i.e., in input string $s = \text{bcbaabbaccab}$ and s^R . By Lemma 5, the output LCS of MYERS is also bcabbaab , which is non-palindromic. $\square$

The goal of Myers was to produce an algorithm, which contrary to the algorithm of Hunt and Szymanski, was faster when the length of the LCS is long. As any LCS may be computed, there is no requirement that the Myers algorithm needs to produce the same LCS string as Hunt and Szymanski, however, as a consequence of showing that both algorithms follow ALGLOWERSKIP, they do in fact produce exactly the same LCS string, which therefore allows them to produce the same edit script.

Corollary 1. *For all pairs of input strings s and t , algorithms HUNT SZYMANSKI and MYERS produce the same LCS string.*

Proof. Follows immediately from Lemma 2 and 5. $\square$

19.4.1 Inverting Myers

The greedy nature of MYERS follows as many $\nwarrow$ steps as possible from the top left cell of the LCS table. The Wagner and Fischer algorithm, denoted as the ALGMATCH algorithm in [35], similarly follows as many $\nwarrow$ steps as possible, but from the bottom right cell of the LCS table. From [35, Theorem 14] it holds that ALGMATCH produces palindromes on input strings s and s^R .

By reversing both input strings to MYERS, it is possible to reverse MYERS to perform greedy steps from the bottom right cell. We shall denote this algorithm as MYERSINV. However, this only guarantees that the path prioritizes $\nwarrow$ from the bottom right cell, but it does not place an order on the $\leftarrow$ and $\uparrow$ steps. If no such secondary order is given, then the proof of [35, Lemma 9], showing that ALGMATCH and ALGCONTOUR produce the same LCS string, does not work. Similarly, it does not hold that ALGMATCH and MYERSINV produce the same output LCS. This can

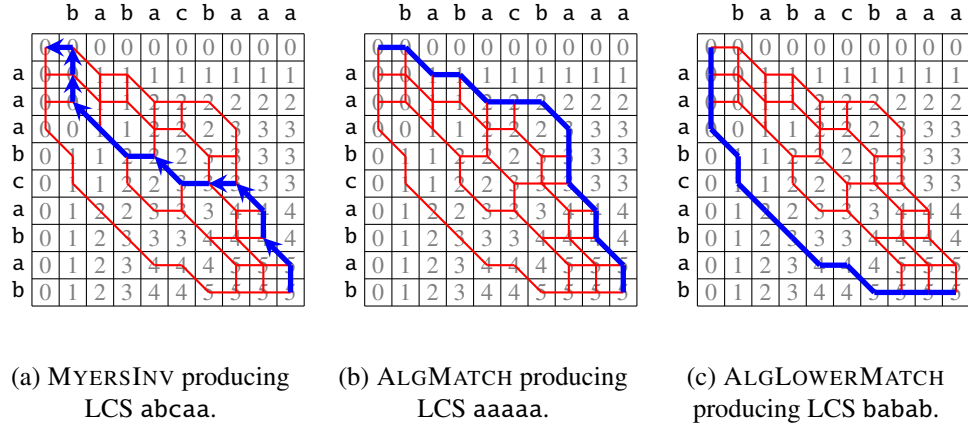


Figure 19.8: LCS table of $s = aaabcbabab$ and s^R , with the profile overlayed in red. The blue path, is the path taken by the given listed algorithm, and the LCS produced by each algorithm is listed below.

be seen in Figure 19.8, where the result of executing MYERSINV, ALGMATCH, and ALGLOWERMATCH in input $s = aaabcbabab$ and $t = s^R$, demonstrates that MYERS differs from the matching strategy. Specifically, the nondeterministic choice between $\leftarrow$ and $\uparrow$ of MYERSINV can be seen demonstrated, as it seven times is allowed to choose between the two steps, where the choices are $[\uparrow, \uparrow, \leftarrow, \leftarrow, \leftarrow, \uparrow, \uparrow]$.

Further, MYERSINV is not guaranteed to produce palindromes.

Theorem 6. *The string produced by MYERSINV on s and s^R can be non-palindromic.*

Proof. On input string $s = aaabcbabab$ and the reversed string s^R the output LCS string of MYERSINV is abcaa, which is non-palindromic. See Figure 19.8a. $\square$

Myers presents multiple versions of the initial greedy algorithm, we denoted MYERS. One of these version, described in [102, Section 4.2], is using that LCS tables may be traversed in both directions, similar to what MYERSINV does above. The version then traverses from both the top left and the bottom right corner of the LCS table at the same time, to find a path connecting the two sides. The idea of this technique is to lower the *width* of the traversed part of the table, i.e., limiting the number of diagonals needed to be visited before locating an LCS path. Interestingly, as the MYERS and MYERSINV are mirrored algorithms, then depending on their internal tie breaking, then what MYERS does from the top left corner may be mirrored by MYERSINV from the bottom right corner. This would then produce a symmetric LCS path, which by [35, Lemma 12] produces a palindromic LCS output string. However, from the details provided in the pseudocode [102, Section 4.2], it is unclear if this mirrored behavior is present, and therefore unclear if the algorithm produces palindromes or not.

19.5 The use of LCS algorithms in Real World Applications

The edit distance on files shows how changes to one file will result in the second file. This may also be viewed as the *difference* between the two files. This difference is used inside the `diff` command from GNU, as well as in the version control program `git`. As Myers algorithm is efficient when the edit distance is small, this is the algorithm both of these programs have chosen to use [69, 68, 95]. As the paper of Myers presents algorithms which are more efficient than MYERS, these more efficient variants are used in the programs. Note, that the programs are not necessarily interested in getting the smallest edit script when the difference between the two files are large, and for this case they use *heuristics* to speed up the program.

The programs do not produce palindromes, even with the heuristics turned off. This is shown using $s = \text{bcfafdaafabdacefdbcbfffaeaecadaaaafdbbbcc}$ to compute the LCS of s and s^R , using the `diff` (GNU `diffutils`) version 3.7, which produces the non-palindromic LCS string `cfaaadacefbbbfecadaaad`. On the same input MYERS, MYERSINV, as well as versions with the inputs swapped, produce output LCS strings among `cfaaadacefbcfbecadaaad`, `bfaaadacefbcfbecadaaaf`, and `cdaaadacefbcfbecadaaaf`, none of which match the output of the `diff`. Note that this does not disprove if the double ended version of Myers algorithm produces non-palindromes, as it is unclear if this variant is used in `diff`.

Chapter 20

Extending Pure Binary Finger Search Trees with New Algorithms

In our paper “Pure Binary Finger Search Trees” [40], many constructions are presented. In this chapter I will consider changes and simplifications to the construction, leading to simpler and faster algorithms for finger searching. Further, I have implemented the old and new algorithms, and run experiments to measure their runtime, which I use to conclude for which cases which algorithm is better. For an overview of the construction, see Chapter 9.

20.1 Simplified Pure Binary Finger Search Trees

The folded trees presented in the paper [40] relies on the red-black trees of Guibas and Sedgwick [74]. The folding of a red-black tree is in itself simple, and allows the folded tree to remain a balanced binary search tree. However, maintaining the folding under updates is not simple. Specifically, many cases are needed to even show that insertions are possible to support. One path for future work is to simplify this folding, to make it easier to maintain.

In the paper, the presented *metanodes* are simple, and allows for abstracting the tree to consider each node to contain pointers to the parent and left and right child, as well as an arbitrary constant number of bits. We shall therefore consider how to maintain a folding of this type of nodes.

Traversing around the folded tree during queries, where the traversal happens on the conceptual unfolded tree, uses the access to the predecessor and successor links. Notice that during the queries, the color bit stored in the nodes, used for the red-black tree structure, is not used, both for the algorithms of the red-black tree and also the algorithms finding the unfolded parent and left and right child, as well as the predecessor and successor of a given node. On the contrary, the bit denoting the path type is used during the queries. We may further notice, that the color bit need not even be stored in the lower chunks, except for the last lower chunk, as the structure of the chunk is sufficient to extract the color information.

During updates, the red-black tree relies heavily on the colors of the tree, and therefore the folded tree also does. The restructuring of the unfolded red-black tree uses color changes and rotations, which have limited effect on the unfolded tree. For color changes, the partition of nodes in the lower chunks may be altered, requiring restructuring, and during rotations of the unfolded tree, multiple entire folded paths needs to be reorganized.

It is my belief that the folded tree may be simplified, along with the update routines, by not basing it on a red-black tree and an unfolded view, but instead using the folded structure itself to maintain balance. In this simplification, the size of the chunks may be used to allow for incremental backwards updates, as in a binary counter, which ensures the update time is amortized $\mathcal{O}(1)$. Each metanode then only needs one bit for the symmetry of the folding, which I believe is still required. I leave it as an open problem to design this simplified structure.

20.2 Overview of Finger Searching

As stated in the introductory text of Chapter 9, finger searching on binary trees cannot be made to run in $\mathcal{O}(\log d)$ time, however, this does not imply that finger searching is impossible on regular binary trees. We will assume in the following that the binary trees are balanced, in the sense that the height is $\Theta(\log n)$ when the tree contains n elements. In the worst case, finger searching may take $\Omega(\log n)$, which happens when searching from the root node to its successor node, as the successor must be at the bottom of the tree, which, in the pointer model of the tree, is relatively far away. This is also the worst case difference in the runtime to the efficient $\mathcal{O}(\log d)$ finger search time, as the rank difference is 1, and the finger search therefore is expected to run in $\mathcal{O}(1)$ time.

Brodal describes how to perform finger searching on regular binary search trees in [29, Section 11.4.1]. Let x be the node the search starts from, and let y be the node the search ends at. In order to traverse the tree from x to y in as few steps as possible, the search must search upwards, until some *turning* node is reached, and then search downwards. This turning node must for the shortest of such path be the *least common ancestor* (LCA) node of x and y . Let l denote the total optimal distance from x to y over the LCA node. However, as Brodal describes, it cannot be determined which node is the LCA during the search. It can, however, when searching up the tree, be determined if the current node is a *candidate* for the LCA and at the same time reject the last candidate LCA. Therefore, determining the LCA can only be done for certain when the upwards search terminates at the root of the tree, at which point it would have been just as effective to simple search down from the root at the beginning.

Brodal therefore suggests a simultaneous search downward from the last LCA candidate, at the same time as the upwards search. The downwards search from each wrong candidate is bounded by the correct upwards search from x for the right LCA. When the right LCA node is reached it will then never be rejected. The further upwards search from the LCA is then bounded by the correct downwards search

towards y . In total the algorithm is running in the optimal $\mathcal{O}(l)$ time. This algorithm is therefore an optimal algorithm on regular binary search trees, where no additional information is stored in the nodes. However, as mentioned, there is no guarantee that $l = \mathcal{O}(\log d)$, and therefore the algorithm is only optimal in regard to the path length, and not the rank difference.

To create trees, which have efficient finger searching with respect to the rank difference, the results of the literature created trees with additional information in the nodes. For example, the level-linked (a, b) -trees by Huddleston and Mehlhorn [77] requires maintaining extra pointers for each node, which links the *levels* of the tree. The search consists of traversing up the tree until either the current node, or its neighbor on the same level contains the value to locate, at which point the search completes by searching downwards. These pointers allow for efficient finger searching, as the shortcut over a level results in not needing to traverse up to the LCA, but simply to the first level k where the distance from x to the target node y is $\mathcal{O}(2^k)$, effectively performing an exponential search [21]. While the searching routine is straight forward, the level linked pointers must be maintained during updates to the tree, for each node touched during the rebalancing of the tree.

In contrast, our paper [40] present a finger search algorithm, which only requires each node to contain a pointer to its predecessor and successor node, essentially adding the nodes to an ordered linked list. During updates, only constantly many nodes need to update these extra pointers, to account for the change, however, the rebalancing does not need to care for these extra pointers, as the sorted order of the nodes are unchanged. The finger searching algorithm presented only requires that these pointers are available in $\mathcal{O}(1)$ time, and that the tree is “balanced”, in the sense that for each node, the height of the shortest and longest path to a leaf is a constant fraction apart. We shall denote this constant as the *balance factor*. The scheme therefore directly applies to red-black trees (Guibas and Sedgewick [74]), as well as AVL trees (Adelson-Velsky and Landis [5]), by adding in the extra pointers.

20.3 Limitations of Finger Searching

From a purely theoretical point of view, we may compare the time of a *regular search* starting at the root of the tree to a *finger search* starting at some given node, both searching for the same value. Theoretically, the regular search is completed in time $\mathcal{O}(\log n)$ and the finger search in $\mathcal{O}(\log d)$, and as the rank difference of any values in our tree may be at most n , then clearly finger searching is always the better option. Unpacking the $\mathcal{O}$ -notation, however, reveals a culprit, which may intuitively be seen as follows: When performing a finger search, the search starts from some node, and proceeds up the tree until a turning point is found, where the search then continues back down the tree. This results in traversing both up and down the tree. Immediately, we may conclude that if such a search starts at the bottom of the tree and must traverse up more than half of the tree height, then clearly searching from the root is faster. Assuming that the tree is a complete binary tree over n nodes of height $\log n$, then only

searching halfway up the tree, only allows for finger searching from some node to be efficient for $\sqrt{n}$ many other values of the tree. Therefore, in practice, finger searching may not always be efficient, but is only efficient for smaller values of d .

If we do roughly unpack the $\mathcal{O}$ -notation, and let the regular searching constant be c_r and the finger searching constant be c_f , then finger searching is only efficient when

$$c_f \log d \leq c_r \log n \quad \Leftrightarrow \quad d \leq \sqrt[c_f]{n^{c_r}}.$$

As c_r only depends on the balance factor of the tree, this constant is typically small, where the constant of the finger searching may be much larger, as the algorithm needs to be sophisticated. This therefore limits the effect of finger searches in practice.

20.4 A Simple Optimization to Finger Searching with Predecessor and Successor Links

Our paper [40] present a new algorithm for finger searching in a balanced binary search tree, when the successor and predecessor of any node can be found in $\mathcal{O}(1)$ time, which is described in [40, Section 3]. As mentioned, the balance is defined such that for any node, the difference in the height of the shortest and longest path to a leaf is a constant fraction apart. This balance is used for two things in the analysis of the runtime. Firstly, it prompts that the time spend to walk up to a given node is within a constant factor of searching downwards from that node. Secondly, it shows that any node of height h has at least $2^{\Omega(h)}$ nodes in the subtree spanning from the node. This is then used to argue for the minimum number of nodes a given search *cover*, in the sense of the number of nodes reachable in that search.

The algorithm presented by the paper is based on a search, where some amount of time is given to the algorithm. Let the time the algorithm may spend be h . Let the starting node be v and the node searched for be u , where the rank difference of the nodes is d . It then holds that the algorithm spends $\mathcal{O}(h)$ time, and either finds u , or concludes that $d \geq 2^{\Omega(h)}$. Therefore, when $h = \Omega(\log d)$, the algorithm must have found u . If a too large value of h is given, then the algorithm spends too much time to find u , however, if h is too small, then the algorithm is unsuccessful. The main algorithm therefore tries for exponential values of $h = 1, 2, 4, \dots, 2^i$, until u is found. It then holds that for the last search that $h = \Omega(\log d)$, but as the second to last search $h/2 = \mathcal{O}(\log d)$. Therefore, $h = \Theta(\log d)$, and the total time spend is

$$\sum_{i=0}^{\mathcal{O}(\log \log d)} \mathcal{O}(2^i) = \mathcal{O}(\log d).$$

The search algorithm presented in the paper starts each of the searches from node v . However, as hinted to in the text above, the search covers a continuous range of elements, to either find u or conclude that u is not contained in the at least $2^{\Omega(h)}$ elements after v . As the search restarts with a doubling of h after an unsuccessful search from the same node v , then the first range of covered elements, have already

been concluded to not contain u . One simple optimization of the algorithm in the paper, is to start the next search not from node v , but the furthest node v' covered in the previous unsuccessful search. This then prompts the algorithm to not search over the same node space more than once. Intuitively, we then expect from the doubling nature of h , that the search is twice as fast. However, our intuition fails us. The algorithm covers at least $2^{\Omega(h)}$ nodes, but as h is in the exponent, a sum of doubly exponentially many elements are covered. Assume for simplicity that the tree is perfectly balanced, and that exactly 2^h elements are covered by the search. Then the optimized version searches k times, until the sum of the sizes of the ranges cover u , i.e., when

$$\sum_{i=0}^k 2^{2^i} \geq d ,$$

where the paper version searches until the last search is successful, which is when

$$2^{2^k} \geq d .$$

An exponential function grows faster than our intuition truly allows us to understand, and a doubly exponential function grows even absurdly faster than that. Where the difference between a sum of exponential factors is bounded by twice the last term, the sum of the doubly exponential factors is bounded by an exponentially decreasing function in the number of terms. In other words, had the algorithm covered exponentially many elements in the i 'th iteration, then the next search would exactly double that range. For the doubly exponential case, the next search covers *squared* as many elements as the previous term. The “optimized” version is therefore only faster, when the rank difference perfectly allows it to skip a term, which as a function of the rank difference, is for a decreasing amount of nodes.

Note that for the optimized version, if only incrementing the values of h by one instead of doubling, then the final value of h can be shown to be $\Theta(\log d)$, however, the time spend would be $\sum_{h=1}^{\mathcal{O}(\log d)} h = \mathcal{O}(\log^2 d)$. The values of h used in the optimized version must therefore still double, to allow the algorithm to run in time $\mathcal{O}(\log d)$.

20.5 A New Algorithm for Finger Searching with Predecessor and Successor Links

As mentioned, general finger searching in a regular balanced binary search in time $\mathcal{O}(\log d)$ is not possible. However, this is only true for the general case, where we wish to finger search from any node of the tree. Tarjan and Van Wyk [129] observed that in a BST, when searching for a larger element from some node in the tree, then as long as the upwards search path only follows a left spine, then the search time is $\mathcal{O}(\log d)$. This stems from the fact that every step upwards doubles the covered area below, therefore also doubling the number of nodes that must exist between the starting node and the goal. Specifically, this allows for finger searching in a BST from the smallest and largest value. Tarjan and Van Wyk dubbed this scheme as

heterogeneous finger search trees, as they allow finger searching from some specific nodes, but not all.

This observation may be used to allow for designing a new finger search algorithm, which allows searching from any node of the tree, when predecessor and successor links are available for the nodes. We shall assume for the following that we start at some node v and search for a node u of greater value, where the rank difference between v and u is d . The algorithm is symmetric for the case when u is smaller than v . As in the algorithm presented in our paper [40, Section 3], we shall first, if v is not a leaf, use the successor link to step to a leaf, to ensure that the search starts from the bottom of the tree. The search then proceeds as follows. Let the current node the search is at be c . If then c is a left child, we take a left spine step to the parent c' . Otherwise, c must be on a right spine. Walking up this spine is not exploring nodes of greater values, and therefore does not help in finding u . We shall therefore start a *chase*, to find the upper left spine the right spine is stemming from, to allow exploring nodes of greater values. This chase is illustrated in Figure 20.1. In the chase, the algorithm simultaneously walk up and down the right spine containing c , using the parent and right links of the nodes. At some point one of the following two things will happen. Either the upwards search steps out of the right spine, and take a left spine step to some node c' , which can be detected by the previous node being the left node of c' . Otherwise, the downwards search cannot step further down right, in which case, the successor link of the down node must point to node c' , which similarly makes the algorithm take a single left spine step.

After the chase concludes by taking a left spine step, we must then verify that the search is allowed to be at the current height at node c' . Let the current node c' have height h' , and the previous node c have height h . If the searched value is greater than the value of the right child of c' , denoted as c'_r on the figure, then it must hold that the rank difference between v and u is at least $2^{\Omega(h')}$, prompting the search to be allowed to be at the current height h' . We shall then denote node c' as *verified*. Otherwise, it holds that u is either smaller than c' , or between c' and c'_r . If the first case holds, then u must be between the previous visited node c and c' . As the height of c is h , and as c must be a verified node, then the procedure may terminate with a standard downward search from c . If the second case holds, then u is in the left subtree of c'_r . However, as the height h' of c' is not verified, a standard downwards search may take more than $\mathcal{O}(\log d)$ time. This may be circumvented by following the successor link of c' to node s , as s must then contain the smallest value in the left subtree of c'_r . As u is contained in this subtree, then by searching up the left spine of the tree from s , using the algorithm of Tarjan and Van Wyk [129], the search then terminates successfully in time $\mathcal{O}(\log d')$, with d' denoting the rank difference between s and u .

To analyze the time complexity, we shall split the search into the upwards search on verified nodes, and the terminating search. For simplicity, we shall assume that the tree is perfectly balanced. First, for the upwards search, we show that when the upwards search is at a verified node of height h , it holds that at most $2h$ time was spent to reach that node, and it holds that the rank difference $d \geq 2^{\Omega(h)}$. The time spent on the upwards search follows by induction. Assume that the search is at some verified

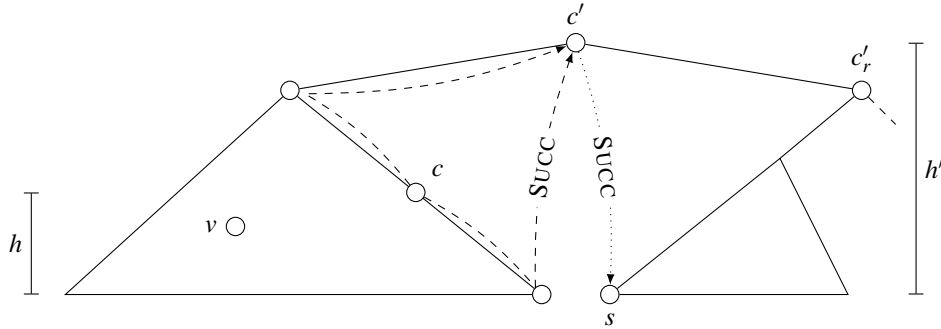


Figure 20.1: Chasing a right spine from node c of height h to the upper left spine node c' at height h' . The dashed arrows show the nodes visited by the upwards and downwards search.

node c of height h and takes a left spine step to a node c' of height h' . If the step is following the parent up a left spine, then $h + 1 = h'$, and at most $2h + 1 \leq 2h'$ steps are taken. Otherwise, the step is up a right spine. If the upwards search finds the node first, then at most $h' - h$ additional steps from c to c' are made. Otherwise, it must hold that $h' \geq 2h$, and the downwards search reaches the node first, spending at most h steps to reach from c to c' . In all cases, the number of steps used to reach c' is at most $2h'$, as the right spine chase ends when the first step to c' is taken. To then verify c' , $\mathcal{O}(1)$ time is spent. For any verified node c of height h it holds that $d \geq 2^{\Omega(h)}$, as all elements of the right-left subtree separates v and u . Therefore, $h = \mathcal{O}(\log d)$ time is spent to reach the verified node. The final search then either spends h additional time in a downwards search from some verified node of height h , or a search up a left spine from some smallest subtree node s , which takes time $\mathcal{O}(\log d')$ time, when the rank difference of s and u is d' . As the rank of s must be greater than or equal to v , then $d' \leq d$. In total $\mathcal{O}(\log d)$ time is spent for the algorithm to perform the finger search.

This approach was developed in collaboration with Gerth Stølting Brodal, after the paper was published, and was only ever recorded on a whiteboard, which prompted me to name the algorithm in my internal notes as the “Whiteboard version”, as opposed to the “Paper version”, which is described in the paper. This algorithm uses the same setup (successor and predecessor links) as the Paper version and has the same runtime ($\mathcal{O}(\log d)$), so one may beg to ask: why even design such another algorithm? When designing the Whiteboard version, we tried to rid of the behavior of the Paper version, in exponentially searching for the value of $\log d$, having to restart the search over and over again until it finds the target node. The first inspiration was that searching up a left spine for a larger value does not need to do anything clever, as it is already efficient. We next observed that in the Paper version, that if the first search conducted in the first subtree on the left is unsuccessful, then there are two problems which can be tackled. Firstly, if the top node r_ℓ found in the left subtree search is not far from c , then traversing down the right spine to ℓ to jump to c via the successor link is less efficient than searching upwards. Secondly, by comparing the searched value of u

to the right child of c , the second search may be entirely omitted. At the same time, the search may continue upwards from c , which the Paper version does not utilize, as it simply restarts the process. These observations are exactly the chase of the right spine and the verification of the top node c , that the Whiteboard version does. So, apart from the Whiteboard version being an optimized version of the Paper, another nice implication of the new algorithm is, that the version presented in the paper is not the only possible way to use the successor and predecessor links to perform finger searching. This shows that there is not one unique way to use the links, and possibly there are more algorithms using these links, which shows their versatile behavior.

In reading references for this dissertation, I came across the paper by Blelloch, Maggs and Woo [24], which describes a structure that allows finger searching on a balanced tree, without altering the tree. In order to do so, they maintain a *hand* for each finger v , consisting of $\Theta(\log n)$ pointers into the structure. The hand allows for efficient finger searching in $\mathcal{O}(\log d)$ time from v , and in the same time, altering the hand to be a finger to the found node. The hand consists of two symmetric structures, allowing for searching for values smaller and greater than v . The structure, which allows for searching for greater values, consists of a queue of the *right parents* of v , which are the ancestors of v of greater values, including v . Additionally, the structure contains for each right parent, a pointer to the bottom of the right-left spine of that parent. To then search for a given value u , the parents are traversed in the queue from the bottom up, until an upper bound node on u is found, at which point the right-left spine of the found upper bound node is searched along from the bottom up. The right parents of v is exactly the possible verified nodes of the Whiteboard version of finger searching, and the procedure, apart from finding the right parents, is identical. The structure of Blelloch, Maggs and Woo is efficient in the sense that they do not need to alter the balanced tree. However, they need $\Theta(\log n)$ space for each finger, leading to $\Theta(n \log n)$ space if a finger needs to be maintained for each node. In comparison, the algorithms we present alter the tree by adding successor and predecessor links, which only uses $\Theta(n)$ space. The cutoff between the two structures is at $\mathcal{O}\left(\frac{n}{\log n}\right)$ fingers, where they both use linear space. This is already many fingers, and assuming that not many fingers are needed, their structure is indeed space efficient. This algorithm further supports the idea that there might be more ways of using the successor and predecessor links, possibly based on other finger searching techniques.

20.6 Finger Searching in Folded Trees

In this section we shall explore how the presented finger search algorithms of the paper [40] and in Sections 20.4 and 20.5 above, behave on the folded tree. Further, we shall optimize the finger searching for specifically folded trees.

The discussed finger searching algorithms, are efficient when access is possible to the parent, left and right child, as well as the predecessor and successor nodes using $\mathcal{O}(1)$ pointer accesses, and when the relative height of the shortest and longest path from any node is within a constant. The folded trees of the paper offers exactly the

guarantee on the fast access to the predecessor and successor, without maintaining exact pointers to them. However, the balance factor on folded trees are not a constant, as there exists nodes where the height of one subtree is constant, but the height of the other is $\Omega(\log n)$. Instead, we view the folded tree as a conceptual unfolded tree, which is a regular red-black tree, which therefore guarantees the balance factor on the height of the unfolded nodes. Additionally, in the folded tree the unfolded parent as well as the left and right child is within distance $\mathcal{O}(1)$, and the predecessor and successor remains the same nodes. The finger search algorithms utilizing predecessor and successor pointer may therefore be applied to the folded trees, which yields finger searching with runtime $\mathcal{O}(\log d)$. Note, however, that the constant does increase by multiple of the constant hidden in the time to find the unfolded parent, left and right child, as well as the predecessor and successor nodes for any node in the folded tree.

If simply searching for the successor y to some node x in the folded tree, we know that this is within a distance $\mathcal{O}(1)$. Assume that we know the value of the successor y up front. Then by applying the finger searching algorithm traversing over the LCA starting in node x searching for the value of y , we may find the successor y within $\mathcal{O}(1)$ time. The rank difference of these nodes are 1, and the finger searching for the immediate successor is therefore efficient, even without taking advantage of the folded structure within the algorithm. In general, for the folded tree, elements, that are at the bottom of long paths in the unfolded tree, are folded to be close to the root of the path. In the finger search algorithm using LCA, the only time the algorithm wastes over the needed $\Omega(\log d)$ time, is the time spend traversing up the tree from the first time two visited nodes are neighbors in a level of the tree. This crawl is a walk up a single long path, which must be folded to be short in the folded tree. This prompts the question: Is the regular LCA algorithm efficient to simply apply on the folded trees?

To show that this is indeed the case, we can use a simple *existential* argument. Let the search start at some node x and terminate at some other node y . By symmetry, we may assume that $x < y$. Let the rank difference between x and y be d . When finger searching using the LCA algorithm, the algorithm traverses from x towards the LCA node, and then down to y . The time spend is proportional to the distance from x to the LCA and from y to the LCA.

Any of the discussed finger search algorithms show that only access the predecessor and successor is sufficient to finger search in $\mathcal{O}(\log d)$ time. The folded tree provides access to the parent and the left and right child node in the unfolded tree, as well as the predecessor and successor, all in $\mathcal{O}(1)$ time, by following $\mathcal{O}(1)$ pointers. As only the parent and the left and right child in the folded tree can be found, it must hold that the distance between the start and end node for each of these five operations are $\mathcal{O}(1)$ continuous nodes apart. By using the finger search algorithm, there then exists some path in the folded tree from x to y which is of length $\mathcal{O}(\log d) \cdot \mathcal{O}(1) = \mathcal{O}(\log d)$. On this path, the LCA of x and y in the folded tree must be visited. Ergo, the distance from x to the LCA and symmetrically the distance from the LCA to y is $\mathcal{O}(\log d)$ in the folded tree. The regular LCA algorithm may therefore be applied on the folded tree, where it will spend $\mathcal{O}(\log d)$ time.

The hidden constant, using this argument, is defined from the hidden constant of

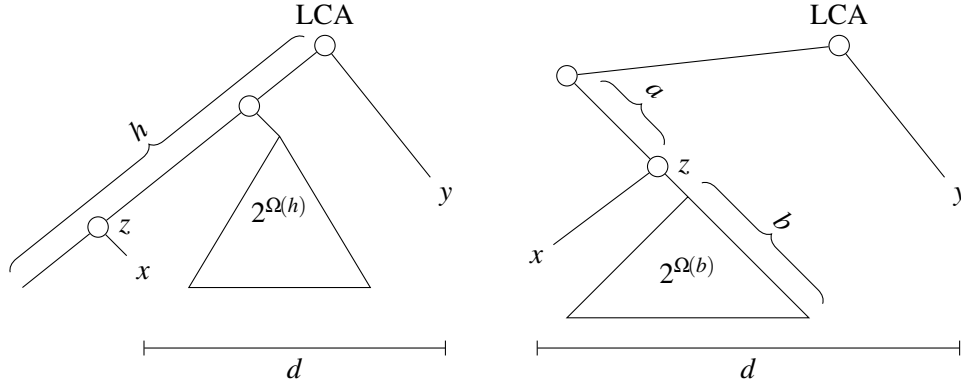


Figure 20.2: Searching from x to y in an unfolded balanced binary tree, where the rank difference is d . The traversal starts upwards from x up to the LCA node, and then downwards to y . (left) Along the up traversal from x the node z is the first node visited on the left spine up to LCA, which is at height h . The size of the right subtree of the left child of LCA is $2^{\Omega(h)}$, and all nodes in this subtree has rank between x and y . (right) Along the up traversal from x the node z is the first node visited on the right spine of the left subtree of LCA. On the spine, there are a nodes on and above z , and b nodes below. The size of the right subtree of z is $2^{\Omega(b)}$, and all nodes in this subtree has rank between x and y .

the chosen finger search algorithm, which we may expect to be large, as well as the constant of the LCA finger search algorithm. This constant may be bounded more tightly, by the below *constructive* argument, taking into account how the path to the LCA may be. We shall argue for the distance from x to the LCA in the folded tree, as the distance from the LCA to y holds by a symmetric argument.

If x is the LCA node or the left child of LCA in the unfolded tree, then the distance is $\mathcal{O}(1) = \mathcal{O}(\log d)$. Otherwise, x may reside in the left or right subtree of the left child of LCA in the unfolded tree. If x is in the left subtree, then the final traversal towards LCA is along a left spine in the unfolded tree, consisting of more than one node. This case is illustrated in Figure 20.2 (left). Let z denote the first node visited on the left spine of the unfolded tree, and let h denote the height of LCA. It must then hold that the right subtree of the left node of the LCA contains $2^{\Omega(h)}$ nodes, which in rank is between x and y . Therefore, the traversal in the unfolded tree from x to LCA is of distance $\mathcal{O}(h) = \mathcal{O}(\log d)$. In the folded tree, the LCA of x and y may be a different node LCA^f . By the folded tree structure, then LCA^f may only be a node between z and the LCA on the unfolded left spine. The distance between any pair of nodes in the folded tree is at most a constant factor larger than the distance in the unfolded tree. As the unfolded tree allows traversing this path, the folded tree may therefore traverse up to LCA^f . The distance traversed upwards in the folded tree is then bounded by $\mathcal{O}(\log d)$.

Otherwise, x resides in the right subtree of the left child of LCA in the unfolded tree. The final step to the LCA in the unfolded tree from x is therefore on a left spine,

but the traversal before that is along a right spine. The unfolded tree for this setup is illustrated in Figure 20.2 (right). Similarly to before, the LCA in the folded tree LCA^f may be different from the LCA in the unfolded tree. However, as the traversal enters the same left spine as LCA in the node immediately before it, the traversal between the LCA and LCA^f is bounded by $\mathcal{O}(1)$. Let the first node entered on the unfolded right spine of the unfolded tree when traversing from x be z . Let the number of nodes on or above z on the right spine be a and the number of nodes below be b . By the balanced property, the number of elements in the right subtree of z is $2^{\Omega(b)}$, and all of these nodes must have a rank between x and y , which resolves in $b = \mathcal{O}(\log d)$. The balance property also ensures that the height of any node in the left subtree of z , and therefore the height of x is bounded by $\mathcal{O}(b) = \mathcal{O}(\log d)$. In the folded tree, the height of any node may only increase by a multiplicative constant, and the traversal time up the tree from x to z in the folded tree is therefore also $\mathcal{O}(\log d)$. We can therefore disregard where x actually resides in the left subtree of z . Note that in the folded tree x may not visit z first, but there is $\mathcal{O}(1)$ nodes between the node x first visits and z . There may now be two cases: z is an upper node on the unfolded right spine or z is a lower node. If z is an upper node, then by the folded structure, it must hold that $a = \mathcal{O}(b) = \mathcal{O}(\log d)$. As the height of the upper nodes only multiplicative increase by a constant in the folded tree, the distance from z to the LCA in the folded tree, and therefore also the distance from x to the LCA, is $\mathcal{O}(\log d)$. Otherwise, z is a lower node. By the folded structure, as the lower nodes are reversed towards to the root of the path, then the distance from z to LCA is bounded by $\mathcal{O}(b)$, and therefore the distance from x to the LCA is similarly bounded by $\mathcal{O}(\log d)$.

By symmetry, the distance from the LCA to y in the folded tree is $\mathcal{O}(\log d)$. This therefore concludes that the distance from x to y in the folded tree is $\mathcal{O}(\log d)$, and the finger search algorithm traversing over the LCA may therefore be applied to gain a finger search algorithm on folded trees with runtime $\mathcal{O}(\log d)$. Note that the constant in this analysis is bounded by performing two searches, one upward from x and one downward to y . The folded tree increases the length of any paths by a small constant, as lower chunks and upper nodes are intermixed. The metanodes further increases the height by a factor 2 as well as a small additive factor. This allows to more tightly bound the constant, than simply by the constant of the finger search algorithm of the paper.

20.7 Implementation and Experiments

The implementation discussed in this section is available at [109].

As part of writing the paper, Brodal and I convinced ourselves that the structure we had created would be easy enough to implement. I therefore spend quite a while trying to implement the structure in Python. The implementation never completed, as time constraints forced us to work on the paper at the end. However, the implementation was not a waste of time, as it showed that some of the cases we had constructed initially could be removed.

For instance, we did not initially require that the number of upper nodes and lower chunks on a folded path was perfectly equal, when not considering the edge case of a single node, and instead allowed to have one extra upper node than number of lower chunks. In this scheme we still needed the possibility of the last lower chunk to be a single red node, so it did not simplify the structure to have all lower chunks be of the same kind. Additionally, the implementation revealed that determining if the extra upper node was indeed an upper or lower node proved to be hard. This led to the version presented in the paper, where the last lower chunk consists of whatever is left over when folding the path. It can be shown that the last lower chunk consists of at most three nodes, which is of constant size, thereby allowing to compute on the last lower chunk in $\mathcal{O}(1)$ time. This further allows for a more simple check on if a node is an upper or lower node, as any upper node on a left (right) path either is the top node on the path, which parent is not existing or is a right (left) path node, or it must be the case that the left (right) child is a left (right) path node.

After completing the paper, I have spent some additional time continuing the implementation, to obtain a proof of concept code base. In this, I have implemented the finger search algorithms described earlier in this chapter and in the paper, to compare their runtimes. To avoid timing issues in the experiments, I have measured the runtime using software counters on how many pointers and bits are accessed and modified, as well as on counting comparisons. These counters are easily implementable in Python, using Python *properties*. In essence, they allow accessing and updating attributes via a defined function, in which the software counter can be placed on the lowest level of the nodes.

To implement the finger search algorithms, we must first determine what we want to return, when the searched value is not contained in the structure, as this is not well-defined in the descriptions of the algorithms. Here I have chosen to return the predecessor node in case the value was not found. For the search trees where the successor of any node may be accessed in $\mathcal{O}(1)$ time, this choice is arbitrary. Note however, that since the successor is accessible in $\mathcal{O}(1)$ time, then given a node, it can be determined if a given node is the predecessor node, by comparing the searched value to the value of the node and the successor node.

For simplicity and scalability, I opted out of using metanodes, and instead chose to use regular red-black trees with explicit predecessor and successor pointers in the nodes. These pointers are relatively easy to maintain under insertions and deletions in the tree, and therefore allows for creating the red-black trees from simple updates. For further simplicity, the tree contains all integer values from 0 to $n - 1$. The total size is therefore n , and the rank difference between pairs of nodes is easy to calculate, as the rank of a node is equal to the value of the node. For the queries in the experiments, only exact queries are made, to not let the choice of returning the predecessor instead of the successor affect the measured outcome. Further, this guarantees that a node is always returned, otherwise the edge case that the predecessor of the first node does not exist occur. I made two options for creating trees; one where the values are inserted in random order into the red-black tree, and one where a complete tree of height h is created, with all black nodes. The last option is to test if runtimes are dependent on

the balance factor or not.

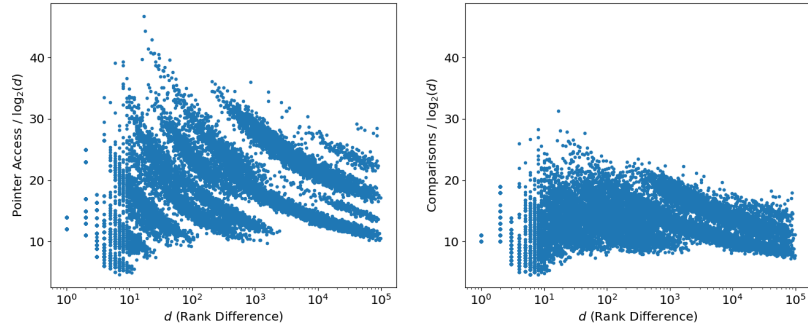
Paper version. First, the Paper version was tested. This algorithm is described in [40, Section 3]. In the Paper version, the search may terminate early on a downwards search if the result is found. To better measure the upper bound on the runtime, I therefore chose to only search for leaves of the tree, which therefore cannot terminate early. By the balance factor of red-back trees, we may define leafs to be any nodes where neither the left nor right child exists. To then obtain a good spread over the rank differences, I tried exponentially increasing values for the rank difference, with base 1.3. The search then starts from some node exponentially far away from the leaf to search for.

On the produced plot in Figure 20.3a, each dot represents a single search. The x -axis denotes the rank difference of the search performed, and is logarithmic. The y -axis denotes the runtime, be that either the number of pointer access to the left or comparisons to the right. As the theoretical runtime is $\mathcal{O}(\log d)$, the runtime is divided by $\log d$, such that the plot shows the constant of the $\mathcal{O}$ -notation. As it can be seen, the constant on the pointer access is larger than the constant on the comparisons. This is due to the nature of the algorithm, which performs multiple upwards and downward traversals. On the upwards traversals, the number of steps is simply h , and no comparisons are performed. On the downwards searches, comparisons are performed to navigate the search. This unbalance between the comparisons only being performed on the downwards traversals is then leading to the difference in the constant.

On the initial plot over the number of pointer access and comparisons, it is clear to see that the search times are grouped. This is expected, as the Paper version tries different values of h until the final call finds the searched node. When trying an additional h value, we therefore expect a jump in the runtime. At the same time, we see that each group drops as the rank difference increases. This similarly is expected, as a search with a given h value covers a large range of nodes, and therefore, as the rank difference increases, but the runtime does not, as the same final value of h is used, then the constant drops.

To demonstrate that the groups are based on the final value of h , the dots are colored to group by h value, which can be seen in Figure 20.3b. However, inside a single group, there still appears to be further groups. This is similarly expected, as a successful search performed with a given h value may terminate in the first downwards search, or in the second downwards search. If only performing one downward search, surely the runtime is smaller. The experiments are therefore further grouped into the number of total downwards searches performed, which can be seen in Figure 20.3c.

There, however, still appears to be two groups inside each of these groups, which is unexpected. The only option left as to why this grouping appears is from the symmetry of the algorithm, i.e., whether the searched value is smaller or greater than the value of the starting node. By symmetry, there should not be any difference. However, by further grouping by the direction of the search, which is shown in Figure 20.4a, for



(a) Ungrouped experiments.

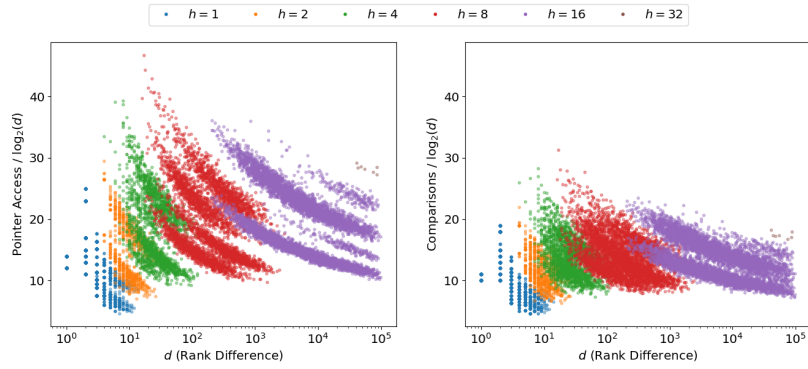
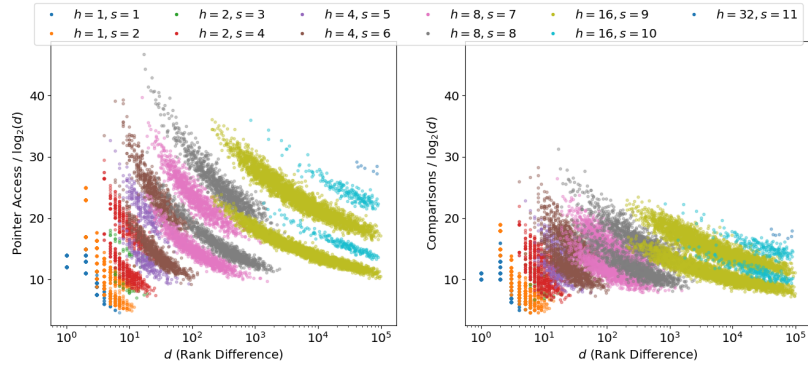
(b) Grouped by final value of h .(c) Grouped by final value of h and total number of downwards searches s performed.

Figure 20.3: Plot of measured runtime of the Paper version, when only searching for leaves. The trees tested on are red-black trees with explicit predecessor and successor links. Test trees of size $n = 100000$ are created by inserting the values in random order over 10 random seeds. Each dot corresponds to a search. The x -axis is logarithmic and shows the rank difference d . The plots on the left shows the number of pointer accesses made divided by $\log d$ on the y -axis, and the plots on the right the number of comparisons.

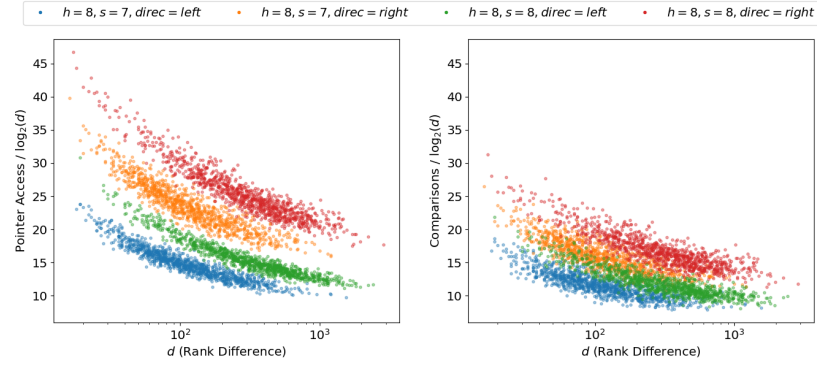
the groups with final value of $h = 8$, we see that the four groups is exactly matching with the direction being the other splitting criteria for the group. This is unexpected, as the symmetry should not care on the direction. The explanation is therefore to be found in the implementation, and not the theory, and indeed the code is the culprit. As mentioned earlier, checking if a node is the predecessor node of the searched value, the successor link is used, and the value is checked to be between the values of the two nodes. This, in Python, is done with the code below.

```
def _is_the_pred_node(node, value):
    return node.value <= value and \
           (node.succ is None or value < node.succ.value)
```

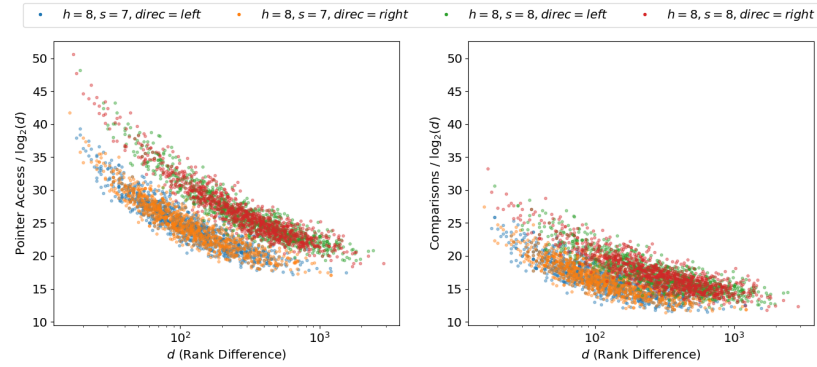
Here, the 'and' in Python is short-circuiting, meaning that if the left side of the expression is false, then the second half is never executed. When searching for values larger than the searched value, then for all failing h values, it must hold that all the nodes checked are smaller than the searched value, and the left-hand side of the expression is therefore always true, leading to the second half of the expression always being executed. When searching for values smaller, then the left-hand side is always false for all failing h values, and therefore the second half of the expression is never executed. This is enough to separate the runtimes when searching for values smaller or greater. To demonstrate this, the short-circuiting of this specific check was removed, to always execute both sides of the expression, and as expected, this aligns the runtime for searching for smaller and larger values, such that there is no difference, which is shown in Figure 20.4b. The grouping is then only on the final value of h and the total number of downward searches performed.

The groups are wide, and has a large concentration around the middle. This stems from the balance factor of red-black trees, which allows paths to the same node to vary by a factor 2. Therefore, when searching exactly h levels up, the consecutive downward search may spend between $h/2$ and $2h$ time, dependent on how many red nodes are on the upwards and downwards path, with the extremes not being the regular case.

To demonstrate this, the experiment was rerun on perfectly balanced trees, i.e., complete trees, where all nodes are black. The height of the tree is 20, chosen to have a sufficiently large tree, and such that searches requiring the final value of $h = 16$ would not reach the root, but $h = 32$ will. The search remains only for a leaf, but the starting node may be any node in the tree. The termination check is forced to not short-circuit, to not get a widening of the group, as in the last downwards search the termination check may or may not trigger the second half of the expression, yielding to a difference in the runtime. The searches are then grouped by the final value of h and the total number of downwards searches performed. The resulting plot can be seen in Figure 20.5. On the first plot in Figure 20.5a we see that the groups have collapsed to be narrow. This supports the hypothesis, that the wideness of the groups on searches in regular red-black trees stems from the balance factor. The groups are, however, not a single line when observing the number of pointer access, which stems from the search starting at any node, which therefore also may be an internal node.

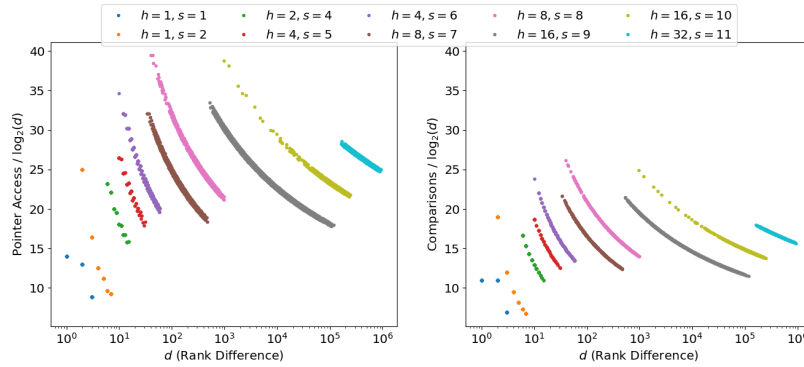


(a) With short-circuiting the termination check.

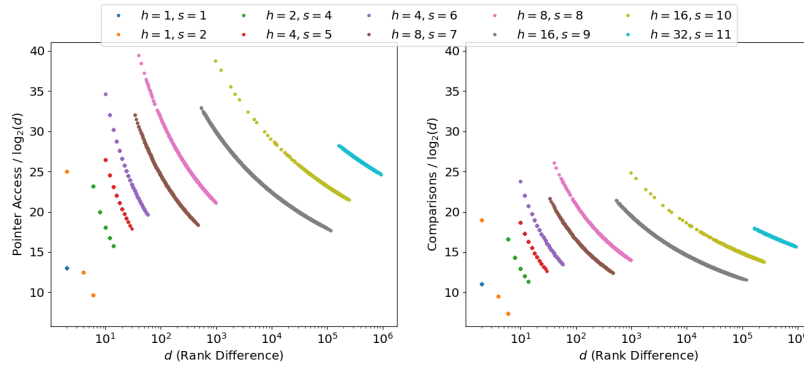


(b) Without short-circuiting the termination check.

Figure 20.4: Plot of measured runtime of the Paper version, when only searching for leaves. The trees tested on are red-black trees with explicit predecessor and successor links. Test trees of size $n = 100000$ are created by inserting the values in random order over 10 random seeds. Each dot corresponds to a search. The x -axis is logarithmic and shows the rank difference d . The plots on the left shows the number of pointer accesses made divided by $\log d$ on the y -axis, and the plots on the right the number of comparisons. Only experiments where the final value of $h = 8$ is shown. The experiments are then grouped by the total number of downwards searches s performed and the direction of the search, i.e., if smaller or greater values than the starting node is searched for.



(a) Starting from any node.



(b) Starting only from a leaf.

Figure 20.5: Plot of measured runtime of the Paper version, when only searching for leaves, with short-circuiting the termination check. The trees tested on are complete red-black trees with explicit predecessor and successor links, of height 20 with size $n = 1048575$. Each dot corresponds to a search. The x -axis is logarithmic and shows the rank difference d . The plots on the left shows the number of pointer accesses made divided by $\log d$ on the y -axis, and the plots on the right the number of comparisons. The experiments are grouped by the final value of h and total number of downwards searches s performed.

Such a node requires the algorithm to take an initial step to a leaf of the tree, which yields a difference depending on the type of the start node. By only allowing the starting node to be a leaf, the resulting plot can be seen in Figure 20.5b. On this plot the number of comparisons performed collapse to a line for each plot, thereby supporting that the difference was the leaf starting node.

Optimized Paper version. Next, experiments on the Optimized Paper version was conducted. This algorithm was discussed in Section 20.4. The results can be seen in Figure 20.6. To avoid many groups, the short-circuiting on the termination check was

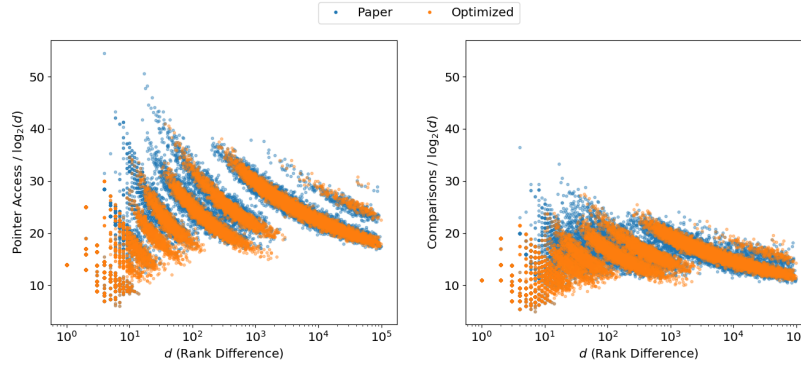
disabled for these experiments. The trees, start nodes, and searched values used is identical to the experiments on the Paper version, for ease of comparison.

In Section 20.4, it was discussed that the optimized version actually does not improve the overall runtime. That section was written after conducting these experiments, as a way of explaining why the plots in this section look as they do. Because, just like the theory tells us, the plots shows that there is not much improvement from the Paper version to the Optimized Paper version, as the dots are almost grouped at the same locations. However, it can be seen on the plot in Figure 20.6a that the left upper part of each group mostly only consists of the experiments from the Paper version, which hints that the Paper version needs to perform more searches for these experiments than the Optimized Paper version needs. Further, remember that the y-axis is divided by $\log d$, which distorts the intuition a bit. As the experiments performed on the two versions are identical, with respect to the input to the algorithms, we may look for each search performed how the runtimes compares. For each column, the rank difference is the same, and here we see that the start of each group consisting of the Paper version experiments, must have the matching experiment for the Optimized Paper version below it. By using the equal experiment set, we can then for each search experiment performed divide the runtime of the Paper version by the Optimized Paper version, to obtain the improvement factor.

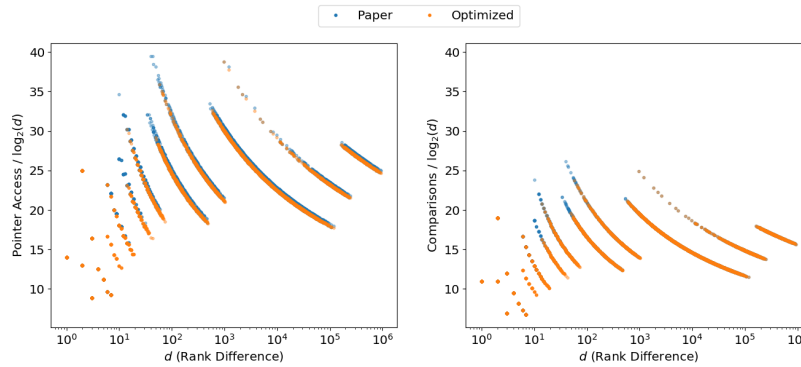
This resulting plot can be seen in Figure 20.7a. Here we can see that most of the time, the improvement of the Optimized version is close to zero, with some rare times, where the Optimized Paper version is twice as fast, and sometimes even 2.5 times as fast. Further, the hypothesis that the Optimized version is faster only for the few rank differences where it performs fewer searches, and that the number of these rank differences decrease over time, is further strengthened, as we see holes in the groups of speed-ups for some ranges of rank differences. Towards the larger rank differences, the time improvement is negligible, simply due to both versions reaching an h value large enough, that the root can be reached. Taking the geometric median of the improvements, we see that the Optimized Paper version on average does not yield a great improvement over the Paper version.

What is surprising about the plot, is the fact that the Optimized Paper version is slower than the Paper version on some instances. According to our intuition, this should not be the case. This can be explained by the balance factor of the tree; the Optimized Paper version searches from new nodes every time the h value increases, and therefore traverses different paths than the Paper version. If these different paths have a worse balance on the downwards search than the path of the Paper version, then they spend more time. This however, is not the only reason why the Optimized version is slower. By coloring the dots by the difference in how many searches the two algorithms perform, it can be seen that the Optimized version actually performs more searches than the regular Paper version for these cases. This is due to the balance factor allowing the Optimized Paper version to traverse far down paths, where the h value does not allow it to traverse far back up, where the Paper version instead would skip over this local deeper part of the tree.

To confirm that the balance factor is to blame for the time when the Optimized



(a) Testing on trees of size $n = 100000$, created by inserting the values in random order over 10 random seeds.

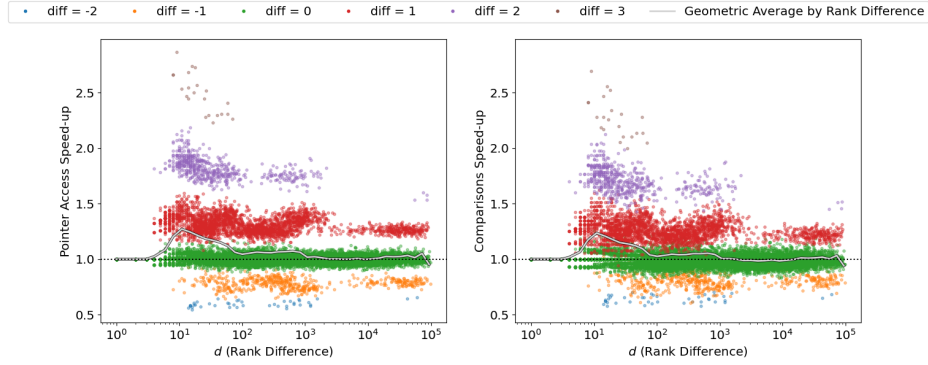


(b) Testing on complete trees of height 20 with size $n = 1048575$.

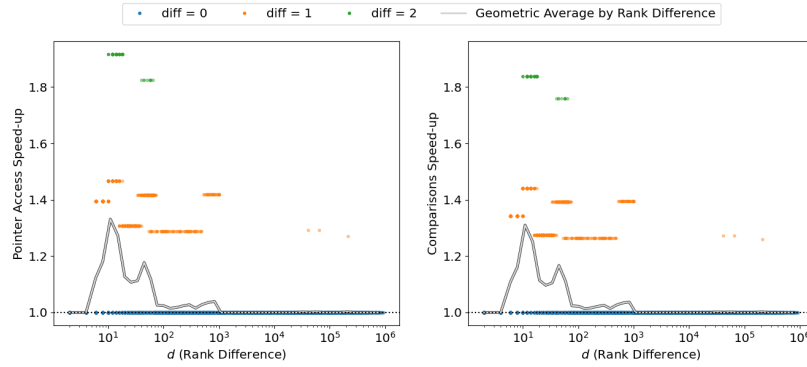
Figure 20.6: Plot of measured runtime of the Paper version and Optimized Paper version, when only searching for leaves, with short-circuiting the termination check. The trees tested on are red-black trees with explicit predecessor and successor links. Each dot corresponds to a search. The x-axis is logarithmic and shows the rank difference d . The plots on the left shows the number of pointer accesses made divided by $\log d$ on the y-axis, and the plots on the right the number of comparisons.

version is slower, the experiment was repeated with complete trees, where the balance factor is 1. The resulting plot of the runtimes can be seen on the plot in Figure 20.6b. Observing the improvement factor seen on the plot in Figure 20.7b, it can be seen that the Optimized Paper version now is no longer slower on any instances, and that it no longer performs more searches. Observing the geometric median, the Optimized Paper version have a larger improvement on some small ranges of rank differences, as expected.

Whiteboard version. The Whiteboard version is next to be tested. This algorithm was discussed in Section 20.5. Similarly to the Paper version, this was tested using red-black trees where the values are inserted in random order. Figure 20.8a shows a



(a) Testing on trees of size $n = 100000$, created by inserting the values in random order over 10 random seeds.



(b) Testing on complete trees of height 20 with size $n = 1048575$.

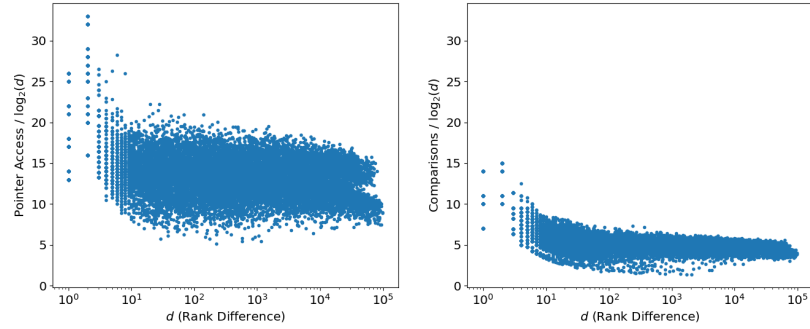
Figure 20.7: Plot of measured improvement in runtime of the Optimized Paper version over the Paper version, when only searching for leaves, with short-circuiting the termination check. The trees tested on are red-black trees with explicit predecessor and successor links. Each dot corresponds to the ratio of the same search performed in the same tree of the two different algorithms. The x -axis is logarithmic and shows the rank difference d . The plots on the left shows the speed-up on the number of pointer accesses made on the y -axis, and the plots on the right the number of comparisons. The experiments are grouped by the difference in the number of total downwards searches performed by the two algorithms, with positive numbers denoting that the Paper version performed more searches than the Optimized Paper version.

plot of the measured runtime, divided by $\log d$, as a function of the rank difference d . Similarly to the Paper version, it can be seen that the plot shows a function bounded by a constant, supporting the theoretical runtime of $\mathcal{O}(\log d)$. And similarly to the Paper version, the constant measured for pointer accesses is larger than the constant for comparisons. This is due to the chases performed by the Whiteboard version being expensive on the pointer accesses performed, but requiring no comparisons. Only the validation checks and the final search requires comparisons.

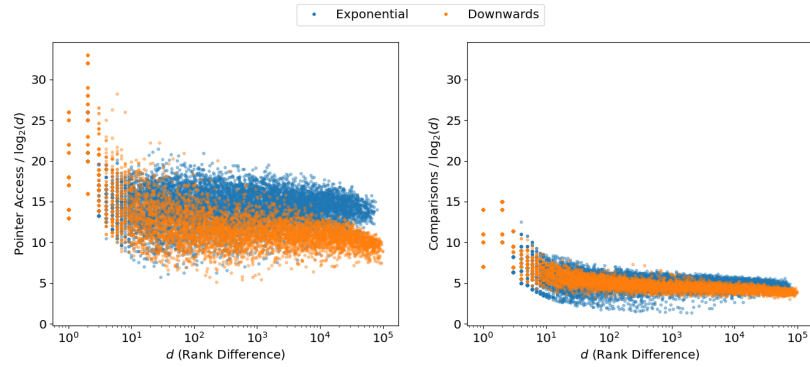
For the Paper version there was a clear set of groups on the plot, which appears to not be present for the Whiteboard version. However, observing the runtime for large values of rank difference on the number of pointer access, reveals some separation. To determine why this separation appears, I tested several features of the performed search to find an underlying parameter causing this separation. At first the hypothesis was that the short-circuiting on the termination check might be causing the separation, but disabling short-circuiting showed no difference. It did however reveal that disabling short-circuiting does not increase the runtime by more than a factor 1.2. This is due to the Whiteboard version only using the short-circuiting function in the final terminating search, thereby not affecting the overall runtime by much. Coloring the dots by the search direction, as was the case with the separation in the Paper version, also shows no separation for the Whiteboard version. The separation only occur for large values of rank difference, which may be requiring the search to traverse over the root for searches across the subtrees spanning from the root, with some searches only existing entirely inside one of the subtrees. Coloring the dots based on if the search traversed over the root similarly showed no separation.

The difference in the groups must then lie in some asymmetry of the Whiteboard algorithm itself. The algorithm may progress upwards by either simple steps or chases. Assume that the search is for a greater value. The Whiteboard version then spends more time doing chases on right spines, than traversing left spines. To test if this caused the separation, I therefore colored the experiments by the number of chases performed, which also showed no separation. We may then observe that when the downwards search of a chase completes first, then the search may jump far up the tree, thereby advancing the search far. Coloring the nodes based on the number of winning downwards searches also showed no separation. Finally, we can observe that the algorithm may terminate by either a final downwards search or an exponential search along a spine of some subtree. According to Section 20.3, then the exponential search is a finger search, which by definition must be slower than the downwards search, as it searches both up and down the tree. However, it is only slower by a constant factor, and the total runtime remains $\mathcal{O}(\log d)$. By coloring the experiments by the terminating search type, the plot in Figure 20.8b is obtained, which indeed shows that this final difference is enough to yield a separation in the runtime.

Comparing the Paper version to the Whiteboard version. From both theory and the above experiments it holds that both the Paper and Whiteboard version has a runtime of $\mathcal{O}(\log d)$. However, the experiments reveals the hidden constants of



(a) Ungrouped experiments.



(b) Grouped by the type of the final terminating search.

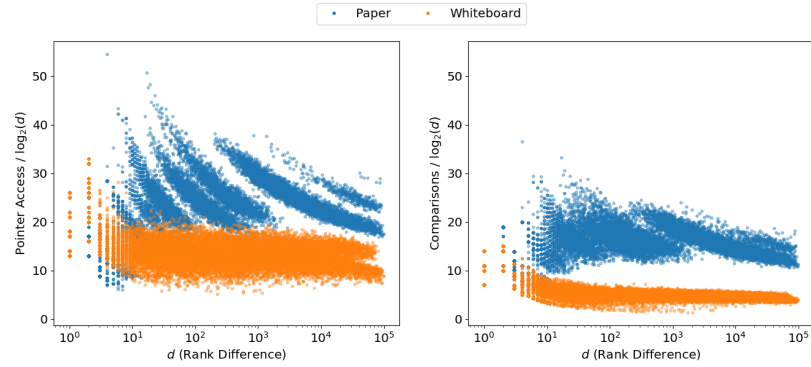
Figure 20.8: Plot of measured runtime of the Whiteboard version, when only searching for leaves. The trees tested on are red-black trees with explicit predecessor and successor links. Test trees of size $n = 100000$ are created by inserting the values in random order over 10 random seeds. Each dot corresponds to a search. The x -axis is logarithmic and shows the rank difference d . The plots on the left shows the number of pointer accesses made divided by $\log d$ on the y -axis, and the plots on the right the number of comparisons.

the algorithms, at least according to one specific implementation. As mentioned in Section 20.5, the Whiteboard version was created in part to improve on the Paper version, and we therefore expect it to perform better than the Paper version. For ease of comparison, Figure 20.9a contains both experiments in the same plot. The experiments run are without short-circuiting the termination check, to not favor searching in one direction.

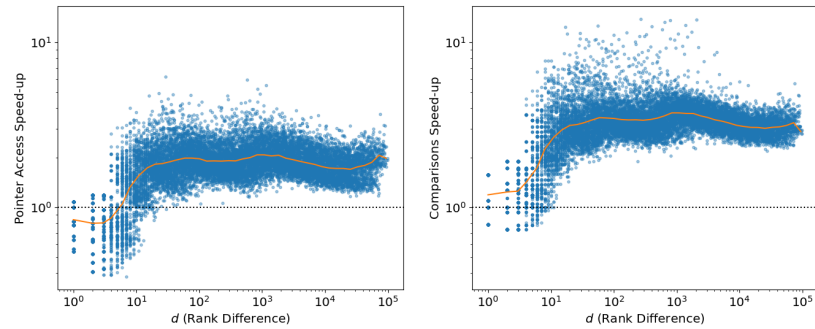
As the experiments performed are identical, we can compare the improvement factor of the Whiteboard version over the Paper version, which results in the plot in Figure 20.9b. Note that the scale of the y-axis is logarithmic, s.t. an improvement of 2 over the equal baseline at 1 matches with a decline of $1/2$ under the baseline. As expected, the Whiteboard version performs best, but only does so for rank differences larger than about 10. Observing the comparison plot, the Whiteboard version has about a factor 2 speed-up in the number of pointer chases and 3.5 speed-up in the number of comparisons, over the Paper version. The factor on pointer chases speed up matches with the intuition that the second search of the Paper version is skipped in the Whiteboard version. The improvement may be in part due to work being skipped, but also the additional work the downward searches performs, as they check every node on the way if it is the correct node to return. Here, the Paper version calls the downwards search often, but the Whiteboard version only calls it once. Note that for rank differences smaller than 10 the Whiteboard version is about twice as slow as the Paper version. This may stem from the Whiteboard version being very effective on long spines, which rarely happens for smaller rank differences.

LCA version Performing finger searches from x to y by traversing in a regular tree over the LCA node, was presented in details in [29, Section 11.4.1], and discussed in Section 20.2. Following the detailed description, this algorithm was implemented, and the measured runtime on red-black trees with values inserted in random order can be seen in Figure 20.10. The measured runtime of each search is divided by the logarithm of the rank difference d of the values x and y of the search to measure the constant of the $\mathcal{O}$ -notation. First we may notice that the number of comparisons is higher than the number of pointer accesses, which is contrary to the previous experiments. This difference stems from the implementation details, where for the previous algorithms a node is checked to be a left child by checking if the node is equal to the parents left child, but following the details of [29, Section 11.4.1], the LCA instead checks this property using a comparison on the values of the nodes.

From the theory, we know that the runtime is not dependent on the rank difference d , but only the height of the LCA. However, on the plot we can see that the runtime divided by $\log d$ decreases as the rank difference increases. There are two theoretical reasons for this. Firstly, a lower bound on the LCA can be obtained from the rank difference. The subtree spanning from the LCA must contain both endpoints of the search, and therefore also all values inbetween. This implies that the LCA must reside at height $\Omega(\log d)$, in order for the subtree to contain the required d elements. There is, however, no upper bound on the height of the LCA as a function of the rank difference,



(a) Measured runtimes.



(b) Improvement in runtime of the Whiteboard version over the Paper version. Note that the y-axis is logarithmic.

Figure 20.9: Plot of runtimes of the Paper version and Whiteboard version, when only searching for leaves, without short-circuiting the termination check. The trees tested on are red-black trees with explicit predecessor and successor links. Test trees of size $n = 100000$ are created by inserting the values in random order over 10 random seeds. Each dot corresponds to a search. The x-axis is logarithmic and shows the rank difference d . The plots on the left shows the pointer accesses made on the y-axis, and the plots on the right the number of comparisons.

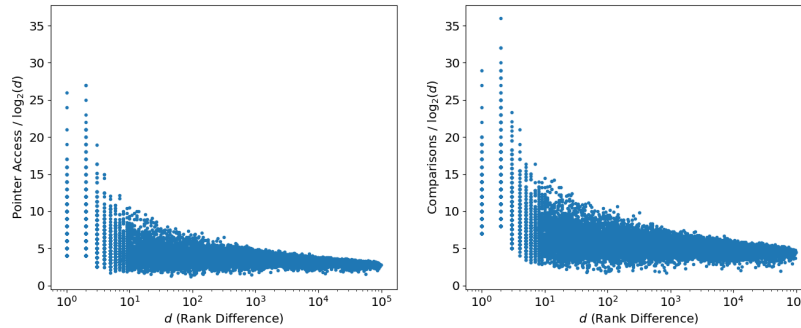


Figure 20.10: Plot of measured runtime of the LCA version, when only searching for leaves. The trees tested on are red-black trees with explicit predecessor and successor links. Note that the algorithm does not use these links. Test trees of size $n = 100000$ are created by inserting the values in random order over 10 random seeds. Each dot corresponds to a search. The x -axis is logarithmic and shows the rank difference d . The plot on the left show the number of pointer accesses made divided by $\log d$ on the y -axis, and the plot on the right the number of comparisons.

which is why we expect the runtime to be larger.

Secondly, the endpoints of the tree, where the search is performed between, are randomly sampled. Note that for a search from x to y to traverse over a given LCA node, then x and y must reside in different subtrees of the LCA. Fix the LCA node, and consider the sampling of an element x in the left subtree of the LCA. From the proof of Section 20.6 we can observe in Figure 20.2, that the only “wasted time” the LCA algorithm performs, is the number of nodes on the right spine above x . As x is randomly sampled, and the tree is balanced, we can roughly assume that the sampling is done by choosing a random traversal from the left child of the LCA and downwards. The length of the right spine above x is then the number of times the right child is selected before the first choice of a left child on this path. We can therefore expect the right spine path to be of constant length, imposing that the expected height of the LCA is at $\mathcal{O}(\log d)$. In other words: the worst-case explained in Section 20.2 rarely happens for randomly selected elements.

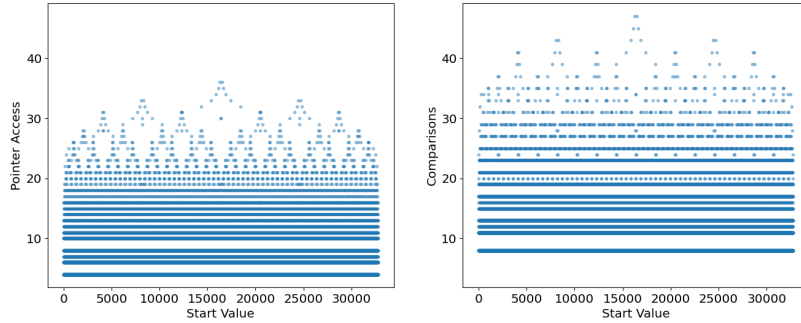
To see that there is a dependency on the height of the LCA and not the rank difference, I measured the runtime of the LCA version, when searching in a complete tree, from each leaf to its neighboring leaf. The rank difference is then always 2, but the height of the LCA differs over the different nodes. The measurement of this experiment can be seen in Figure 20.11a. Here we can see that the runtime differs as the starting node varies, and in the runtimes we can see the structure of the binary tree. The tree is however not completely visible. This is due to the simultaneous upwards and downwards search being performed by the LCA version, which alters slightly the runtime. As the search continues upwards, until it can ensure that the previously found

LCA candidate is the correct candidate, it may first do so when either taking a step up and finding a node bounding the search, or reaching the root. For some searches this step is not reached, and for others it is, prompting the varying in the runtime.

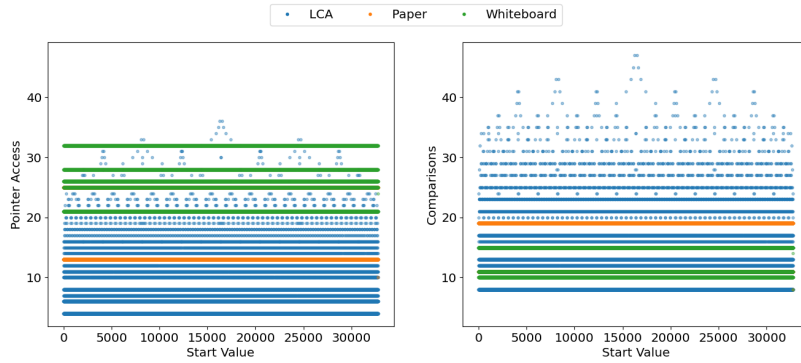
To compare this with the Paper version and Whiteboard version, which for the constant rank difference takes constant time, the plot in Figure 20.11b shows the three algorithms run on complete trees, when searching from each node to a node at rank distance 2. As expected, the other versions are constant time, at different constants, depending on the structure of the local tree the search appears in. At around the largest valued node, we do see faster single outliers, which simply stem from the fact that the search terminates, when determining that a node does not have a successor, and thus the current node must be the correct node to return. From a theoretical standpoint they are therefore better than the LCA version. From a practical standpoint, their relatively large constant, when compared to the LCA version, makes the LCA version faster for most cases, and either a large tree, or an unlucky search would be needed, in order for them to outperform the LCA version.

To show that the runtime is indeed dependent on the height of the LCA, I measured the runtime of the LCA version, when searching in a complete tree, from each leaf over increasing rank differences. By then showing the plot not as a function of the rank difference, but the LCA height, it can be seen that the runtime is dependent on the LCA height. The plot can be seen in Figure 20.12. Note that the groups by rank difference shows that the same rank difference appears in multiple groups, with the tendency that larger rank differences use larger LCA heights, as the theory also supports. Note that to show all groups on the rank difference, the points have been slightly shifted along the x -axis. We can then clearly see, that the runtime is dependent on the LCA height. The runtime, however, taper off at the midpoint among LCA heights. This is due to the upwards and downwards search reaching the root of the tree if the LCA is higher than halfway up the tree. It can then be confirmed that the found candidate is the LCA, and the search is then only directed downwards.

Jump to the root and search down In Section 20.3, the limitations of finger searching was discussed, when comparing to simply searching downwards from the root of the tree. Specifically, it was computed that at most a root of the n elements of the tree would be reachable more efficiently with finger searching, when considering the hidden constants in the $\mathcal{O}$ -notation. To test this theory, I analyzed the runtime of searching for elements in the red-black trees, creating a finger search algorithm which ignores the finger, and simply jumps to the root and searches down, yielding the Search from Root version of finger searching. This has the theoretical runtime of $\mathcal{O}(\log n)$ for both the number of pointer access and the number of comparisons. The measured runtime can be seen on the plot in Figure 20.13a. Note that the displayed runtime decreases as the rank difference increases, which is simply due to the runtime being divided by $\log d$, as with the other finger search algorithms. When removing the division we get the plot in Figure 20.13b. Here we see that the runtime is constant with respect to the rank difference, which is as expected, as we test on trees of a fixed



(a) Only including the LCA version.



(b) Including the LCA version, Paper version, and Whiteboard version.

Figure 20.11: Plot of measured runtime when searching for the constant rank difference 2 from each leaf of the tree. The trees tested on are complete red-black trees with explicit predecessor and successor links, of height 15 with size $n = 32767$. Each dot corresponds to a search. The x -axis shows the starting node. The plots on the left shows the number of pointer accesses made on the y -axis, and the plots on the right the number of comparisons.

size, leading to $\log n$ being a constant. The runtimes do vary from 14 to 20, which is due to the balance factor of red-black trees. The chosen size of the trees is $n = 100000$, and thus $\log n \approx 16.6$, which matches with the measured runtimes on average.

In Figure 20.14a, a plot of the runtimes of the Paper version, Whiteboard version, and Search from Root version can be seen. The theory of Section 20.3 tells us that the runtime of the finger search algorithms should be better for small rank differences, where the downwards search then takes over and becomes more efficient for larger rank differences. This is reflected on the plot, as we see for the larger rank differences that the simple downwards search is the fastest.

We may then measure the runtime improvement of the Search from Root version over the Paper and Whiteboard versions, which are shown in the plots in Fig-

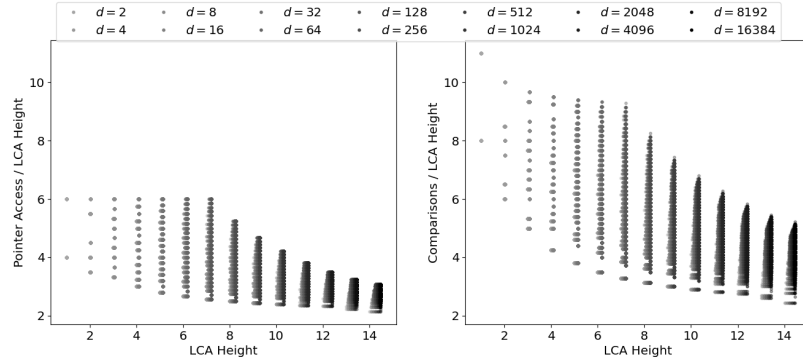
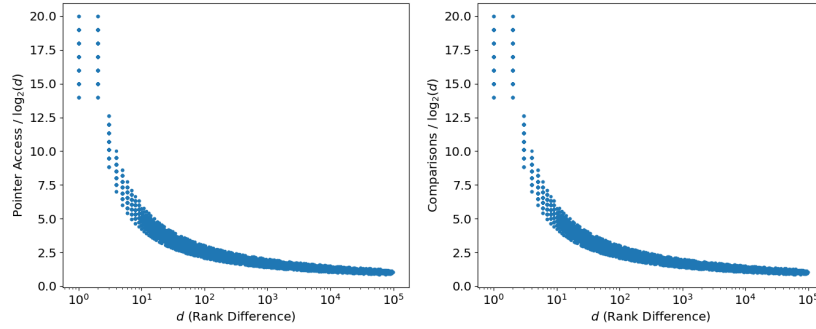
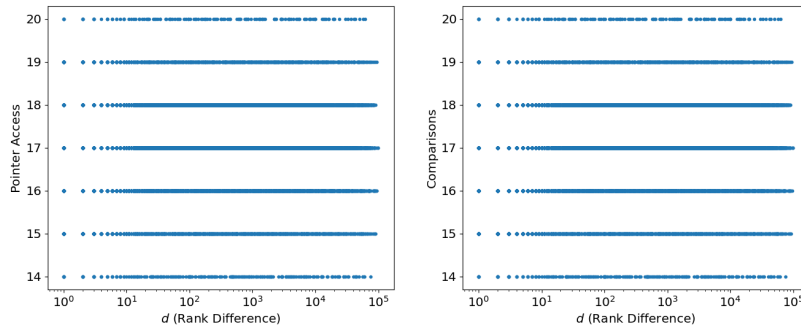


Figure 20.12: Plot of measured runtime of the LCA version, when only searching for leaves. The trees tested on are complete red-black trees with explicit predecessor and successor links, of height 15 with size $n = 32767$. Each dot corresponds to a search. The dots are slightly shifted on the x -axis by the rank difference d of the search, to not have overlapping points. The darker the dot is, the higher is the value of the rank difference. The x -axis shows the height of the LCA. The plot on the left show the number of pointer accesses made divided by the LCA height on the y -axis, and the plot on the right the number of comparisons.

ure 20.14b and 20.14c. As expected, we see that the Search from Root version is slower for small rank differences, but then increase in improvement as the rank difference increases, as the downwards search over the whole tree becomes more effective, than searching both up and down the whole tree.

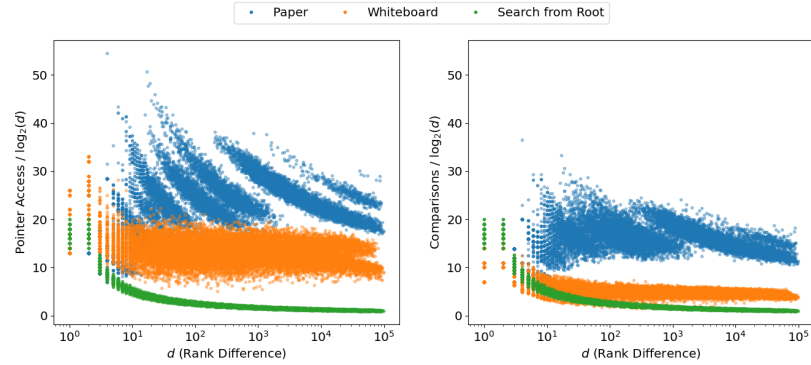
Level-linked (a, b) -trees In the paragraphs above, the runtime of the finger search algorithms based on predecessor and successor access was measured, as well as the versions searching over the tree without using the extra information in the nodes. To put the runtimes of these algorithms into context, I have implemented the level-linked (a, b) -trees by Huddleston and Mehlhorn [77], as $(2, 4)$ -trees. The runtime of finger searching in the level-linked trees can then be measured. To obtain the trees, n elements are inserted in random order into the tree. The plot of the runtime of finger searching can be seen in Figure 20.15a. First we notice that the constant on both the number of pointer accesses and comparisons is much smaller than the constants of the previous finger search algorithms. This is expected, as the level linked trees are designed specifically for finger searching. All elements reside at the leafs of the tree, and the inner elements are then only routing elements. The level links themselves then allows for essentially searching up a single left spine, independent on where in the tree the search is performed.

The next thing we may notice is that there is clearly two groups in the plot. This grouping is based on if the searched value is smaller or greater than the starting value, as demonstrated in Figure 20.15b. This grouping is therefore similar to that found in

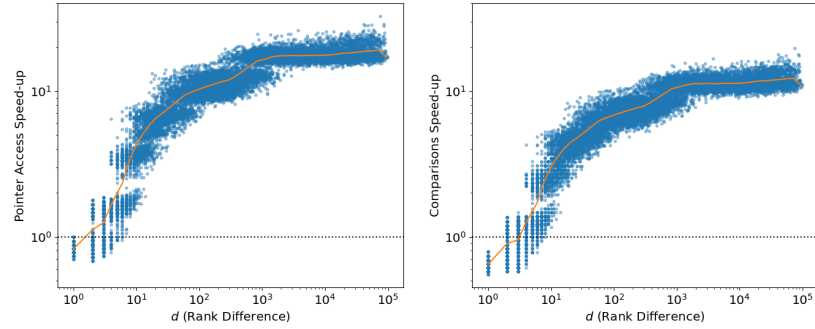
(a) Dividing the runtime by $\log d$.

(b) Without division.

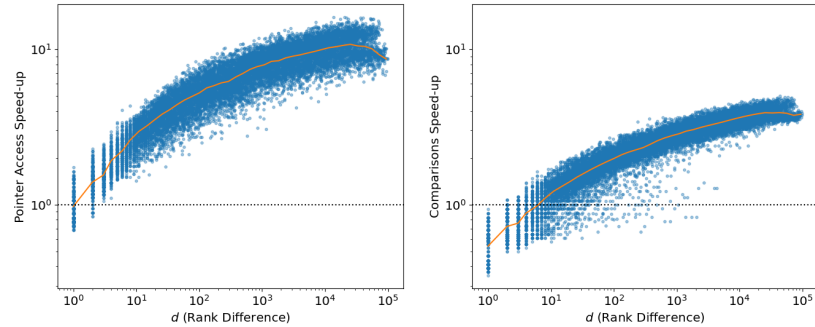
Figure 20.13: Plot of measured runtime of the Search from Root version, when only searching for leaves. The trees tested on are red-black trees with explicit predecessor and successor links. Test trees of size $n = 100000$ are created by inserting the values in random order over 10 random seeds. Each dot corresponds to a search. The x -axis is logarithmic and shows the rank difference d . The plots on the left shows the number of pointer accesses made on the y -axis, and the plots on the right the number of comparisons.



(a) Measured runtimes.



(b) Improvement in runtime of the Search from Root version over the Paper version. Note that the y-axis is logarithmic.



(c) Improvement in runtime of the Search from Root version over the Whiteboard version. Note that the y-axis is logarithmic.

Figure 20.14: Plot of runtimes of the Search from Root, Paper version, and Whiteboard version, when only searching for leaves, without short-circuiting the termination check. The trees tested on are red-black trees with explicit predecessor and successor links. Test trees of size $n = 100000$ are created by inserting the values in random order over 10 random seeds. Each dot corresponds to a search. The x -axis is logarithmic and shows the rank difference d . The plots on the left show the pointer accesses made on the y -axis, and the plots on the right the number of comparisons.

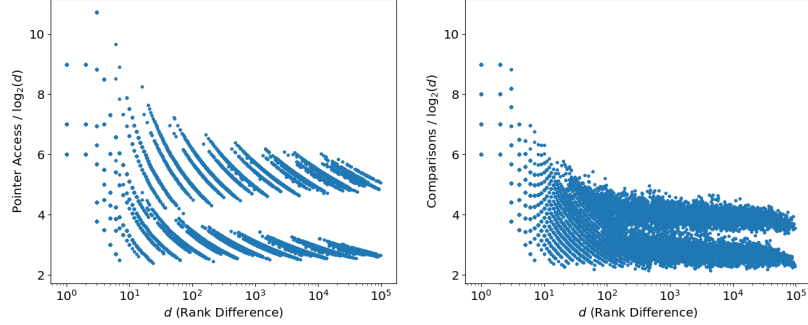
the plots of the Paper version. The finger searching in the level-linked trees proceeds by searching upwards until the current node or the neighboring node contains the desired value in the subtree. This check is done similarly to the check for the correct node being found for the Paper version. The check is stated below.

```
def contains_value(node):
    return node.splitters[0] <= value and \
        (node.succ is None or value < node.succ.splitters[0])
```

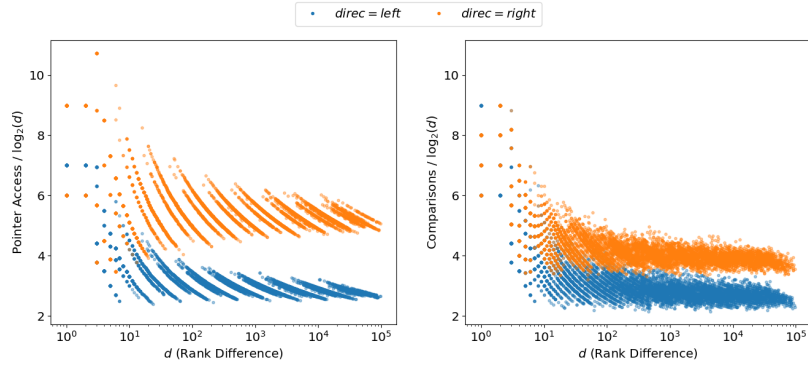
We can notice that the 'and', as previously, uses short-circuit evaluation. By disabling the short-circuit evaluation on this check, resulting in the plot in Figure 20.15c, the measured runtime becomes similar when searching for smaller or larger values.

We can now collect the runtimes of the level-linked (a, b) -trees with the runtimes of the previous search algorithms, to compare the runtimes. These plots can be seen in Figure 20.16, for all algorithms using short-circuit evaluation (Figure 20.16a), and all not using it (Figure 20.16b). For the level-linked trees, the access to each child pointer is counted as a separate access, and each comparison to each child is counted separately. The runtimes may therefore roughly be compared directly at face value. As we can see, the level-linked trees outperforms both the Paper version and the Whiteboard version, and as expected, the Search from Root and LCA version outperforms the level-linked trees for larger rank differences, on pointer accesses. However, as most clearly visible on the measurement without short-circuit evaluation, the Whiteboard version is competitive to the level-linked trees, when measuring comparisons. This stems from the level-linked trees performing double comparisons checks on the upwards search, checking both the current node and the neighbor. On the downwards search, multiple values in each node may be compared, however, this is analogous to the comparisons performed on red-black trees on a downwards search. The implementation does however not binary search in each node, which would save a single comparison in each node of degree 4. For comparison to the red-black trees, nodes of degree 4 would be a black node, with two red children, where not all 3 values are compared to, but at most 2 of them.

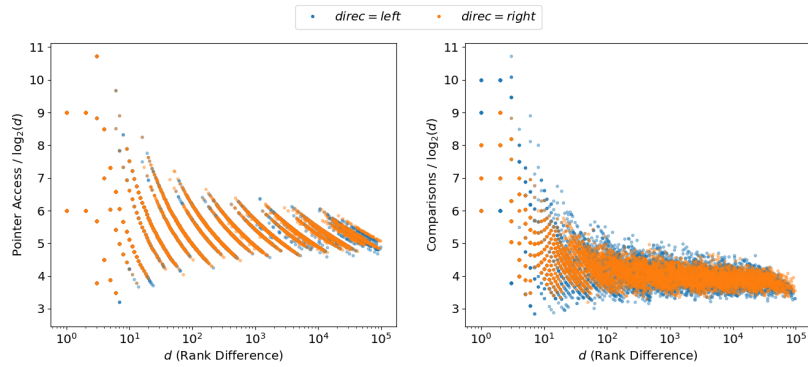
Folded Trees The folded trees guarantees access to the predecessor and successor in time $\mathcal{O}(1)$, without maintaining additional pointers. The parent and left and right child in the unfolded tree may similarly be found in time $\mathcal{O}(1)$. The Paper version of the finger search algorithm may therefore be applied in folded trees to finger search in time $\mathcal{O}(\log d)$, which is the main result of the paper [40]. The folded trees can then be maintained under insertions and deletions, however, their structure is consistent to an unfolded red-black tree, and during queries they are static. In this work I have only tested the runtime of the finger search itself, and therefore it is not needed to implement the updates on folded trees, but only a function which *folds* a red-black tree into a folded tree. The folded tree requires two bits in each node. For the queries the color bit is not needed, but the bit denoting the path type is. This is maintained in the pure model by metanodes. For this implementation, the metanodes, as well as the pure nodes with only two pointers, are not made, but the behavior stubbed, by using



(a) Ungrouped experiments, with short-circuit evaluation.

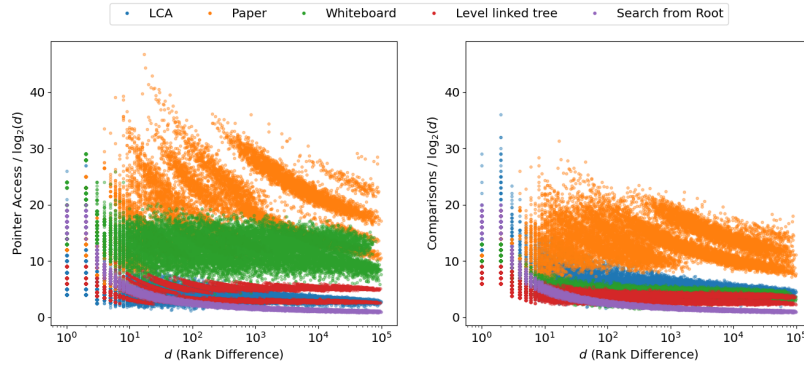


(b) Grouped by the direction of the search, with short-circuit evaluation.

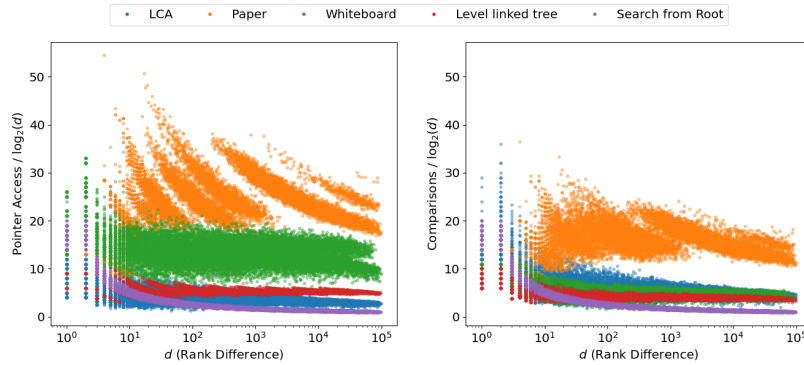


(c) Grouped by the direction of the search, without short-circuit evaluation.

Figure 20.15: Plot of runtimes of the Level-linked trees, when only searching for leaves. Test trees of size $n = 100000$ are created by inserting the values in random order over 10 random seeds. Each dot corresponds to a search. The x -axis is logarithmic and shows the rank difference d . The plots on the left show the pointer accesses made divided by $\log d$ on the y -axis, and the plots on the right show the number of comparisons.



(a) With short-circuit evaluation.



(b) Without short-circuit evaluation.

Figure 20.16: Plot of measured runtime of the Paper, Whiteboard, Search from Root, LCA, and Level-linked version, when only searching for leaves. The trees tested on are red-black trees with explicit predecessor and successor links. Test trees of size $n = 100000$ are created by inserting the values in random order over 10 random seeds. Each dot corresponds to a search. The x -axis is logarithmic and shows the rank difference d . The plots on the left shows the number of pointer accesses made divided by $\log d$ on the y -axis, and the plots on the right the number of comparisons.

nodes with three pointers, two bits, and a single value. Clearly the measured runtimes would increase if using metanodes, and two pointer pure nodes, however, this is not affecting the overall $\mathcal{O}(\log d)$ runtime, that we wish to show being present.

The folding is simple to implement as two mutually recursive procedure over the downwards spines, folding left and right spines respectively, which then yields the folded tree. The functions for accessing the parent and left and right child in the unfolded tree, as well as the predecessor and successor was then implemented. The functions were tested by checking for each node in the folded and unfolded tree, if the accessed nodes was the same. The values used were integers from 0 to $n - 1$, which allowed for easily testing the predecessor and successor functions.

The Paper version was then applied on the nodes of the folded tree, using the unfolded access functions, which corresponds to the construction of the paper [40]. The resulting measured runtime can be seen in Figure 20.17. First we may notice that the plot on the pointer accesses looks similar to the plot on the Paper version applied on the red-black tree with explicit predecessor and successor links, as expected. The constant, however, have largely increased. This is due to the many additional pointer accesses on the folded tree needed to locate the desired nodes of the unfolded tree. If the implementation used metanodes, each pointer access of the folded tree would require additional accesses to traverse through the metanodes. Notice that the number of times a bit is accessed is not accounted for in the runtime. In the metanodes, these bit accesses would further increase the number of pointer accesses, as the part of the metanode containing the bit would need to be accessed. The runtime of the comparisons is, however, very low for the Paper version applied on the folded tree. This is due to the unfolded access functions only using the pointers and access to the bits, without any comparisons being performed.

In Section 20.6, it was shown that the Paper version and the unfolded access functions are not needed to finger search in folded trees, as the LCA of the searched nodes are $\mathcal{O}(\log d)$ distance away. This may be shown by computing the shortest path for pairs of nodes in a folded tree, and then comparing this distance to the rank difference. The distance is computed by a BFS traversal from each node, and the measured distance can be seen in Figure 20.18. Here it can be seen that the distance between nodes x and y with rank difference d in a folded tree is $\approx \log d$. What is interesting to see is that the distance is not truly constant for large or small rank differences, as the constant is decreasing for the really large rank differences. We may first notice, that in a folded tree, the distance between the largest and smallest value is $\mathcal{O}(1)$, as they are at the bottom of the left and right spine of the root. The difference to the theory may be seen in the constructive proof of Section 20.6, where the traversed distance from x to the spine is crudely upper bounded by $\mathcal{O}(\log d)$, where it due to the folded structure may be much closer.

By the theory of Section 20.6, and the experiment confirming the stated theory, we may therefore apply the LCA version directly on the folded trees, without utilizing the structure of the folded tree inside the LCA algorithm. When doing so, the measured runtime can be seen in the plot of Figure 20.19a. At first, it may look as if the runtime is faster than $\mathcal{O}(\log d)$, as the curve decreases over time. This, however, simply stems

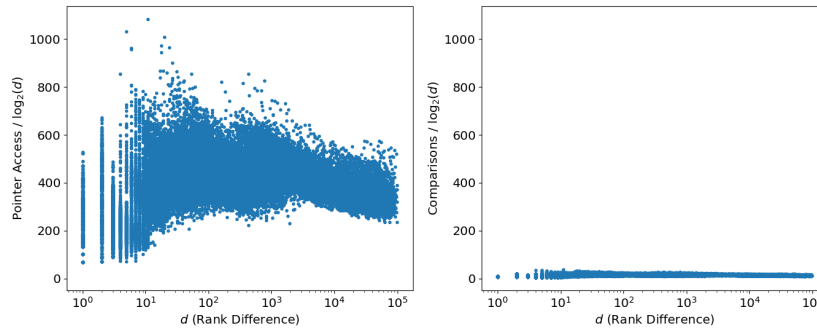


Figure 20.17: Measured runtime of the Paper version when applied on folded trees. The folded trees are generated by generating random red-black trees of size $n = 100000$ by inserting the values in random order over 10 random seeds, and then folding the resulting tree. Each dot is a single search. The x -axis is logarithmic and shows the rank difference d . The plots on the left shows the number of pointer accesses made divided by $\log d$ on the y -axis, and the plots on the right the number of comparisons.

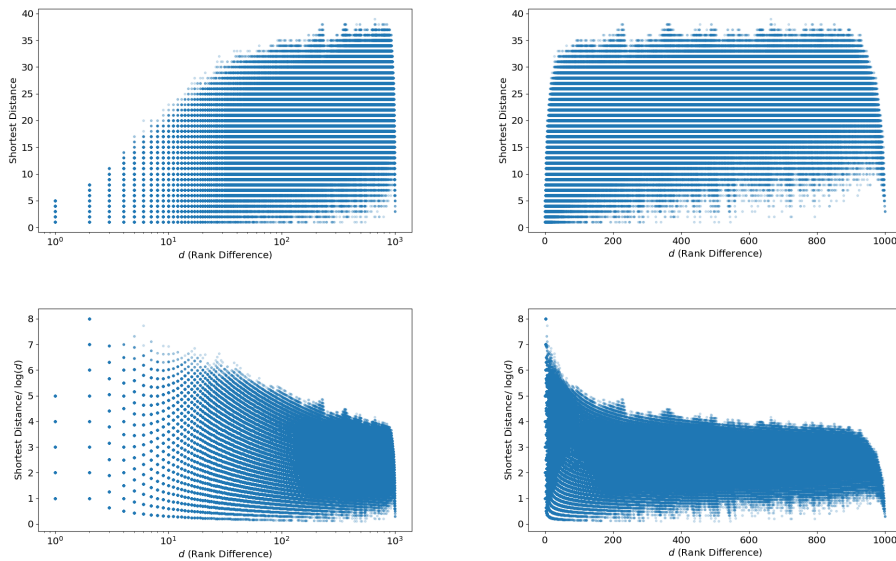


Figure 20.18: The distance between all pairs of nodes in a folded tree compared to the rank difference. The folded trees are generated by generating random red-black trees of size $n = 1000$ by inserting the values in random order over 5 random seeds, and then folding the resulting tree. The top plots show the pure distance, where the bottom plots show the distance divided by $\log d$. The left plots show the rank difference on the x -axis as a logarithmic scale, where the right is a linear scale.

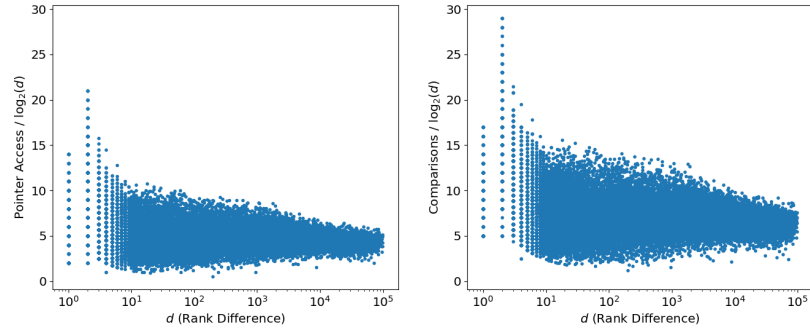
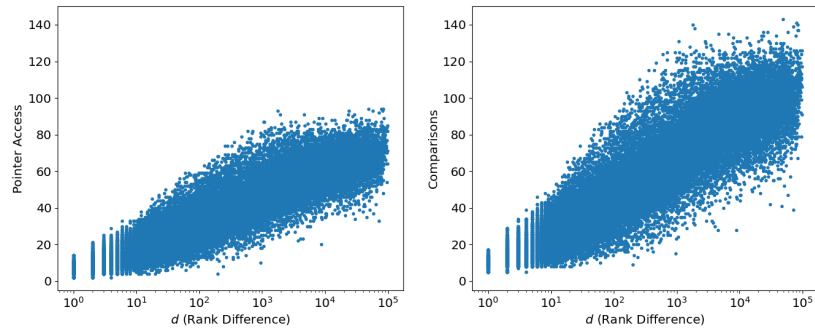
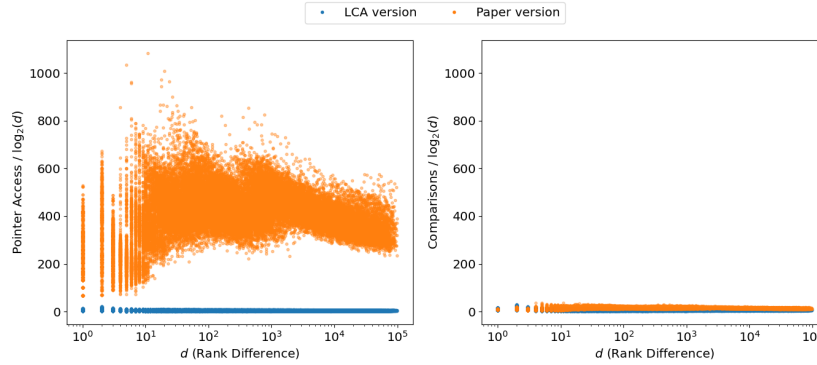
(a) Dividing by the logarithm of the rank difference d .(b) Without dividing by the logarithm of the rank difference d .

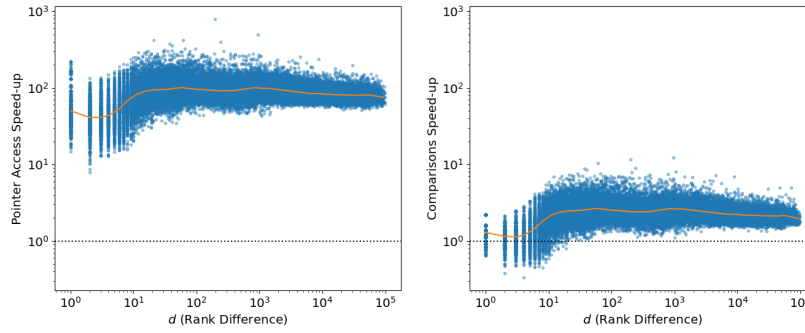
Figure 20.19: Measured runtime of the LCA version when applied on folded trees. The folded trees are generated by generating random red-black trees of size $n = 100000$ by inserting the values in random order over 10 random seeds, and then folding the resulting tree. Each dot is a single search. The x -axis is logarithmic and shows the rank difference d . The plots on the left shows the number of pointer accesses made on the y -axis, and the plots on the right the number of comparisons.

from the LCA algorithm capping the runtime, when the LCA is more than halfway up the tree, as also shown previously. It still holds that the runtime increases as the rank difference increases, until the rank difference prompts the search to traverse far up the tree, where the LCA search then caps. This is also clear when removing the division on the runtime by $\log d$, as seen in Figure 20.19b.

The LCA algorithm was analyzed to be applicable both for the simplicity of the algorithm, but the analysis also showed that the hidden constant is better. This is remarkably clear, when plotting the runtimes of the two approaches in the same plot, as seen in Figure 20.20a. Here, the pointer accesses performed by the Paper version applied on the folded tree is visible, where the LCA algorithm reduces to a line on the far larger plot. The comparisons are, however, still competitive between the two



(a) Individual runtimes.



(b) Improvement factor of the LCA version over the Paper version. Note that the y-axis is logarithmic.

Figure 20.20: Measured runtime of the LCA version and Paper version when applied on folded trees. The folded trees are generated by generating random red-black trees of size $n = 100000$ by inserting the values in random order over 10 random seeds, and then folding the resulting tree. Each dot is a single search. The x-axis is logarithmic and shows the rank difference d . The plots on the left show the number of pointer accesses made divided by $\log d$ on the y-axis, and the plots on the right show the number of comparisons.

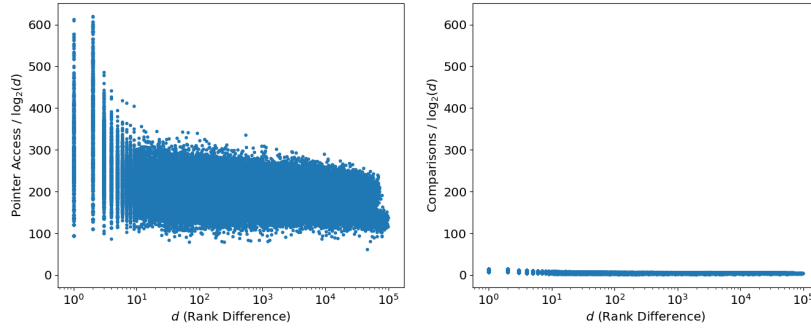
algorithms. The improvement factor of the LCA version over the Paper version can be seen in Figure 20.20b. Here we note that the LCA algorithm is about 100 times faster than the Paper version when counting pointer access, and only about 2.5 times faster when counting comparisons. This clearly shows that the constant of the LCA version is far better.

Similarly, the Whiteboard version may be applied to the folded trees. In the above analysis, it was shown that the Whiteboard version is about twice as fast on pointer access and 3.5 times faster on the comparisons, when comparing to the Paper version. We therefore expect the Whiteboard version to perform better than the Paper version on the folded trees, however, still not as good as the LCA version. The runtime

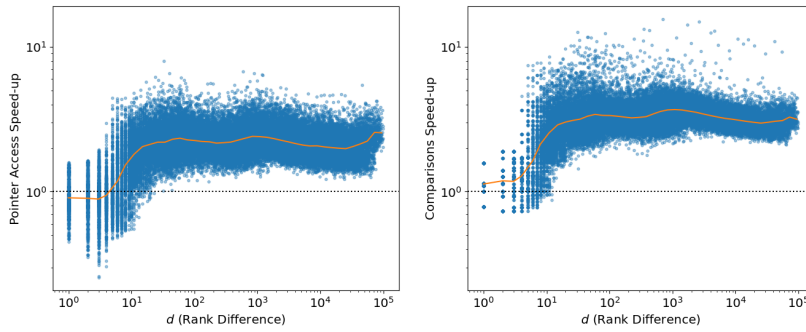
of the Whiteboard version was measured on folded trees and can be seen on the plot in Figure 20.21a. The comparison to the Paper version may then be seen in Figure 20.21b, where it can be seen that it is better than the Paper version, by about the same amount as it outperformed the Paper version on unfolded trees. This is not in itself interesting, but it does hint that the different unfolded access functions may have about the same constant, as we do not expect the two versions to ask for access to the parent, left and right child, and the predecessor and successor equally often. Finally, the LCA version can be seen compared to the Whiteboard version in Figure 20.21c, where it is about 50 times faster on pointer access, but interestingly the LCA version is slower on comparisons by about a factor 0.7 compared to the Whiteboard version. Therefore, if the only measurement we care about is the number of comparisons, the Whiteboard version is the fastest algorithm on folded trees.

Measuring the Unfolded Access Constants of Folded Trees The folded trees must access the parent and left and right child of the unfolded tree, as well as the predecessor and successor to allow for applying the Paper version of the finger searching. In the paper, it was simply mentioned that these operations existed and that they take $\mathcal{O}(1)$ time to perform. With the implementation we get access to a concrete algorithm for how to find these nodes, and therefore the ability to measure what exactly the constants of the unfolded access functions are. This measurement can be seen on the plots in Figure 20.22, for folded trees created from inserting n element into a red-black tree in random order, and for folding a complete binary tree without and with layers of red nodes. The plot is made using boxplots, to show the median, lower and upper quarter, as well as the outliers in the runtimes. We may first notice that the constant is smaller when the tree does not contain any red nodes. This is expected, as then all lower chunks but the last must be a single node. Traversing between pairs of upper nodes is therefore faster. The median runtime measured on access to the parent is larger than to the left and right child. This stems from the parent having more cases to check, as we implicitly need to check if the node is a left or right child, where this information is given when accessing the left or right child. The outliers of access to the left and right child are however larger than the maximum access time to the parent. The predecessor and successor access is larger than the access to the unfolded neighbors. This is due to these functions using the unfolded access functions of the parent and left and right child for simplicity.

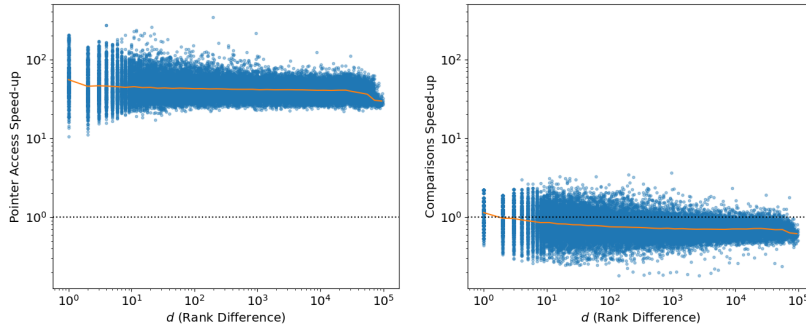
One odd thing we also may notice is that the access to the left is not equal to the right, even though these functions should be symmetrical, and similarly for the predecessor and successor runtimes. The first intuition is that this stems from the first path being left folded, which may make the measurements asymmetrical. As the access times are constant, then this would only have an effect if the first path, being the longest, has more nodes in the lower chunk that must be traversed. The larger last lower chunk may, however, appear anywhere in the tree at roughly the same probability. If the first path being the asymmetrical factor, this would imply that the functions are then not constant, as they are dependent truly on the path length.



(a) Individual runtime of the Whiteboard version.

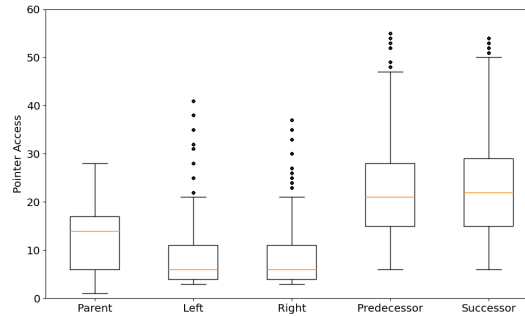


(b) Improvement factor of the Whiteboard version over the Paper version. Note that the y-axis is logarithmic.

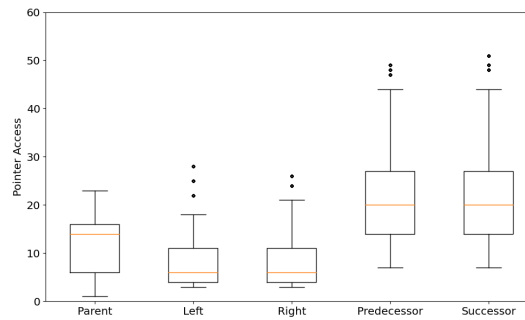


(c) Improvement factor of the LCA version over the Whiteboard version. Note that the y-axis is logarithmic.

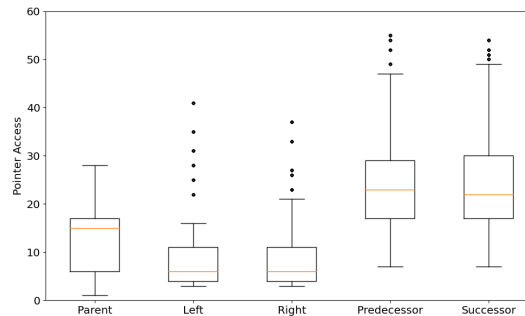
Figure 20.21: Measured runtime of the LCA version, Whiteboard version, and Paper version when applied on folded trees. The folded trees are generated by generating random red-black trees of size $n = 100000$ by inserting the values in random order over 10 random seeds, and then folding the resulting tree. Each dot is a single search. The x -axis is logarithmic and shows the rank difference d . The plots on the left shows the number of pointer accesses made divided by $\log d$ on the y -axis, and the plots on the right the number of comparisons.



(a) Random folded tree obtained from folding the resulting red-black tree from inserting $n = 10000$ values in random order over 10 seeds.



(b) Folded tree obtained from folding complete red-black tree of height $h = 16$ with $n = 65535$ values, where all nodes are black.



(c) Folded tree obtained from folding complete red-black tree of height $h = 16$ with $n = 65535$ values nodes, where the layers alternate between red and black nodes.

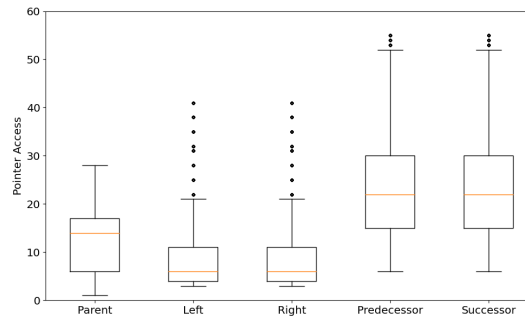
Figure 20.22: Measured Pointer Accesses performed when accessing the unfolded parent and left and right child, as well as the predecessor and successor in a folded tree. The boxplots show in orange the median, the box covers from the lower quarter to the upper quarter. The whiskers are then marked at the furthest point within 1.5 the height of the box distance from the box. The circles are the outliers outside this interval.

This may be tested by folding the first path right, which shows that the folding of the outermost spine does not affect this runtime.

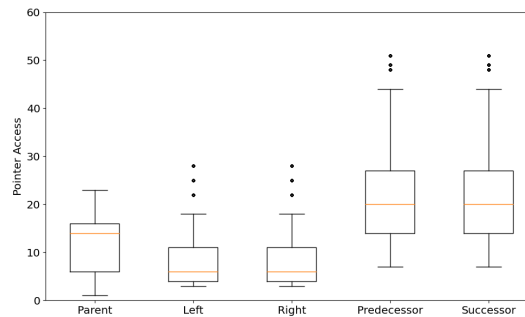
I then turned to the idea that maybe short-circuit evaluation was the culprit, as it has been seen earlier. One predicate the functions use, is to check if some node is the last lower node on the folded path. This can be checked by first checking if the node is a lower node, and then if the child where the next path node would be is on a different path. The function therefore needs to take into account if the node to check is on a left or right path for this check, and therefore has a difference in the order these checks are performed. The function may also simply check that the node is a lower node, and that both children are on a different path. The function is implemented as below in Python.

```
def is_last_lower(self) -> bool:
    return self.is_lower \
        and (self.left is None or \
            self.left.is_left_path != self.is_left_path) \
        and (self.right is None or \
            self.right.is_left_path != self.is_left_path)
```

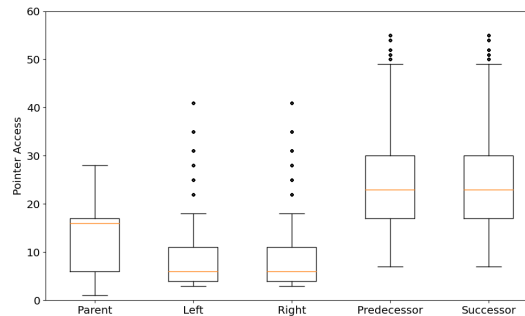
This function includes short-circuit evaluation on the check for the left and right path type. This allows the check to finish after only checking the left and not also the right child, when the left child is in the same path. By disabling the short-circuit evaluation on the 'and' in the function, the measured runtimes can then be seen in Figure 20.23, where we no longer see a difference on access to the left or right unfolded child.



(a) Random folded tree obtained from folding the resulting red-black tree from inserting $n = 10000$ values in random order over 10 seeds.



(b) Folded tree obtained from folding complete red-black tree of height $h = 16$ with $n = 65535$ values, where all nodes are black.



(c) Folded tree obtained from folding complete red-black tree of height $h = 16$ with $n = 65535$ values, where the layers alternate between red and black nodes.

Figure 20.23: Measured Pointer Accesses performed when accessing the unfolded parent and left and right child, as well as the predecessor and successor in a folded tree. The boxplots show in orange the median, the box covers from the lower quarter to the upper quarter. The whiskers are then marked at the furthest point within 1.5 the height of the box distance from the box. The circles are the outliers outside this interval. The short-circuit evaluation on the check on the node being the last lower node is disabled.

Chapter 21

Using Lazy B-Trees for External-Memory Priority-Queues

In our paper “Lazy B-trees” [110], it is presented that the construction may be used as an external-memory minimum priority-queue, without altering the construction, as long as queries and deletions are performed only on the element of the smallest rank. Symmetrically, the construction may be used as a maximum priority-queue. This priority-queue supports insertions using worst-case $\mathcal{O}(\log_B \log_B N)$ I/Os, extract of the smallest rank element using amortized $\mathcal{O}(\frac{1}{B} \log_2 N + \log_B \log_B N)$ I/Os, as well as decrease-key in worst-case $\mathcal{O}(\log_B \log_B N)$ I/Os [110, Corollary 2].

These bounds do not match the most efficient external-memory priority-queues, with the most recent improvement by Brodal et al. achieving insertions using amortized $\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os and extract using amortized $\mathcal{O}(1/B)$ I/Os [36]. This structure thereby allows for sorting a collection of N elements in the optimal sorting bound. However, the construction does not support decrease-key operations, which our construction based on lazy B-trees does. This is in part due to the lower bound on external-memory priority-queues of Eenberg, Larsen, and Yu, which shows that constructions supporting decrease-key operations cannot be optimal with regard to the sorting bound [62]. Therefore, even though our construction does not match the state-of-the-art bounds on insertions and extractions, it is still relevant, as it supports further operations.

As mentioned, the construction of the lazy B-tree need not be altered to be used as a priority-queue, only the pattern of the updates and queries must. If, however, the construction is only used as an external-memory priority-queue, the construction may be greatly simplified, supporting the operations in the same or improved I/O bounds as stated in [110, Corollary 2]. This new result is made in collaboration with Sebastian Wild from the University of Marburg. See Table 21.1 for an overview of the related work and the new results presented in the sections below.

Table 21.1: Overview of results on external-memory priority-queues for both cache-aware and -oblivious constructions. All stated bounds are on the I/O cost, with “N/A” denoting that the construction does not support the given operation. Results marked by “am.” are amortized, “rnd” denotes randomized expected, and all other results are worst-case. Results marked by $\parallel$ uses $M > B \log^{0.01} N$, and results marked by $\dagger$ uses the tall cache assumption $M \geq B^2$, and $\ddagger$ uses $M \geq B^{1+\varepsilon}$. The result marked by $*$ supports deletion of any key efficiently, however, this does not lead to a general decrease-key operation, as the value of the key to decrease must be known in advance.

	Insert	Extract-Min	Decrease-Key
Cache-Aware			
Kumar and Schwabe [87]	$\mathcal{O}\left(\frac{1}{B} \log_2 \frac{N}{B}\right)$ am.	$\mathcal{O}\left(\frac{1}{B} \log_2 \frac{N}{B}\right)$ am.	$\mathcal{O}\left(\frac{1}{B} \log_2 \frac{N}{B}\right)$ am.
Iacono, Jacob, and Tsakalidis [80]	$\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ am.	$\mathcal{O}\left(\frac{M^\varepsilon}{B} \log_{M/B}^2 \frac{N}{B}\right)$ am.	$\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ am.
Jiang and Larsen [82] $\parallel$	$\mathcal{O}\left(\frac{1}{B} \log_{\log N} \frac{N}{B}\right)$ rnd.	$\mathcal{O}\left(\frac{1}{B} \log_{\log N} \frac{N}{B}\right)$ rnd.	$\mathcal{O}\left(\frac{1}{B} \log_{\log N} \frac{N}{B}\right)$ rnd.
Brodal et al. [36]	$\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ am.	$\mathcal{O}\left(\frac{1}{B}\right)$ am.	N/A
Rysgaard and Wild [110, Corollary 2]	$\mathcal{O}(\log_B \log_B N)$	$\mathcal{O}\left(\frac{1}{B} \log_2 N + \log_B \log_B N\right)$ am.	$\mathcal{O}(\log_B \log_B N)$
New (Section 21.1)	$\mathcal{O}(\log_B \log_B N)$	$\mathcal{O}\left(\frac{1}{B} \log_2 N\right)$ am.	$\mathcal{O}(\log_B \log_B N)$
Cache-Oblivious			
Arge et al. [12] $\dagger*$	$\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$	$\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$	N/A
Brodal and Fagerberg [31] $\ddagger$			
Brodal et al. [34]	$\mathcal{O}\left(\frac{1}{B} \log_2 \frac{N}{B}\right)$ am.	$\mathcal{O}\left(\frac{1}{B} \log_2 \frac{N}{B}\right)$ am.	$\mathcal{O}\left(\frac{1}{B} \log_2 \frac{N}{B}\right)$ am.
Chowdhury and Ramachandran. [52]			
New (Section 21.2)	$\mathcal{O}(\log_B \log_B N)$	$\mathcal{O}\left(\frac{1}{B} \log_2 N\right)$ am.	$\mathcal{O}(\log_B \log_B N)$

21.1 Simplified External-Memory Priority-Queue

The lazy B-tree may be simplified to a priority-queue as follows. First, the construction will only ever contain a single gap, as a query isolates the element of the smallest rank. The gap structure, which is used to locate gaps upon updates and queries, may therefore be entirely omitted. The construction is then only a single instance of the interval structure.

In the regular lazy B-tree, when a query occur in some gap, the merge rule is first applied to all intervals, to obtain structure needed for the analysis. Next, the interval containing the queried rank element is split around the query rank, and the resulting left and right interval is then split twice each. Finally, the merge rule is applied again to ensure the invariant bounding the number of intervals.

For the priority-queue case, the query may only touch the element of the smallest rank. This inherently lets the gap be *1-sided*, which immediately simplifies the

potential function, as the dependence on the number of elements contained in 0- and 1-sided gaps may be omitted, as no elements can be moved to a 2-sided gap. The 1-sidedness, combined with decrease-key operations only moving elements to a smaller rank, may simplify the definition of *outside* to always be the number of elements to the left of an interval. Further, that queries only are for the smallest rank element, collapses the analysis of the splitting to a single case, where the structure of the other intervals is unneeded, as the outside is well-defined, and the query cannot create elements on both sides of some cut. Therefore, the first application of the merge rule may be omitted. Additionally, the double split of intervals was applied to ensure that at least $1/4$ of the elements of the original interval obtained an outside matching the size of the interval now containing them to free potential. As the outside is only towards the left, the second split may be omitted, as splitting the interval once ensures that the new interval on the right contains as many elements as its outside, thereby ensuring that $1/2$ of the elements free up the needed potential.

Remaining is the guarantee that there is $\mathcal{O}(\log_2 N)$ intervals, and therefore, a B-tree over the intervals allow insertions and decrease-key operation to locate and insert elements into the structure using $\mathcal{O}(\log_B \log_B N)$ I/Os in worst-case, as the potential does not increase by more than $1/B$. Querying for the smallest rank element simply scans and splits the first interval using amortized $\mathcal{O}(1)$ I/Os. The B-tree over intervals need not be queried to fetch the smallest interval, and when inserting the new intervals into the tree, the location must be in the smallest leaf, allowing for updating the tree using amortized $\mathcal{O}(1)$ I/Os, as the leaf location is known. Then the merge rule is applied, to ensure the number of intervals is $\mathcal{O}(\log_2 N)$. This uses a linear scan, where merging of the intervals are paid for by the amortization, allowing the query to use amortized $\mathcal{O}(\frac{1}{B} \log_2 N)$ I/Os. The simplified construction therefore allows removing the additive $\mathcal{O}(\log_B \log_B N)$ I/O overhead on queries, while matching the bounds on insertions and decrease-key operations. Note that if $M = \Omega(\log_2 N)$, then the B-tree over the intervals may be omitted, and replaced by a simple list stored in internal memory. The I/O cost of insertions and decrease-key then decrease to using worst-case $\mathcal{O}(1)$ I/Os and extract-min using amortized $\mathcal{O}(1)$ I/Os.

21.2 Cache-Oblivious External-Memory Priority-Queue

The construction presented in the previous section uses B-trees and blocked-linked-lists, and is therefore not cache-oblivious. By altering these primitives, the construction may be made cache-oblivious.

First, the B-tree is replaced by a cache-oblivious B-tree. This may be a dynamic tree, supporting queries, insertions, and deletions using $\mathcal{O}(\log_B I)$ I/Os, for the set of I intervals, as in [19, 33]. However, the tree is only updated upon a query, where all intervals are scanned. A static cache-oblivious tree may therefore be applied instead, where updates are made by rebuilding the tree in linear time, while the $\mathcal{O}(\log_B I)$ I/O query cost is maintained [18]. To ensure that the intervals can be scanned, this tree is accompanied by a simple list over the intervals, which is scanned and updated upon

queries.

The blocked-linked-list must be replaced by a cache-oblivious linked-list. One such structure is described in [56, Section 5.4], which supports linked-lists by using arrays linked by pointers. This ensures that merging intervals may be done using $\mathcal{O}(1)$ I/Os. Any access to the intervals either only touch a single element, or scans over the entire interval. Assume that an interval consists of K elements, which are stored in P linked arrays. The scan then uses $\mathcal{O}(K/B + P)$ I/Os. During the scan, the elements may be compacted into a single array, which thereby removed $P - 1$ pointers. By an amortized argument, placing $\mathcal{O}(1)$ I/Os as potential on each pointer, the scan thereby uses amortized $\mathcal{O}(K/B)$ I/Os, without increasing the amortized cost of merging intervals. By using standard garbage-collection techniques, where the arrays are reallocated when more than half of the allocated space is not used for the structure, the space usage can be shown to be $\mathcal{O}(N/B)$ blocks of storage, when containing N keys.

When splitting an interval, deterministic selection [25] is applied to split the interval at the median rank element. Following [56, Theorem 2], this selection is cache-oblivious. The two resulting intervals may then be placed into single arrays, thereby not creating new pointers, which would increase the potential.

If the decrease-key operation is to be supported, a pointer to elements must be supportable. This is trivial in the external-memory case, as the blocked-linked-lists allow $\mathcal{O}(B)$ incoming pointers to any block, which therefore allows decrease-key operations, while maintaining efficient scan times. In the cache-oblivious construction of this section, the lists used are moved to arrays upon any scan. Any incoming pointers would then need to be updated, which increases the I/O cost from $\mathcal{O}(K/B)$ to $\mathcal{O}(K)$ when scanning K elements. In the construction, lists are only scanned when the list is also split at the median, which frees $\Omega(K/B)$ I/Os from the potential. By increasing the potential function by a factor B , the freed potential is then $\Omega(K)$, which pays for the operation. The potential increase is then $\mathcal{O}(1)$ I/O and not $\mathcal{O}(1/B)$ I/O, however, this increase only occur upon insertions or decrease-key operations, which has an I/O cost of $\mathcal{O}(\log_B \log_B N)$, thereby not increasing the amortized asymptotic cost. Note, however, that even though the amortized cost is not increased, the underlying cost is increased by a factor B , which generally is unwanted in the I/O model. Therefore, if decrease-key operations are not needed, the construction is faster in practice by a factor B for queries.

21.3 Corrections to the Result

After this dissertation was completed, we found an error in the stated I/O costs, as also described in Section 5.3.

The error lies in the analysis of the I/O cost of splitting a list. It does hold that a (cache oblivious) external list with K elements can be scanned and partitioned using $\mathcal{O}(K/B)$ I/Os. However, this forgets the K incoming pointers to the list, which must

be updated when the elements move, which the section above also mentions. Recall that these pointers are crucial for the efficient decrease-key operation.

This has the implication, that the main selling point of the structure, the fast decrease-key operations, does not hold. Specifically, the decrease-key operations increase to use amortized $\mathcal{O}(\log_2 N)$ I/Os. The error was caught, when trying to simplify this structure to be cache oblivious, as further analysis showed that internal linked lists would be sufficient for the external structure. This is then where we caught the cascading increase in the I/O cost imposed by the missing pointer updates in the I/O cost of splitting lists. Instead, we pivoted the result to the internal model, where the pointer problem disappears, as the linked list nodes may be reused in a stable manner, to create simple heaps, with insertion and decrease-key operations in $\mathcal{O}(\log \log n)$ time, and extract-min and delete operations in $\mathcal{O}(\log n)$ time. This result was published at LATIN'26 [37], with the full version on arXiv [38], as a collaboration with Gerth Stølting Brodal, John Iacono, and Sebastian Wild. The paper is a merge of three constructions all building on using $\mathcal{O}(\log n)$ linked lists maintained using concatenation and partitioning, with three separate invariants, rules, and analysis for maintaining the lists.

Note that the I/O costs stated in the result above still holds, if delete and decrease-key operations are not to be supported, as the pointers into the lists are then no longer needed. The problem of the pointers cannot be directly removed. However, we do not need actual outside pointers directly into the lists, but only the functionality of finding a given element using $\mathcal{O}(\log_B \log_B N)$ I/Os. To achieve this, a secondary data structure, which uses some other pointer representation to find the elements faster, must be constructed.

Chapter 22

Simplified Adaptive Sorting using Inversions

In internal memory, it is well-known that sorting by comparisons has a lower bound of $\Omega(n \log n)$, when sorting n elements $x_1, x_2, \dots, x_n$. However, by applying further metrics than just the number of elements, this bound may be surpassed. We call such sorting algorithm which uses further metrics *adaptive sorting algorithms*. One such metric is *inversions*. An inversion is a pair of indexes (i, j) in the input list, where $i < j$ but $x_i > x_j$, i.e., the elements at indexes i and j are in the wrong order. Let the input list contain I inversions. Insertion sort then uses $\mathcal{O}(n + I)$ comparisons and work when sorting the list [86, Section 5.2.1], resulting in it being an adaptive sorting algorithm with respect to the number of inversions – the fewer inversions the list contains, the faster the algorithms is, independent on the size of the list.

Guibas, McCreight, Plass, and Roberts [73] presented in 1977 the first finger search trees. Included in their paper, they give one motivation for the use of finger search trees, to be that of a better adaptive insertion sort algorithm. In the improved algorithm, instead of linearly inserting every element into the sorted prefix, a finger search tree on the prefix is maintained, and the elements are then inserted by performing a finger search from the largest element in the prefix. As the rank difference for each such query is corresponding to the number of inversions the inserted element has with the prefix, they show that this yields an algorithm which uses only $\mathcal{O}(n \log \frac{I}{n})$ comparisons and work. Further they show a matching lower bound, based on generating functions.

In this chapter I present joined work with Kasper Green Larsen and Rolf Svenning from Aarhus University, where we simplify the constructions leading to a matching upper and lower bounds. Some of the presented results may be folklore, but have, to our knowledge, not been typed up in a complete work. An overview of the previous work and new results is shown in Table 22.1.

Table 22.1: Overview of sorting algorithms on input lists of size n , which adapt to the number of inversions I .

	Comparisons	Work
Upper Bound		
Insertion Sort [86, Section 5.2.1]	$\mathcal{O}(n + I)$	$\mathcal{O}(n + I)$
General Finger Search Trees [73]	$\mathcal{O}(n \log \frac{I}{n})$	$\mathcal{O}(n \log \frac{I}{n})$
New Exponential Search [21]	$\mathcal{O}(n \log \frac{I}{n})$	$\mathcal{O}(n + I)$
New Two-layer Trees	$\mathcal{O}(n \log \frac{I}{n})$	$\mathcal{O}(\min\{n\sqrt{n}, n + I\})$
New Heterogeneous Finger Search Trees [129]	$\mathcal{O}(n \log \frac{I}{n})$	$\mathcal{O}(n \log \frac{I}{n})$
Lower Bound		
Generating Functions [73]	$\Omega(n \log \frac{I}{n})$	
New Inversion Blocks	$\Omega(n \log \frac{I}{n})$	

22.1 Simplified Upper Bound

We have observed, that finger search trees are not needed in this algorithm, as the searches are only from the largest element, and therefore a simple balanced tree suffices for efficient finger searches. This result is due to Tarjan and Van Wyk [129] and was named *heterogeneous finger search trees*, as finger searching is only allowed from some nodes of the tree. This result yields an adaptive algorithm with $\mathcal{O}(n \log \frac{I}{n})$ comparisons and work. If only the number of comparisons is to be efficient, another simple algorithm may be applied. Here, we let the sorted prefix be maintained in a list, as in insertion sort. When searching for the location of the next element to insert into the prefix, we apply exponential search [21] for the insertion index, and then insert the element in linear time. This yields an algorithm with $\mathcal{O}(n + I)$ work, but only $\mathcal{O}(n \log \frac{I}{n})$ comparisons.

The linear work performed by the last algorithm is inserting into a list, which by caching and low level instructions is somewhat efficient, when compared to the random access performed in the work optimal heterogeneous trees. A tradeoff inbetween these two extrema exists, using *two-layer trees*, which is a tree with fan-out $\sqrt{n}$, which takes up two layers, with the root and all leaves storing $\sqrt{n}$ elements in arrays. This tree is in essence a B-tree with $B = \sqrt{n}$. Searching in the tree is performed by binary search in the root and a single leaf using $2 \cdot \mathcal{O}(\log \sqrt{n}) = \mathcal{O}(\log n)$ comparisons. To update the tree, an element is inserted or deleted in a leaf, and similarly to a B-tree, nodes are split or merged to ensure that each leaf contains $\Theta(\sqrt{n})$ elements. Updates therefore perform $\mathcal{O}(\sqrt{n})$ work.

This tree may be used for adaptive sorting on inversions as follows. As n is known in advance, and only insertions are performed, we shall split a leaf only when it contains $2\sqrt{n}$ elements, and insert a new element in the root. To finger search for a new element, exponential search is first applied in the largest leaf from the largest element.

If the location of the searched element resides in this leaf, then $\mathcal{O}(\log d)$ comparisons is spent, for the tree containing d elements larger than the searched element. Otherwise, it must hold that the largest leaf contains $\geq \sqrt{n}$ elements, and therefore $d \geq \sqrt{n}$. We may then search regularly from the root, using $\mathcal{O}(\log n) = \mathcal{O}(\log \sqrt{n}) = \mathcal{O}(\log d)$ comparisons. Inserting the new element then spends $\mathcal{O}(\sqrt{n})$ work. This therefore yields an algorithm, using $\mathcal{O}(n \log \frac{I}{n})$ comparisons and $\mathcal{O}(n\sqrt{n})$ work. Note that the work bound uses that at most $\mathcal{O}(\sqrt{n})$ elements are to be moved upon an insertion to make room for the new element in the array, and to copy $\sqrt{n}$ elements upon splitting a leaf. By amortized analysis, the splitting is amortized $\mathcal{O}(1)$ work per element, amounting to $\mathcal{O}(n)$ total work for rebalancing. When moving elements in the arrays, only larger elements than the inserted element is moved, of which there is at most d . The bound on work may therefore be tightened to $\mathcal{O}(\min\{n\sqrt{n}, n + I\})$.

22.2 Simplified Lower Bound

The lower bound of Guibas, McCreight, Plass, and Roberts [73] is based on generating functions to lower bound the number of permutations of lists with n elements containing at most I inversions.

We note that this lower bound may be matched by a simpler and more intuitive *block* structure on the lists, using that only a lower bound on the count is needed. We shall assume that $I = c \cdot n$ for some integer value $c \geq 1$, by rounding down the real I to a multiple of n . Note that if this rounds I down to 0, we may still apply the $\Omega(n)$ lower bound needed to verify that the input list is sorted, which for $I < n$ still yields $\Omega(n \log \frac{I}{n})$ as a comparison lower bound.

We then divide the list into blocks of size k , where elements are sorted between the blocks but may be permuted within each block. Any inversion must then reside inside a block. Each block then contributes with at most k^2 inversions, and as there are n/k blocks, there must be at most kn inversions in total. As the number of inversions must be at most I , we therefore choose $k = I/n$. Note that this ensures that k is an integer, leading to each block containing a whole number of elements. We let the number of blocks be $\lfloor n/k \rfloor$, with the remaining $< k$ elements in the last block being sorted. This structure allows each block to be freely permuted, while still maintaining the bound on the number of inversions. There are then at least $(k!)^{\lfloor n/k \rfloor}$ lists with at most I inversions.

The lower bound on comparisons then comes from taking the logarithm of this count, as the algorithm must be able to separate all inputs with at most I inversions, and each comparison may separate two sets of input lists. This leads to the lower bound

$$\log \left((k!)^{\lfloor n/k \rfloor} \right) = \lfloor n/k \rfloor \Omega(k \log k) = \Omega(n \log k) ,$$

which by the definition of k yields the lower bound $\Omega(n \log \frac{I}{n})$ on comparisons.

Bibliography

- [1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.14. 267
- [2] Mikkel Abrahamsen, Anna Adamaszek, and Tillmann Miltzow. The art gallery problem is $\exists\mathbb{R}$ -complete. *J. ACM*, 69(1), December 2021. doi:10.1145/3486220. 51, 52
- [3] Mikkel Abrahamsen, Joakim Blikstad, André Nusser, and Hanwen Zhang. Minimum star partitions of simple polygons in polynomial time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024*, pages 904–910, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3618260.3649756. 52
- [4] Mikkel Abrahamsen and Bartosz Walczak. Common tangents of two disjoint polygons in linear time and constant workspace. *ACM Trans. Algorithms*, 15(1), December 2018. doi:10.1145/3284355. 54
- [5] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)*, 146:263–266, 1962. 10, 35, 36, 75, 285
- [6] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535. 4, 5
- [7] Arne Andersson. Improving partial rebuilding by using simple balance criteria. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989. doi:10.1007/3-540-51542-9_33. 75
- [8] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Workshop on Algorithms and Data Structures (WADS)*, pages 334–345. Springer, 1995. doi:10.1007/3-540-60220-8_74. 27, 36

- [9] Lars Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *6th International Symposium on Algorithms and Computations (ISAAC)*, volume 1004 of *Lecture Notes in Computer Science*, pages 82–91, 1995. doi:10.1007/BFb0015411. 43, 45
- [10] Lars Arge. The I/O-complexity of ordered binary-decision diagram. In *BRICS RS preprint series*, volume 29. Department of Computer Science, University of Aarhus, 1996. doi:10.7146/brics.v3i29.20010. 43, 45
- [11] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. doi:10.1007/S00453-003-1021-X. 27, 36
- [12] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM J. Comput.*, 36(6):1672–1695, 2007. doi:10.1137/S0097539703428324. 326
- [13] Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithmics*, 8:1.2–es, December 2004. doi:10.1145/996546.996549. 6, 14, 29, 264, 266
- [14] Amitabha Bagchi, Adam L. Buchsbaum, and Michael T. Goodrich. Biased skip lists. *Algorithmica*, 42(1):31–48, 2005. doi:10.1007/S00453-004-1138-6. 37, 38, 39
- [15] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683. 5, 13, 14, 27, 29, 36
- [16] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996. doi:10.1007/s007780050028. 14, 29
- [17] Michael A. Bender, Rathish Das, Martín Farach-Colton, Rob Johnson, and William Kuszmaul. Flushing without cascades. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 650–669. Society for Industrial and Applied Mathematics, 2020. doi:10.1137/1.9781611975994.40. 29, 32
- [18] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, Redondo Beach, California, USA, November 12-14, 2000*, pages 399–409. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892128. 327

- [19] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. *J. Algorithms*, 53(2):115–136, 2004. doi:10.1016/J.JALGOR.2004.04.014. 327
- [20] Samuel W. Bent, Daniel Dominic Sleator, and Robert Endre Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–568, 1985. doi:10.1137/0214041. 37, 38, 39
- [21] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976. doi:10.1016/0020-0190(76)90071-5. 73, 285, 332
- [22] Ahmad Biniiaz, Anil Maheshwari, Magnus Christian Ring Merrild, Joseph S. B. Mitchell, Saeed Odak, Valentin Polishchuk, Eliot W. Robson, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, Thomas C. Shermer, Jack Spalding-Jamieson, Rolf Svenning, and Da Wei Zheng. Polynomial-time algorithms for contiguous art gallery and related problems. In Oswin Aichholzer and Haitao Wang, editors, *41st International Symposium on Computational Geometry, SoCG 2025, June 23-27, 2025, Kanazawa, Japan*, volume 332 of *LIPICs*, pages 20:1–20:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.SOCG.2025.20. 3, 51, 52, 53, 61
- [23] Ahmad Biniiaz, Anil Maheshwari, Joseph S. B. Mitchell, Saeed Odak, Valentin Polishchuk, and Thomas Shermer. Contiguous boundary guarding, 2025. URL: <https://arxiv.org/abs/2412.15053>, arXiv:2412.15053. 51, 52, 61
- [24] Guy E. Blelloch, Bruce M. Maggs, and Shan Leung Maverick Woo. Space-efficient finger search on degree-balanced search trees. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '03*, pages 374–383, USA, 2003. Society for Industrial and Applied Mathematics. URL: <http://dl.acm.org/citation.cfm?id=644108.644169>. 290
- [25] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9. 40, 328
- [26] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th Design Automation Conference (DAC)*, pages 40–45. Association for Computing Machinery, 1990. doi:10.1109/DAC.1990.114826. 43, 45
- [27] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 79–97. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.15. 267

- [28] Gerth Stølting Brodal. Worst-case efficient priority queues. In Éva Tardos, editor, *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, 28-30 January 1996, Atlanta, Georgia, USA, pages 52–58. ACM/SIAM, 1996. URL: <http://dl.acm.org/citation.cfm?id=313852.313883>. 35, 36
- [29] Gerth Stølting Brodal. Finger search trees. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 11, page 11. CRC Press, 2005. URL: <https://www.cs.au.dk/~gerth/papers/finger05.pdf>, doi:10.1201/9781420035179. 284, 305
- [30] Gerth Stølting Brodal. Bottom-Up Rebalancing Binary Search Trees by Flipping a Coin. In Andrei Z. Broder and Tami Tamir, editors, *12th International Conference on Fun with Algorithms (FUN 2024)*, volume 291 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:15, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FUN.2024.6. 74
- [31] Gerth Stølting Brodal and Rolf Fagerberg. Funnel heap - A cache oblivious priority queue. In Prosenjit Bose and Pat Morin, editors, *Algorithms and Computation, 13th International Symposium, ISAAC 2002 Vancouver, BC, Canada, November 21-23, 2002, Proceedings*, volume 2518 of *Lecture Notes in Computer Science*, pages 219–228. Springer, 2002. doi:10.1007/3-540-36136-7_20. 326
- [32] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554. Society for Industrial and Applied Mathematics, 2003. URL: <http://dl.acm.org/citation.cfm?id=644108.644201>. 28, 29, 36
- [33] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, pages 39–48. ACM/SIAM, 2002. URL: <http://dl.acm.org/citation.cfm?id=545381.545386>. 327
- [34] Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 480–492. Springer, 2004. doi:10.1007/978-3-540-27810-8_41. 326

- [35] Gerth Stølting Brodal, Rolf Fagerberg, and Casper Moldrup Rysgaard. On Finding Longest Palindromic Subsequences Using Longest Common Subsequences. In *32nd Annual European Symposium on Algorithms (ESA 2024)*, volume 308 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:16, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi : 10.4230/LIPIcs.ESA.2024.35. 3, 65, 66, 67, 68, 69, 70, 71, 267, 268, 273, 275, 279, 280
- [36] Gerth Stølting Brodal, Michael T. Goodrich, John Iacono, Jared Lo, Ulrich Meyer, Victor Pagan, Nodari Sitchinava, and Rolf Svenning. External-memory priority queues with optimal insertions. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms, ESA 2025, Warsaw, Poland, September 15-17, 2025*, volume 351 of *LIPIcs*, pages 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi : 10.4230/LIPICS.ESA.2025.5. 325, 326
- [37] Gerth Stølting Brodal, John Iacono, Casper Moldrup Rysgaard, and Sebastian Wild. Partition-based simple heaps. In *17th Latin American Theoretical Informatics Symposium (LATIN 2026)*, Lecture Notes in Computer Science. Springer, 2026. To appear. 41, 329
- [38] Gerth Stølting Brodal, John Iacono, Casper Moldrup Rysgaard, and Sebastian Wild. Partition-based simple heaps. *CoRR*, abs/2603.01206, 2026. arXiv: 2603.01206, doi : 10.48550/ARXIV.2603.01206. 41, 329
- [39] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict fibonacci heaps. In *Symposium on Theory of Computing (STOC)*, STOC’12, page 1177–1184. ACM, May 2012. doi : 10.1145/2213977.2214082. 35, 36
- [40] Gerth Stølting Brodal and Casper Moldrup Rysgaard. Pure binary finger search trees. In Ioana Oriana Bercea and Rasmus Pagh, editors, *2025 Symposium on Simplicity in Algorithms, SOSA 2025, New Orleans, LA, USA, January 13-15, 2025*, pages 172–195. SIAM, 2025. doi : 10.1137/1.9781611978315.14. 3, 73, 74, 75, 76, 77, 78, 79, 80, 283, 285, 286, 288, 290, 295, 313, 316
- [41] Gerth Stølting Brodal, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, and Rolf Svenning. Persistent-functional-trees, 2022. URL: <https://github.com/Crowton/Persistent-Functional-Trees>. 10
- [42] Gerth Stølting Brodal, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, and Rolf Svenning. Space-efficient functional offline-partially-persistent trees with applications to planar point location. In Pat Morin and Subhash Suri, editors, *Algorithms and Data Structures - 18th International Symposium, WADS 2023, Montreal, QC, Canada, July 31 - August 2, 2023, Proceedings*, volume 14079 of *Lecture Notes in Computer Science*, pages 644–659. Springer, 2023. doi : 10.1007/978-3-031-38906-1_43. 3, 7, 8, 9

- [43] Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. External memory fully persistent search trees. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 1410–1423. ACM, 2023. doi:10.1145/3564246.3585140. 3, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 24, 28, 29
- [44] Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. Buffered partially-persistent external-memory search trees. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms, ESA 2025, Warsaw, Poland, September 15-17, 2025*, volume 351 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 82:1–82:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.ESA.2025.82. 3, 24, 27, 28, 29, 30, 31, 32, 33, 34, 263
- [45] Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. Buffered partially-persistent external-memory search trees. *CoRR*, abs/2503.08211, 2025. arXiv:2503.08211. 27, 33
- [46] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. Fully persistent B-trees. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 602–614. SIAM, 2012. doi:10.1137/1.9781611973099.51. 13, 14
- [47] Mark R. Brown. A storage scheme for height-balanced trees. *Information Processing Letters*, 7(5):231–232, 1978. doi:10.1016/0020-0190(78)90005-4. 75
- [48] Mark R. Brown. Addendum to “a storage scheme for height-balanced trees”. *Information Processing Letters*, 8(3):154–156, 1979. doi:10.1016/0020-0190(79)90009-7. 75, 76
- [49] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986. doi:10.1109/TC.1986.1676819. 43, 44, 45
- [50] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986. doi:10.1007/BF01840440. 38
- [51] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '95*, pages 139—149. Society for Industrial and Applied Mathematics, 1995. 43, 45

- [52] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In Phillip B. Gibbons and Micah Adler, editors, *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, pages 245–254. ACM, 2004. doi:10.1145/1007912.1007949. 326
- [53] Vasek Chvátal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory, Series B*, 18(1):39–41, 1975. 51
- [54] Richard Cole. Searching and storing similar lists. *Journal of Algorithms*, 7(2):202–220, 1986. doi:10.1016/0196-6774(86)90004-0. 9
- [55] Rathish Das, John Iacono, and Yakov Nekrich. External-memory dictionaries with worst-case update cost. In Sang Won Bae and Heejin Park, editors, *33rd International Symposium on Algorithms and Computation, ISAAC 2022, December 19-21, 2022, Seoul, Korea*, volume 248 of *LIPICs*, pages 21:1–21:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ISAAC.2022.21. 29, 32, 33, 34
- [56] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*, BRICS, University of Aarhus, Denmark, 2002. URL: <https://erikdemaine.org/papers/BRICS2002/>. 5, 27, 328
- [57] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, STOC '87*, pages 365–372, New York, NY, USA, 1987. ACM. doi:10.1145/28395.28434. 6, 13
- [58] Paul F. Dietz and Rajeev Raman. Persistence, randomization and parallelization: On some combinatorial games and their applications (abstract). In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11-13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 289–301. Springer, 1993. doi:10.1007/3-540-57155-8_256. 27, 33
- [59] Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 365–372. Association for Computing Machinery, 1987. doi:10.1145/28395.28434. 27, 33
- [60] David P. Dobkin and J. Ian Munro. Efficient uses of the past. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science, SFCS '80*, pages 200–206, USA, 1980. IEEE Computer Society. doi:10.1109/SFCS.1980.18. 9

- [61] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. doi:10.1016/0022-0000(89)90034-2. 6, 7, 8, 9
- [62] Kasper Eenberg, Kasper Green Larsen, and Huacheng Yu. Decreasekeys are expensive for external memory priority queues. In *Symposium on Theory of Computing (STOC)*, page 1081–1093. ACM, June 2017. doi:10.1145/3055399.3055437. 325
- [63] H El Gindy and D Avis. A linear algorithm for computing the visibility polygon from a point. *Journal of Algorithms*, 2(2):186–197, 1981. doi:10.1016/0196-6774(81)90019-5. 54
- [64] Rolf Fagerberg. The I/O-model, slides. Accessed on the 22. juli 2025. URL: <https://imada.sdu.dk/u/rolf/Edu/DM79/E04/Slides/IOModel.pdf>. 4
- [65] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987. doi:10.1145/28869.28874. 35
- [66] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS 1999, New York, NY, USA, October 17-18, 1999*, pages 285–298. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814600. 5
- [67] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 165–174. ACM/SIAM, 1993. URL: <https://dl.acm.org/doi/10.5555/313559.313676>. 75
- [68] Git. Documentation of the git diff command. URL: <https://git-scm.com/docs/git-diff/2.48.0#Documentation/git-diff.txt-myers>. 281
- [69] Git. Source code of the git diff command. URL: <https://github.com/git/git/blob/e813a0200a7121b97fec535f0d0b460b0a33356c/xdiff/xdiffi.c#L37>. 281
- [70] Ronald Goldman. Intersection of two lines in three-space. In ANDREW S. GLASSNER, editor, *Graphics Gems*, page 304. Morgan Kaufmann, San Diego, 1990. doi:10.1016/B978-0-08-050753-8.50064-4. 58, 60
- [71] Michael T. Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry (preliminary version). In *34th*

- Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA, November 3-5, 1993*, pages 714–723. IEEE Computer Society, 1993. doi:10.1109/SFCS.1993.366816. 266
- [72] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988. doi:10.1007/978-3-642-97881-4. 58
- [73] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, STOC '77*, page 49–60, New York, NY, USA, 1977. Association for Computing Machinery. doi:10.1145/800105.803395. 331, 332, 333
- [74] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3. 10, 35, 36, 74, 75, 283, 285
- [75] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. doi:10.1145/360825.360861. 268
- [76] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675, 1977. doi:10.1145/322033.322044. 269
- [77] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982. doi:10.1007/BF00288968. 74, 75, 285, 310
- [78] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest subsequences. *Communications of the ACM*, 20(5):350–353, 1977. doi:10.1145/359581.359603. 268, 269, 270
- [79] James Wayne Hunt and M Douglas MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976. URL: <https://www.cs.dartmouth.edu/~doug/diff.pdf>. 268
- [80] John Iacono, Riko Jacob, and Konstantinos Tsakalidis. External memory priority queues with decrease-key and applications to graph algorithms. In *European Symposium on Algorithms (ESA)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ESA.2019.60. 36, 326
- [81] Intel®. Intel® 64 and ia-32 architectures optimization reference manual volume 1, version 050. Accessed on the 22. juli 2025. URL: <https://www.intel.com/content/www/us/en/processors/ia-32-ia-64/optimization-reference-manual-volume-1.html>

//www.intel.com/content/www/us/en/content-details/821612/
intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.
html. 4

- [82] Shunhua Jiang and Kasper Green Larsen. A faster external memory priority queue with decreasekeys. In *Symposium on Discrete Algorithms (SODA)*, page 1331–1343. Society for Industrial and Applied Mathematics, January 2019. doi:10.1137/1.9781611975482.81. 326
- [83] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, nov 1962. doi:10.1145/368996.369025. 9
- [84] J. Mark Keil. Decomposing a polygon into simpler components. *SIAM Journal on Computing*, 14(4):799–817, 1985. doi:10.1137/0214056. 52
- [85] David Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983. doi:10.1137/0212002. 9
- [86] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998. 73, 77, 331, 332
- [87] Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing, SPDP 1996, New Orleans, Louisiana, USA, October 23-26, 1996*, pages 169–176. IEEE Computer Society, 1996. doi:10.1109/SPDP.1996.570330. 36, 326
- [88] S. Kuo and G. R. Cross. An improved algorithm to find the length of the longest common subsequence of two strings. *SIGIR Forum*, 23(3-4):89–99, apr 1989. doi:10.1145/74697.74702. 275
- [89] Sitaram Lanka and Eric Mays. Fully persistent B+-trees. *SIGMOD Records*, 20(2):426–435, April 1991. doi:10.1145/119995.115861. 14
- [90] Aldo Laurentini. Guarding the walls of an art gallery. *The Visual Computer*, 15(6):265–278, 1999. doi:10.1007/S003710050177. 51, 52
- [91] C.C. Lee and D.T. Lee. On a circle-cover minimization problem. *Information Processing Letters*, 18(2):109–115, 1984. doi:10.1016/0020-0190(84)90033-4. 56
- [92] D. T. Lee and Arthur K. Lin. *Computational Complexity of Art Gallery Problems*, pages 303–309. Springer New York, New York, NY, 1990. doi:10.1007/978-1-4613-8997-2_23. 51, 52
- [93] Christos Levkopoulos and Mark H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Informatica*, 26(3):269–277, 1988. doi:10.1007/BF00299635. 75

- [94] David B. Lomet and Betty Salzberg. Exploiting a history database for backup. In *Proceedings of the 19th International Conference on Very Large Data Bases, VLDB '93*, pages 380–390, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. URL: <https://dl.acm.org/doi/10.5555/645919.672672>. 14
- [95] David MacKenzie, Paul Eggert, , and Richard Stallman. Documentation of the GNU diff command. URL: <https://www.gnu.org/software/diffutils/manual/diffutils.pdf>. 281
- [96] Francisco Martínez, Antonio Jesús Rueda, and Francisco Ramón Feito. A new algorithm for computing boolean operations on polygons. *Computers & Geosciences*, 35(6):1177–1185, 2009. doi:10.1016/j.cageo.2008.08.009. 54
- [97] John W. Mauchly. *Theory and Techniques for the Design of Electronic Digital Computers*. Moore School of Electrical Engineering, 1946. 73
- [98] Magnus Christian Ring Merrild, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, and Rolf Svenning. The contiguous art gallery problem is solvable in polynomial time, 2025. URL: <https://arxiv.org/abs/2412.13938>, arXiv:2412.13938. 51, 52, 53, 54, 55, 56, 57, 58, 60, 61
- [99] Ulrich Meyer, Peter Sanders, and Jop Sibeyn. *Algorithms for Memory Hierarchies: Advanced Lectures*. Springer, Berlin, Heidelberg, 2003. doi:10.1007/3-540-36574-5. 43, 45
- [100] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *30th Design Automation Conference (DAC)*, pages 272–277. Association for Computing Machinery, 1993. doi:10.1145/157485.164890. 49
- [101] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *27th Design Automation Conference (DAC)*, pages 52–57. Association for Computing Machinery, 1990. doi:10.1145/123186.123225. 43, 45
- [102] Eugene W Myers. An $O(nd)$ difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986. doi:10.1007/BF01840446. 276, 280
- [103] Chris Okasaki. Purely functional random-access lists. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA '95*, pages 86–95, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224164.224187. 10, 11
- [104] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, USA, 1999. 7

- [105] Joseph O'Rourke. *Art gallery theorems and algorithms*. Oxford University Press, Inc., USA, 1987. 51, 52
- [106] Mark H Overmars. Dynamization of order decomposable set problems. *Journal of Algorithms*, 2(3):245–260, 1981. doi:10.1016/0196-6774(81)90025-0. 57
- [107] Jason Robert Carey Patterson. Modern microprocessors, a 90-minute guide! Accessed on the 22. juli 2025. URL: <https://www.lighterra.com/papers/modernmicroprocessors/>. 4
- [108] Eliot W. Robson, Jack Spalding-Jamieson, and Da Wei Zheng. The analytic arc cover problem and its applications to contiguous art gallery, polygon separation, and shape carving, 2025. URL: <https://arxiv.org/abs/2412.15567>, arXiv:2412.15567. 51, 52, 61, 62
- [109] Casper Moldrup Rysgaard. Pure binary finger search trees implementation, 2025. URL: <https://github.com/Crowton/PureBinaryFingerSearchTrees>. 293
- [110] Casper Moldrup Rysgaard and Sebastian Wild. Lazy B-trees. In Pawel Gawrychowski, Filip Mazowiecki, and Michal Skrzypczak, editors, *50th International Symposium on Mathematical Foundations of Computer Science, MFCS 2025, Warsaw, Poland, August 25-29, 2025*, volume 345 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 87:1–87:19, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.MFCS.2025.87. 3, 35, 36, 37, 39, 40, 41, 325, 326
- [111] Casper Moldrup Rysgaard and Sebastian Wild. Lazy B-trees, 2025. URL: <https://arxiv.org/abs/2507.00277>, arXiv:2507.00277. 35, 37, 40
- [112] Bryce Sandlund and Sebastian Wild. Lazy search trees. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 704–715. IEEE, 2020. doi:10.1109/FOCS46700.2020.00071. 35, 36, 37
- [113] Bryce Sandlund and Sebastian Wild. Lazy search trees, 2020. arXiv:2010.08840. 35, 37
- [114] Bryce Sandlund and Lingyi Zhang. Selectable heaps and optimal lazy search trees. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1962–1975, 2022. doi:10.1137/1.9781611977073.78. 36, 37
- [115] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, jul 1986. doi:10.1145/6138.6151. 7, 8, 9, 264

- [116] C. Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961. doi:10.4153/CJM-1961-015-3. 269
- [117] Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry*, 1(1):51–64, 1991. doi:10.1016/0925-7721(91)90012-4. 9
- [118] Raimund Seidel. Maintaining ideally distributed random search trees without extra space. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 134–142. Springer, 2009. doi:10.1007/978-3-642-03456-5_9. 75
- [119] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. doi:10.1007/BF01940876. 10, 75
- [120] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835. 75
- [121] Steffan Christ Sølvesten. I/O-efficient manipulation of binary decision diagrams, slides. Accessed on the 24. juli 2025. URL: <https://raw.githubusercontent.com/SSoelvsten/slides/out/adiar.core.pdf>. 46
- [122] Steffan Christ Sølvesten. Use random access for internal/prod2 when a decision diagram has small width, github issue. Accessed on the 24. juli 2025. URL: <https://github.com/ssoelvsten/adiar/issues/462>. 49
- [123] Steffan Christ Sølvesten, Jaco Van de Pol, Anna Blume Jakobsen, and Mathias Weller Berg Thomasen. Adiar: Binary Decision Diagrams in External Memory. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13244 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2022. doi:10.1007/978-3-030-99527-0_16. 43, 44, 45, 50
- [124] Steffan Christ Sølvesten, Casper Moldrup Rysgaard, and Jaco van de Pol. Adiar 2.0.0-beta.3 : Experiment data, 01 2024. doi:10.5281/zenodo.10493770. 50
- [125] Steffan Christ Sølvesten, Casper Moldrup Rysgaard, and Jaco van de Pol. Random access on narrow decision diagrams in external memory. In Thomas Neele and Anton Wijs, editors, *Model Checking Software - 30th International Symposium, SPIN 2024, Luxembourg City, Luxembourg, April 8-9, 2024, Proceedings*, volume 14624 of *Lecture Notes in Computer Science*, pages 137–145. Springer, 2024. doi:10.1007/978-3-031-66149-5_7. 3, 43, 47, 48

- [126] Steffan Christ Sølvsten and Jaco van de Pol. Adiar 1.1 - zero-suppressed decision diagrams in external memory. In Kristin Yvonne Rozier and Swarat Chaudhuri, editors, *NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May 16-18, 2023, Proceedings*, volume 13903 of *Lecture Notes in Computer Science*, pages 464–471. Springer, 2023. doi: 10.1007/978-3-031-33170-1_28. 49, 50
- [127] Steffan Christ Sølvsten, Jaco van de Pol, Anna Blume Jakobsen, and Mathias Weller Berg Thomasen. Efficient binary decision diagram manipulation in external memory. *CoRR*, abs/2104.12101, 2021. arXiv:2104.12101. 50
- [128] Jack Stade. The Point-Boundary Art Gallery Problem Is $\exists\mathbb{R}$ -Hard. In Oswin Aichholzer and Haitao Wang, editors, *41st International Symposium on Computational Geometry (SoCG 2025)*, volume 332 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 74:1–74:23, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.SoCG.2025.74. 51, 52, 63
- [129] Robert E. Tarjan and Christopher J. Van Wyk. An $O(n \log \log n)$ -time algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, 17(1):143–178, 1988. doi:10.1137/0217010. 287, 288, 332
- [130] Robert Endre Tarjan. *Data structures and network algorithms*, volume 44 of *CBMS-NSF regional conference series in applied mathematics*. SIAM, 1983. doi:10.1137/1.9781611970265. 78
- [131] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. doi:10.1137/0606031. 75, 77
- [132] Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997. doi:10.1109/69.599929. 14, 29
- [133] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811. 65, 66, 68, 70, 71, 267
- [134] J.W.J. Williams. Algorithm 232: Heapsort. *Commun. ACM*, 7(6):347–348, May 2025. doi:10.1145/512274.3734138. 35