Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components

KRISHNENDU CHATTERJEE, IST Austria
RASMUS IBSEN-JENSEN, IST Austria
AMIR KAFSHDAR GOHARSHADY, IST Austria
ANDREAS PAVLOGIANNIS, IST Austria

We study algorithmic questions wrt algebraic path properties in concurrent systems, where the transitions of the system are labeled from a complete, closed semiring. The algebraic path properties can model dataflow analysis problems, the shortest path problem, and many other natural problems that arise in program analysis. We consider that each component of the concurrent system is a graph with constant treewidth, a property satisfied by the controlflow graphs of most programs. We allow for multiple possible queries, which arise naturally in demand driven dataflow analysis. The study of multiple queries allows us to consider the tradeoff between the resource usage of the *one-time* preprocessing and for *each individual* query. The traditional approach constructs the product graph of all components and applies the best-known graph algorithm on the product. In this approach, even the answer to a single query requires the transitive closure (i.e., the results of all possible queries), which provides no room for tradeoff between preprocessing and query time.

Our main contributions are algorithms that significantly improve the worst-case running time of the traditional approach, and provide various tradeoffs depending on the number of queries. For example, in a concurrent system of two components, the traditional approach requires hexic time in the worst case for answering one query as well as computing the transitive closure, whereas we show that with one-time preprocessing in almost cubic time, each subsequent query can be answered in at most linear time, and even the transitive closure can be computed in almost quartic time. Furthermore, we establish conditional optimality results showing that the worst-case running time of our algorithms cannot be improved without achieving major breakthroughs in graph algorithms (i.e., improving the worst-case bound for the shortest path problem in general graphs). Preliminary experimental results show that our algorithms perform favorably on several benchmarks.

CCS Concepts: • **Theory of computation → Graph algorithms analysis; Program analysis;**

Additional Key Words and Phrases: Concurrent systems, Dataflow analysis, Constant-treewidth graphs, Algebraic path properties, Shortest path.

ACM Reference Format:
Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Amir Kafshdar Goharshady and Andreas Pavlogiannis, 2016. Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 0 (0), 42 pages.
DOI: 0000001.0000001

1. INTRODUCTION

In this work we consider concurrent finite-state systems where each component is a constant-treewidth graph, and the algorithmic question is to determine algebraic path properties between

The research was partly supported by Austrian Science Fund (FWF) Grant No P23499- N23, FWF NFN Grant No S11407-N23 (RiSE/SHiNE), and ERC Start grant (279307: Graph Games).
Author's addresses: K. Chatterjee, A. K. Goharshady, R. Ibsen-Jensen and A. Pavlogiannis, IST Austria (Institute of Science and Technology Austria) Klosterneuburg, Austria;

DOI: 0000001.0000001

pairs of nodes in the system. Our main contributions are algorithms which significantly improve the worst-case running time of the existing algorithms. We establish conditional optimality results for some of our algorithms in the sense that they cannot be improved without achieving major breakthroughs in the algorithmic study of graph problems. Finally, we provide a prototype implementation of our algorithms which significantly outperforms the existing algorithmic methods on several benchmarks.

Concurrency and algorithmic approaches. The analysis of concurrent systems is one of the fundamental problems in computer science in general, and programming languages in particular. A finite-state concurrent system consists of several components, each of which is a finite-state graph, and the whole system is a composition of the components. Since errors in concurrent systems are hard to reproduce by simulations due to combinatorial explosion in the number of interleavings, formal methods are necessary to analyze such systems. In the heart of the formal approaches are graph algorithms, which provide the basic search procedures for the problem. The basic graph algorithmic approach is to construct the product graph (i.e., the product of the component systems) and then apply the best-known graph algorithms on the product graph. While there are many practical approaches for the analysis of concurrent systems, a fundamental theoretical question is whether special properties of graphs that arise in analysis of programs can be exploited to develop asymptotically faster algorithms as compared to the basic approach.

Special graph properties for programs. A very well-studied notion in graph theory is the concept of *treewidth* of a graph, which is a measure of how similar a graph is to a tree (a graph has treewidth 1 precisely if it is a tree) [Robertson and Seymour 1984]. The treewidth of a graph is defined based on a *tree decomposition* of the graph [Halin 1976], see Section 2 for a formal definition. On one hand the treewidth property provides a mathematically elegant way to study graphs, and on the other hand there are many classes of graphs which arise in practice and have constant treewidth. The most important example is that the controlflow graph for goto-free programs for many programming languages are of constant treewidth [Thorup 1998], and it was also shown in [Gustedt et al. 2002] that typically all Java programs have constant treewidth.

Algebraic path properties. To specify properties of traces of concurrent systems we consider a very general framework, where edges of the system are labeled from a complete, closed semiring (which subsumes bounded and finite distributive semirings), and we refer to the labels of the edges as weights. For a given path, the weight of the path is the semiring product of the weights on the edges of the path, and the weights of different paths are combined using the semiring plus operator. For example, (i) the Boolean semiring (with semiring product as AND, and semiring plus as OR) expresses the reachability property; (ii) the tropical semiring (with real numbers as edge weights, semiring product as standard sum, and semiring plus as minimum) expresses the shortest path property; and (iii) with letter labels on edges, semiring product as string concatenation and semiring plus as union we can express the regular expression of reaching from one node to another. The algebraic path properties subsumes the dataflow analysis of the IFDS/IDE frameworks [Reps et al. 1995; Sagiv et al. 1996] in the intraprocedural setting, which consider compositions of distributive dataflow functions, and meet-over-all-paths as the semiring plus operator. Since IFDS/IDE is a special case of our framework, a large and important class of dataflow analysis problems that can be expressed in IFDS/IDE can also be expressed in our framework. However, the IFDS/IDE framework works for sequential interprocedural analysis, whereas we focus on intraprocedural analysis, but in the concurrent setting.

Expressiveness of algebraic path properties. The algebraic path properties provide an expressive framework with rich modeling power. Here we elaborate on three important classes.

(1) *Weighted shortest path.* The algebraic paths framework subsumes several problems on weighted graphs. The most well-known such problem is the shortest path problem [Floyd 1962; Warshall

1962; Bellman 1958; Ford 1956; Johnson 1977], phrased on the tropical semiring. For example, the edge weights (positive and negative) can express energy consumptions, and the shortest path problem asks for the least energy consuming path. Another important quantitative property is the *mean-payoff* property, where each edge weight represents a reward or cost, and the problem asks for a path that minimizes the average of the weights along a path. Many quantitative properties of relevance for program analysis (e.g., to express performance or resource consumption) can be modeled as mean-payoff properties [Chatterjee et al. 2015c; Cerny et al. 2013]. The mean-payoff and other fundamental problems on weighted graphs (e.g., the most probable path and the minimum initial credit problem) can be reduced to the shortest-path problem [Viterbi 1967; Lawler 1976; Karp 1978; Bouyer et al. 2008; Chatterjee et al. 2010; Cerny et al. 2013; Wilhelm et al. 2008; Chatterjee et al. 2015a].

(2) *Dataflow problems.* A wide range of dataflow problems have an algebraic paths formulation, expressed as a "meet-over-all-paths" analysis [Kildall 1973]. Perhaps the most well-known case is that of distributive flow functions considered in the IFDS framework [Reps et al. 1995; Sagiv et al. 1996]. Given a finite domain D and a universe F of distributive dataflow functions $f : 2^D \to 2^D$, a weight function $\mathsf{wt} : E \to F$ associates each edge of the controlflow graph with a flow function. The weight of a path is then defined as the composition of the flow functions along its edges, and the dataflow distance between two nodes u, v is the meet $\sqcap$ (union or intersection) of the weights of all $u \rightsquigarrow v$ paths. The problem can be formulated on the meet-composition semiring $(F, \sqcap, \circ, \emptyset, I)$, where I is the identity function. We note, however, that the IFDS/IDE framework considers interprocedural paths in sequential programs. In contrast, the current work focuses on intraprocedural analysis of concurrent programs. The dataflow analysis of concurrent programs has been a problem of intensive study (e.g. [Grunwald and Srinivasan 1993; Knoop et al. 1996; Farzan and Madhusudan 2007; Chugh et al. 2008; Kahlon et al. 2009; De et al. 2011]), where (part of) the underlying analysis is based on an algebraic, "meet-over-all-paths" approach.

(3) *Regular expressions.* Consider the case that each edge is annotated with an observation or action. Then the regular expression to reach from one node to another represents all the sequences of observable actions that lead from the start node to the target. The regular languages of observable actions have provided useful formulations in the analysis and synthesis of concurrent systems [Dwyer et al. 2004; Farzan et al. 2013; Cerny et al. 2015]. Regular expressions have also been used as algebraic relaxations of interprocedurally valid paths in sequential and concurrent systems [Yan et al. 2011; Bouajjani et al. 2003].

The algorithmic problem. In graph theoretic parlance, graph algorithms typically consider two types of queries: (i) a *pair query* given nodes u and v (called (u, v)-pair query) asks for the algebraic path property from u to v; and (ii) a *single-source* query given a node u asks for the answer of (u, v)-pair queries for all nodes v. In the context of concurrency, in addition to the classical pair and single-source queries, we also consider *partial* queries. Given a concurrent system with k components, a node in the product graph is a tuple of k component nodes. A *partial* node $\overline{u}$ in the product only specifies nodes of a nonempty strict subset of all the components. Our work also considers partial pair and partial single-source queries, where the input nodes are partial nodes. Queries on partial nodes are very natural, as they capture properties between local locations in a component, that are shaped by global paths in the whole concurrent system. For example, constant propagation and dead code elimination are local properties in a program, but their analysis requires analyzing the concurrent system as a whole.

Preprocess vs query. A topic of widespread interest in the programming languages community is that of on-demand analysis [Babich and Jazayeri 1978; Zadeck 1984; Horwitz et al. 1995; Duesterwald et al. 1995; Reps 1995; Sagiv et al. 1996; Reps 1997; Yuan et al. 1997; Naeem et al. 2010; Chatterjee et al. 2015b]. Such analysis has several advantages, such as (quoting from [Horwitz et al. 1995; Reps 1997]) (i) narrowing down the focus to specific points of interest, (ii) narrowing down the focus to specific dataflow facts of interest, (iii) reducing work in preliminary phases, (iv) sidestepping incremental updating problems, and (v) offering demand analysis as a user-level operation. For example, in alias analysis, the question is whether two pointers may point to the same object, which is by definition modeled as a question between a pair of nodes. Similarly, in constant propagation a relevant question is whether some variable remains constant between a pair of controlflow locations. The problem of on-demand analysis allows us to distinguish between a single preprocessing phase (one time computation), and a subsequent query phase, where queries are answered on demand. The two extremes of the preprocessing and query phase are: (i) *complete preprocessing* (aka *transitive closure* computation) where the result is precomputed for every possible query, and hence queries are answered by simple table lookup; and (ii) *no preprocessing* where every query requires a new computation. However, in general, there can be a tradeoff between the preprocessing and query computation. Most of the existing works for on-demand analysis do not make a formal distinction between preprocessing and query phases, as the provided complexities only guarantee the *same worst-case complexity* property, namely that the total time for handling any sequence of queries is no worse than the complete preprocessing. Hence most existing tradeoffs are practical, without any theoretical guarantees.

	Preprocess		**Query time**			
	Time	Space	Single-source	Pair	Partial single-source	Partial pair
Previous results [Lehmann 1977; Floyd 1962] [Warshall 1962; Kleene 1956]	$O(n^6)$	$O(n^4)$	$O(n^2)$	$O(1)$	$O(n^2)$	$O(1)$
Our result Corollary 5.9 ($\epsilon > 0$)	$\mathbf{O(n^3)}$	$\mathbf{O(n^{2+\epsilon})}$	$\mathbf{O(n^{2+\epsilon})}$	$\mathbf{O(n^2)}$	$\mathbf{O(n^{2+\epsilon})}$	$\mathbf{O(n^2)}$
Our result Theorem 5.6 ($\epsilon > 0$)	$\mathbf{O(n^{3+\epsilon})}$	$\mathbf{O(n^3)}$	$\mathbf{O(n^{2+\epsilon})}$	$\mathbf{O(n)}$	$\mathbf{O(n^2)}$	$\mathbf{O(1)}$
Our result Corollary 5.10 ($\epsilon > 0$)	$\mathbf{O(n^{4+\epsilon})}$	$\mathbf{O(n^4)}$	$\mathbf{O(n^2)}$	$\mathbf{O(1)}$	$\mathbf{O(n^2)}$	$\mathbf{O(1)}$

Table I: The algorithmic complexity for computing algebraic path queries wrt a closed, complete semiring on a concurrent graph G which is the product of two constant-treewidth graphs G_1, G_2, with n nodes each.

Previous results. In this work we consider finite-state concurrent systems, where each component graph has constant treewidth, and the trace properties are specified as algebraic path properties. Our framework can model a large class of problems: typically the controlflow graphs of programs have constant treewidth [Thorup 1998; Gustedt et al. 2002; Burgstaller et al. 2004], and if there is a constant number of synchronization variables with constant-size domains, then each component graph has constant treewidth. Note that this imposes little practical restrictions, as typically synchronization variables, such as locks, mutexes and condition variables have small (even binary) domains (e.g. locked/unlocked state). The best-known graph algorithm for the algebraic path property problem is the classical Warshall-Floyd-Kleene [Lehmann 1977; Floyd 1962; Warshall 1962; Kleene 1956] style dynamic programming, which requires cubic time. Two well-known special cases of the algebraic paths problem are (i) computing the shortest path from a source to a target node in a weighted graph, and (ii) computing the regular expression from a source to a target node in an automaton whose edges are labeled with letters from a finite alphabet. In the first case, the best-known algorithm is the Bellman-Ford algorithm with time complexity $O(n \cdot m)$, where n and m are the number of nodes and edges, respectively. In the second case, the well-known construction of Kleene's [Kleene 1956] theorem requires cubic time. The only existing algorithmic approach for the problem we consider is to first construct the product graph (thus if each component graph has size n, and there are k components, then the product graph has size $O(n^k)$), and then apply the best-known graph algorithm (thus the overall time complexity is $O(n^{3 \cdot k})$ for algebraic path properties). Hence for the important special case of two components we obtain a hexic-time (i.e., $O(n^6)$) algorithm. Moreover, for algebraic path properties the current best-known algorithms for one pair query (or one single-source query) computes the entire transitive closure. Hence the existing approach does not allow a tradeoff of preprocessing and query as even for one query the entire transitive closure is computed.

Our contributions. Our main contributions are improved algorithmic upper bounds, proving several optimality results of our algorithms, and experimental results. Below all the complexity measures (time and space) are in the number of basic machine operations and number of semiring operations. We elaborate our contributions below.

(1) *Improved upper bounds.* We present improved upper bounds both for generally k components, and the important special case of two components.
  — *General case.* We show that for $k \geqslant 3$ components with n nodes each, after $O(n^{3\cdot(k-1)})$ preprocessing time, we can answer (i) single-source queries in $O(n^{2\cdot(k-1)})$ time, (ii) pair queries in $O(n^{k-1})$ time, (iii) partial single-source queries in $O(n^k)$ time, and (iv) partial pair queries in $O(1)$ time; while using at all times $O(n^{2\cdot k-1})$ space. In contrast, the existing methods [Lehmann 1977; Floyd 1962; Warshall 1962; Kleene 1956] compute the transitive closure even for a single query, and thus require $O(n^{3\cdot k})$ time and $O(n^{2\cdot k})$ space.
  — *Two components.* For the important case of two components, the existing methods require $O(n^6)$ time and $O(n^4)$ space even for one query. In contrast, we establish a variety of tradeoffs between preprocessing and query times, and the best choice depends on the number of expected queries. In particular, for any fixed $\epsilon > 0$, we establish the following three results.

    Three results. First, we show (Corollary 5.9) that with $O(n^3)$ preprocessing time and using $O(n^{2+\epsilon})$ space, we can answer single-source queries in $O(n^{2+\epsilon})$ time, and pair and partial pair queries require $O(n^2)$ time. Second, we show (Theorem 5.6) that with $O(n^{3+\epsilon})$ preprocessing time and using $O(n^3)$ space, we can answer pair and partial pair queries in time $O(n)$ and $O(1)$, respectively. Third, we show (Corollary 5.10) that the transitive closure can be computed using $O(n^{4+\epsilon})$ preprocessing time and $O(n^4)$ space, after which single-source queries require $O(n^2)$ time, and pair and partial pair queries require $O(1)$ time (i.e., all queries require linear time in the size of the output).

    Tradeoffs. Our results provide various tradeoffs: The first result is best for answering $O(n^{1+\epsilon})$ pair and partial pair queries; the second result is best for answering between $\Omega(n^{1+\epsilon})$ and $O(n^{3+\epsilon})$ pair queries, and $\Omega(n^{1+\epsilon})$ partial pair queries; and the third result is best when answering $\Omega(n^{3+\epsilon})$ pair queries. Observe that the transitive closure computation is preferred when the number of queries is large, in sharp contrast to the existing methods that compute the transitive closure even for a single query. Our results are summarized in Table I and the tradeoffs are pictorially illustrated in Figure 1.

(2) *Optimality of our results.* Given our significant improvements for the case of two components, a very natural question is whether the algorithms can be improved further. While presenting matching bounds for polynomial-time graph algorithms to establish optimality is very rare in the whole of computer science, we present *conditional lower bounds* which show that our combined preprocessing and query time cannot be improved without achieving a major breakthrough in graph algorithms.
  — *Almost optimality.* First, note that in the first result (obtained from Corollary 5.9) our space usage and single-source query time are arbitrarily close to optimal, as both the input and the output have size $\Theta(n^2)$. Moreover, the result is achieved with preprocessing time less than $\Omega(n^4)$, which is a lower bound for computing the transitive closure (which has n^4 entries). Furthermore, in our third result (obtained from Corollary 5.10) the $O(n^{4+\epsilon})$ preprocessing time is arbitrarily close to optimal, and the $O(n^4)$ preprocessing space is indeed optimal, as the transitive closure computes the distance among all n^4 pairs of nodes (which requires $\Omega(n^4)$ time and space).

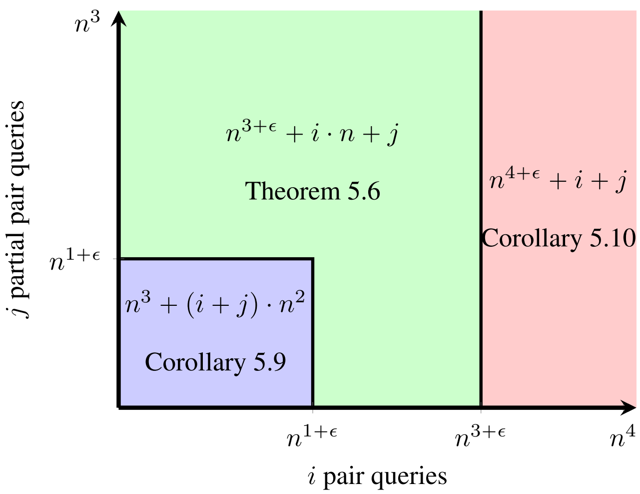

Fig. 1: Given a concurrent graph G of two constant-treewidth graphs of n nodes each, the figure illustrates the time required by the variants of our algorithms to preprocess G, and then answer i pair queries and j partial pair queries. The different regions correspond to the best variant for handling different number of such queries. In contrast, the current best solution requires $O(n^6 + i + j)$ time. For ease of presentation we omit the $O(\cdot)$ notation.

— *Conditional lower bound.* In recent years, the conditional lower bound problem has received vast attention in complexity theory, where under the assumption that certain problems (such as matrix multiplication, all-pairs shortest path) cannot be solved faster than the existing upper bounds, lower bounds for other problems (such as dynamic graph algorithms) are obtained [Abboud and Williams 2014; Abboud et al. 2015; Henzinger et al. 2015]. The current best-known algorithm for algebraic path properties for general (not constant-treewidth) graphs is cubic in the number of nodes. Even for the special case of shortest paths with positive and negative weights, the best-known algorithm (which has not been improved over five decades) is $O(n \cdot m)$, where m is the number of edges. Since m can be $\Omega(n^2)$, the current best-known worst-case complexity is cubic in the number of nodes. We prove that pair queries require more time in a concurrent graph of two constant-treewidth graphs, with n nodes each, than in general graphs with n nodes. This implies that improving the $O(n^3)$ combined preprocessing and query time over our result (from Corollary 5.9) for answering r queries, for $r = O(n)$, would yield the same improvement over the $O(n^3)$ time for answering r pair queries in general graphs. That is, the combination of our preprocessing and query time (from Corollary 5.9) cannot be improved without equal improvement on the long standing cubic bound for the shortest path and the algebraic path problems in general graphs. Additionally, our result (from Theorem 5.6) cannot be improved much further even for n^2 queries, as the combined time for preprocessing and answering n^2 queries is $O(n^{3+\epsilon})$ using Theorem 5.6, while the existing bound is $O(n^3)$ for general graphs.

(3) *Experimental results.* We provide a prototype implementation of our algorithms which significantly outperforms the baseline methods on several benchmarks.

Technical contributions. The results of this paper rely on several novel technical contributions.

(1) *Upper bounds*. Our upper bounds depend on a series of technical results.
  (a) The first key result is an algorithm for constructing a *strongly balanced* tree-decomposition T. A tree is called (β, γ)-balanced if for every node u and descendant v of u that appears γ levels below, the size of the subtree of T rooted at v is at most a β fraction of the size of the

subtree of T rooted at u. For any fixed $\delta > 0$ and $\lambda \in \mathbb{N}$ with $\lambda \geqslant 2$, let $\beta = ((1+\delta)/2)^{\lambda-1}$ and $\gamma = \lambda$. We show that a (β, γ)-balanced tree decomposition of a constant-treewidth graph with n nodes can be constructed in $O(n \cdot \log n)$ time and $O(n)$ space. To our knowledge, this is the first algorithm that constructs a tree decomposition with such a strong notion of balance. This property is crucial for achieving the resource bounds of our algorithms for algebraic paths. The construction is presented in Section 3.

(b) Given a concurrent graph G obtained from k constant-treewidth graphs G_i, we show how a tree-decomposition of G can be constructed from the strongly balanced tree-decompositions T_i of the components G_i, in time that is linear in the size of the output. We note that G can have large treewidth, and thus determining the treewidth of G can be computationally expensive. Instead, our construction avoids computing the treewidth of G, and directly constructs a tree-decomposition of G from the strongly balanced tree decompositions T_i. The construction is presented in Section 4.

(c) Given the above tree-decomposition algorithm for concurrent graphs G, in Section 5 we present the algorithms for handling algebraic path queries. In particular, we introduce the *partial expansion* $\overline{G}$ of G for additionally handling partial queries, and describe the algorithms for preprocessing and querying $\overline{G}$ in the claimed time and space bounds.

(2) *Lower bound*. Given an arbitrary graph G (not of constant treewidth) of n nodes, we show how to construct a constant-treewidth graph G'' of $2 \cdot n$ nodes, and a graph G' that is the product of G'' with itself, such that algebraic path queries in G coincide with such queries in G'. This construction requires quadratic time on n. The conditional optimality of our algorithms follows, as improvement over our algorithms must achieve the same improvement for algebraic path properties on arbitrary graphs.

All our algorithms are simple to implement and provided as pseudocode in the Appendix.

A preliminary version of this work has appeared in [Chatterjee et al. 2016]. The current version expands upon [Chatterjee et al. 2016] by including detailed algorithms, full proofs, and additional examples.

1.1. Related Works

Treewidth of graphs. The notion of treewidth of graphs as an elegant mathematical tool to analyze graphs was introduced in [Robertson and Seymour 1984]. The significance of constant treewidth in graph theory is large mainly because several problems on graphs become complexity-wise easier. Given a tree decomposition of a graph with low treewidth t, many NP-complete problems for arbitrary graphs can be solved in time polynomial in the size of the graph, but exponential in t [Arnborg and Proskurowski 1989; Bern et al. 1987; Bodlaender 1988; 1993; 2005]. Even for problems that can be solved in polynomial time, faster algorithms can be obtained for low treewidth graphs, e.g., for the distance problem [Chaudhuri and Zaroliagis 1995]. The constant-treewidth property of graphs has also been used in the context of logic: Monadic Second Order (MSO) logic is a very expressive logic, and a celebrated result of [Courcelle 1990] showed that for constant-treewidth graphs the decision questions for MSO can be solved in polynomial time; and the result of [Elberfeld et al. 2010] shows that this can even be achieved in deterministic log-space. Various other models (such as probabilistic models of Markov decision processes and games played on graphs for synthesis) with the constant-treewidth restriction have also been considered [Chatterjee and Lacki 2013; Obdrzálek 2003]. The problem of computing a balanced tree decomposition for a constant treewidth graph was considered in [Reed 1992]. More importantly, in the context of programming languages, it was shown in [Thorup 1998] that the controlflow graphs of goto-free programs in many programming languages have constant treewidth. This theoretical result was subsequently followed up in several practical approaches, and although in the presence of gotos the treewidth is not guaranteed to be bounded, it has been shown that programs in several programming languages have typically low treewidth [Gustedt et al. 2002; Burgstaller et al. 2004]. The constant-treewidth property of graphs has been used to develop faster algorithms for sequential interprocedural analysis [Chatterjee et al. 2015b], and on the analysis of automata with auxiliary storage (e.g., stacks and queues) [Madhusudan and Parlato 2011]. These results have been followed in practice, and some compilers (e.g., SDCC) implement tree-decomposition-based algorithms for performance optimizations [Krause 2013].

Concurrent system analysis. The problem of concurrent system analysis has been considered in several works, both for intraprocedural as well context-bounded interprocedural analysis [Harel et al. 1997; Alur et al. 1999; Farzan et al. 2013; Qadeer and Rehof 2005; Bouajjani et al. 2005; La Torre et al. 2008; Lal et al. 2008; Lal and Reps 2009; Kahlon et al. 2013], and many practical tools have been developed as well [Qadeer and Rehof 2005; Lal and Reps 2009; Suwimonteerabuth et al. 2008; Lal et al. 2012]. In this work we focus on the intraprocedural analysis with constant-treewidth graphs, and present algorithms with better asymptotic complexity. To our knowledge, none of the previous works consider the constant-treewidth property, nor do they improve the asymptotic complexity of the basic algorithm for the algebraic path property problem.

2. DEFINITIONS

In this section we present definitions related to semirings, graphs, concurrent graphs, and tree decompositions. We start with some basic notation on sets and sequences.

Notation on sets and sequences. Given a number $r \in \mathbb{N}$, we denote by $[r] = \{1, 2, \dots, r\}$ the natural numbers from 1 to r. Given a set X and a $k \in \mathbb{N}$, we denote by $X^k = \prod_{i=1}^{k} X$, the k times Cartesian product of X. A sequence $x_1, \dots x_k$ is denoted for short by $(x_i)_{1 \leqslant i \leqslant k}$, or $(x_i)_i$ when k is implied from the context. Given a sequence Y, we denote by $y \in Y$ the fact that y appears in Y.

2.1. Complete, closed semirings

Definition 2.1 (*Complete, closed semirings*). We fix a complete *semiring* $S = (\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$ where Σ is a countable set, $\oplus$ and $\otimes$ are binary operators on Σ, and $\overline{\mathbf{0}}, \overline{\mathbf{1}} \in \Sigma$, and the following properties hold:

(1) $\oplus$ is infinitely associative, commutative, and $\overline{\mathbf{0}}$ is the neutral element,
(2) $\otimes$ is associative, and $\overline{\mathbf{1}}$ is the neutral element,
(3) $\otimes$ infinitely distributes over $\oplus$,
(4) $\overline{\mathbf{0}}$ absorbs in multiplication, i.e., $\forall a \in \Sigma : a \otimes \overline{\mathbf{0}} = \overline{\mathbf{0}}$.

Additionally, we consider that S is *idempotent*, that is, $\forall s \in \Sigma$ we have that $s \oplus s = s$. The idempotence property defines a partial order $\preceq \subseteq \Sigma \times \Sigma$, such that $\forall s_1, s_2 \in \Sigma$, we have that $s_1 \preceq s_2$ iff $s_1 \oplus s_2 = s_1$. Finally, we consider that S is equipped with a *closure* operator *, such that $\forall s \in \Sigma :\ s^* = \overline{\mathbf{1}} \oplus (s \otimes s^*) = \overline{\mathbf{1}} \oplus (s^* \otimes s)$ (i.e., the semiring is *closed*).

In the remaining of this document we fix a semiring $S = (\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$, and we will consider graphs labeled with elements of Σ.

2.2. Graphs and tree decompositions

Graphs and weighted paths. Let $G = (V, E)$ be a weighted finite directed graph (henceforth called simply a graph) where V is a set of n nodes and $E \subseteq V \times V$ is an edge relation, along with a weight function $\mathsf{wt} : E \to \Sigma$ that assigns to each edge of G an element from Σ. Given a set of nodes $X \subseteq V$, we denote by $G[X] = (X, E \cap (X \times X))$ the subgraph of G induced by X. A path $P : u \rightsquigarrow v$ is a sequence of nodes $(x_1, \dots, x_k)$ such that $x_1 = u$, $x_k = v$, and for all $1 \leqslant i < k$

we have $(x_i, x_{i+1}) \in E$. The length of P is $|P| = k-1$, and a single node is itself a 0-length path. A path P is *simple* if no node repeats in the path (i.e., it does not contain a cycle). Given a path $P = (x_1, \ldots, x_k)$, the weight of P is $\otimes(P) = \bigotimes(\mathsf{wt}(x_i, x_{i+1}))_i$ if $|P| \geqslant 1$ else $\otimes(P) = \overline{\mathbf{1}}$. Given nodes $u, v \in V$, the *semiring distance* $d(u,v)$ is defined as $d(u,v) = \bigoplus_{P:u \rightsquigarrow v} \otimes(P)$, and $d(u,v) = \overline{\mathbf{0}}$ if no such P exists.

Trees. A *tree* $T = (V, E)$ is an undirected graph with a root node u_0, such that between every two nodes there is a unique simple path. For a node u we denote by $\mathsf{Lv}(u)$ the *level* of u which is defined as the length of the simple path from u_0 to u. A *child* of a node u is a node v such that $\mathsf{Lv}(v) = \mathsf{Lv}(u) + 1$ and $(u, v) \in E$, and then u is the *parent* of v. For a node u, any node (including u itself) that appears in the path from u_0 to u is an *ancestor* of u, and if v is an ancestor of u, then u is a *descendant* of v. Given two nodes u, v, the *lowest common ancestor (LCA)* is the common ancestor of u and v with the highest level. Given a tree T, a *contiguous subtree* is subgraph (X, E') of T such that $E' = E \cap (X \times X)$ and for every pair $u, v \in X$, every node that appears in the unique path from u to v belongs to X. A tree is *k-ary* if every node has at most k-children (e.g., a binary tree has at most two children for every node). In a *full k-ary* tree, every node has 0 or k-children.

Tree decompositions. A *tree-decomposition* $\mathrm{Tree}(G) = T = (V_T, E_T)$ of a graph G is a tree, where every node B_i in T is a subset of nodes of G such that the following conditions hold:

C1 $V_T = \{B_0, \ldots, B_b\}$ with $B_i \subseteq V$, and $\bigcup_{B_i \in V_T} B_i = V$ (every node is covered).
C2 For all $(u, v) \in E$ there exists $B_i \in V_T$ such that $u, v \in B_i$ (every edge is covered).
C3 For all i, j, k such that there is a bag B_k that appears in the simple path $B_i \rightsquigarrow B_j$ in $\mathrm{Tree}(G)$, we have $B_i \cap B_j \subseteq B_k$ (every node appears in a contiguous subtree of T).

The sets B_i which are nodes in V_T are called *bags*. We denote by $|T| = |V_T|$ the number of bags in T. Conventionally, we call B_0 the root of T, and denote by $\mathsf{Lv}(B_i)$ the level of B_i in $\mathrm{Tree}(G)$. For a bag B of T, we denote by $T(B)$ the subtree of T rooted at B. A bag B is called the *root bag* of a node u if $u \in B$ and every B' that contains u appears in $T(B)$. We often use B_u to refer to the root bag of u, and define $\mathsf{Lv}(u) = \mathsf{Lv}(B_u)$. Given a bag B, we denote by

(1) $V_T(B)$ the nodes of G that appear in bags in $T(B)$,
(2) $\mathcal{V}_T(B)$ the nodes of G that appear in B and its ancestors in T.

The *width* of the tree-decomposition T is the size of the largest bag minus 1. The treewidth t of G is the smallest width among the widths of all tree decompositions of G. We say that G has constant treewidth if t is fixed and independent of n (i.e., $t = O(1)$). Note that if T achieves the treewidth of G, we have $|V_T(B)| \leqslant (t+1) \cdot |T(B)|$. Given a graph G with treewidth t and a fixed $\alpha \in \mathbb{N}$, a tree-decomposition $\mathrm{Tree}(G)$ is called *α-approximate* if it has width at most $\alpha \cdot (t+1) - 1$. Figure 2 illustrates the above definitions on a small example.

Remark 2.2. It follows directly from the definition of tree decompositions that for every bag B and nodes $u, v \in B$, if $\mathsf{Lv}(u) \leqslant \mathsf{Lv}(v)$ then B_v is a descendant of B_u. We make use of this property in the proofs of our algorithms.

2.3. Concurrent graphs

Product graphs. A graph $G_{\mathsf{p}} = (V_{\mathsf{p}}, E_{\mathsf{p}})$ is said to be the *product graph* of k graphs $(G_i = (V_i, E_i))_{1 \leqslant i \leqslant k}$ if $V_{\mathsf{p}} = \prod_i V_i$ and E_{p} is such that for all $u, v \in V_{\mathsf{p}}$ with $u = \langle u_i \rangle_{1 \leqslant i \leqslant k}$ and $v = \langle v_i \rangle_{1 \leqslant i \leqslant k}$, we have $(u, v) \in E_{\mathsf{p}}$ iff there exists a set $\mathcal{I} \subseteq [k]$ such that (i) $(u_i, v_i) \in E_i$ for all $i \in \mathcal{I}$, and (ii) $u_i = v_i$ for all $i \notin \mathcal{I}$. In words, an edge $(u, v) \in E_{\mathsf{p}}$ is formed in the product graph by traversing a set of edges $\{(u_i, v_i) \in E_i\}_{i \in \mathcal{I}}$ in some component graphs $\{G_i\}_{i \in \mathcal{I}}$, and traversing no edges in the remaining $\{G_i\}_{i \notin \mathcal{I}}$. We say that G_{p} is the *k-self-product* of a graph G' if $G_i = G'$ for all $1 \leqslant i \leqslant k$.

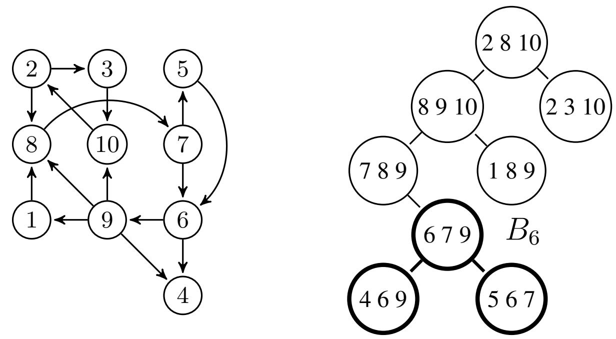

Fig. 2: A graph G with treewidth 2 (left) and a corresponding tree-decomposition $T = \mathrm{Tree}(G)$ of 8 bags and width 2 (right). The distinguished bag B_6 is the root bag of node 6. We have $V_T(B_6) = \{6, 7, 9, 4, 5\}$ and $\mathcal{V}_T(B_6) = \{6, 7, 9, 8, 10, 2\}$. The subtree $T(B_6)$ is shown in bold.

Concurrent graphs. A graph $G = (V, E)$ is called a *concurrent graph* of k graphs $(G_i = (V_i, E_i))_{1 \leqslant i \leqslant k}$ if $V = V_{\mathsf{p}}$ and $E \subseteq E_{\mathsf{p}}$, where $G_{\mathsf{p}} = (V_{\mathsf{p}}, E_{\mathsf{p}})$ is the product graph of $(G_i)_i$. Given a concurrent graph $G = (V, E)$ and a node $u \in V$, we will denote by u_i the i-th constituent of u. We say that G is a *k-self-concurrent* of a graph G' if G_{p} is the *k-self-product* of G'.

Various notions of composition. The framework we consider is quite general as it captures various different notions of concurrent composition. Indeed, the edge set of the concurrent graph is any possible subset of the edge set of the corresponding product graph. Then, two well-known composition notions can be modeled as follows. For any edge $(u, v) \in E$ of the concurrent graph G, let $\mathcal{I}_{u,v} = \{i \in [k] : (u_i, v_i) \in E_i\}$ denote the components that execute a transition in (u, v).

(1) In *synchronous composition* at every step all components make one move each simultaneously. This is captured by $\mathcal{I}_{u,v} = [k]$ for all $(u, v) \in E$.
(2) In *asynchronous composition* at every step only one component makes a move. This is captured by $|\mathcal{I}_{u,v}| = 1$ for all $(u, v) \in E$.

Thus the framework we consider is not specific to any particular notion of composition, and all our results apply to various different notions of concurrent composition that exist in the literature.

Partial nodes of concurrent graphs. A *partial node* $\overline{u}$ of a concurrent graph G is an element of $\prod_i (V_i \cup \{\bot\})$, where $\bot \notin \bigcup_i V_i$. Intuitively, $\bot$ is a fresh symbol to denote that a component is unspecified. A partial node $\overline{u}$ is said to *refine* a partial node $\overline{v}$, denoted by $\overline{u} \sqsubseteq \overline{v}$ if for all $1 \leqslant i \leqslant k$ either $\overline{v}_i = \bot$ or $\overline{v}_i = \overline{u}_i$. We say that the partial node $\overline{u}$ *strictly refines* $\overline{v}$, denoted by $\overline{u} \sqsubset \overline{v}$, if $\overline{u} \sqsubseteq \overline{v}$ and $\overline{u} \neq \overline{v}$ (i.e., for at least one constituent i we have $\overline{v}_i = \bot$ but $\overline{u}_i \neq \bot$). A partial node $\overline{u}$ is called *strictly partial* if it is strictly refined by some node $u \in V$ (i.e., $\overline{u}$ has at least one $\bot$). The notion of semiring distances is extended to partial nodes, and for partial nodes $\overline{u}, \overline{v}$ of G we define the semiring distance from $\overline{u}$ to $\overline{v}$ as

$$d(\overline{u}, \overline{v}) = \bigoplus_{u \sqsubseteq \overline{u}, v \sqsubseteq \overline{v}} d(u, v)$$

where $u, v \in V$. In the sequel, a partial node $\overline{u}$ will be either (i) a node of V, or (ii) a strictly partial node. We refer to nodes of the first case as actual nodes, and write u (i.e., without the bar). Distances where one endpoint is a strictly partial node $\overline{u}$ succinctly quantify over all nodes of all the components for which the corresponding constituent of $\overline{u}$ is $\bot$. Observe that the distance still depends on the unspecified components.

The algebraic paths problem on concurrent graphs of constant-treewidth components. In this work we are interested in the following problem. Let $G = (V, E)$ be a concurrent graph of $k \geqslant 2$

Method: DiningPhilosophers

```
while True do
  while fork not mine or knife not mine
    do
    if fork is free then
      lock(ℓ)
      acquire(fork)
      unlock(ℓ)
    end
    if knife is free then
      lock(ℓ)
      acquire(knife)
      unlock(ℓ)
    end
  end
  dine(fork, knife)// for some time
  lock(ℓ)
  release(fork)
  release(knife)
  unlock(ℓ)
  discuss()// for some time
end
```

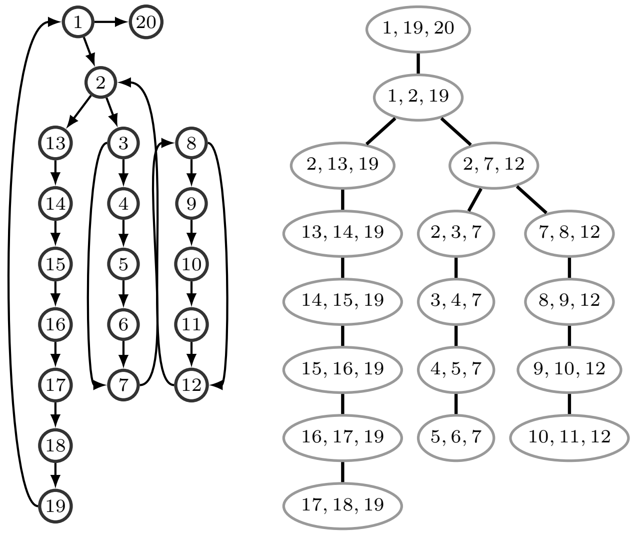

Fig. 3: A concurrent program (left), its controlflow graph (middle), and a tree decomposition of the controlflow graph (right).

constant-treewidth graphs $(G_i = (V_i, E_i))_{1 \leqslant i \leqslant k}$, and $\mathsf{wt} : E \to \Sigma$ be a weight function that assigns to every edge of G a weight from a set Σ that forms a complete, closed semiring $S = (\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$. The *algebraic path problem* on G asks the following types of queries:

(1) *Single-source query.* Given a partial node $\overline{u}$ of G, return the distance $d(\overline{u}, v)$ to every node $v \in V$. When the partial node $\overline{u}$ is an actual node of G, we have a traditional single-source query.
(2) *Pair query.* Given two nodes $u, v \in V$, return the distance $d(u, v)$.
(3) *Partial pair query.* Given two partial nodes $\overline{u}, \overline{v}$ of G where at least one is strictly partial, return the distance $d(\overline{u}, \overline{v})$.

Figure 3 presents the notions introduced in this section on a toy example on the dining philosophers problem. See Section 7 for an example on pair and partial pair queries in the analysis of the dining philosophers program.

Input parameters. For technical convenience, we consider a uniform upper bound n on the number of nodes of each G_i (i.e. $|V_i| \leqslant n$). Similarly, we let $t = O(1)$ be an upper bound on the treewidth of each G_i. The number k is taken to be fixed and independent of n. The input of the problem consists of the graphs $(G_i)_{1 \leqslant i \leqslant k}$, together with some representation of the edge relation E of G.

Complexity measures. The complexity of our algorithms is measured as a function of n. In particular, we ignore the size of the representation of E when considering the size of the input. This has the advantage of obtaining complexity bounds that are independent of the representation of E, which can be represented implicitly (such as synchronous or asynchronous composition) or explicitly, depending on the modeling of the problem under consideration. The time complexity of our algorithms is measured in number of operations, with each operation being either a basic machine operation, or an application of one of the operations of the semiring.

3. STRONGLY BALANCED TREE DECOMPOSITIONS

In this section we introduce the notion of strongly balanced tree decompositions, and present an algorithm for computing them efficiently on constant-treewidth graphs. Informally, a strongly balanced tree-decomposition is a binary tree-decomposition in which the number of descendants of each bag is typically approximately half of that of its parent. The following sections make use of this construction.

Strongly balanced tree decompositions. Given a binary tree-decomposition T and constants $0 < \beta < 1$, $\gamma \in \mathbb{N}^+$, a bag B of T is called (β,γ)-*balanced* if for every descendant B_i of B with $\mathsf{Lv}(B_i) - \mathsf{Lv}(B) = \gamma$, we have $|T(B_i)| \leqslant \beta \cdot |T(B)|$, i.e., the number of bags in $T(B_i)$ is at most a β-fraction of those in $T(B)$. A tree-decomposition T is called a (β,γ)-balanced tree-decomposition if every bag of T is (β,γ)-balanced. A (β,γ)-balanced tree-decomposition that is α-approximate is called an (α,β,γ)-balanced tree-decomposition. The following theorem is central to the results obtained in this paper. The proof is technical and presented later in this section. Here we provide a sketch of the algorithm for obtaining it.

THEOREM 3.1. *For every graph G with n nodes and constant treewidth, for any fixed $\delta > 0$ and $\lambda \in \mathbb{N}$ with $\lambda \geqslant 2$, let $\alpha = 6 \cdot \lambda/\delta$, $\beta = ((1+\delta)/2)^{\lambda-1}$, and $\gamma = \lambda$. A binary (α,β,γ) tree-decomposition $\mathrm{Tree}(G)$ with $O(n)$ bags can be constructed in $O(n \cdot \log n)$ time and $O(n)$ space.*

Sketch of Theorem 3.1. The construction of Theorem 3.1 considers that a tree-decomposition T' of G that has width t and $O(n)$ bags is given (which can be obtained using e.g. [Bodlaender 1996] in $O(n)$ time). Given the parameters $\delta > 0$ and $\lambda \in \mathbb{N}$ with $\lambda \geqslant 2$, T' is turned to an (α,β,γ)-balanced tree-decomposition, for $\alpha = 6 \cdot \lambda/\delta$, $\beta = ((1+\delta)/2)^{\lambda-1}$, and $\gamma = \lambda$, in two conceptual steps.

(1) A tree of bags $\mathsf{R_G}$ is constructed, which is (β,γ)-balanced.
(2) $\mathsf{R_G}$ is turned to an α-approximate tree decomposition of G.

The first construction is obtained by a recursive algorithm Rank, which operates on inputs $(\mathcal{C},\ell)$. The argument $\mathcal{C}$ represents a component of T', defined as a set of bags of T'. The argument ℓ is such that $\ell \in [\lambda]$, and it specifies the type of operation the algorithm performs on $\mathcal{C}$. Given such a component $\mathcal{C}$, we denote by $\mathsf{Nh}(\mathcal{C})$ the *neighborhood* of $\mathcal{C}$, defined as the set of bags of T' that are incident to $\mathcal{C}$ (but not including bags of $\mathcal{C}$). Informally, on input $(\mathcal{C},\ell)$, the algorithm partitions $\mathcal{C}$ into two sub-components $\overline{\mathcal{C}}_1$ and $\overline{\mathcal{C}}_2$ such that either (i) the size of each $\overline{\mathcal{C}}_i$ is approximately half the size of $\mathcal{C}$, or (ii) the size of the neighborhood of each $\overline{\mathcal{C}}_i$ is approximately half the size of the neighborhood of $\mathcal{C}$. In more detail,

(1) If $\ell > 0$, then $\mathcal{C}$ is partitioned into components $\mathcal{Y} = (\mathcal{C}_1,\dots,\mathcal{C}_r)$, by removing a list of bags $\mathcal{X} = (B_1,\dots B_m)$, such that $|\mathcal{C}_i| \leqslant \frac{\delta}{2} \cdot |\mathcal{C}|$. The union of $\mathcal{X}$ yields a new bag $\mathcal{B}$ in $\mathsf{R_G}$. Then $\mathcal{Y}$ is merged into two components $\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2$ with $|\overline{\mathcal{C}}_1| \leqslant |\overline{\mathcal{C}}_2| \leqslant \frac{1+\delta}{2} \cdot |\mathcal{C}|$. Finally, each $\overline{\mathcal{C}}_i$ is passed on to the next recursive step with $\ell = (\ell+1) \mod \lambda$.
(2) If $\ell = 0$, then $\mathcal{C}$ is partitioned into two components $\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2$ such that $|\mathsf{Nh}(\overline{\mathcal{C}}_i) \cap \mathsf{Nh}(\mathcal{C})| \leqslant \frac{2\cdot|\mathsf{Nh}(\mathcal{C})|}{3}$, by removing a single bag B and appropriately merging the resulting connected components created by such removal. This bag becomes a new bag $\mathcal{B}$ in $\mathsf{R_G}$, and each $\overline{\mathcal{C}}_i$ is passed on to the next recursive step with $\ell = (\ell+1) \mod \lambda$.

Figure 4 provides an illustration. The second construction is obtained simply by inserting in each bag $\mathcal{B}$ of $\mathsf{R_G}$ the nodes contained in the neighborhood $\mathsf{Nh}(\mathcal{C})$ of the component $\mathcal{C}$ from which $\mathcal{B}$ was constructed.

Use of (α,β,γ)-balanced tree-decompositions. For ease of presentation we consider that every $\mathrm{Tree}(G)$ is a full binary tree. Since our tree decompositions are (β,γ)-balanced, we can always

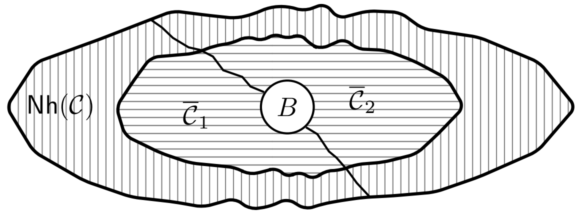

Fig. 4: Illustration of one recursive step of Rank on a component $\mathcal{C}$ (horizontal lines). $\mathcal{C}$ is partitioned into two sub-components $\overline{\mathcal{C}}_1$ and $\overline{\mathcal{C}}_2$ by removing a list of bags $\mathcal{X} = (B_i)_i$. Once every λ recursive calls, $\mathcal{X}$ contains one bag, such that the neighborhood $\mathsf{Nh}(\overline{\mathcal{C}}_i)$ of each $\overline{\mathcal{C}}_i$ is at most half the size of $\mathsf{Nh}(\mathcal{C})$ (i.e., the area with vertical lines is partitioned in half). In the remaining $\lambda - 1$ recursive calls, $\mathcal{X}$ contains m bags, such that the size of each $\overline{\mathcal{C}}_i$, is at most $\frac{1+\delta}{2}$ fraction the size of $\mathcal{C}$. (i.e., the area horizontal lines is partitioned in almost half).

attach empty children bags to those that have only one child, while increasing the size of $\mathrm{Tree}(G)$ by a constant factor only. In the sequel, $\mathrm{Tree}(G)$ will denote a full binary (α, β, γ)-balanced tree-decomposition of G. The parameters δ and λ will be chosen appropriately in later sections.

Remark 3.2. The notion of balanced tree decompositions exists in the literature [Elberfeld et al. 2010; Bodlaender and Hagerup 1998], but balancing only requires that the height of the tree is logarithmic in its size. Here we develop a stronger notion of balancing, which is crucial for proving the complexity results of the algorithms presented in this work.

3.1. Constructing Strongly Balanced Tree Decompositions

We now present in detail the construction of a strongly balanced tree decomposition. Given constants $0 < \delta \leqslant 1$ and $\lambda \geqslant 2$, throughout this section we fix

$$\alpha = 6 \cdot \lambda/\delta; \quad \beta = ((1+\delta)/2)^{\lambda-1}; \quad \gamma = \lambda$$

We show how given a graph G of treewidth t and a tree-decomposition T' of b bags and width t, we can construct in $O(b \cdot \log b)$ time and $O(b)$ space a (α, β, γ)-balanced tree-decomposition with b bags. That is, the resulting tree-decomposition has width at most $\alpha \cdot (t+1)$, and for every bag B and descendant B' of B that appears γ levels below, we have that $|T(B')| \leqslant \beta \cdot |T(B)|$ (i.e., the number of bags in $T(B')$ is at most β times as large as that in $T(B)$). Intuitively, the parameter δ specifies how well-balanced the new tree decomposition is, and the parameter λ specifies the frequency at which this well-balancing takes place along the levels. The result is established in two steps.

Tree components and operations Split**,** Merge **and** NhPartition**.** Given a tree-decomposition $T = (V_T, E_T)$, a *component* of T is a subset of bags of T. The *neighborhood* $\mathsf{Nh}(\mathcal{C})$ of $\mathcal{C}$ is the set of bags in $V_T \setminus \mathcal{C}$ that have a neighbor in $\mathcal{C}$, i.e.

$$\mathsf{Nh}(\mathcal{C}) = \{B \in V_T \setminus \mathcal{C} : (\{B\} \times \mathcal{C}) \cap E_T \neq \emptyset\}$$

em Operation Split. Given a component $\mathcal{C}$, we define the operation Split as $\mathsf{Split}(\mathcal{C}) = (\mathcal{X}, \mathcal{Y})$, where $\mathcal{X} \subseteq \mathcal{C}$ is a list of bags $(B_1, \dots B_{2/\delta})$ and $\mathcal{Y}$ is a list of sub-components $(\mathcal{C}_1, \dots \mathcal{C}_r)$ such that removing each bag B_i from $\mathcal{C}$ splits $\mathcal{C}$ into the subcomponents $\mathcal{Y}$, and for every i we have $|\mathcal{C}_i| \leqslant \frac{\delta}{2} \cdot |\mathcal{C}|$. Note that since $\mathcal{C}$ is a component of a tree, we can find a single separator bag that splits $\mathcal{C}$ into sub-components of size at most $\frac{|\mathcal{C}|}{2}$. Applying this step recursively for $\log(2/\delta)$ levels yields the desired separator set $\mathcal{X}$. For technical convenience, if this process yields less than $2/\delta$ bags, we repeat some of these bags until we have $2/\delta$ many.

Operation Merge. Consider a list of components $\mathcal{Y} = (\mathcal{C}_1, \dots \mathcal{C}_r)$, and let $z = \sum_i |\mathcal{C}_i|$. Let j be the largest integer such that $\sum_{i=1}^{j} |\mathcal{C}_i| \leqslant \frac{z}{2}$. We define the operation $\mathsf{Merge}(\mathcal{Y}) = (\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2)$, where $\overline{\mathcal{C}}_1 = \bigcup_{i=1}^{j} \mathcal{C}_i$ and $\overline{\mathcal{C}}_2 = \bigcup_{i=j+1}^{r} \mathcal{C}_i$. The following claim is trivially obtained.

CLAIM 1. *If $|\mathcal{C}_i| < \frac{\delta}{2} \cdot z$ for all i, then $|\overline{\mathcal{C}}_1| \leqslant |\overline{\mathcal{C}}_2| \leqslant \frac{1+\delta}{2} \cdot z$.*

PROOF. By construction, $\frac{1-\delta}{2} \cdot z < |\overline{\mathcal{C}}_1| \leqslant \frac{1}{2} \cdot z$, and since $\overline{\mathcal{C}}_1$ and $\overline{\mathcal{C}}_2$ partition $\mathcal{Y}$, we have $|\overline{\mathcal{C}}_1| + |\overline{\mathcal{C}}_2| = z$. The result follows. □

Intuitively, if a component $\mathcal{C}$ is split into sub-components $\mathcal{Y} = (\mathcal{C}_1, \dots \mathcal{C}_r)$ using the operation Split, then $z < |\mathcal{C}|$, as none of the sub-components $\mathcal{C}_i$ contains any of the bags of the main component $\mathcal{C}$ that where chosen as separator bags in Split.

Operation NhPartition. Finally, consider a component $\mathcal{C}$ such that $|\mathsf{Nh}(\mathcal{C})| \geqslant 2$. We define $\mathsf{NhPartition}(\mathcal{C}) = (B, \overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2)$, as follows. The bag $B \in \mathcal{C}$ is chosen such that its removal partitions $\mathcal{C}$ into connected components $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ such that $|\mathsf{Nh}(\mathcal{C}_i) \cap \mathsf{Nh}(\mathcal{C})| \leqslant \frac{|\mathsf{Nh}(\mathcal{C})|}{2}$ for each $i \in \{1, 2, 3\}$. We note that possibly $\mathcal{C}_i = \emptyset$ for some i. We construct two (not necessarily connected) components $\overline{\mathcal{C}_1}, \overline{\mathcal{C}_2}$ as follows. We let $\overline{\mathcal{C}_1} = \mathcal{C}_j$, where $j = \arg\max_i \mathsf{Nh}(\mathcal{C}_i)$ and $\overline{\mathcal{C}_2} = \bigcup_{i \neq j} \mathcal{C}_i$. The following claim follows easily.

CLAIM 2. *For each $i \in 1, 2$ we have that $|\mathsf{Nh}(\overline{\mathcal{C}}_i) \cap \mathsf{Nh}(\mathcal{C})| \leqslant \frac{2 \cdot |\mathsf{Nh}(\mathcal{C})|}{3}$.*

Construction of a (β, γ)-balanced rank tree. In the following, we consider that $T_G = (V_T, E_T)$ is a tree-decomposition of G and has $|V_T| = b$ bags. Given the parameters $\lambda \in \mathbb{N}$ with $\lambda \geqslant 2$ and $0 < \delta < 1$, we use the following algorithm Rank to construct a tree of bags $\mathsf{R_G}$. Rank operates recursively on inputs $(\mathcal{C}, \ell)$ where $\mathcal{C}$ is a component of T_G and $\ell \in \{0\} \cup [\lambda - 1]$, as follows.

1. If $|\mathcal{C}| \cdot \frac{\delta}{2} \leqslant 1$, construct a bag $\mathcal{B} = \bigcup_{B \in \mathcal{C}} B$, and return $\mathcal{B}$.
2. Else, if $\ell > 0$, let $(\mathcal{X}, \mathcal{Y}) = \mathsf{Split}(\mathcal{C})$. Construct a bag $\mathcal{B} = \bigcup_{B_i \in \mathcal{X}} B_i$, and let $(\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2) = \mathsf{Merge}(\mathcal{Y})$. Call Rank recursively on input $(\overline{\mathcal{C}}_1, (\ell + 1) \mod \lambda)$ and $(\overline{\mathcal{C}}_2, (\ell + 1) \mod \lambda)$, and let $\mathcal{B}_1$, $\mathcal{B}_2$ be the returned bags. Make $\mathcal{B}_1$ and $\mathcal{B}_2$ the left and right child of $\mathcal{B}$, and return the resulting tree.
3. Else, if $\ell = 0$, if $|\mathsf{Nh}(\mathcal{C})| > 1$, let $(B, \overline{\mathcal{C}_1}, \overline{\mathcal{C}_2}) = \mathsf{NhPartition}(\mathcal{C})$. Let $\mathcal{B} = B$. Call Rank recursively on input $(\overline{\mathcal{C}}_1, (\ell + 1) \mod \lambda)$ and $(\overline{\mathcal{C}}_2, (\ell + 1) \mod \lambda)$, and let $\mathcal{B}_1$, $\mathcal{B}_2$ be the returned bags. Make $\mathcal{B}_1$ and $\mathcal{B}_2$ the left and right child of $\mathcal{B}$, and return the resulting tree. Finally, if $|\mathsf{Nh}(\mathcal{C})| \leqslant 1$, call Rank recursively on input $(\mathcal{C}, (\ell - 1) \mod \lambda)$, and return the tree obtained by this recursive call.

In the following we use the symbols B and $\mathcal{B}$ to refer to bags of T_G and $\mathsf{R_G}$ respectively. Given a bag $\mathcal{B}$, we denote by $\mathcal{C}(\mathcal{B})$ the input component of Rank when $\mathcal{B}$ was constructed, and define the *neighborhood* of $\mathcal{B}$ as $\mathsf{Nh}(\mathcal{B}) = \mathsf{Nh}(\mathcal{C}(\mathcal{B}))$. Additionally, we denote by $\mathsf{Bh}(\mathcal{B})$ the set of separator bags $B_1, \dots B_r$ of $\mathcal{C}$ that were used to construct $\mathcal{B}$ (note that in case 1 of the algorithm, $\mathsf{Bh}(\mathcal{B})$ equals the bags of the input component $\mathcal{C}$). It is straightforward that $\mathsf{Bh}(\mathcal{B}_1) \cap \mathsf{Bh}(\mathcal{B}_2) = \emptyset$ for every distinct $\mathcal{B}_1$ and $\mathcal{B}_2$.

CLAIM 3. *Let $\mathcal{B}$ and $\mathcal{B}'$ be respectively a bag and its parent in $\mathsf{R_G}$. Then $\mathsf{Nh}(\mathcal{B}) \subseteq \mathsf{Nh}(\mathcal{B}') \cup \mathsf{Bh}(\mathcal{B}')$, and thus $|\mathsf{Nh}(\mathcal{B})| \leqslant |\mathsf{Nh}(\mathcal{B}')| + 2/\delta$.*

PROOF. Every bag in $\mathsf{Nh}(\mathcal{C}(\mathcal{B}))$ is either a bag in $\mathsf{Nh}(\mathcal{C}(\mathcal{B}'))$, or a separator bag of $\mathcal{C}(\mathcal{B}')$, and thus a bag of $\mathsf{Bh}(B')$. □

Note that every bag B of T_G belongs in $\mathsf{Bh}(\mathcal{B})$ of some bag $\mathcal{B}$ of $\mathsf{R_G}$, and thus the bags of $\mathsf{R_G}$ already cover all nodes and edges of G (i.e., properties C1 and C2 of a tree decomposition). In the following we show how $\mathsf{R_G}$ can be modified to also satisfy condition C3, i.e., that every node u appears in a

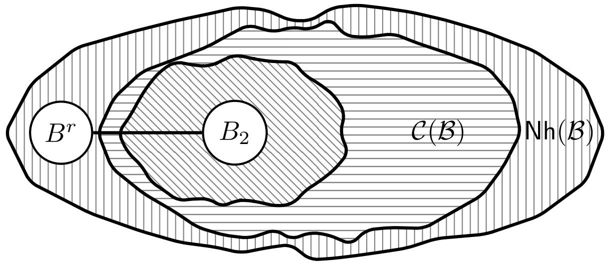

Fig. 5: Illustration of Lemma 3.3. Since B_2 belongs to $\mathcal{C}(\mathcal{B})$ and the sub-component with diagonal lines has not been split yet, the bag B^r is in the neighborhood of the sub-component with diagonal lines, and thus in the neighborhood of $\mathcal{C}(\mathcal{B})$.

contiguous subtree of $\mathsf{R_G}$. Given a bag $\mathcal{B}$, we denote by $\mathsf{NhV}(\mathcal{B}) = \mathcal{B} \cup \bigcup_{B\in\mathsf{Nh}(\mathcal{B})} B$, i.e., $\mathsf{NhV}(\mathcal{B})$ is the set of nodes of G that appear in $\mathcal{B}$ and its neighborhood. In the sequel, to distinguish between paths in different trees, given a tree of bags T (e.g. T_G or $\mathsf{R_G}$) and bags B_1, B_2 of T, we write $B_1 \rightsquigarrow_T B_2$ to denote the unique simple path from B_1 to B_2 in T.

We say that a pair of bags $(\mathcal{B}_1, \mathcal{B}_2)$ *form a gap* of some node u in a tree of bags T (e.g., $\mathsf{R_G}$) if $u \in \mathcal{B}_1 \cap \mathcal{B}_2$ and for the unique simple path $P : \mathcal{B}_1 \rightsquigarrow_T \mathcal{B}_2$ we have that $|P| \geqslant 2$ (i.e., there is at least one intermediate bag in P) and for all intermediate bags $\mathcal{B}$ in P we have $u \notin \mathcal{B}$. The following crucial lemma shows that if $\mathcal{B}_1$ and $\mathcal{B}_2$ form a gap of u in $\widehat{\mathsf{R_G}}$, then for every intermediate bag $\mathcal{B}$ in the path $P : \mathcal{B}_1 \rightsquigarrow_{\mathsf{R_G}} \mathcal{B}_2$, u must appear in some bag of $\mathsf{Nh}(\mathcal{B})$ (i.e., $u \in \mathsf{NhV}(\mathcal{B})$).

LEMMA 3.3. *For every node u, and pair of bags $(\mathcal{B}_1, \mathcal{B}_2)$ that form a gap of u in $\mathsf{R_G}$, such that $\mathcal{B}_1$ is an ancestor of $\mathcal{B}_2$, for every intermediate bag $\mathcal{B}$ in $P : \mathcal{B}_1 \rightsquigarrow_{\mathsf{R_G}} \mathcal{B}_2$ in $\mathsf{R_G}$, we have that $u \in \mathsf{NhV}(\mathcal{B})$.*

PROOF. Fix any such a bag $\mathcal{B}$, and since $\mathcal{B}_1$ and $\mathcal{B}_2$ form a gap of u, there exist bags $B_1 \in \mathsf{Bh}(\mathcal{B}_1)$ and $B_2 \in \mathsf{Bh}(\mathcal{B}_2)$ with $u \in B_1 \cap B_2$. Consider the time point j that bag $\mathcal{B}$ was constructed. Let B^r be the rightmost bag of the path $P_1 : B_1 \rightsquigarrow_{T_G} B_2$ that had been chosen as a separator in some previous step $j' < j$ of the algorithm. Note that B_1 has been chosen as such a separator, therefore B^r is well defined. We argue that $B^r \in \mathsf{Nh}(\mathcal{B})$, which implies that $u \in \mathsf{NhV}(\mathcal{B})$. This is done in two steps.

(1) Since $\mathcal{B}_2$ is a descendant of $\mathcal{B}$, we have that $B_2 \in \mathcal{C}(\mathcal{B})$, i.e., B_2 is a bag of the component when $\mathcal{B}$ was constructed.
(2) By the choice of B^r, for every intermediate bag B^i in the path $B^r \rightsquigarrow_{T_G} B_2$ we have that at the time $\mathcal{B}$ was constructed, each B^i belongs to the same component as B_2, and hence B^r is incident to that component.

These two points imply that $B^r \in \mathsf{Nh}(\mathcal{B})$. From the properties of tree decomposition we know that $u \in B^r$. It follows that $u \in \mathsf{NhV}(\mathcal{B})$, as desired. Figure 5 provides an illustration of the argument. □

Turning the rank tree to a tree decomposition. Lemma 3.3 suggests a way to turn the rank tree $\mathsf{R_G}$ to a tree-decomposition. Let $\widehat{\mathsf{R_G}} = \mathsf{Replace}(\mathsf{R_G})$ be the tree obtained by replacing each bag B of $\mathsf{R_G}$ with $\mathsf{NhV}(B)$. For a bag $\mathcal{B}$ in $\mathsf{R_G}$ let $\widehat{\mathcal{B}}$ be the corresponding bag in $\widehat{\mathsf{R_G}}$ and vice versa.

CLAIM 4. *If there is a pair of bags $\widehat{\mathcal{B}}_1$, $\widehat{\mathcal{B}}_2$ that form a gap of some node u in $\widehat{\mathsf{R_G}}$, then there is a pair of bags $\widehat{\mathcal{B}}'_1$, $\widehat{\mathcal{B}}'_2$ that also form a gap of u, and $\widehat{\mathcal{B}}'_1$ is ancestor of $\widehat{\mathcal{B}}'_2$.*

PROOF. First, note that neither parent of the bags $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ in $\widehat{\mathsf{R_G}}$ contains u. Assume that neither of $\widehat{\mathcal{B}}_1$, $\widehat{\mathcal{B}}_2$ is ancestor of the other.

(1) If for some $i \in \{1,2\}$ there is no bag $B_i \in \mathsf{Bh}(\mathcal{B}_i)$ such that $u \in B_i$, then there exists a bag $B \in \mathsf{Nh}(\mathcal{B}_i)$ with $u \in B$. Let $\mathcal{B}'_i$ be the parent of $\mathcal{B}_i$, and by Claim 3, we have that $B \in \mathsf{Nh}(\mathcal{B}'_i) \cup \mathsf{Bh}(\mathcal{B}'_i)$ and thus $u \in \mathsf{NhV}(\mathcal{B}'_i)$ and $u \in \widehat{\mathcal{B}}'_i$. This contradicts that $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ form a gap of u.
(2) Else, there exists a $B_1 \in \mathsf{Bh}(\mathcal{B}_1)$ and $B_2 \in \mathsf{Bh}(\mathcal{B}_2)$ such that $u \in B_1 \cap B_2$. Let B be first bag in the path $B_1 \rightsquigarrow_{T_G} B_2$ that was chosen as a separator. We have $B \in \mathsf{Bh}(\mathcal{B})$ for some ancestor $\mathcal{B}$ of $\mathcal{B}_1$ and $\mathcal{B}_2$, therefore $u \in \mathsf{NhV}(\mathcal{B})$, and thus $\widehat{\mathcal{B}}$ forms a gap of u with both $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ in $\widehat{\mathsf{R_G}}$.

It follows that in both cases there exists an ancestor $\widehat{\mathcal{B}}'_i$ of some $\widehat{\mathcal{B}}_i$ so that the two form a gap of u in $\widehat{\mathsf{R_G}}$. $\square$

The following lemma states that $\widehat{\mathsf{R_G}}$ is a tree decomposition of G.

LEMMA 3.4. $\widehat{\mathsf{R_G}} = \mathsf{Replace}(\mathsf{R_G})$ *is a tree-decomposition of* G.

PROOF. It is straightforward to see that the bags of $\widehat{\mathsf{R_G}}$ cover all nodes and edges of G (properties C1 and C2 of the definition of tree-decomposition), because for each bag $\mathcal{B}$, we have that $\mathcal{B} \subseteq \widehat{\mathcal{B}}$. It remains to show that every node u appears in a contiguous subtree of $\widehat{\mathsf{R_G}}$ (i.e., that property C3 is satisfied).

Assume towards contradiction otherwise, and by Claim 4 it follows that there exist bags $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ in $\widehat{\mathsf{R_G}}$ that form a gap of some node u such that $\widehat{\mathcal{B}}_1$ is an ancestor of $\widehat{\mathcal{B}}_2$. Let $\widehat{P} : \widehat{\mathcal{B}}_1 \rightsquigarrow_{\widehat{\mathsf{R_G}}} \widehat{\mathcal{B}}_2$ be the path between them, and $P : \mathcal{B}_1 \rightsquigarrow_{\mathsf{R_G}} \mathcal{B}_2$ the corresponding path in $\mathsf{R_G}$. By Lemma 3.3 we have $u \notin \mathcal{B}_1 \cap \mathcal{B}_2$, otherwise for every intermediate bag $\mathcal{B} \in \widehat{P}$ we would have $u \in \mathsf{NhV}(\mathcal{B})$ and thus $u \in \widehat{\mathcal{B}}$. Additionally, we have $u \in \mathcal{B}_2$, otherwise by Claim 3, we would have $u \in \mathsf{NhV}(\mathcal{B}'_2)$, where $\mathcal{B}'_2$ is the parent of $\mathcal{B}_2$, and thus $u \in \widehat{\mathcal{B}}'_2$, contradicting the assumption that $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ form a gap of u. Hence $u \notin \mathcal{B}_1$. A similar argument as that of Claim 4 shows that for the parent $\mathcal{B}'_1$ of $\mathcal{B}_1$, we have that $u \in \mathcal{B}'_1$, and wlog, take $\mathcal{B}'_1$ to be the lowest ancestor of $\mathcal{B}_1$ with this property. Then $\mathcal{B}'_1$ is also an ancestor of $\mathcal{B}_2$, and $\mathcal{B}'_1$ and $\mathcal{B}_2$ form a gap of u in $\mathsf{R_G}$. Then by Lemma 3.3, for every intermediate bag $\mathcal{B}$ in the path $\mathcal{B}'_1 \rightsquigarrow_{\mathsf{R_G}} \mathcal{B}_2$ we have that $u \in \mathsf{NhV}(\mathcal{B})$, thus $u \in \widehat{\mathcal{B}}$. Since the path $\widehat{\mathcal{B}}_1 \rightsquigarrow_{\widehat{\mathsf{R_G}}} \widehat{\mathcal{B}}_2$ is a suffix of $\widehat{\mathcal{B}}'_1 \rightsquigarrow_{\widehat{\mathsf{R_G}}} \widehat{\mathcal{B}}_2$, we have that $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ cannot form a gap of u. We have thus arrived at a contradiction, and the desired result follows. $\square$

Properties of the tree-decomposition $\widehat{\mathsf{R_G}}$. Lemma 3.4 states that $\widehat{\mathsf{R_G}}$ obtained by replacing each bag of $\mathsf{R_G}$ with $\mathsf{NhV}(B)$ is a tree-decomposition of G. The remaining of the section focuses on showing that $\widehat{\mathsf{R_G}}$ is a (α, β, γ)-balanced tree-decomposition of G, and that it can be constructed in $O(b \cdot \log b)$ time and $O(b)$ space. Recall the definition of the parameters

$$\alpha = 6 \cdot \lambda/\delta; \quad \beta = ((1+\delta)/2)^{\lambda-1}; \quad \gamma = \lambda$$

LEMMA 3.5. *The following assertions hold:*

(1) *Every bag $\widehat{\mathcal{B}}$ of $\widehat{\mathsf{R_G}}$ is (β, γ)-balanced.*
(2) *For every bag $\widehat{\mathcal{B}}$ of $\widehat{\mathsf{R_G}}$, we have $|\widehat{\mathcal{B}}| \leqslant \alpha \cdot (t+1)$.*

PROOF. We prove each item separately.

(1) For every bag $\mathcal{B}$ constructed by Rank, in at least $\gamma - 1$ out of every γ levels, Item 2 of the algorithm applies, and by Claim 1, the recursion proceeds on components $\overline{\mathcal{C}}_1$ and $\overline{\mathcal{C}}_2$ that are at most $\frac{1+\delta}{2}$ times as large as the input component $\mathcal{C}$ in that recursion step. Thus $\mathcal{B}$ is (β, γ)-balanced in $\mathsf{R_G}$, and hence $\widehat{\mathcal{B}}$ is (β, γ)-balanced in $\widehat{\mathsf{R_G}}$.

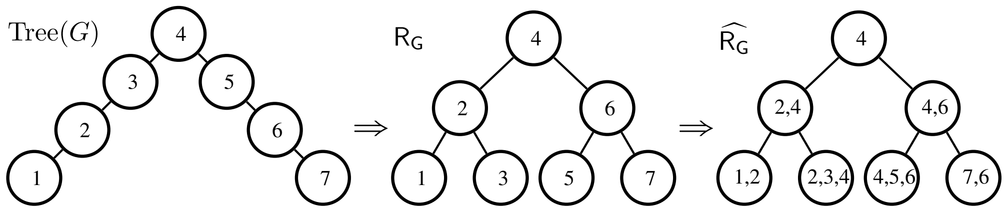

Fig. 6: Given the tree-decomposition $\mathrm{Tree}(G)$ on the left, the graph in the middle is the corresponding $\mathsf{R_G}$ and the one on the right is the corresponding tree-decomposition $\widehat{\mathsf{R_G}} = \mathsf{Replace}(\mathsf{R_G})$ after replacing each bag B with $\mathsf{NhV}(B)$.

(2) It suffices to show that for every bag $\mathcal{B}$, we have $|\mathsf{Nh}(\mathcal{B})| \leqslant \alpha - 1 = 3 \cdot (2/\delta) \cdot \lambda - 1$. Assume towards contradiction otherwise. Let $\mathcal{B}$ be the first bag that Rank constructed such that $|\mathsf{Nh}(\mathcal{B})| \geqslant 3 \cdot (2/\delta) \cdot \lambda$. Let $\mathcal{B}'$ be the lowest ancestor of $\mathcal{B}$ in $\mathsf{R_G}$ that was constructed by Rank on some input $(\mathcal{C}, \ell)$ with $\ell = 1$, and let $\mathcal{B}''$ be the parent of $\mathcal{B}'$ in $\mathsf{R_G}$ (note that $\mathcal{B}'$ can be $\mathcal{B}$ itself). By Item 3 of Rank and Claim 2, it follows that $|\mathsf{Nh}(\mathcal{B}')| \leqslant \lfloor \frac{2 \cdot |\mathsf{Nh}(\mathcal{B}'')|}{3} \rfloor + 1$. Note that $\mathcal{B}'$ is at most $\lambda - 1$ levels above $\mathcal{B}$ (as we allow $\mathcal{B}'$ to be $\mathcal{B}$). By Claim 3, the neighborhood of a bag can increase by at most $(2/\delta)$ from the neighborhood of its parents, hence $\mathsf{Nh}(\mathcal{B}') \geqslant 2 \cdot (2/\delta) \cdot (\lambda + 1)$. The last two inequalities lead to $|\mathsf{Nh}(\mathcal{B}'')| \geqslant 3 \cdot (2/\delta) \cdot \lambda$, which contradicts our choice of B.

The desired result follows. $\square$

A minimal example. Figure 6 illustrates an example of $\widehat{\mathsf{R_G}}$ constructed out of a tree-decomposition T' of G. First, T' is turned into a binary and balanced tree $\mathsf{R_G}$ and then into a binary and balanced tree $\widehat{\mathsf{R_G}}$. If the numbers are pointers to bags, such that T' is a tree-decomposition for G, then $\widehat{\mathsf{R_G}}$ is a binary and balanced tree-decomposition of G. The values of λ and δ are immaterial for this example, as $\widehat{\mathsf{R_G}}$ becomes perfectly balanced (i.e., $(1/2, 1)$-balanced).

THEOREM 3.1. *For every graph G with n nodes and constant treewidth, for any fixed $\delta > 0$ and $\lambda \in \mathbb{N}$ with $\lambda \geqslant 2$, let $\alpha = 6 \cdot \lambda/\delta$, $\beta = ((1+\delta)/2)^{\lambda-1}$, and $\gamma = \lambda$. A binary (α, β, γ) tree-decomposition $\mathrm{Tree}(G)$ with $O(n)$ bags can be constructed in $O(n \cdot \log n)$ time and $O(n)$ space.*

PROOF. By [Bodlaender 1996] an initial tree-decomposition T' of G with width t and $b = O(n)$ bags can be constructed in $O(n)$ time. Lemma 3.4 and Lemma 3.5 prove that the constructed $\widehat{\mathsf{R_G}}$ is a (α, β, γ)-balanced tree-decomposition of G. The time and space complexity come from the construction of $\mathsf{R_G}$ by the recursion of Rank. It can be easily seen that every level of the recursion processes disjoint components $\mathcal{C}_i$ of T' in $O(|\mathcal{C}_i|)$ time (recall that operations Split and Merge require linear time), thus one level of the recursion requires $O(b)$ time in total. There are $O(\log b)$ such levels, since every λ levels, the size of each component has been reduced to at most a factor $((1+\delta)/2)^{\lambda-1}$. Hence the time complexity is $O(b \cdot \log b) = O(n \cdot \log n)$. The space complexity is that of processing a single level of the recursion, hence $O(b) = O(n)$. $\square$

4. CONCURRENT TREE DECOMPOSITION

In this section we present the construction of a tree-decomposition $\mathrm{Tree}(G)$ of a concurrent graph $G = (V, E)$ of k constant-treewidth graphs. In general, G can have treewidth which depends on the number of its nodes (e.g., G can be a grid, which has treewidth n, obtained as the product of two lines, which have treewidth 1). While the treewidth computation for constant-treewidth graphs is linear time [Bodlaender 1996], it is NP-complete for general graphs [Bodlaender 1993]. Hence computing a tree decomposition that achieves the treewidth of G can be computationally expensive

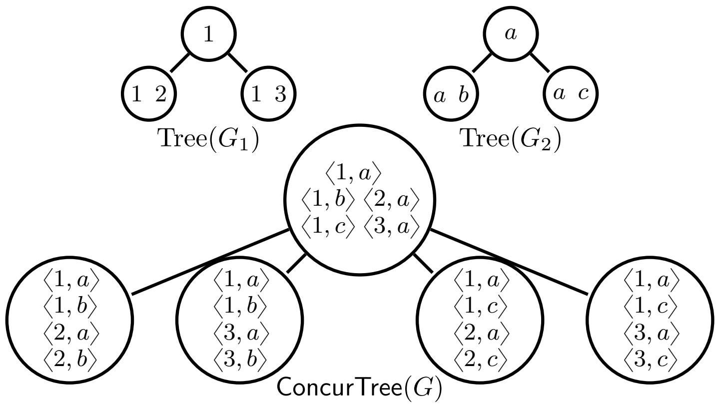

Fig. 7: The tree-decomposition $\mathsf{ConcurTree}(G)$ of a concurrent graph G of two constant-treewidth graphs G_1 and G_2.

(e.g., exponential in the size of G). Here we develop an algorithm $\mathsf{ConcurTree}$ which constructs a tree-decomposition $\mathsf{ConcurTree}(G)$ of G, given an (α, β, γ) tree-decomposition of the components, in $O(n^k)$ time and space (i.e., linear in the size of G), such that the following properties hold: (i) the width is $O(n^{k-1})$; and (ii) for every bag in level at least $i \cdot \gamma$, the size of the bag is $O(n^{k-1} \cdot \beta^i)$ (i.e., the size of the bags decreases geometrically along the levels).

Algorithm $\mathsf{ConcurTree}$ **for concurrent tree decomposition.** Let G be a concurrent graph of k graphs $(G_i)_{1 \leqslant i \leqslant k}$. The input consists of a full binary tree-decomposition T_i of constant width for every graph G_i. In the following, B_i ranges over bags of T_i, and we denote by $B_{i,r}$, with $r \in [2]$, the r-th child of B_i. We construct the *concurrent tree-decomposition* $T = \mathsf{ConcurTree}(G) = (V_T, E_T)$ of G using the recursive procedure $\mathsf{ConcurTree}$, which operates as follows. On input $(T_i(B_i))_{1 \leqslant i \leqslant k}$, return a tree decomposition where

(1) The root bag B is

$$B = \bigcup_{1 \leqslant i \leqslant k} \left(\left(\prod_{j<i} V_{T_j}(B_j) \right) \times B_i \times \left(\prod_{j>i} V_{T_j}(B_j) \right) \right) \tag{1}$$

(2) If every B_i is a non-leaf bag of T_i, for every choice of $\langle r_1, \ldots, r_k \rangle \in [2]^k$, repeat the procedure for $(T_i(B_{i,r_i}))_{1 \leqslant i \leqslant k}$, and let B' be the root of the returned tree. Make B' a child of B.
(3) If some B_i is a leaf bag of T_i, then the algorithm terminates.

Let B_i be the root of the tree-decomposition T_i. We denote by $\mathsf{ConcurTree}(G)$ the application of the recursive procedure $\mathsf{ConcurTree}$ on $(T_i(B_i))_{1 \leqslant i \leqslant k}$. Figure 7 provides an illustration.

Remark 4.1. Recall that for any bag B_j of a tree-decomposition T_j, we have $V_{T_j}(B_j) = \bigcup_{B'_j} B'_j$, where B'_j ranges over bags in $T_j(B_j)$. Then, for any two bags B_{i_1}, B_{i_2}, of tree-decompositions T_{i_1} and T_{i_2} respectively, we have

$$V_{T_{i_1}}(B_{i_1}) \times V_{T_{i_2}}(B_{i_2}) = \bigcup_{B'_{i_1}, B'_{i_2}} \left(B'_{i_1} \times B'_{i_2} \right)$$

where B'_{i_1} and B'_{i_2} range over bags in $T_{i_1}(B_{i_1})$ and $T_{i_2}(B_{i_2})$ respectively. Since each tree-decomposition T_i has constant width, it follows that $|V_{T_{i_1}}(B_{i_1}) \times V_{T_{i_2}}(B_{i_2})| = O(|T_{i_1}(B_{i_1})| \cdot |T_{i_2}(B_{i_2})|)$. Thus, the size of each bag B of $\mathsf{ConcurTree}(G)$ constructed in Eq. 1 on some input $(T_i(B_i))_i$ is $|B| = O(\sum_i \prod_{j \neq i} n_j)$, where $n_i = |T_i(B_i)|$.

In view of Remark 4.1, the time and space required by ConcurTree to operate on input $(T_i(B_i))_{1\leqslant i\leqslant k}$ where $|T_i(B_i)| = n_i$, is given, up to constant factors, by

$$\mathcal{T}(n_1,\dots,n_k) \leqslant \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j + \sum_{(r_i)_i\in[2]^k}\mathcal{T}(n_{1,r_1},\dots,n_{k,r_k}) \tag{2}$$

such that for every i we have that $\sum_{r_i\in[2]} n_{i,r_i} \leqslant n_i$.

The following lemma establishes the correctness of the construction.

LEMMA 4.2. $\mathsf{ConcurTree}(G)$ *is a tree decomposition of* G.

PROOF. We show that T satisfies the three conditions of a tree decomposition.

C1 For each node $u = \langle u_i\rangle_{1\leqslant i\leqslant k}$, let $j = \arg\min_i \mathsf{Lv}(u_i)$. Then $u \in B$, where B is the bag constructed by step 1 of ConcurTree when it operates on input $(T_i(B_i))_{1\leqslant i\leqslant k}$, where each T_i is a tree decomposition, and additionally $B_j = B_{u_j}$ (i.e., B_j is the root bag of u_j in T_j).

C2 Similarly, for each edge $(u,v) \in E$ with $u = \langle u_i\rangle_{1\leqslant i\leqslant k}$ and $v = \langle v_i\rangle_{1\leqslant i\leqslant k}$, let $j = \arg\min_i(\max(\mathsf{Lv}(u_i),\mathsf{Lv}(v_i)))$, where $\arg\min_i f(i)$ returns the value of i that minimizes f. Then $(u,v)\in B$, where B is a bag similar to C1.

C3 For any node $u = \langle u_i\rangle_{1\leqslant i\leqslant k}$ and path $P : B \rightsquigarrow B'$ with $u \in B \cap B'$, let B'' be any bag of P. Since at least one of B, B' is a descendant of B'', we have $V_T(B) \subseteq V_T(B'')$ or $V_T(B') \subseteq V_T(B'')$, and because $u \in B\cap B'$, if B'' was constructed on input $(T_i(B''_i))_{1\leqslant i\leqslant k}$, where each T_i is a tree decomposition, we have $u_i \in V_{T_i}(B''_i)$. Let $(T_i(B_i))_{1\leqslant i\leqslant k}$ and $(T_i(B'_i))_{1\leqslant i\leqslant k}$ be the inputs to the algorithm when B and B' were constructed, and it follows that for some $1\leqslant j\leqslant k$ we have $u_j \in B_j \cap B'_j$. Then B''_j is an intermediate bag in the path $P_j : B_j \rightsquigarrow B'_j$ in T_j, thus $u_j \in B''_j$ and hence $u \in B''$.

The desired result follows. □

We now turn our attention to the complexity. We start with analyzing the following recurrence, which will be useful in the complexity analysis afterwards.

LEMMA 4.3. *Consider the following recurrence*

$$\mathcal{T}(n_1,\dots,n_k) \leqslant \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j + \sum_{(r_i)_i\in[2]^k}\mathcal{T}(n_{1,r_1},\dots,n_{k,r_k}) \tag{3}$$

such that for every i we have that $n_{i,1}, n_{i,2} \geqslant 1$ and $\sum_{r_i\in[2]} n_{i,r_i} \leqslant n_i$ and as the base case we have that if $n_i = 1$ for some i, then

$$\mathcal{T}(n_1,\dots,n_k) \leqslant \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j \tag{4}$$

Then Eq. 3 has the solution

$$\mathcal{T}(n_1,\dots,n_k) \leqslant 2\cdot k\cdot \prod_{1\leqslant i\leqslant k} n_i - \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j. \tag{5}$$

PROOF. Observe that the right hand side of Eq. 5 is always larger than the right hand side of Eq.4. Hence, in order to verify that Eq. 3 has Eq. 5 as a solution, it suffices to substitute Eq. 5 in Eq. 3 (i.e., we take Eq. 5 also as the base case solution). Indeed, substituting Eq. 5 to the recurrence

Eq. 3 we have

$$\begin{aligned}\mathcal{T}(n_1,\dots,n_k) &\leqslant \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j + \sum_{(r_i)_i\in[2]^k}\left(2\cdot k\cdot \prod_{1\leqslant i\leqslant k} n_{i,r_i} - \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_{j,r_j}\right)\\ &= \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j + 2\cdot k\cdot X - Y \qquad (6)\end{aligned}$$

where

$$X = \sum_{(r_i)_i\in[2]^k}\left(\prod_{1\leqslant i\leqslant k} n_{i,r_i}\right) \qquad \text{and} \qquad Y = \sum_{(r_i)_i\in[2]^k}\left(\sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_{j,r_j}\right)$$

We compute X and Y respectively.

$$X = \sum_{(r_i)_i\in[2]^k}\left(\prod_{1\leqslant i\leqslant k} n_{i,r_i}\right) = \sum_{r_1\in[2]} n_{1,r_1}\cdot\left(\sum_{r_2\in[2]} n_{2,r_2}\cdot\left(\dots\sum_{r_k\in[2]} n_{k,r_k}\right)\right) \leqslant \prod_{1\leqslant i\leqslant k} n_i \qquad (7)$$

by factoring out every n_{i,r_i} of the sum. Similarly,

$$\begin{aligned}Y &= \sum_{(r_i)_i\in[2]^k}\left(\sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_{j,r_j}\right) = \sum_{1\leqslant i\leqslant k}\left(\sum_{(r_i)_i\in[2]^k}\prod_{j\neq i} n_{j,r_j}\right)\\ &=2\cdot\sum_{1\leqslant i\leqslant k}\left(\sum_{r_1\in[2]} n_{1,r_1}\cdot\ldots(\sum_{r_{i-1}\in[2]} n_{i-1,r_{i-1}}\cdot(\sum_{r_{i+1}\in[2]} n_{i+1,r_{i+1}}\cdot\ldots(\sum_{r_k\in[2]} n_{k,r_k})))\right)\\ &\geqslant 2\cdot\sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j \qquad (8)\end{aligned}$$

The second equality is obtained by swapping the inner with the outer sum. The third equality follows by expanding the sum over $(r_i)_i \in [2]^k$. The final inequality is obtained since for all $1 \leqslant i \leqslant k$ we have $n_{i,1} + n_{i,2} \leqslant n_i$. Substituting inequalities Eq. 7 and 8 to Eq. 6 we obtain

$$\mathcal{T}(n_1,\dots,n_k) \leqslant \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j + 2\cdot k\cdot X - Y \leqslant 2\cdot k\cdot \prod_{1\leqslant i\leqslant k} n_i - \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j$$

as desired. □

LEMMA 4.4. ConcurTree *requires $O(n^k)$ time and space.*

PROOF. It is easy to verify that ConcurTree(G) performs a constant number of operations per node per bag in the returned tree decomposition. Hence we will bound the time taken by bounding the size of ConcurTree(G). Consider a recursion step of ConcurTree on input $(T_i(B_i))_{1\leqslant i\leqslant k}$. Let $n_i = |T_i(B_i)|$ for all $1 \leqslant i \leqslant k$, and $n_{i,r_i} = |T_i(B_{i,r_i})|$, $r_i \in [2]$, where B_{i,r_i} is the r_i-th child of B_i. Without loss of generality, we assume that $n_i \geqslant 2$ for all i. In view of Remark 4.1, the time required by ConcurTree on this input is given by the recurrence in Eq. 3, up to a constant factor. The desired result follows from Lemma 4.3. □

We summarize the results of this section with the following theorem.

THEOREM 4.5. *Let $G = (V, E)$ be a concurrent graph of k constant-treewidth graphs $(G_i)_{1\leqslant i\leqslant k}$ of n nodes each. Let a binary (α, β, γ) tree-decomposition T_i for every graph G_i be given, for some constant α.* ConcurTree *constructs a 2^k-ary tree-decomposition* ConcurTree(G)

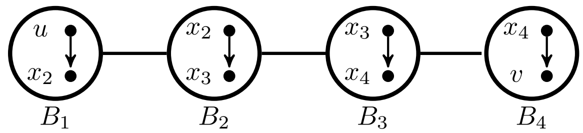

Fig. 8: Illustration of Lemma 5.1. If P is the unique simple path $B_1 \rightsquigarrow B_4$ in $\mathrm{Tree}(G)$, then there exist (not necessarily distinct) $x_i \in B_{i-1} \cap B_i$ with $1 < i \leqslant 4$ such that $d(u,v) = d(u,x_2) \otimes d(x_2,x_3) \otimes d(x_3,x_4) \otimes d(x_4,v)$.

of G in $O(n^k)$ time and space, with the following property. For every $i \in \mathbb{N}$ and bag B at level $\mathsf{Lv}(B) \geqslant i \cdot \gamma$, we have $|B| = O(n^{k-1} \cdot \beta^i)$.

PROOF. Lemma 4.2 proves the correctness and Lemma 4.4 the complexity. Here we focus on bounding the size of a bag B with $\mathsf{Lv}(B) \geqslant i \cdot \gamma$. Let $(T_i(B_i))_{1 \leqslant i \leqslant k}$ be the input on ConcurTree when it constructed B using Eq. 1 and $n_i = |T_i(B_i)|$. Observe that $\mathsf{Lv}(B) = \mathsf{Lv}(B_i)$ for all i, and since each T_i is (β, γ)-balanced, we have that $n_i \leqslant O(n \cdot \beta^i)$. Since each T_i is α-approximate, $|B_i| = O(1)$ for all i. It follows from Eq. 1 and Remark 4.1 that $|B| = O(n^{k-1} \cdot \beta^i)$. □

5. CONCURRENT ALGEBRAIC PATHS

We now turn our attention to the core algorithmic problem of this paper, namely answering semiring distance queries in a concurrent graph G of k constant-treewidth graphs $(G_i)_{1 \leqslant i \leqslant k}$. To this direction, we develop a data-structure ConcurAP (for *concurrent algebraic paths*) which will preprocess G and afterwards support single-source, pair, and partial pair queries on G.

Semiring distances on tree decompositions. The preprocessing and query of our data-structure exploits a key property of semiring distances on tree decompositions. This property is formally stated in Lemma 5.1, and concerns any two nodes u, v that appear in some distinct bags B_1, B_j of $\mathrm{Tree}(G)$. Informally, the semiring distance $d(u,v)$ can be written as the semiring multiplication of distances $d(x_i, x_{i+1})$, where x_i is a node that appears in the i-th and $(i-1)$-th bags of the unique simple path $B_1 \rightsquigarrow B_j$ in $\mathrm{Tree}(G)$ (recall that since $\mathrm{Tree}(G)$ is a tree decomposition, there is a unique simple path between every pair of its bags). Figure 8 provides an illustration.

LEMMA 5.1. *Consider a graph $G = (V, E)$ with a weight function $\mathsf{wt} : E \to \Sigma$, and a tree-decomposition $\mathrm{Tree}(G)$. Let $u, v \in V$, and $P : B_1, B_2, \dots, B_j$ be any simple path in T such that $u \in B_1$ and $v \in B_j$. Let $A = \{u\} \times \left(\prod_{1 < i \leqslant j} (B_{i-1} \cap B_i)\right) \times \{v\}$. Then $d(u,v) = \bigoplus_{(x_1, \dots, x_{j+1}) \in A} \bigotimes_{i=1}^{j} d(x_i, x_{i+1})$.*

PROOF. By [Chatterjee et al. 2015b, Lemma 1], for every bag B_i with $i > 1$ and path $P : u \rightsquigarrow v$, there exists a node $x_i \in B_{i-1} \cap B_i \cap P$. Denote by $P_{x,y}$ a path $x \rightsquigarrow y$ in G. Then

$$d(u,v) = \bigoplus_{P_{u,v}} \otimes(P_{u,v}) = \bigoplus_{x_i \in B_{i-1} \cap B_i \cap P} \left(\bigoplus_{P_{u,x_i}} \otimes(P_{u,x_i}) \otimes \bigoplus_{P_{x_i,v}} \otimes(P_{x_i,v}) \right)$$
$$= \bigoplus_{x_i \in B_{i-1} \cap B_i} \left(d(u,x_i) \otimes d(x_i,v) \right)$$

and the proof follows an easy induction on i. □

Informal description of the preprocessing. The preprocessing phase of ConcurAP is handled by algorithm ConcurPreprocess, which performs the following steps.

(1) First, the *partial expansion* $\overline{G}$ of G is constructed by introducing a pair of strictly partial nodes $\overline{u}^1$, $\overline{u}^2$ for every strictly partial node $\overline{u}$ of G, and edges between strictly partial nodes and the corresponding nodes of G that refine them.
(2) Second, the concurrent tree-decomposition $T = \mathsf{ConcurTree}(G)$ of G is constructed, and modified to a tree-decomposition $\overline{T}$ of the partial expansion graph $\overline{G}$.
(3) Third, a standard, two-way pass of $\overline{T}$ is performed to compute *local distances*. In this step, for every bag $\overline{B}$ in $\overline{T}$ and all partial nodes $\overline{u}, \overline{v} \in \overline{B}$, the distance $d(\overline{u}, \overline{v})$ is computed (i.e., all-pair distances in $\overline{B}$). Since we compute distances between nodes that are *local* in a bag, this step is called local distance computation. This information is used to handle (i) single-source queries and (ii) partial pair queries in which both nodes are strictly partial.
(4) Finally, a top-down pass of $\overline{T}$ is performed in which for every node u and partial node $\overline{v} \in \mathcal{V}_{\overline{T}}(\overline{B}_u)$ (i.e., $\overline{v}$ appears in some ancestor of $\overline{B}_u$) the distances $d(u, \overline{v})$ and $d(\overline{v}, u)$ are computed. This information is used to handle pair queries in which at least one node is a node of G (i.e., not strictly partial).

Bottom-up and top-down traversals. In the description of the preprocessing algorithm $\mathsf{ConcurPreprocess}$, we make use of two types of traversals of the tree decomposition. A *bottom-up traversal* is any traversal of the tree in which a bag B is visited after all children of B have been visited. A *top-down traversal* is any traversal of the tree in which a bag B is visited after the parent of B has been visited.

Algorithm $\mathsf{ConcurPreprocess}$. We now formally describe algorithm $\mathsf{ConcurPreprocess}$ for preprocessing the concurrent graph $G = (V, E)$ for the purpose of answering algebraic path queries. For any desired $0 < \epsilon \leqslant 1$, we choose appropriate constants α, β, γ, which will be defined later for the complexity analysis. On input $G = (V, E)$, where G is a concurrent graph of k constant-treewidth graphs $(G_i = (V_i, E_i))_{1 \leqslant i \leqslant k}$, and a weight function $\mathsf{wt} : E \to \Sigma$, $\mathsf{ConcurPreprocess}$ operates as follows:

(1) Construct the *partial expansion* $\overline{G} = (\overline{V}, \overline{E})$ of G together with an extended weight function $\overline{\mathsf{wt}} : \overline{E} \to \Sigma$ as follows.
  (a) The node set is $\overline{V} = V \cup \{\overline{u}^1, \overline{u}^2 : \exists u \in V \text{ s.t. } u \sqsubset \overline{u}\}$; i.e., $\overline{V}$ consists of nodes in V and two copies for every partial node $\overline{u}$ that is strictly refined by a node u of G.
  (b) The edge set is $\overline{E} = E \cup \{(\overline{u}^1, u), (u, \overline{u}^2) : \overline{u}^1, \overline{u}^2 \in \overline{V} \text{ and } u \in V \text{ s.t. } u \sqsubset \overline{u}^1, \overline{u}^2\}$, i.e., along with the original edges E, the first (resp., second) copy of every strictly partial node has outgoing (resp., incoming) edges to (resp., from) the nodes of G that refine it.
  (c) For the weight function we have $\overline{\mathsf{wt}}(\overline{u}, \overline{v}) = \mathsf{wt}(\overline{u}, \overline{v})$ if $\overline{u}, \overline{v} \in V$, and $\overline{\mathsf{wt}}(\overline{u}, \overline{v}) = \overline{\mathbf{1}}$ otherwise. That is, the original weight function is extended with value $\overline{\mathbf{1}}$ (which is neutral for semiring multiplication) to all new edges in $\overline{G}$.
(2) Construct the tree-decomposition $\overline{T} = (\overline{V}_T, \overline{E}_T)$ of $\overline{G}$ as follows.
  (a) Obtain an (α, β, γ)-balanced tree-decomposition $T_i = \mathrm{Tree}(G_i)$ of every graph G_i using Theorem 3.1.
  (b) Construct the concurrent tree-decomposition $T = \mathsf{ConcurTree}(G)$ of G using $(T_i)_{1 \leqslant i \leqslant k}$.
  (c) Let $\overline{T}$ be identical to T, with the following exception: For every bag B of T and $\overline{B}$ the corresponding bag in $\overline{T}$, for every node $u \in B$, insert in $\overline{B}$ all strictly partial nodes $\overline{u}^1$, $\overline{u}^2$ of $\overline{V}$ that u refines. Formally, set $\overline{B} = B \cup \{\overline{u}^1, \overline{u}^2 : \exists u \in B \text{ s.t. } u \sqsubset \overline{u}\}$. Note that also $u \in \overline{B}$.

  Observe that the root bag of $\overline{T}$ contains all strictly partial nodes.
(3) Perform the *local distance computation* on $\overline{T}$ as follows. For every partial node $\overline{u}$, maintain two map data-structures $\textsc{Fwd}_{\overline{u}}, \textsc{Bwd}_{\overline{u}} : \overline{B}_{\overline{u}} \to \Sigma$. Intuitively, $\textsc{Fwd}_{\overline{u}}$ (resp., $\textsc{Bwd}_{\overline{u}}$) aims to store the forward (resp., backward) distance, i.e., distance from (resp., to) $\overline{u}$ to (resp., from) vertices in $\overline{B}_{\overline{u}}$. Initially set $\textsc{Fwd}_{\overline{u}}(\overline{v}) = \overline{\mathsf{wt}}(\overline{u}, \overline{v})$ and $\textsc{Bwd}_{\overline{u}}(\overline{v}) = \overline{\mathsf{wt}}(\overline{v}, \overline{u})$ for all partial nodes $\overline{v} \in \overline{B}_{\overline{u}}$ (and $\textsc{Fwd}_{\overline{u}}(\overline{v}) = \textsc{Bwd}_{\overline{u}}(\overline{v}) = \overline{\mathbf{0}}$ if $(\overline{u}, \overline{v}) \notin \overline{E}$). At any point in the computation, given a bag $\overline{B}$

we denote by $wt_{\overline{B}} : \overline{B} \times \overline{B} \to \Sigma$ a map data-structure such that for every pair of partial nodes $\overline{u}, \overline{v}$ with $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$ we have $wt_{\overline{B}}(\overline{u}, \overline{v}) = \textsc{Fwd}_{\overline{u}}(\overline{v})$ and $wt_{\overline{B}}(\overline{v}, \overline{u}) = \textsc{Bwd}_{\overline{u}}(\overline{v})$.

(a) Traverse $\overline{T}$ bottom-up, and for every bag $\overline{B}$, execute an all-pairs algebraic path computation on $\overline{G}[\overline{B}]$ with weight function $wt_{\overline{B}}$. This is done using classical algorithms for the transitive closure, e.g. [Lehmann 1977; Floyd 1962; Warshall 1962; Kleene 1956]. For every pair of partial nodes $\overline{u}, \overline{v}$ with $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$, set $\textsc{Bwd}_{\overline{u}}(\overline{v}) = d'(\overline{v}, \overline{u})$ and $\textsc{Fwd}_{\overline{u}}(\overline{v}) = d'(\overline{u}, \overline{v})$, where $d'(\overline{u}, \overline{v})$ and $d'(\overline{v}, \overline{u})$ are the computed distances in $\overline{G}[\overline{B}]$ (recall that $\overline{G}[\overline{B}]$ denotes the restriction of the graph $\overline{G}$ on the node set $\overline{B}$).

(b) Traverse $\overline{T}$ top-down, and for every bag $\overline{B}$ perform the computation of Item 3a.

(4) Perform the *ancestor distance computation* on $\overline{T}$ as follows. For every node u, maintain two map data-structures $\textsc{Fwd}^+_u, \textsc{Bwd}^+_u : \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u) \to \Sigma$ from partial nodes that appear in the ancestor bags of $\overline{B}_u$ to Σ. These maps aim to capture distances between the node u and nodes in the ancestor bags of $\overline{B}_u$ (in contrast to $\textsc{Fwd}_u$ and $\textsc{Bwd}_u$ which store distances only between u and nodes in $\overline{B}_u$). Initially, set $\textsc{Fwd}^+_u(\overline{v}) = \textsc{Fwd}_u(\overline{v})$ and $\textsc{Bwd}^+_u(\overline{v}) = \textsc{Bwd}_u(\overline{v})$ for every partial node $\overline{v} \in \overline{B}_u$. Given a pair of partial nodes $\overline{u}, \overline{v}$ with $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$ we denote by $wt^+(\overline{u}, \overline{v}) = \textsc{Fwd}^+_{\overline{u}}(\overline{v})$ and $wt^+(\overline{v}, \overline{u}) = \textsc{Bwd}^+_{\overline{u}}(\overline{v})$. Traverse $\overline{T}$ via a DFS starting from the root, and for every encountered bag $\overline{B}$ with parent $\overline{B}'$, for every node u such that $\overline{B}$ is the root bag of u, for every partial node $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u)$, assign

$$\textsc{Fwd}^+_u(\overline{v}) = \bigoplus_{x \in \overline{B} \cap \overline{B}'} \textsc{Fwd}_u(x) \otimes wt^+(x, \overline{v}) \tag{9}$$

$$\textsc{Bwd}^+_u(\overline{v}) = \bigoplus_{x \in \overline{B} \cap \overline{B}'} \textsc{Bwd}_u(x) \otimes wt^+(\overline{v}, x) \tag{10}$$

If $\overline{B}$ is the root of $\overline{T}$, simply initialize the maps $\textsc{Fwd}^+_u$ and $\textsc{Bwd}^+_u$ according to the corresponding maps $\textsc{Fwd}_u$ and $\textsc{Bwd}_u$ constructed from Item 3.

(5) Preprocess $\overline{T}$ to answer LCA queries in $O(1)$ time [Harel and Tarjan 1984].

The following claim states that the first (resp., second) copy of each strictly partial node inserted in Item 1 captures the distance from (resp., to) the corresponding strictly partial node of $\overline{G}$.

CLAIM 5. *For every partial node $\overline{u}$ and strictly partial node $\overline{v}$ we have $d(\overline{u}, \overline{v}) = d(\overline{u}, \overline{v}^2)$ and $d(\overline{v}, \overline{u}) = d(\overline{v}^1, \overline{u})$.*

PROOF. By construction, for every node $v \in V$ that strictly refines $\overline{v}$ (i.e., $v \sqsubset \overline{v}$), we have $\overline{\mathsf{wt}}(\overline{v}^1, v) = d(\overline{v}^1, v) = \overline{\mathbf{1}}$ and $\overline{\mathsf{wt}}(v, \overline{v}^2) = d(v, \overline{v}^2) = \overline{\mathbf{1}}$, i.e., every such v can reach (resp., be reached from) $\overline{v}^2$ (resp., $\overline{v}^1$) without changing the distance from $\overline{u}$. The claim follows easily. □

Key novelty and insights. The key novelty and insights of our algorithm are as follows:

(1) A partial pair query can be answered by breaking it down to several pair queries. Instead, preprocessing the partial expansion of the concurrent graph allows to answer partial pair queries directly. Moreover, the partial expansion does not increase the asymptotic complexity of the preprocessing time and space.
(2) ConcurPreprocess computes the transitive closure only during the local distance computation in each bag (Item 3 above), instead of a global computation on the whole graph. The key reason of our algorithmic improvement lies on the fact that the local computation is cheaper than the global computation, and is also sufficient to handle queries fast.
(3) The third key aspect of our algorithm is the strongly balanced tree decomposition, which is crucially used in Theorem 4.5 to construct a tree decomposition for the concurrent graph such that the size of the bags decreases geometrically along the levels. By using the cheaper local distance

computation (as opposed to the transitive closure globally) and recursing on a geometrically decreasing series we obtain the desired complexity bounds for our algorithm. Both the strongly balanced tree decomposition and the fast local distance computation play important roles in our algorithmic improvements.

We now turn our attention to the analysis of ConcurPreprocess.

LEMMA 5.2. *$\overline{T}$ is a tree decomposition of the partial expansion $\overline{G}$.*

PROOF. By Theorem 4.5, $\mathsf{ConcurTree}(G)$ is a tree decomposition of G. To show that $\overline{T}$ is a tree decomposition of the partial expansion $\overline{G}$, it suffices to show that the conditions C1-C3 are met for every pair of nodes $\overline{u}^1$, $\overline{u}^2$ that correspond to a strict partial node $\overline{u}$ of $\overline{G}$. We only focus on $\overline{u}^1$, as the other case is similar.

C1 This condition is met, as $\overline{u}^1$ appears in every bag of $\overline{T}$ that contains a node u that refines $\overline{u}^1$.
C2 Since every node $\overline{u}^1$ is connected only to nodes u of G that refine $\overline{u}$, this condition is also met.
C3 First, observe that $\overline{u}^1$ appears in the root bag $\overline{B}$ of $\overline{T}$. Then, for every simple path $P : \overline{B} \rightsquigarrow \overline{B}'$ from the root to some leaf bag $\overline{B}'$, if $\overline{B}''$ is the first bag in P where $\overline{u}^1$ does not appear, then some non-$\perp$ constituent of u does not appear in bags of $\overline{T}_{\overline{B}''}$, hence neither does $\overline{u}^1$. Thus, $\overline{u}^1$ appears in a contiguous subtree of $\overline{T}$.

The desired result follows. □

In Lemma 5.3 we establish that the forward and backward maps computed by ConcurPreprocess store the distances between nodes.

LEMMA 5.3. *At the end of* ConcurPreprocess*, the following assertions hold:*

(1) *For all nodes $u, v \in V$ such that $\overline{B}_u$ appears in $\overline{T}(\overline{B}_v)$, we have* $\textsc{Fwd}^+_u(v) = d(u,v)$ *and* $\textsc{Bwd}^+_u(v) = d(v,u)$.
(2) *For all strictly partial nodes $\overline{v} \in \overline{V}$ and nodes $u \in V$ we have* $\textsc{Fwd}^+_u(\overline{v}^2) = d(u,\overline{v})$ *and* $\textsc{Bwd}^+_u(\overline{v}^1) = d(\overline{v},u)$.
(3) *For all strictly partial nodes $\overline{u}, \overline{v} \in \overline{V}$ we have* $\textsc{Fwd}_{\overline{u}^1}(\overline{v}^2) = d(\overline{u},\overline{v})$ *and* $\textsc{Bwd}_{\overline{u}^2}(\overline{v}^1) = d(\overline{v},\overline{u})$.

PROOF. We describe the key invariants that hold during the traversals of $\overline{T}$ by ConcurPreprocess in Item 3a, Item 3b and Item 4 after the algorithm processes a bag $\overline{B}$.

Item 3a For every pair of partial nodes $\overline{u}$, $\overline{v} \in \overline{B}$ such that $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$ we have $\textsc{Fwd}_{\overline{u}}(\overline{v}) \leq \bigoplus_{P_1} \otimes(P_1)$ and $\textsc{Bwd}_{\overline{u}}(\overline{v}) \leq \bigoplus_{P_2} \otimes(P_2)$ where P_1 and P_2 are $\overline{u} \rightsquigarrow \overline{v}$ and $\overline{v} \rightsquigarrow \overline{u}$ paths respectively that only traverse nodes in $\overline{V}_{\overline{T}}(\overline{B})$. The statement follows by a straightforward induction on the levels processed by the algorithm in the bottom-up pass. Note that if $\overline{u}$ and $\overline{v}$ are partial nodes in the root of $\overline{T}$, the statement yields $\textsc{Fwd}_{\overline{u}}(\overline{v}) = d(u,v)$ and $\textsc{Bwd}_{\overline{u}}(\overline{v}) = d(v,u)$.

Item 3b The invariant is similar to the previous, except that P_1 and P_2 range over all $\overline{u} \rightsquigarrow \overline{v}$ and $\overline{v} \rightsquigarrow \overline{u}$ paths in $\overline{G}$ respectively. Hence now $\textsc{Fwd}_{\overline{u}}(\overline{v}) = d(\overline{u},\overline{v})$ and $\textsc{Bwd}_{\overline{u}}(\overline{v}) = d(\overline{v},\overline{u})$. The statement follows by a straightforward induction on the levels processed by the algorithm in the top-down pass. Note that the base case on the root follows from the previous item, where the maps $\textsc{Bwd}$ and $\textsc{Fwd}$ store actual distances.

Item 4 For every node $u \in \overline{B}$ and partial node $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B})$ we have $\textsc{Fwd}^+_u(\overline{v}) = d(u,\overline{v})$ and $\textsc{Bwd}^+_u(\overline{v}) = d(\overline{v},u)$. The statement follows from Lemma 5.1 and a straightforward induction on the length of the path from the root of $\overline{T}$ to the processed bag $\overline{B}$. Indeed, the statement is true when $\overline{B}$ is the root of the tree decomposition, which serves as the basis of the induction. This follows from the correctness Item 3b, as at this point the maps $\textsc{Fwd}^+$ and $\textsc{Bwd}^+$ restricted to nodes of $\overline{B}$ are identical to the maps $\textsc{Fwd}$ and $\textsc{Bwd}$ restricted to nodes of $\overline{B}$. For the inductive step, consider any bag $\overline{B}$, and assume that the statement holds for the parent bag $\overline{B}'$ of $\overline{B}$. Lemma 5.1 yields that for bag $\overline{B}'' \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B})$, for every pair of partial nodes $\overline{u}, \overline{v}$ such that $\overline{u} \in \overline{B}$ and $\overline{v} \in \overline{B}''$, we have that

$$d(\overline{u}, \overline{v}) = \bigoplus_{\overline{w} \in \overline{B}'} (d(\overline{u}, \overline{w}) \otimes d(\overline{w}, \overline{v}))$$

By the induction hypothesis, the distances $d(\overline{w}, \overline{v})$ are found in the map $\textsc{Fwd}_u^+$, whereas the distances $d(\overline{u}, \overline{w})$ are found, by the correctness of Item 3b in the maps $\textsc{Fwd}_u$ and $\textsc{Bwd}_u$. It follows that the algorithm combines the distances computed in these maps to compute the distance $d(\overline{u}, \overline{v})$.

Statement 1 of the lemma follows from Item 4. Similarly for statement 2, together with the observation that every strictly partial node $\overline{v}$ appears in the root of $\overline{T}$, and thus $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u)$. Finally, statement 3 follows again from the fact that all strictly partial nodes appear in the root bag of $\overline{T}$. The desired result follows. □

Complexity analysis. We now consider the complexity analysis of ConcurPreprocess. Recall that ConcurPreprocess takes as part of its input a desired constant $0 < \epsilon \leqslant 1$. We choose a $\lambda \in \mathbb{N}$ and $\delta \in \mathbb{R}$ such that $\lambda \geqslant 4/\epsilon$ and $\delta \leqslant \epsilon/18$. Additionally, we set $\alpha = 6 \cdot \lambda/\delta$, $\beta = ((1+\delta)/2)^{\lambda-1}$ and $\gamma = \lambda$, which are the constants used for constructing an (α, β, γ)-balanced tree-decomposition $T_i = \mathrm{Tree}(G_i)$ in Item 2a of ConcurPreprocess. We start with a technical lemma on two recurrence relations, $\mathcal{T}_k$ and $\mathcal{S}_k$, which are parameterized by k, and will help us bound the time and space, respectively, spent by ConcurPreprocess.

LEMMA 5.4. *Consider the recurrences in Eq. 11 and Eq. 12.*

$$\mathcal{T}_k(n) \leqslant n^{3\cdot(k-1)} + 2^{\lambda\cdot k} \cdot \mathcal{T}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right) \tag{11}$$

$$\mathcal{S}_k(n) \leqslant n^{2\cdot(k-1)} + 2^{\lambda\cdot k} \cdot \mathcal{S}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right) \tag{12}$$

Then

(1) $\mathcal{T}_k(n) = O(n^{3\cdot(k-1)})$*, and*
(2) (i) $\mathcal{S}_k(n) = O(n^{2\cdot(k-1)})$ *if* $k \geqslant 3$*, and (ii)* $\mathcal{S}_2(n) = O(n^{2+\epsilon})$*.*

PROOF. We analyze each recurrence separately. First we consider Eq. 11. Note that

$$\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)^{3\cdot(k-1)} = \left(\frac{1+\delta}{2}\right)^{3\cdot(\lambda-1)\cdot(k-1)} \cdot n^{3\cdot(k-1)} \tag{13}$$

and

$$2^{\lambda\cdot k} \cdot \left(\frac{1+\delta}{2}\right)^{3\cdot(\lambda-1)\cdot(k-1)} = \frac{(1+\delta)^{3\cdot(\lambda-1)\cdot(k-1)}}{2^{2\cdot k\cdot\lambda+3\cdot(k+\lambda-1)}} \tag{14}$$

and since $\log(1+\delta) = \frac{\ln(1+\delta)}{\ln 2} < \frac{\delta}{\ln 2} < 2\cdot\delta$, we have

$$(1+\delta)^{3\cdot(\lambda-1)\cdot(k-1)} = 2^{\log(1+\delta)\cdot 3\cdot(\lambda-1)\cdot(k-1)} < 2^{6\cdot\delta\cdot(\lambda-1)\cdot(k-1)}$$

Hence the expression in Eq. 14 is bounded by 2^x with

$$\begin{aligned} x &\leqslant 6\cdot\delta\cdot(\lambda-1)\cdot(k-1) - 2\cdot k\cdot\lambda + 3\cdot(\lambda+k-1) \\ &= -2\cdot\lambda\cdot k\cdot(1-3\cdot\delta) + 3\cdot(\lambda+k-1)\cdot(1-2\cdot\delta) \end{aligned}$$

Let $f(k) = -2\cdot\lambda\cdot k\cdot(1-3\cdot\delta)+3\cdot(\lambda+k-1)\cdot(1-2\cdot\delta)$ and note that since $\lambda \geqslant \frac{4}{\epsilon} \geqslant 4$ and $\delta \leqslant \frac{\epsilon}{18} \leqslant \frac{1}{18}$, $f(k)$ is decreasing, and thus maximized for $k=2$, for which we obtain $f(2) = -4\cdot\lambda\cdot(1-3\cdot\delta)+3\cdot(\lambda+1)\cdot(1-2\cdot\delta) = -\lambda\cdot(1-6\cdot\delta)\leqslant 0$ as $\delta\leqslant\frac{1}{18}$. It follows that there exists a constant $c<1$ for which

$$2^{\lambda\cdot k}\cdot\mathcal{T}_k\left(n\cdot\left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)\leqslant c\cdot n^{3\cdot(k-1)}$$

which yields that Eq. 11 follows a geometric series, and thus $\mathcal{T}_k(n) = O(n^{3\cdot(k-1)})$.

We now turn our attention to Eq. 12. When $k\geqslant 3$, an analysis similar to Eq. 11 yields the bound $O(n^{2\cdot(k-1)})$. When $k=2$, since $\epsilon>0$, we write Eq. 12 as

$$\mathcal{S}_2(n)\leqslant n^{2+\epsilon}+2^{2\cdot\lambda}\cdot\mathcal{S}_2\left(n\cdot\left(\frac{1+\delta}{2}\right)^{\lambda-1}\right) \tag{15}$$

Similarly as above, we have

$$\left(n\cdot\left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)^{2+\epsilon} = \left(\frac{1+\delta}{2}\right)^{(2+\epsilon)\cdot(\lambda-1)}\cdot n^{2+\epsilon} \tag{16}$$

and

$$2^{2\cdot\lambda}\cdot\left(\frac{1+\delta}{2}\right)^{(2+\epsilon)\cdot(\lambda-1)} = \frac{(1+\delta)^{(2+\epsilon)\cdot(\lambda-1)}}{2^{-2+\epsilon\cdot(\lambda-1)}} \tag{17}$$

and since $\log(1+\delta)=\frac{\ln(1+\delta)}{\ln 2}<\frac{\delta}{\ln 2}<2\cdot\delta$, we have

$$(1+\delta)^{(2+\epsilon)\cdot(\lambda-1)}<2^{2\cdot\delta\cdot(2+\epsilon)\cdot(\lambda-1)}$$

Hence the expression in Eq. 17 is bounded by 2^x with

$$\begin{aligned}
x &\leqslant 2\cdot\delta\cdot(2+\epsilon)\cdot(\lambda-1)+2-\epsilon\cdot(\lambda-1)\\
&=(\lambda-1)\cdot(2\cdot\delta\cdot(2+\epsilon)-\epsilon)+2\\
&\leqslant(\lambda-1)\cdot\frac{4\cdot\epsilon+2\cdot\epsilon^2-18\cdot\epsilon}{18}+2\\
&\leqslant(1-\lambda)\cdot\epsilon\cdot\frac{2}{3}+2\\
&\leqslant-(4-\epsilon)\cdot\frac{2}{3}+2\leqslant 0
\end{aligned}$$

since $\delta\leqslant\frac{\epsilon}{18}$ and $\lambda\geqslant\frac{4}{\epsilon}$ and $\epsilon\leqslant 1$. It follows that there exists a constant $c<1$ for which

$$2^{2\cdot\lambda}\cdot\mathcal{S}_2(n)\leqslant c\cdot n^{2+\epsilon}$$

which yields that Eq. 15 follows a geometric series, and thus $\mathcal{S}_2(n)=O(n^{2+\epsilon})$.

□

The following lemma analyzes the complexity of ConcurPreprocess, and makes use of the above recurrences.

LEMMA 5.5. ConcurPreprocess *requires $O(n^{2\cdot k-1})$ space and (1) $O(n^{3\cdot(k-1)})$ time if $k\geqslant 3$, and (2) $O(n^{3+\epsilon})$ time if $k=2$.*

PROOF. We examine each step of the algorithm separately.

(1) The time and space required for this step is bounded by the number of nodes introduced in the partial expansion $\overline{G}$, which is $2 \cdot \sum_{i<k} \binom{n}{i} = O(n^{k-1})$.
(2) By Theorem 4.5, $\mathsf{ConcurTree}(G)$ is constructed in $O(n^k)$ time and space. In $\overline{T}$, the size of each bag $\overline{B}$ is increased by constant factor, hence this step requires $O(n^k)$ time and space.
(3) In each pass, $\mathsf{ConcurPreprocess}$ spends $|\overline{B}|^3$ time to perform an all-pairs algebraic paths computation in each bag $\overline{B}$ of $\overline{T}$ [Lehmann 1977; Floyd 1962; Warshall 1962; Kleene 1956]. The space usage for storing all maps $\textsc{Fwd}_{\overline{u}}$ and $\textsc{Bwd}_{\overline{u}}$ for every node $\overline{u}$ whose root bag is $\overline{B}$ is $O(|\overline{B}|^2)$, since there are at most $|\overline{B}|$ such nodes $\overline{u}$, and each map has size $|\overline{B}|$. By the previous item, we have $|\overline{B}| = O(|B|)$, where B is the corresponding bag of T before the partial expansion of G. By Theorem 4.5, we have $|B| = O(n^{k-1} \cdot \beta^i)$, where $\mathsf{Lv}(B) \geqslant i \cdot \gamma = i \cdot \lambda$, and $\beta = ((1+\delta)/2)^{\lambda-1}$. Then, since $\overline{T}$ is a full 2^k-ary tree, the time and space required for preprocessing every $\gamma = \lambda$ levels of $\overline{T}$ is given by the following recurrences respectively (ignoring constant factors for simplicity).

$$\mathcal{T}_k(n) \leqslant n^{3\cdot(k-1)} + 2^{\lambda\cdot k} \cdot \mathcal{T}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)$$

$$\mathcal{S}_k(n) \leqslant n^{2\cdot(k-1)} + 2^{\lambda\cdot k} \cdot \mathcal{S}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)$$

By the analysis of Eq. 11 and Eq. 12 of Lemma 5.4, we have that $\mathcal{T}_k(n) = O(n^{3\cdot(k-1)})$ and (i) $\mathcal{S}_k(n) = O(n^{2\cdot(k-1)})$ if $k \geqslant 3$, and (ii) $\mathcal{S}_2(n) = O(n^{2+\epsilon})$.
(4) We first focus on the space usage. Let $\overline{B}^i_u$ denote the ancestor bag of $\overline{B}_u$ at level i. We have

$$\begin{aligned}|\overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u)| &\leqslant \sum_i |\overline{B}^i_u| \leqslant c_1 \cdot \sum_i |\overline{B}^{\lfloor i/\gamma \rfloor}_u| \leqslant c_2 \cdot \sum_i |B^{\lfloor i/\gamma \rfloor}_u| \\ &\leqslant c_3 \cdot \sum_i \left(n^{k-1} \cdot \beta^i\right) = O(n^{k-1})\end{aligned}$$

for some constants c_1, c_2, c_3. The first inequality comes from expressing the size of all (constantly many) ancestors $\overline{B}^i_u$ with $\lfloor i/\gamma \rfloor = j$ as a constant factor the size of $\overline{B}^{\lfloor i/\gamma \rfloor}_u$. The second inequality comes from Item 1 of this lemma, which states that $O(|\overline{B}|) = O(|B|)$ for every bag $\overline{B}$. The third inequality comes from Theorem 4.5. The final equality holds because β is a constant, and thus the sum forms a geometric series. By Item 2, there are $O(n^k)$ such nodes u in $\overline{T}$, hence the space required is $O(n^{2\cdot k-1})$.
We now turn our attention to the time requirement. For every bag $\overline{B}$, the algorithm requires $O(|\overline{B}|^2)$ time to iterate over all pairs of nodes u and x in Eq. 9 and Eq. 10 to compute the values $\textsc{Fwd}^+_u(\overline{v})$ and $\textsc{Bwd}^+_u(\overline{v})$ for every $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B})$. Hence the time required for all nodes u and one partial node $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B})$ to store the maps values $\textsc{Fwd}^+_u(\overline{v})$ and $\textsc{Bwd}^+(\overline{v})$ is given by the recurrence

$$\mathcal{T}_k(n) \leqslant n^{2\cdot(k-1)} + 2^{\lambda\cdot k} \cdot \mathcal{T}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)$$

The analysis of Eq. 11 and Eq. 12 of Lemma 5.4 gives $\mathcal{T}_k(n) = O(n^{2\cdot(k-1)})$ for $k \geqslant 3$ and $\mathcal{T}_2(n) = O(n^{2+\epsilon})$ (i.e., the above time recurrence is analyzed as the recurrence for $\mathcal{S}_k$ of Lemma 5.4). From the space analysis we have that there exist $O(n^{k-1})$ partial nodes $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B})$

for every node u whose root bag is $\overline{B}$. Hence the total time for this step is $O(n^{3\cdot(k-1)})$ for $k \geqslant 3$, and $O(n^{3+\epsilon})$ for $k = 2$.

(5) This step requires time linear in the size of $\overline{T}$ [Harel and Tarjan 1984].

The desired result follows. □

Algorithm ConcurQuery**.** In the query phase, ConcurAP answers distance queries using the algorithm ConcurQuery. We distinguish three cases, according to the type of the query. Recall that all strictly partial nodes appear in the root bag of the tree decomposition.

(1) *Single-source query.* Given a source node u, initialize a map data-structure $A : V \to \Sigma$, and initially set $A(v) = \textsc{Fwd}_u(v)$ for all $v \in \overline{B}_u$, and $A(v) = \overline{\mathbf{0}}$ for all other nodes $v \in V \setminus \overline{B}_u$. Perform a BFS on $\overline{T}$ starting from $\overline{B}_u$, and for every encountered bag $\overline{B}$ and nodes $x, v \in \overline{B}$ with $\mathsf{Lv}(v) \leqslant \mathsf{Lv}(x)$, set $A(v) = A(v) \oplus (A(x) \otimes \textsc{Fwd}_x(v))$. Return the map A.
(2) *Pair query.* Given two nodes $u, v \in V$, find the LCA $\overline{B}$ of bags $\overline{B}_u$ and $\overline{B}_v$. Return $\bigoplus_{x \in \overline{B} \cap V} (\textsc{Fwd}^+_u(x) \otimes \textsc{Bwd}^+_v(x))$.
(3) *Partial pair query.* Given two partial nodes $\overline{u}, \overline{v}$,
   (a) If both $\overline{u}$ and $\overline{v}$ are strictly partial, return $\textsc{Fwd}_{\overline{u}^1}(\overline{v}^2)$, else
   (b) If $\overline{u}$ is strictly partial, return $\textsc{Bwd}^+_v(\overline{u}^1)$, else
   (c) Return $\textsc{Fwd}^+_u(\overline{v}^2)$.

We thus establish the following theorem.

THEOREM 5.6. *Let $G = (V, E)$ be a concurrent graph of k constant-treewidth graphs $(G_i)_{1 \leqslant i \leqslant k}$, and* wt $: E \to \Sigma$ *a weight function of G. For any fixed $\epsilon > 0$, the data-structure* ConcurAP *correctly answers single-source and pair queries and requires:*

(*1*) *Preprocessing time*
   (*a*) $O(n^{3\cdot(k-1)})$ *if* $k \geqslant 3$, *and* (*b*) $O(n^{3+\epsilon})$ *if* $k = 2$.
(*2*) *Preprocessing space* $O(n^{2\cdot k-1})$.
(*3*) *Single-source query time*
   (*a*) $O(n^{2\cdot(k-1)})$ *if* $k \geqslant 3$, *and* (*b*) $O(n^{2+\epsilon})$ *if* $k = 2$.
(*4*) *Pair query time* $O(n^{k-1})$.
(*5*) *Partial pair query time* $O(1)$.

PROOF. The correctness of ConcurQuery for handling all queries follows from Lemma 5.1 and the properties of the preprocessing established in Lemma 5.3. The preprocessing complexity is stated in Lemma 5.5. The time complexity for the single-source query comes from the observation that ConcurQuery spends quadratic time in each encountered bag, and the result follows from the recurrence analysis of Eq. 12 in Lemma 5.4. The time complexity for the pair query follows from the $O(1)$ time to access the LCA bag $\overline{B}$ of $\overline{B}_u$ and $\overline{B}_v$, and the $O(|\overline{B}|) = O(n^{k-1})$ time required to iterate over all nodes $x \in \overline{B} \cap V$. Finally, the time complexity for the partial pair query follows from the $O(1)$ time lookup in the constructed maps $\textsc{Fwd}$, $\textsc{Fwd}^+$ and $\textsc{Bwd}^+$. □

Note that a single-source query from a strictly partial node $\overline{u}$ can be answered in $O(n^k)$ time by breaking it down to n^k partial pair queries. The most common case in analysis of concurrent programs is that of two threads, for which we obtain the following corollary.

COROLLARY 5.7. *Let $G = (V, E)$ be a concurrent graph of two constant-treewidth graphs G_1, G_2, and* wt $: E \to \Sigma$ *a weight function of G. For any fixed $\epsilon > 0$, the data-structure* ConcurAP *correctly answers single-source and pair queries and requires:*

(*1*) *Preprocessing time* $O(n^{3+\epsilon})$.
(*2*) *Preprocessing space* $O(n^3)$.
(*3*) *Single-source query time* $O(n^{2+\epsilon})$.

(*4*) *Pair query time* $O(n)$.
(*5*) *Partial pair query time* $O(1)$.

Remark 5.8. In contrast to Corollary 5.7, the existing methods for handling even one pair query require hexic time and quartic space [Lehmann 1977; Floyd 1962; Warshall 1962; Kleene 1956] by computing the transitive closure. While our improvements are most significant for algebraic path queries, they imply improvements also for special cases like reachability (expressed in Boolean semirings). For reachability, the complete preprocessing requires quartic time, and without preprocessing every query requires quadratic time. In contrast, with almost cubic preprocessing we can answer pair (resp., partial pair) queries in linear (resp., constant) time.

Note that Item 4 of ConcurPreprocess is required for handling pair queries only. By skipping this step, we can handle every (partial) pair query $\overline{u}, \overline{v}$ similarly to the single source query from $\overline{u}$, but restricting the BFS to the path $P : \overline{B}_{\overline{u}} \rightsquigarrow \overline{B}_{\overline{v}}$, and spending $O(|\overline{B}|^2)$ time for each bag $\overline{B}$ of P. Recall (Theorem 4.5) that the size of each bag B in T (and thus the size of the corresponding bag $\overline{B}$ in $\overline{T}$) decreases geometrically every γ levels. Then, the time required for this operation is $O(|\overline{B}'|^2) = O(n^2)$, where $\overline{B}'$ is the bag of P with the smallest level. This leads to the following corollary.

COROLLARY 5.9. *Let $G = (V, E)$ be a concurrent graph of two constant-treewidth graphs G_1, G_2, and* wt $: E \to \Sigma$ *a weight function of G. For any fixed ϵ, the data-structure* ConcurAP *(by skipping Item 4 in* ConcurPreprocess*) correctly answers single-source and pair queries and requires:*

(*1*) *Preprocessing time* $O(n^3)$.
(*2*) *Preprocessing space* $O(n^{2+\epsilon})$.
(*3*) *Single-source query time* $O(n^{2+\epsilon})$.
(*4*) *Pair and partial pair query time* $O(n^2)$.

Finally, we can use ConcurAP to obtain the transitive closure of G by performing n^2 single-source queries. The preprocessing space is $O(n^{2+\epsilon})$ by Corollary 5.9, and the space of the output is $O(n^4)$, since there are n^4 pairs for the computed distances. Hence the total space requirement is $O(n^4)$. The time requirement is $O(n^{4+\epsilon})$, since by Corollary 5.9, every single-source query requires $O(n^{2+\epsilon})$ time. We obtain the following corollary.

COROLLARY 5.10. *Let $G = (V, E)$ be a concurrent graph of two constant-treewidth graphs G_1, G_2, and* wt $: E \to \Sigma$ *a weight function of G. For any fixed $\epsilon > 0$, the transitive closure of G wrt* wt *can be computed in $O(n^{4+\epsilon})$ time and $O(n^4)$ space.*

6. CONDITIONAL OPTIMALITY FOR TWO GRAPHS

In the current section we establish the optimality of Corollary 5.9 in handling algebraic path queries in a concurrent graph that consists of two constant-treewidth components. The key idea is to show that for any arbitrary graph (i.e., without the constant-treewidth restriction) G of n nodes, we can construct a concurrent graph G' as a 2-self-concurrent asynchronous composition of a constant-treewidth graph G'' of $2 \cdot n$ nodes, such that semiring queries in G coincide with semiring queries in G'.

Arbitrary graphs as composition of two constant-treewidth graphs. We fix an arbitrary graph $G = (V, E)$ of n nodes, and a weight function wt $: E \to \Sigma$. Let x_i, $1 \leqslant i \leqslant n$ range over the nodes V of G, and construct a graph $G'' = (V'', E'')$ such that $V'' = \{x_i, y_i : 1 \leqslant i \leqslant n\}$ and $E'' = \{(x_i, y_i), (y_i, x_i) : 1 \leqslant i \leqslant n\} \cup \{(y_i, y_{i+1}), (y_{i+1}, y_i) : 1 \leqslant i < n\}$.

CLAIM 6. *The treewidth of G'' is* 1.

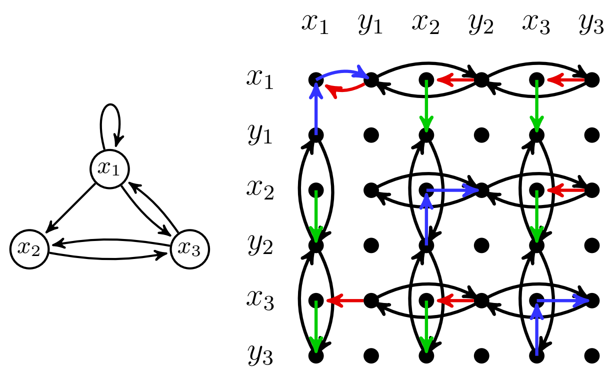

Fig. 9: A graph G (left), and G' that is a 2-self-product of a graph G'' of treewidth 1 (right). The weighted edges of G correspond to weighted red edges on G'. The distance $d(x_i, x_j)$ in G equals the distance $d(\langle x_i, x_i\rangle, \langle x_j, x_j\rangle) = d(\langle \perp, x_i\rangle, \langle \perp, x_j\rangle)$ in G'.

PROOF. Observe that if we (i) ignore the direction of the edges and (ii) remove multiple appearances of the same edge, we obtain a tree. It is known that trees have treewidth 1. □

Given G'', we construct a graph G' as a 2-self-concurrent asynchronous composition of G''. Informally, a node x_i of G corresponds to the node $\langle x_i, x_i\rangle$ of G'. An edge (x_i, x_j) in G is simulated by two paths in G'.

(1) The first path has the form $P_1 : \langle x_i, x_i\rangle \rightsquigarrow \langle x_i, x_j\rangle$, and is used to witness the weight of the edge in G, i.e., $\mathsf{wt}(x_i, x_j) = \otimes(P_1)$. It traverses a sequence of nodes, where the first constituent is fixed to x_i, and the second constituent forms the path $x_i \to y_i \to y_{i'} \to \cdots \to y_j \to x_j$. The last transition will have weight equal to $\mathsf{wt}(x_i, x_j)$, and the other transitions have weight $\overline{\mathbf{1}}$. Any path that has the above form can be taken as P_1.
(2) The second path has the form $P_2 : \langle x_i, x_j\rangle \rightsquigarrow \langle x_j, x_j\rangle$, it has no weight (i.e., $\otimes(P_2) = \overline{\mathbf{1}}$), and is used to reach the node $\langle x_j, x_j\rangle$. It traverses a sequence of nodes, where the second constituent is fixed to x_j, and the first constituent forms the path $x_i \to y_i \to y_{i'} \to \cdots \to y_j \to x_j$. Any path that has the above form can be taken as P_2.

Then the concatenation of P_1 and P_2 creates a path $P : \langle x_i, x_i\rangle \rightsquigarrow \langle x_j, x_j\rangle$ with $\otimes(P) = \otimes(P_1) \otimes \otimes(P_2) = \mathsf{wt}(x_i, x_j) \otimes \overline{\mathbf{1}} = \mathsf{wt}(x_i, x_j)$.

Formal construction. We construct a graph $G' = (V', E')$ as a 2-self-concurrent asynchronous composition of G'', by including the following edges.

(1) *Black edges.* For all $1 \leqslant i \leqslant n$ and $1 \leqslant j < n$ we have $(\langle x_i, y_j\rangle, \langle x_i, y_{j+1}\rangle), (\langle x_i, y_{j+1}\rangle, \langle x_i, y_j\rangle) \in E'$, and for all $1 \leqslant i < n$ and $1 \leqslant j \leqslant n$ we have $(\langle y_i, x_j\rangle, \langle y_{i+1}, x_j\rangle), (\langle y_{i+1}, x_j\rangle, \langle y_i, x_j\rangle) \in E'$.
(2) *Blue edges.* For all $1 \leqslant i \leqslant n$ we have $(\langle x_i, x_i\rangle, \langle x_i, y_i\rangle), (\langle y_i, x_i\rangle, \langle x_i, x_i\rangle) \in E'$.
(3) *Red edges.* For all $(x_i, x_j) \in E$ we have $(\langle x_i, y_j\rangle, \langle x_i, x_j\rangle) \in E'$.
(4) *Green edges.* For all $1 \leqslant i, j \leqslant n$ with $i \neq j$ we have $(\langle x_i, x_j\rangle, \langle y_i, x_j\rangle) \in E'$.

Additionally, we construct a weight function such that $\mathsf{wt}'(\langle x_i, y_j\rangle, \langle x_i, x_j\rangle) = \mathsf{wt}(x_i, x_j)$ for every red edge $(\langle x_i, y_j\rangle, \langle x_i, x_j\rangle)$, and $\mathsf{wt}'(u, v) = \overline{\mathbf{1}}$ for every other edge (u, v). Figure 9 provides an illustration of the construction.

LEMMA 6.1. *For every $x_i, x_j \in V$, there exists a path $P : x_i \rightsquigarrow x_j$ with $\otimes(P) = z$ in G iff there exists a path $P' : \langle x_i, x_i\rangle \rightsquigarrow \langle x_j, x_j\rangle$ with $\otimes(P') = z$ in G'.*

PROOF. Recall that only red edges contribute to the weights of paths in G'. We argue that there is path $\overline{P} : \langle x_i, x_i\rangle \rightsquigarrow \langle x_j, x_j\rangle$ in G' that traverses a single red edge iff there is an edge (x_i, x_j) in G with $\otimes(\overline{P}) = \mathsf{wt}(x_i, x_j)$.

(1) Given the edge (x_i, x_j), the path $\overline{P}$ is formed by traversing the red edge $(\langle x_i, y_j\rangle, \langle x_i, x_j\rangle)$ as

$$\langle x_i, x_i\rangle \to \langle x_i, y_i\rangle \rightsquigarrow \langle x_i, y_j\rangle \to \langle x_i, x_j\rangle \to \langle y_i, x_j\rangle \rightsquigarrow \langle y_i, x_j\rangle \to \langle x_j, y_j\rangle$$

Since $\mathsf{wt}((\langle x_i, y_j\rangle, \langle x_i, x_j\rangle)) = \mathsf{wt}(x_i, x_j)$ and all other edges of $\overline{P}$ have weight $\overline{\mathbf{1}}$, we have that $\otimes(\overline{P}) = \mathsf{wt}(x_i, x_j)$.

(2) Every path $\overline{P}$ that traverses a red edge $\langle x_{i'}, y_{j'}\rangle \to \langle x_{i'}, x_{j'}\rangle$ has to traverse a blue edge to $\langle x_{j'}, x_{j'}\rangle$. Then $x_{j'}$ must be x_j, otherwise $\overline{P}$ will traverse a second red edge before reaching $\langle x_j, x_j\rangle$.

The result follows easily from the above. □

Lemma 6.1 implies that for every $x_i, x_j \in V$, we have $d(x_i, x_j) = d(\langle x_i, x_i\rangle, \langle x_j, x_j\rangle)$, i.e., pair queries in G for nodes x_i, x_j coincide with pair queries $(\langle x_i, x_i\rangle, \langle x_j, x_j\rangle)$ in G'. Observe that in G' we have $d(\langle x_i, x_i\rangle, \langle x_j, x_j\rangle) = d(\langle \perp, x_i\rangle, \langle \perp, x_j\rangle)$, and hence pair queries in G also coincide with partial pair queries in G'.

THEOREM 6.2. *For every graph $G = (V, E)$ and weight function $\mathsf{wt} : E \to \Sigma$ there exists a graph $G' = (V \times V, E')$ that is a 2-self-concurrent asynchronous composition of a constant-treewidth graph, together with a weight function $\mathsf{wt}' : E' \to \Sigma$, such that for all $u, v \in V$, and $\langle u, u\rangle, \langle v, v\rangle \in V'$ we have $d(u, v) = d(\langle u, u\rangle, \langle v, v\rangle) = d(\langle \perp, u\rangle, \langle \perp, v\rangle)$. Moreover, the graph G' can be constructed in quadratic time in the size of G.*

This leads to the following corollary.

COROLLARY 6.3. *Let $\mathcal{T}_S(n) = \Omega(n^2)$ be a lower bound on the time required to answer a single algebraic paths query wrt to a semiring S on arbitrary graphs of n nodes. Consider any concurrent graph G which is an asynchronous self-composition of two constant-treewidth graphs of n nodes each. For any data-structure DS, let $\mathcal{T}_{\mathsf{DS}}(G, r)$ be the time required by DS to preprocess G and answer r pair queries. That is, DS is an oracle that has a build phase for preprocessing G, and and query phase for handing queries on the semiring distance between nodes of G. We have $\mathcal{T}_{\mathsf{DS}}(G, 1) = \Omega(\mathcal{T}_S(n))$.*

Conditional optimality of Corollary 5.9. Note that for $r = O(n)$ pair queries, Corollary 5.9 yields that the time spent by our data-structure ConcurAP for preprocessing G and answering r queries is $\mathcal{T}_{\mathsf{ConcurAP}}(G, r) = O(n^3)$. The long-standing (over five decades) upper bound for answering even one pair query for algebraic path properties in arbitrary graphs of n nodes is $O(n^3)$. Theorem 6.2 implies that any improvement upon our results would yield the same improvement for the long-standing upper bound, which would be a major breakthrough.

Almost-optimality of Theorem 5.6 and Corollary 5.10. Finally, we highlight some almost-optimality results obtained by variants of ConcurAP for the case of two graphs. By almost-optimality we mean that the obtained bounds are $O(n^\epsilon)$ factor worse that optimal, for any fixed $\epsilon > 0$ arbitrarily close to 0.

(1) According to Theorem 5.6, after $O(n^{3+\epsilon})$ preprocessing time, single-source queries are handled in $O(n^{2+\epsilon})$ time, and partial pair queries in $O(1)$ time. The former (resp. later) query time is almost linear (resp. exactly linear) in the size of the output. Hence the former queries are handled almost-optimally, and the latter indeed optimally. Moreover, this is achieved using $O(n^{3+\epsilon})$ preprocessing time, which is far less than the $\Omega(n^4)$ time required for the transitive closure computation (which computes the distance between all n^4 pairs of nodes).

(2) According to Corollary 5.10, the transitive closure can be computed in $O(n^{4+\epsilon})$ time, for any fixed $\epsilon > 0$, and $O(n^4)$ space. Since the size of the output is $\Theta(n^4)$, the transitive closure is computed in almost-optimal time and optimal space.

7. MODELING EXAMPLE

Figure 3 illustrates the introduced notions in a small example of the well-known k dining philosophers problem. For the purpose of the example, lock is considered a blocking operation. Consider the case of $k = 2$ threads being executed in parallel. The graphs G_1 and G_2 that correspond to the two threads have nodes of the form (i, ℓ), where $i \in [20]$ is a node of the controlflow graph, and $\ell \in [3]$ denotes the thread that controls the lock ($\ell = 3$ denotes that ℓ is free, whereas $\ell = i \in [2]$ denotes that it is acquired by thread i). The concurrent graph G is taken to be the asynchronous composition of G_1 and G_2, and consists of nodes $\langle x, y\rangle$, where x and y is a node of G_1 and G_2 respectively, such that x and y agree on the value of ℓ (all other nodes can be discarded). For brevity, we represent nodes of G as triplets $\langle x, y, \ell\rangle$ where now x and y are nodes in the controlflow graphs G_1 and G_2 (i.e., without carrying the value of the lock), and ℓ is the value of the lock. A transition to a node $\langle x, y, \ell\rangle$ in which one component G_i performs a lock is allowed only from a node where $\ell = 3$, and sets $\ell = i$ in the target node (i.e., $\langle x, y, i\rangle$). Similarly, a transition to a node $\langle x, y, \ell\rangle$ in which one component G_i performs an unlock is allowed from a node where $\ell = i$, and sets $\ell = 3$ in the target node(i.e., $\langle x, y, 3\rangle$).

Suppose that we are interested in determining (1) whether the first thread can execute dine(fork, knife) without owning fork or knife, and (2) whether a deadlock can be reached in which each thread owns one resource. These questions naturally correspond to partial pair and pair queries respectively, as in case (1) we are interested in a local property of G_1, whereas in case (2) we are interested in a global property of G. We note, however, that case (1) still requires an analysis on the concurrent graph G. In each case, the analysis requires a set of datafacts D, along with dataflow functions $f : 2^D \to 2^D$ that mark each edge. These functions are distributive, in the sense that $f(A) = \bigcup_{a\in A} f(a)$. Hence, with a slight abuse of notation, we can define f as functions $f : D \to D$, and their extension to $2^D \to 2^D$ is according to the distributivity property.

Local property as a partial pair query. Assume that we are interested in determining whether the first thread can execute dine(fork, knife) without owning fork or knife. A typical datafact set is $D = \{\mathsf{fork}, \mathsf{knife}, \mathsf{null}\}$, where each datafact denotes that the corresponding resource must be owned by the first thread. The concurrent graph G is associated with a weight function wt of dataflow functions $f : 2^D \to 2^D$. The dataflow function $\mathsf{wt}(e)$ along an edge e behaves as follows on input datafact F (we only describe the case where $F = \mathsf{fork}$, as the other case is symmetric).

(1) If e transitions to a node in which the second thread acquires fork or the first thread releases fork, then $\mathsf{wt}(e)(\mathsf{fork}) \to \mathsf{null}$ (i.e., fork is removed from the datafacts).
(2) Else, if e transitions to a node in which the first thread acquires fork, then $\mathsf{wt}(e)(\mathsf{null}) \to \mathsf{fork}$ (i.e., fork is inserted to the datafacts).

Similarly for the $F = \mathsf{knife}$ datafact. The "meet-over-all-paths" operation is set intersection. Then the question is answered by testing whether $d(\langle 1, 1, 3\rangle, \langle 14, \bot, 3\rangle) = \{\{\mathsf{fork}, \mathsf{knife}\}\}$, i.e., by performing a *partial pair query*, in which the node of the second thread is unspecified.

Global property as a pair query. Assume that we are interested in determining whether the two threads can cause a deadlock. Because of symmetry, we look for a deadlock in which the first thread may hold the fork, and the second thread may hold the knife. A typical datafact set is $D = 2^{\{\mathsf{fork}, \mathsf{knife}\}}$. For a datafact $F \in D$ we have

(1) $\mathsf{fork} \in F$ if fork may be acquired by the *first* thread.
(2) $\mathsf{knife} \in F$ if knife may be acquired by the *second* thread.

The concurrent graph G is associated with a weight function wt of dataflow functions $f : 2^D \to 2^D$. The dataflow function $\mathsf{wt}(e)$ along an edge e behaves as follows on input datafact F.

(1) If e transitions to a node in which the second thread acquires fork or the first thread releases fork, then $\mathsf{wt}(e)(F) \to F \setminus \{\mathsf{fork}\}$ (i.e., the first thread no longer owns fork).
(2) If e transitions to a node in which the first thread acquires fork, then $\mathsf{wt}(e)(F) \to F \cup \{\mathsf{fork}\}$ (i.e., the first thread now owns fork).
(3) If e transitions to a node in which the first thread acquires knife or the second thread releases knife, then $\mathsf{wt}(e)(F) \to F \setminus \{\mathsf{knife}\}$ (i.e., the second thread no longer owns knife).
(4) If e transitions to a node in which the second thread acquires knife, then $\mathsf{wt}(e)(F) \to F \cup \{\mathsf{knife}\}$ (i.e., the second thread now owns knife).

The "meet-over-all paths" operation is set union. Then the question is answered by testing whether $\{\mathsf{fork}, \mathsf{knife}\} \in d(\langle 1, 1, 3\rangle, \langle 2, 2, 3\rangle)$, i.e., by performing a *pair query*, and finding out whether the two threads can start the $while$ loop with each one holding one resource. Alternatively, we can answer the question by performing a single-source query from $\langle 1, 1, 3\rangle$ and finding out whether there exists any node in the concurrent graph G in which every thread owns one resource (i.e., its distance contains $\{\mathsf{fork}, \mathsf{knife}\}$).

8. EXPERIMENTAL RESULTS

In the current section we report on experimental evaluation of our algorithms, in particular of the algorithms of Corollary 5.10. We test their performance for obtaining the transitive closure on various concurrent graphs. We focus on the transitive closure for a fair comparison with the existing algorithmic methods, which compute the transitive closure even for a single query. Since the contributions of this work are algorithmic improvements for algebraic path properties, we consider the most fundamental representative of this framework, namely, the shortest path problem. Our comparison is done against the standard Bellman-Ford algorithm, which (i) has the best worst-case complexity for the problem, and (ii) allows for practical improvements, such as early termination.

Basic setup. We outline the basic setup used in all experiments. We use two different sets of benchmarks, and obtain the controlflow graphs of Java programs using Soot [Vallée-Rai et al. 1999], and use LibTW [van Dijk et al. 2006] to obtain the tree decompositions of the corresponding graphs. For every obtained graph G', we construct a concurrent graph G as a 2-self asynchronous composition of G', and then assign random integer weights in the range $[-10^3, 10^3]$, without negative cycles. Although this last restriction does not affect the running time of our algorithms, it allows for early termination of the Bellman-Ford algorithm (and thus only benefits the latter). The 2-self composition is a natural situation arising in practice, e.g. in concurrent data-structures where two threads of the same method access the data-structure. We note that the 2-self composition is no simpler than the composition of any two constant-treewidth graphs, (recall that the lower-bound of Section 6 is established on a 2-self composition).

DaCapo benchmarks. In our first setup, we extract controlflow graphs of methods from the DaCapo suit [Blackburn 2006]. The average treewidth of the input graphs is around 6. This supplies a large pool of 120 concurrent graphs, for which we use Corollary 5.10 to compute the transitive closure. This allows us to test the scalability of our algorithms, as well as their practical dependence on input parameters. Recall that our transitive closure time complexity is $O(n^{4+\epsilon})$, for any fixed $\epsilon > 0$, which is achieved by choosing a sufficiently large $\lambda \in \mathbb{N}$ and a sufficiently small $\delta \in \mathbb{R}$ when running the algorithm of Theorem 3.1. We compute the transitive closure for various λ. In practice, δ has effects only for very large input graphs. For this, we fix it to a large value ($\delta = 1/3$) which can be proved to have no effect on the obtained running times. Table II shows for each value of λ, the percentage of cases for which that value is at most 5% slower than the smallest time (among all tested λ) for each examined case. We find that $\lambda = 7$ works best most of the time.

λ	2	3	4	5	6	7	8
%	6	7	16	22	25	57	17

Table II: Percentage of cases for which the transitive closure of the graph G for the given value of λ is at most 5% slower than the time required to obtain the transitive closure of G for the best λ.

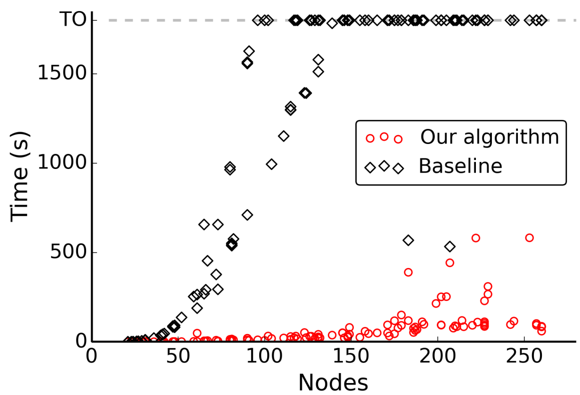

Fig. 10: Time required to compute the transitive closure on concurrent graphs of various sizes. Our algorithm is run for $\lambda = 7$. TO denotes that the computation timed out after 30 minutes.

Figure 10 shows the time required to compute the transitive closure on each concurrent graph G by our algorithm (for $\lambda = 7$) and the baseline Bellman-Ford algorithm. We see that our algorithm significantly outperforms the baseline method. Note that our algorithm seems to scale much better than its theoretical worst-case bound of $O(n^{4+\epsilon})$ of Corollary 5.10.

Concurrency with locks. Our second set of experiments is on methods from containers of the java.util.concurrent library that use locks as their synchronization mechanism. The average treewidth of the input graphs is around 8. In this case, we expand the node set of the concurrent graph G with the lock set $[3]^{\ell}$, where ℓ is the number of locks used by G'. Intuitively, the i-th value of the lock set denotes which of the two components owns the i-th lock (the value is 3 if the lock is free). Transitions to nodes that perform lock operations are only allowed wrt the lock semantics. That is, a transition to a node of G where the value of the i-th lock is

(1) *(Lock aquire):* $j \in [2]$, is only allowed from nodes where the value of that lock is 3, and the respective graph G_j is performing a lock operation on that edge.
(2) *(Lock release):* 3, is only allowed from nodes where the value of that lock is $j \in [2]$, and the respective graph G_j is performing an unlock operation on that edge.

Similarly as before, we compare our transitive closure time with the standard Bellman-Ford algorithm. Table III shows a time comparison between our algorithms and the baseline method. We observe that our transitive closure algorithm is significantly faster, and also scales better.

9. CONCLUSIONS

We have considered the fundamental algorithmic problem of computing algebraic path properties in a concurrent intraprocedural setting, where component graphs have constant treewidth. We have presented algorithms that significantly improve the existing theoretical complexity of the problem, and provide a variety of tradeoffs between preprocessing and query times for on-demand analyses. Moreover, we have proved that further theoretical improvements over our algorithms must achieve major breakthroughs. An interesting direction of future work is to extend our algorithms to the

Java method	$\mathbf{n}$	$\mathbf{T_o(s)}$	$\mathbf{T_b(s)}$
ArrayBlockingQueue: poll	19	19	60
ArrayBlockingQueue: peek	20	20	81
LinkedBlockingDeque: advance	25	29	195
PriorityBlockingQueue: removeEQ	25	32	176
ArrayBlockingQueue: init	26	47	249
LinkedBlockingDeque: remove	26	49	290
ArrayBlockingQueue: offer	26	56	304
ArrayBlockingQueue: clear	28	33	389
ArrayBlockingQueue: contains	32	205	881
DelayQueue: remove	42	267	3792
ConcurrentHashMap: scanAndLockForPut	46	375	2176
ArrayBlockingQueue: next	46	407	3915
ConcurrentHashMap: put	72	1895	> 8 h

Table III: Time required for the transitive closure on 2-self concurrent graphs extracted from methods of the java.util.concurrent library. Each constituent graph has $\mathbf{n}$ nodes. $\mathbf{T_o(s)}$ and $\mathbf{T_b(s)}$ correspond to our method and the baseline method respectively.

interprocedural setting. However, in that case even the basic problem of reachability is undecidable, and other techniques and formulations are required to make the analysis tractable, such as context-bounded formulations and regular approximations of interprocedural paths [Qadeer and Rehof 2005; Bouajjani et al. 2005; Lal and Reps 2009]. The effect of constant-treewidth components in such formulations is an interesting theoretical direction to pursue, with potential for practical use.

APPENDIX: Formal Pseudocode of Our Algorithms

Here we present formally (in pseudocode) the algorithms that appear in the main part of the paper.

ALGORITHM 1: Rank

Input: A component $\mathcal{C}$ of T, a natural number $\ell \in [\lambda]$
Output: A rank tree $\mathsf{R_G}$

Assign $\mathcal{T} \leftarrow$ an empty tree
if $|\mathcal{C}| \cdot \frac{\delta}{2} \leqslant 1$ **then**
  Assign $\mathcal{B} \leftarrow \bigcup_{B \in \mathcal{C}} B$ and make $\mathcal{B}$ the root of $\mathcal{T}$
else if $\ell > 0$ **then**
  Assign $(\mathcal{X}, \mathcal{Y}) \leftarrow \mathsf{Split}(\mathcal{C})$
  Assign $\mathcal{B} \leftarrow \bigcup_{B_i \in \mathcal{X}} B_i$
  Assign $(\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2) \leftarrow \mathsf{Merge}(\mathcal{Y})$
  Assign $\mathcal{T}_1 \leftarrow \mathsf{Rank}(\overline{\mathcal{C}}_1, (\ell+1) \mod \lambda)$
  Assign $\mathcal{T}_2 \leftarrow \mathsf{Rank}(\overline{\mathcal{C}}_2, (\ell+1) \mod \lambda)$
  Make $\mathcal{B}$ the root of $\mathcal{T}$ and $\mathcal{T}_1$ and $\mathcal{T}_2$ its left and right subtree
else
  if $|\mathsf{Nh}(\mathcal{C})| > 1$ **then**
    Let $B \leftarrow$ a bag of $\mathcal{C}$ whose removal splits $\mathcal{C}$ to $\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2$ with $|\mathsf{Nh}(\overline{\mathcal{C}}_i) \cap \mathsf{Nh}(\mathcal{C})| \leqslant \frac{|\mathsf{Nh}(\mathcal{C})|}{2}$
    Assign $\mathcal{B} \leftarrow B$
    Assign $\mathcal{T}_1 \leftarrow \mathsf{Rank}(\overline{\mathcal{C}}_1, (\ell+1) \mod \lambda)$
    Assign $\mathcal{T}_2 \leftarrow \mathsf{Rank}(\overline{\mathcal{C}}_2, (\ell+1) \mod \lambda)$
    Make $\mathcal{B}$ the root of $\mathcal{T}$ and $\mathcal{T}_1$ and $\mathcal{T}_2$ its left and right subtree
  else
    Assign $\mathcal{T} \leftarrow \mathsf{Rank}(\mathcal{C}, (\ell-1) \mod \lambda)$
  end
end
return $\mathcal{T}$

ALGORITHM 2: ConcurTree

Input: Tree-decompositions $T_i = (V_{T_i}, E_{T_i})_{1 \leqslant i \leqslant k}$ with root bags $(B_i)_{1 \leqslant i \leqslant k}$.
Output: A tree decomposition T of the concurrent graph

Assign $B \leftarrow \varnothing$
Assign $T \leftarrow$ a tree with the single bag B as its root
for $i \in [k]$ **do**
  Assign $B \leftarrow B \cup \left(\prod_{1 \leqslant j < i} V_{T_j}(B_j) \times B_i \times \prod_{i < j \leqslant k} V_{T_j}(B_j)\right)$
end
if *none of the B_i's is a leaf in its respective T_i* **then**
  for *every sequence of bags $B'_1, \dots, B'_k$ such that each B'_i is a child of B_i in T_i* **do**
    Assign $T'_i \leftarrow \mathsf{ConcurTree}(T_1(B'_1), \dots, T_k(B'_k))$
    Add T'_i to T_i, setting the root of T'_i as a new child of B
  end
end
return T

ALGORITHM 3: ConcurPreprocess Item 1

Input: Graphs $(G_i = V_i, E_i)_{1 \leqslant i \leqslant k}$, a concurrent graph $G(V, E)$ of G_i's and a weight function $\mathsf{wt} : E \to \Sigma$

/* Construct the partial expansion $\overline{G}$ of G */

Assign $\overline{V} \leftarrow V$
Assign $\overline{E} \leftarrow E$
Create a map $\overline{\mathsf{wt}} : \overline{E} \to \Sigma$
Assign $\overline{\mathsf{wt}} \leftarrow \mathsf{wt}$
foreach $u' \in \prod_i (V_i \cup \{\perp\})$ **do**
  Let $u \in V$ such that $u \sqsubset u'$
  Assign $\overline{V} \leftarrow \overline{V} \cup \{\overline{u}^1, \overline{u}^2\}$
  Assign $\overline{E} \leftarrow \overline{E} \cup \{(\overline{u}^1, u), (u, \overline{u}^2)\}$
  Set $\overline{\mathsf{wt}}(\overline{u}^1, u) \leftarrow \overline{\mathbf{1}}$
  Set $\overline{\mathsf{wt}}(u, \overline{u}^2) \leftarrow \overline{\mathbf{1}}$
end
return $\overline{G} = (\overline{V}, \overline{E})$ and $\overline{\mathsf{wt}}$

ALGORITHM 4: ConcurPreprocess Item 2

Input: A tree-decomposition $T = \mathrm{Tree}(G) = (V_T, E_T)$ and the partial expansion $\overline{G} = (\overline{V}, \overline{E})$

/* Construct the tree-decomposition $\overline{T}$ of $\overline{G}$ */

Assign $V_{\overline{T}} \leftarrow \varnothing$
foreach *bag* $B \in V_T$ **do**
  Assign $\overline{B} \leftarrow B$ **foreach** $u \in B$ **do**
    foreach $\overline{u} \in \overline{V}$ *such that* $u \sqsubset \overline{u}$ **do**
      Assign $\overline{B} \leftarrow \overline{B} \cup \{\overline{u}^1, \overline{u}^2\}$
    end
  end
  Assign $V_{\overline{T}} \leftarrow V_{\overline{T}} \cup \{\overline{B}\}$
end
return $\overline{T} = (V_{\overline{T}}, E_{\overline{T}})$

ALGORITHM 5: ConcurPreprocess Item 3

Input: The partial expansion tree-decomposition $\overline{T} = (\overline{V}_{\overline{T}}, \overline{E}_{\overline{T}})$, and weight function $\overline{\mathsf{wt}}$

`/* Local distance computation */`

foreach *partial node* $\overline{u}$ **do**
    Create two maps $\textsc{Fwd}_{\overline{u}}, \textsc{Bwd}_{\overline{u}} : \overline{B}_{\overline{u}} \to \Sigma$
    for $\overline{v} \in \overline{B}_{\overline{u}}$ **do**
        Assign $\textsc{Fwd}_{\overline{u}}(\overline{v}) \leftarrow \overline{\mathsf{wt}}(\overline{u}, \overline{v})$
        Assign $\textsc{Bwd}_{\overline{u}}(\overline{v}) \leftarrow \overline{\mathsf{wt}}(\overline{v}, \overline{u})$
    end
end
foreach *bag* $\overline{B}$ *of* $\overline{T}$ *in bottom-up order* **do**
    Assign $d' \leftarrow$ the transitive closure of $\overline{G}[\overline{B}]$ wrt $wt_{\overline{B}}$
    foreach $\overline{u}, \overline{v} \in \overline{B}$ **do**
        if $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$ **then**
            Assign $\textsc{Bwd}_{\overline{u}}(\overline{v}) \leftarrow d'(\overline{v}, \overline{u})$
            Assign $\textsc{Fwd}_{\overline{u}}(\overline{v}) \leftarrow d'(\overline{u}, \overline{v})$
        end
    end
end
foreach *bag* $\overline{B}$ *of* $\overline{T}$ *in top-down order* **do**
    Assign $d' \leftarrow$ the transitive closure of $\overline{G}[\overline{B}]$ wrt $wt_{\overline{B}}$
    foreach $\overline{u}, \overline{v} \in \overline{B}$ **do**
        if $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$ **then**
            Assign $\textsc{Bwd}_{\overline{u}}(\overline{v}) \leftarrow d'(\overline{v}, \overline{u})$
            Assign $\textsc{Fwd}_{\overline{u}}(\overline{v}) \leftarrow d'(\overline{u}, \overline{v})$
        end
    end
end

ALGORITHM 6: ConcurPreprocess Item 4

Input: The partial expansion tree-decomposition $\overline{T} = (\overline{V}_{\overline{T}}, \overline{E}_{\overline{T}})$ and maps $\textsc{Fwd}_{\overline{u}}, \textsc{Bwd}_{\overline{u}} : \overline{B}_u \to \Sigma$ for every partial node $\overline{u}$

`/* Ancestor distance computation */`

foreach *node* $u \in V$ **do**
    Create two maps $\textsc{Fwd}^+_u, \textsc{Bwd}^+_u : \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u) \to \Sigma$
end
foreach *bag* $\overline{B}$ *of* $\overline{T}$ *in DFS order starting from the root* **do**
    Let $\overline{B}'$ be the parent of $\overline{B}$
    foreach *node* $u \in \overline{B} \cap V$ *such that* $\overline{B}$ *is the root of* u **do**
        foreach $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u)$ **do**
            Assign $\textsc{Fwd}^+_u(\overline{v}) \leftarrow \bigoplus_{x \in \overline{B} \cap \overline{B}'} \textsc{Fwd}_u(x) \otimes wt^+(x, \overline{v})$
            Assign $\textsc{Bwd}^+_u(\overline{v}) \leftarrow \bigoplus_{x \in \overline{B} \cap \overline{B}'} \textsc{Bwd}_u(x) \otimes wt^+(\overline{v}, x)$
        end
    end
end

ALGORITHM 7: ConcurQuery Single-source query

Input: A source node $u \in V$
Output: A map $A : V \to \Sigma$ that contains distances of vertices from u

Create a map $A : V \to \Sigma$
for $v \in V$ **do**
    Assign $A(v) \leftarrow \overline{\mathbf{0}}$
end
for *every bag* $\overline{B}$ *of* $\overline{T}$ *in BFS order starting from* $\overline{B}_u$ **do**
    for $x, v \in \overline{B} \cap V$ **do**
        if $\mathsf{Lv}(v) \leqslant \mathsf{Lv}(x)$ **then**
            Assign $A(v) \leftarrow A(v) \oplus A(x) \otimes \textsc{Fwd}_x(v)$
        end
    end
end
return A

ALGORITHM 8: ConcurQuery Pair query

Input: Two nodes $u, v \in V$
Output: The distance $d(u, v)$

Let $\overline{B} \leftarrow$ the LCA of $\overline{B}_u$ and $\overline{B}_v$ in $\overline{T}$
Assign $d \leftarrow \overline{\mathbf{0}}$
for $x \in \overline{B} \cap V$ **do**
    Assign $d \leftarrow d \oplus \textsc{Fwd}^+_u(x) \otimes \textsc{Bwd}^+_v(x)$
end
return d

ALGORITHM 9: ConcurQuery Partial pair query

Input: Two partial nodes $\overline{u}, \overline{v} \in \overline{V}$, at least one of which is strictly partial
Output: The distance $d(\overline{u}, \overline{v})$

if *both* $\overline{u}$ *and* $\overline{v}$ *are strictly partial* **then**
    return $\textsc{Fwd}_{\overline{u}^1}(\overline{v}^2)$
else if $\overline{u}$ *is strictly partial* **then**
    return $\textsc{Bwd}^+_v(\overline{u}^1)$
else
    return $\textsc{Fwd}^+_u(\overline{v}^2)$
end

References

Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *FOCS*. 434–443.

Amir Abboud, Virginia Vassilevska Williams, and Huacheng Yu. 2015. Matching Triangles and Basing Hardness on an Extremely Popular Conjecture. In *STOC*. 41–50.

Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. 1999. Communicating Hierarchical State Machines *(ICAL)*.

Stefan Arnborg and Andrzej Proskurowski. 1989. Linear time algorithms for NP-hard problems restricted to partial k-trees . *Discrete Appl Math* (1989).

Wayne A. Babich and Mehdi Jazayeri. 1978. The method of attributes for data flow analysis. *Acta Informatica* 10, 3 (1978).

R. Bellman. 1958. On a Routing Problem. *Quarterly of Applied Mathematics* (1958).

M.W Bern, E.L Lawler, and A.L Wong. 1987. Linear-time computation of optimal subgraphs of decomposable graphs. *J Algorithm* (1987).

Stephen M. et al. Blackburn. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*.

HansL. Bodlaender. 2005. Discovering Treewidth. In *SOFSEM 2005: Theory and Practice of Computer Science*. LNCS, Vol. 3381. Springer.

Hans L. Bodlaender. 1988. Dynamic programming on graphs with bounded treewidth. In *ICALP*. Springer.

Hans L. Bodlaender. 1993. A Tourist Guide through Treewidth. *Acta Cybern.* (1993).

Hans L. Bodlaender. 1996. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM J. Comput.* (1996).

Hans L. Bodlaender and Torben Hagerup. 1998. Parallel Algorithms with Optimal Speedup for Bounded Treewidth. *SIAM J. Comput.* (1998).

Ahmed Bouajjani, Javier Esparza, Stefan Schwoon, and Jan Strejček. 2005. Reachability Analysis of Multithreaded Software with Asynchronous Communication *(FSTTCS)*.

Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. 2003. A Generic Approach to the Static Analysis of Concurrent Programs with Procedures. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, Nicolas Markey, and Jiří Srba. 2008. Infinite Runs in Weighted Timed Automata with Energy Constraints. In *Formal Modeling and Analysis of Timed Systems*. Lecture Notes in Computer Science, Vol. 5215. Springer Berlin Heidelberg, 33–47.

Bernd Burgstaller, Johann Blieberger, and Bernhard Scholz. 2004. On the Tree Width of Ada Programs. In *Reliable Software Technologies - Ada-Europe 2004*.

Pavol Cerny, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. 2015. From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis. In *CAV*.

Pavol Cerny, Thomas A. Henzinger, and Arjun Radhakrishna. 2013. Quantitative Abstraction Refinement. In *POPL*. 115–128.

Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. 2010. Quantitative languages. *ACM Trans. Comput. Log.* 11, 4 (2010).

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In *POPL*. 733–747.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2015a. Faster Algorithms for Quantitative Verification in Constant Treewidth Graphs. In *CAV*.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Andreas Pavlogiannis, and Prateesh Goyal. 2015b. Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth. In *POPL*.

K. Chatterjee and J. Lacki. 2013. Faster Algorithms for Markov Decision Processes with Low Treewidth. In *CAV*.

Krishnendu Chatterjee, Andreas Pavlogiannis, and Yaron Velner. 2015c. Quantitative Interprocedural Analysis. In *POPL*.

Shiva Chaudhuri and Christos D. Zaroliagis. 1995. Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms. *Algorithmica* (1995).

Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. 2008. Dataflow Analysis for Concurrent Programs Using Datarace Detection. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Brouno Courcelle. 1990. Graph Rewriting: An Algebraic and Logic Approach. In *Handbook of Theoretical Computer Science (Vol. B)*. MIT Press, Cambridge, MA, USA.

Arnab De, Deepak D'Souza, and Rupesh Nasre. 2011. Dataflow Analysis for Datarace-free Programs. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software (ESOP'11/ETAPS'11)*. Springer-Verlag, 196–215.

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1995. Demand-driven Computation of Interprocedural Data Flow *(POPL)*.

Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. 2004. Flow Analysis for Verifying Properties of Concurrent Software Systems. *ACM Trans. Softw. Eng. Methodol.* 13, 4 (2004), 359–430.

M. Elberfeld, A. Jakoby, and T. Tantau. 2010. Logspace Versions of the Theorems of Bodlaender and Courcelle. In *FOCS*.

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2013. Inductive Data Flow Graphs *(POPL)*.

Azadeh Farzan and P. Madhusudan. 2007. Causal Dataflow Analysis for Concurrent Programs. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* (1962).

Lester R. Ford. 1956. Network Flow Theory. Report P-923, The Rand Corporation. (1956).

Dirk Grunwald and Harini Srinivasan. 1993. Data Flow Equations for Explicitly Parallel Programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

J. Gustedt, O. Maehle, and J. Telle. 2002. The Treewidth of Java Programs. In *Algorithm Engineering and Experiments*. Springer.

Rudolf Halin. 1976. S-functions for graphs. *Journal of Geometry* (1976).

David Harel, Orna Kupferman, and MosheY. Vardi. 1997. On the complexity of verifying concurrent transition systems. In *CONCUR*.

D. Harel and R. Tarjan. 1984. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.* (1984).

Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *STOC*. 21–30.

Susan Horwitz, Thomas Reps, and Mooly Sagiv. 1995. Demand Interprocedural Dataflow Analysis. *SIGSOFT Softw. Eng. Notes* (1995).

Donald B. Johnson. 1977. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM* (1977).

Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta. 2013. Static analysis for concurrent programs with applications to data race detection. *International Journal on Software Tools for Technology Transfer* 15, 4 (2013), 321–336.

Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. Static Data Race Detection for Concurrent Programs with Asynchronous Calls. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. 13–22.

Richard M. Karp. 1978. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics* (1978).

Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

S. C. Kleene. 1956. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies* (1956).

Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. 1996. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. *ACM Trans. Program. Lang. Syst.* (1996).

Philipp Klaus Krause. 2013. Optimal Register Allocation in Polynomial Time. In *Compiler Construction*, Ranjit Jhala and Koen De Bosschere (Eds.).

Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2008. Context-bounded Analysis of Concurrent Queue Systems *(TACAS)*.

Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. 2012. A Solver for Reachability Modulo Theories *(CAV)*.

Akash Lal and Thomas Reps. 2009. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. *Form. Methods Syst. Des.* (2009).

Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas Reps. 2008. Interprocedural Analysis of Concurrent Programs Under a Context Bound *(TACAS)*.

Eugene Lawler. 1976. *Combinatorial Optimization: Networks and Matroids*. Saunders College Publishing.

Daniel J. Lehmann. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science* (1977).

P. Madhusudan and Gennaro Parlato. 2011. The Tree Width of Auxiliary Storage *(POPL)*.

Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. 2010. Practical Extensions to the IFDS Algorithm *(CC)*.

Jan Obdrzálek. 2003. Fast Mu-Calculus Model Checking when Tree-Width Is Bounded. In *CAV*.

Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software *(TACAS)*.

Bruce A. Reed. 1992. Finding Approximate Separators and Computing Tree Width Quickly. In *STOC*.

ThomasW. Reps. 1995. Demand Interprocedural Program Analysis Using Logic Databases. In *Applications of Logic Databases*. Vol. 296.

Thomas Reps. 1997. Program Analysis via Graph Reachability *(ILPS)*.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*.

Neil Robertson and P.D Seymour. 1984. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B* (1984).

Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* (1996).

Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon. 2008. Symbolic Context-Bounded Analysis of Multithreaded Java Programs *(SPIN)*.

Mikkel Thorup. 1998. All Structured Programs Have Small Tree Width and Good Register Allocation. *Information and Computation* (1998).

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *CASCON '99*. IBM Press.

Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. 2006. *Computing treewidth with LibTW*. Technical Report. University of Utrecht.

A. Viterbi. 1967. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Trans. Inf. Theor.* 13, 2 (1967), 260–269.

Stephen Warshall. 1962. A Theorem on Boolean Matrices. *J. ACM* (1962).

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.* 7, 3 (2008).

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven Context-sensitive Alias Analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*.

Xin Yuan, Rajiv Gupta, and Rami Melhem. 1997. Demand-Driven Data Flow Analysis for Communication Optimization. *Parallel Processing Letters* 07, 04 (1997), 359–370.

Frank Kenneth Zadeck. 1984. Incremental Data Flow Analysis in a Structured Program Editor *(SIGPLAN)*.