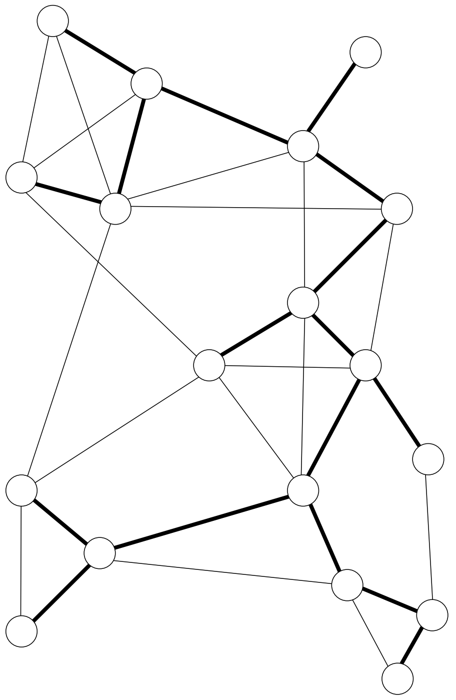
An optimal minimum spanning tree algorithm

Master's Thesis

Claus Andersen, 20030583

November 28, 2008



Department of Computer Science
Aarhus University

Supervisor:
Gerth Stølting Brodal
(gerth@cs.au.dk)

Abstract

This thesis describes the optimal minimum spanning tree algorithm given by Pettie and Ramachandran (in Journal of the ACM, 2002). The algorithm presented finds a minimum spanning tree of a graph with n vertices and m edges deterministically in time $O\left(\mathcal{T}^*\left(m,n\right)\right)$, where $\mathcal{T}^*$ is the minimum number of edge-weight comparisons needed to determine the solution of the problem. The function $\mathcal{T}^*$ is the decision tree complexity of the minimum spanning tree problem, and thus the algorithm shows that the algorithmic complexity of the problem is equal to its decision tree complexity.

Even though the algorithm runs in optimal time, the exact function describing the running time is not known at present, and is thus still an open problem. A trivial lower bound is $\mathcal{T}^*\left(m,n\right) = \Omega\left(m\right)$. A deterministic upper bound is $\mathcal{T}^*\left(m,n\right) = O\left(m \cdot \alpha(m,n)\right)$, due to Chazelle (in Journal of the ACM, 2000). Here, α is the very slow growing inverse of the Ackermann function.

The optimal algorithm uses hardwired optimal decision trees for very small graphs compared to the input graph, but for input graph instances with practical size, the decision trees are not required to obtain the optimal running time. Because the optimal algorithm is relatively advanced, the hidden Big-Oh constant of its running time is relatively large. This thesis implements the optimal algorithm, except for the decision trees, and compares its running time with minimum spanning tree algorithms with theoretically higher complexity. The result of the experiments show that some of these algorithms are significantly faster than the optimal algorithm for practical input graph instances. The overall experiment winner is the algorithm originally discovered by Jarník, and later by Prim and Dijkstra, which in all experiments is significantly faster than the optimal algorithm. However, the experiments show that for a special graph family, the optimal algorithm is faster than the earliest minimum spanning tree algorithm known, namely Borůvka's algorithm.

Consequently, for practical purposes, the algorithm described provides no benefit to the solution of the minimum spanning tree problem.

Contents

Acknowledgements

I thank my supervisor Gerth Brodal for his help and support during the creation of this thesis, and for his reviews of this report.

Furthermore I thank Grzegorz Nowak and Henrik Kirk for proofreading.

This report was written in LaTeX and the figures were created with Ipe and Dia.

Claus Andersen,
November 28, 2008.

Part I

Introduction

1 History of the minimum spanning tree problem

In 1926 the Czech mathematician Otakar Borůvka described a solution to "a certain minimal problem" [Bor26]. He used the solution to find the optimal way to lay out an electrical network. The concepts of graphs and minimum spanning trees were not known at the time Borůvka described the solution. His solution to the problem is what we today know as a *minimum spanning tree* algorithm and is the earliest known minimum spanning tree algorithm. Today this algorithm is known as Borůvka's algorithm. His algorithm is studied in Chapter 8. Throughout this thesis, we will refer to a minimum spanning tree as *"MST"*.

In 1930 another MST algorithm was discovered by the Czech mathematician Jarník [Jar30]. This algorithm was independently discovered by the American mathematician and computer scientist Prim in 1957, and later rediscovered by the Dutch computer scientist Dijkstra in 1959. The algorithm is therefore sometimes called Prim's Algorithm, Jarník's algorithm, the Prim-Jarník algorithm or briefly the DJP algorithm. Throughout this thesis, we will refer to this algorithm as the *DJP* algorithm. The DJP algorithm is studied in Chapter 9.

Many modern MST algorithms utilises the ideas from these two algorithms. In particular the optimal MST algorithm studied in this thesis heavily utilise the ideas from both algorithms. Since the invention of these algorithms, the MST problem has been heavily studied, but no one has found an exact lower bound on the time complexity for the problem. The MST algorithm studied in this thesis was published by Pettie and Ramachandran [PR02] in 2002. We can prove that the algorithm runs in optimal time, but we can not give an exact lower bound. This algorithm is studied in Part IV.

A precise definition of a minimum spanning tree will be given in Chapter 4. An overview of running times of important MST algorithms will be given in Section 4.3. Before stating the MST problem formally, we will give some information of the mathematical notation used in this thesis, as well as a graph theoretic introduction.

2 Mathematical notation and initial notes

Otherwise stated explicitly, $\log n = \log_2 n$, that is the logarithm with base 2.

For integers $i \geq 0$ the term $\log^{(i)}(n)$ is defined inductively by $\log^{(0)}(n) = n$ and $\log^{(i+1)}(n) = \log\log^{(i)}(n)$. For example $\log^{(3)}(n) = \log\log\log n$.

The iterated logarithm $\log^*(n)$ is defined as $\min\left\{i \mid \log^{(i)}(n) \leq 1\right\}$. That is, the number of times the logarithm function must be applied before the result is ≤ 1. This function grows very slowly, and $\log^*(n)$ is less than six for all "practical" values of n.

The factorial of a positive integer n, denoted $n!$ is defined as $\prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$. Hence, it easy to verify that $n! \leq n^n$. It is inductively defined by $0! = 1$ and $n! = (n-1) \cdot n$.

For non-negative integers n and m, the Ackermann function $A(m, n)$ can be defined recursively as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{else .} \end{cases}$$

The interesting property of $A(m, n)$ is that its value grows very rapidly. Let $A'(n) = A(n, n)$. Since A' grows very rapidly, its inverse function A'^{-1} grows very slowly. This inverse Ackermann function is denoted α. The function $\alpha(n)$ is less than five for all "practical" values of n. A two-parameter variation of the inverse Ackermann function can be defined as $\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) \geq \log n\}$, but the important thing is that α grows very slowly.

For integers $m \geq 0$ and $n \geq 1$, the beta function $\beta(m, n)$ is defined as $\beta(m,n) = \min\left\{i \mid \log^{(i)}(n) \leq m/n\right\}$. That is, the number of times the logarithm function must be applied to n before the result is $\leq m/n$.

3 Graph theory

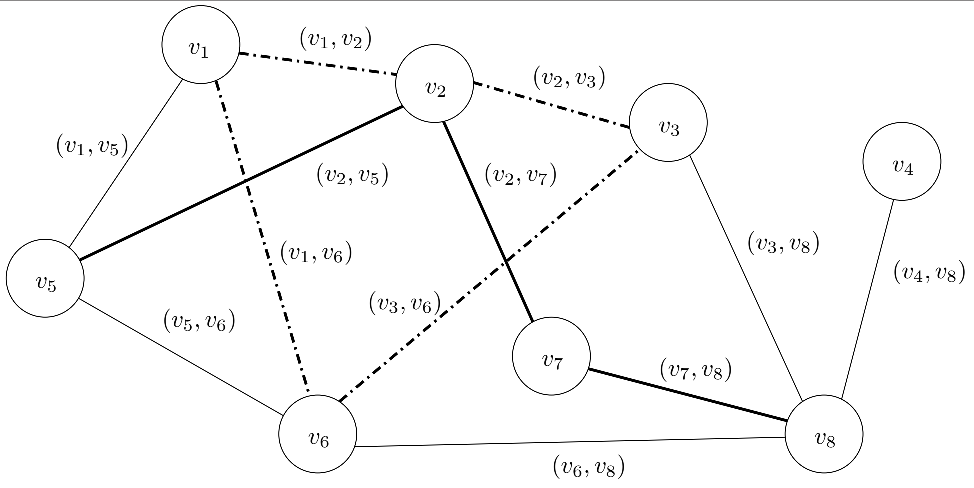


Figure 3.1: A simple connected graph G with $n = 8$ and $m = 12$, defined by $V = \{v_1, \ldots, v_8\}$ and $E = \{(v_1, v_2), (v_1, v_5), \ldots, (v_6, v_8), (v_7, v_8)\}$.
A sequence as $(v_5, (v_2, v_5), v_2, (v_2, v_7), v_7, (v_7, v_8), v_8)$ (bold edges) forms a simple path. A sequence as $(v_2, (v_2, v_3), v_3, (v_3, v_6), v_6, (v_1, v_6), v_1, (v_1, v_2), v_2)$ (dash-dotted edges) forms a simple cycle.

An undirected (multi) graph G is an abstract data type defined by an ordered pair $G = (V, E)$, where V is a set of *vertices* and E is a multiset of *edges*, which are unordered pairs of vertices. By definition $n = |V|$ and $m = |E|$, so we have vertices $V = \{v_1, v_2, \ldots, v_n\}$ and edges $E = \{e_1, e_2, \ldots, e_m\}$. The class of graphs with n vertices and m edges is denoted by $G(m, n)$. The vertex set of a specific graph G is denoted by $V(G)$, and the edge set is denoted by $E(G)$.

An edge e_i connects two vertices, say v_a and v_b, and is denoted (v_a, v_b). By definition of an undirected graph, an edge is an unordered pair, so $(v_a, v_b) = (v_b, v_a)$. A visual example of a graph is given in Figure 3.1. Each edge e_i is assigned a positive *weight* $w(e_i)$, that is the cost of “using” the edge. If $w(e_i) < w(e_j)$ for $i \neq j$, then e_i is said to be *lighter* than e_j. Similarly, e_j is said to be *heavier* than e_i. For simplicity, all example graphs in this thesis will have edge-weights corresponding to the Euclidean distance between endpoints. This is a common assumption in real-world examples, such as road networks and networks of power or data wires.

The *degree* of a vertex $deg(v)$ is the number of edges that connect to it. This is also referred to as the number of edges *incident* to vertex v.

A *path* in a graph is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex, such that each edge is incident to its predecessor and successor vertex. That is a sequence $(v_{\text{start}}, (v_{\text{start}}, v_a), v_a, (v_a, v_b), v_b, \ldots, v_{\text{end}})$. A *cycle* is a path with the same start and end vertex. That is $v_{\text{start}} = v_{\text{end}}$. A *simple path* is a path where each vertex is distinct. Similarly, a *simple cycle* is a cycle where each vertex is distinct, except for the start and end vertex. See Figure 3.1 for an example of a simple path and a simple cycle.

A graph is *connected* if there exists a path between any pair of vertices. That is, if it is possible to get from one vertex to any other vertex in the graph. A *self loop* is an edge that connects a vertex to itself, that is $e = (v_i, v_i)$. Such edges will be counted twice in $deg(v)$. Two distinct edges, say e and e', are *parallel* if their endpoints are the same. That is, if $e = (v_i, v_j) = e'$. A graph containing parallel edges is called a *multi graph*, because E is a multiset as allowed in the first definition of a graph. A *simple graph* is an undirected graph with no self-loops and no parallel edges. By definition of a simple graph, E is a "real" set as opposed to a multiset. The degree of a vertex in a simple graph is at most $n-1$, and the degree is the same as the number of neighbouring vertices. The graph in Figure 3.1 is simple and connected. If a graph is not connected, its maximal connected subgraphs are called *connected components*.

We will show a lower bound on the number of edges in a simple connected graph by induction. For the base case, consider a trivial simple connected graph G consisting of one vertex ($n = 1$), and thus no edges ($m = 0$). Hence the expression $m = n - 1 = 0$ holds. Then inductively consider a simple connected graph with $n - 1$ vertices where $m = (n-1) - 1 = n - 2$ holds. Adding a new vertex to G requires one edge incident to the new vertex for the graph to be connected, so $m = (n-2) + 1 = n - 1$. This proves that any simple connected graph with n vertices has $m \geq n - 1$ edges. It is easy to see that connecting a graph to an extra isolated vertex with exactly one edge does not induce a cycle in the simple connected graph. As the graph is connected and has no cycles, there exists exactly one simple path between each pair of vertices. Hence, adding one more edge to the graph will add an extra simple path between some pair(s) of vertices, and thus a cycle. Hence, a connected simple graph with $m = n - 1$ has no cycles. By definition a *tree* is a connected graph without cycles, equivalent to that $m = n - 1$. A *forest* is by definition the union of a set of one or more vertex disjoint trees.

Another special case of a simple connected graph is the *complete graph*. A complete graph is a graph which contains all possible edges, that is, each pair of distinct vertices (v_i, v_j) where $i \neq j$ are connected by an edge. Every vertex in a complete graph has degree $n-1$, that is one incident edge to every other vertex in the graph. The total number of edges is the number of distinct vertex pairs: $n\left(n-1\right)/2$ or equivalently $\left(n^2 - n\right)/2$ which is $O\left(n^2\right)$ edges. The complete graph with n vertices is denoted K_n.

Throughout this thesis[1], we will define the *density* of a graph as the ratio between the number of edges and number of vertices, that is by m/n. The number of edges m in a simple connected graph with n vertices is $n - 1 \leq m \leq \left(n^2 - n\right)/2$. That is m is $\Omega\left(n\right)$ and $O\left(n^2\right)$. A sparse graph is loosely defined as a graph where m/n is small. Similarly a dense graph is loosely defined as a graph where m/n is large. The actual definition of sparse and dense graphs will be clear in the contexts where the terms are used.

[1]Except for an experiment in Chapter 19.

Yet another special case of a simple connected graph is a *planar graph*. A planar graph is a graph which can be drawn in the plane, such that no edges intersect, except for endpoints at the vertices.

Theorem 3.1. *Simple connected planar graphs with $n \geq 3$ vertices has $m \leq 3n - 6$ edges.*

Proof. Let f be the number of faces, that is regions bounded by edges including the outer infinitely large region. If $f = 1$ (the outer face is the only face), then the graph is a tree and $m = n - 1 \leq 3n - 6$ for $n \geq 3$.
For $f > 1$, a connected graph with $n \geq 3$ vertices has $m \geq 3$ edges. Euler's formula [Som58] states that for a connected planar graph, $n - m + f = 2 \Rightarrow m = n + f - 2$. Each face is bounded by at least 3 edges (a triangular face), and every edge touches at most 2 faces. Hence, $f \leq 2m/3$, so

$$m \leq n + 2m/3 - 2 \quad \Rightarrow \quad m/3 \leq n - 2 \quad \Rightarrow \quad m \leq 3n - 6\ .$$

□

Consequently, for a planar graph, the density is bounded by a constant and m is $O(n)$.

Due to the lower bound of $n - 1$ edges for simple connected graphs, n is $O(m)$, which results in $O(n + m) = O(m)$. So the input size of any simple connected graph with m edges is $O(m)$. Consequently, the time needed to build a simple connected graph with m edges is $\Omega(m)$.

4 Minimum spanning trees

Definition Let G be a connected graph. A *spanning tree* of G is a tree containing all the vertices and a subset of the edges in G. In other words a tree that *spans* over all vertices in G. Hence, each spanning tree of a connected graph with n vertices has exactly $n - 1$ edges.

Definition Let T be a spanning tree of a connected weighted graph. The weight of T is defined by the sum of edge weights in T:

$$w(T) = \sum_{e \in E(T)} w(e) \ .$$

Definition Let G be a connected weighted graph. A *minimum spanning tree (MST)* of G is a spanning tree of G with minimum total edge weight.

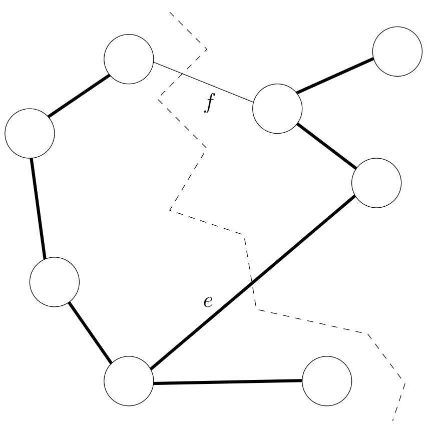
The bold edges in the graph on the front page symbolises a MST. It is clear from the definition of a MST, that a solution to the problem requires comparisons of edge-weights. A graph with equal edge weights may not have a unique minimum spanning tree, since the graph can have multiple spanning trees with equal minimum total edge weight. It will become clear in the subsequent section that if a graph has distinct edges weights, then the graph has a unique minimum spanning tree.

4.1 Properties

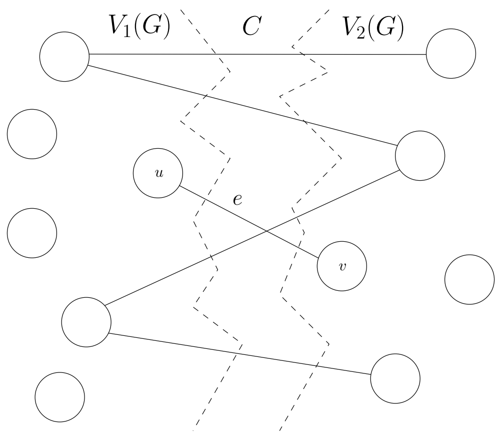
The cycle property

Theorem 4.1. *For each possible simple cycle in a connected weighted graph G with distinct edge weights, the heaviest edge in the cycle does* not *belong to a MST of G.*

Proof. See Figure 4.1a. Assume the contrary, namely that the heaviest edge e belongs to a MST. Deleting e from a MST would split the tree into two disjoint subtrees with the two endpoints of e in different subtrees. There exists some edge $f \neq e$ in the cycle with the two endpoints in different subtrees. As $w(f) < w(e)$, reconnecting the two subtrees with f will produce a spanning tree of smaller weight. Hence e does not belong to a MST. □



(a) The cycle property. The fat edges are the assumed MST. The dashed line illustrates the splitting of the tree.



(b) The cut property. Only edges in the cut C are shown.

Figure 4.1: The cycle property and the cut property.

The cut property

Definition Let $V(G)$ be the vertex set of a graph G such that $|V(G)| \geq 2$. Let $V_1(G)$ and $V_2(G)$ be nonempty disjoint partitions of the vertices in $V(G)$. That is $V_1(G) \neq \emptyset$, $V_2(G) \neq \emptyset$, $V_1(G) \cap V_2(G) = \emptyset$, and $V_1(G) \cup V_2(G) = V(G)$. Let C be the set of edges with one endpoint in both $V_1(G)$ and $V_2(V)$. Then C forms a *cut* in the graph G

Theorem 4.2. *For each possible cut C in a connected weighted graph G with distinct edge weights, the lightest edge in C belongs to a MST of G.*

Proof. See Figure 4.1b. Let u and v denote the endpoint vertices of the lightest edge e. If e is the only edge in C on any simple path between u and v, then the proof is obvious because all vertices in G must be connected in a MST. Otherwise, assume the contrary, namely that the lightest edge e does *not* belong to a MST. It is obvious that there is some edge, say f, in C on the path between u and v in the assumed MST, otherwise the two vertices would not be connected in the MST. As $w(e) < w(f)$, replacing f by e will produce a spanning tree of smaller weight. Hence e belongs to a MST. $\square$

Corollaries

Theorem 4.3. *Let G be a connected weighted graph with distinct edge weights, let T be a MST of G, and let e be an arbitrary edge in $E(G)$. If $e \in T$, then e is the lightest edge in some cut in G. If $e \notin T$, then e is the heaviest edge in some cycle in G.*

Proof. Due to the cycle property (Theorem 4.1) and the cut property (Theorem 4.2), we can prove this theorem by proving that any edge in $E(G)$ is either the lightest edge in some cut in G, or the heaviest edge in some cycle in G.

Assume an edge (u, v) exists where this does not apply. If there is a simple path $\mathcal{P}$ in G with endpoints u and v, where every edge is proven to be in the MST by the cut property, then $\mathcal{P} \cup \{(u, v)\}$ forms a cycle, and thus (u, v) must be the heaviest edge in this cycle. Otherwise, then the graph is not connected by MST edges, and thus there exists a cut without any edges proven to be in the MST, so we can add the lightest edge of this cut to the MST. Then repeat this procedure until (u, v) is either proven to be in the MST because it is the lightest edge in such cut, or proven to be the heaviest edge in a cycle. This will clearly happen at some point, because we repeatedly add a new edge to the MST, so we have a contradiction. □

Uniqueness

Theorem 4.4. *A connected weighted graph with distinct edge weights has a unique MST.*

Proof. From Theorem 4.3, it is clear that we can find a MST by either:

- Initialise the MST, say T, to be G. Then repeatedly remove the heaviest edge in a simple cycle in T until the derived graph is a spanning tree for G.
- Initialise the MST, say T, to be $V(G)$. Then for every cut in G, add the lightest edge to T if it is not already there.

As every edge has a unique weight, there is a unique edge with heaviest weight in any cycle. Similarly, there is a unique edge with lightest weight in any cut. Consequently, the graph has a unique MST. □

Throughout this thesis, we will assume that all graphs have distinct edge weights, otherwise stated explicitly. Thus, all graphs will have a unique MST.

4.2 Minimum spanning forests and graph assumptions

We have only defined the MST problem for connected graphs. The connected part makes sense, since by the definition of a tree, there exists no spanning trees for unconnected graphs.

A *spanning forest* is defined by the union of spanning trees for each connected component in a graph. Similarly a *minimum spanning forest (MSF)* is defined by the spanning forest with minimum total weight. Hence, the MSF problem is a generalisation of the MST problem, and can be solved by solving the MST problem for each connected component in the graph. It is easy to find the connected components in a graph in linear time using Depth-First-Search, and then solve the MST problem for each connected component. Detection of connected components will be described in Chapter 7.

As the MST problem is a specialisation of the MSF problem, we will refer to the two terms interchangeably when the difference is trivial.

It is easy to verify that any pair in a set of parallel edges forms a simple cycle in the graph. Due to the cycle property (Theorem 4.1), it is easy to verify that all edges, except the lightest, in a set of parallel edges not belong to the MST. It is also easy to verify that every self looping edge does not belong to the MST. Hence, for any input graph that is not simple, we can perform a precomputing step to remove these edges without any impact on the final MST result. Such precomputation step can be done in linear time by a call to `contract()`, which will be explained in Chapter 7.

Throughout this thesis, we will assume that all graphs are simple and connected. Otherwise, we can perform the trivial precomputation steps just described. As described in Chapter 3, these assumptions lead to some nice graph properties, such as $E(G)$ is a "real" set, and bounds on the number of edges.

4.3 Running times

In this section, we will present the running times of the MST algorithms covered in this thesis, as well as some other important MST algorithms.

The MST algorithm input graph G has n vertices and m edges. A function describing the exact complexity of the MST problem for general graphs is not known at present[1]. This thesis will show that an algorithm exists which runs in time order of the optimal number of edge-weight comparisons, even though the function for "optimal" is unknown.

It is easy to verify for all $n, m > 2$, that it is possible to build a simple graph where every edge is contained in at least one simple cycle. Hence, to solve the MST problem, all edge weights must be processed at least once, which implies a trivial lower bound of $\Omega(m)$. The best deterministic upper bound at present is $O(m \cdot \alpha(m,n))$ due to Chazelle [Cha00a]. Here, $\alpha(m,n)$ is the very slow growing inverse of the Ackermann function, which loosely speaking makes the MST complexity "almost linear" in the graph size. In 1995, Karger et al. [KKT95] presented a randomised (non-deterministic) MST algorithm with expected running time $O(m)$. This algorithm works under the assumption that we have access to a stream of truly random bits. However, this thesis will focus only on worst-case running times for deterministic MST algorithms.

The function $\mathcal{T}^*(m,n)$ denotes the minimum (optimal) number of edge weight comparisons needed to find the MST of any graph with n vertices and m edges. The function $\mathcal{T}^*(G)$ denotes the minimum (optimal) number of edge weight comparisons needed to find the MST of the specific graph G. Table 4.1 shows the running times of various deterministic MST algorithms (and optimal decision trees) for different graph densities. Notice, the sparse-dense bound is unique for each algorithm. The algorithms covered in

[1]This thesis is written in 2008.

this thesis are written with boldface. The algorithm called "Optimal" is the optimal MST algorithm by Pettie and Ramachandran [PR02], which we will analyse in this thesis.

Table 4.1 shows that MST algorithms exist with linear running times for certain densities. But for the intermediate class of "sparse" graphs, no MST algorithms with provable linear running time exist. However as stated above, this thesis will present an algorithm which runs in order of "optimal" time for all densities.

Algorithm	**Running time**		
	Planar	Sparse	Dense
Borůvka [Bor26], 1926	$O(m)$	$O(m \log n)$	$O(n^2)$
DJP [Jar30, FT87], 1930	$O(n \log n)$		$O(m)$
Kruskal, 1956	$O(m \log n)$		
Kruskal, edges sorted by weight	$O(m\alpha(n))$		
Yao, 1974	$O(m \log \log n)$		
Dense Case [FT87], 1987	$O(m \log^* (n))$		$O(m)$
Borůvka + DJP	$O(m)$	$O(m \log \log n)$	$O(n^2)$
Chazelle [Cha00a], 2000	$O(m\alpha(m,n))$		
Precomputed optimal decision tree for G	$O(\mathcal{T}^*(G))$		
Optimal [PR02], 2002	$O(\mathcal{T}^*(m,n))$		

Table 4.1: Worst case running times of MST algorithms, ordered by year.

5 Thesis goal and results

The goal of this thesis is to describe, analyse, implement, and test the optimal MST algorithm by Pettie and Ramachandran [PR02]. The running time of the algorithm will be compared to running times of other MST algorithms with theoretically higher complexities. The optimal MST algorithm is essentially a composition of known algorithmic methods. They are presented in Fredman and Tarjan [FT87], Chazelle [Cha00b], Mareš [Mar04], as well as the early articles by Borůvka [Bor26] and Jarník [Jar30]. The subsequent two parts of this thesis are dedicated to describe these methods. The optimal MST algorithm is finally described and analysed in Chapter 15 and its running time is compared to other MST algorithms in Chapter 19.

Results

We have tested the running time of the optimal MST algorithm and the other MST algorithms covered in this thesis[1] on various graphs. As the optimal MST algorithm is a composition of other algorithms, it has a high overhead in running time compared to the other MST algorithms we have tested. In other words, the hidden Big-Oh constant of its running time, $O\left(\mathcal{T}^*\left(m,n\right)\right)$, is high compared to the other algorithms. So the hidden Big-Oh constant may dominate the running time of the optimal MST algorithm for practical graphs instances. Notice, there is a practical upper bound on the size of the test graphs we are able to create, so this may also apply to our test graph instances. In brief, this assumption is confirmed by the experiments. With the exception of narrow intervals of densities, the DJP algorithm is generally fastest and the optimal algorithm is generally slowest. However for the densities $O\left(\log\log n\right)$ and $O\left(\log n\right)$ we have observed that the optimal algorithm is slightly faster than Borůvka's algorithm for large graphs. But even for these densities, both algorithms are significantly slower than DJP.

Consequently, for practical graph instances, the optimal MST algorithm given by Pettie and Ramachandran [PR02] provides no benefit to the solution of the MST problem.

[1]That is the boldface algorithms in Table 4.1, except precomputed decision trees.

Part II

Fundamental building blocks

6 Priority queues

A well known abstract data type is the priority queue. A priority queue stores pairs of (e, k), where e is an element and k is the key associated with e. The priority queues presented in this thesis are all based on heap ordered trees, in particular they are all minimum heaps. Briefly a heap is a rooted tree structure, and all nodes are stored in heap order, where the key of a node always is less than or equal to the keys of its child nodes. Consequently, the root node always stores a element with minimum key.

Except for the special soft heap, this thesis is not about priority queues, so we will briefly state the common operation interface of a priority queue:

insert(e, k) Inserts the element e into the queue with key k associated.

findMin Returns the element with minimum key in the queue.

deleteMin Deletes and returns the element with minimum key in the queue.

decreaseKey(e,Δ) Decreases the key of element e by Δ. To make this efficient, some queues also require a reference to the node storing e in the queue.

delete(e) Deletes the element e from the queue. Again, to make this efficient, some queues also require a reference to the node storing e in the queue.

meld(Q) Merges another queue Q into the queue.

In this thesis, there will be no need for deletion of arbitrary elements nor melding heaps. Hence, there will be no need for the **delete** and **meld** operations.

Briefly, the **deleteMin** operation of soft heaps [Cha00b] (fully presented in Chapter 12) may return a wrong element, because the heap corrupts (artificially raises) some of the element keys. The running time of soft heaps depends on an error rate $0 < \varepsilon < 1/2$, that the priority queue must be initialised with. The error rate also dictates an upper bound on the number of corrupted elements in the heap. The running times of the classic simple binary heap [Flo64, Wil64], the more advanced Fibonacci heap [FT87, AK07], and the soft heap[1] are presented in Table 6, where n is the number of elements in the heap.

[1]The decreaseKey operation is not available for the soft heap. See Chapter 12 for more information.

Operation	Binary heap [Flo64, Wil64]	Fibonacci heap [FT87]		Soft heap [Cha00b]
	Worst case	Worst case	Amortised	Amortised
insert	$O(\log n)$	$O(1)$	$O(1)$	$O(\log 1/\varepsilon)$
deleteMin	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
decreaseKey	$O(\log n)$	$O(n)$	$O(1)$	N/A

Table 6.1: Running times of binary heaps, Fibonacci heaps, and soft heaps.

7 Graph contraction and generic MST algorithms

It is well known that if T is a tree of detected MST edges in a graph G, then we can contract the connected component induced by T into a single vertex and maintain the invariant, that the MST of G is the same as the MST edges of the contracted graph plus T. Contraction of a connected component, such as a tree, means to replace all vertices in the connected component by a single vertex. The edges with exactly one endpoint in the connected component are rearranged such that this one endpoint is changed to the new vertex. The edges with both endpoints in the connected component can either be removed, or rearranged as self-loops for the new vertex. In a MST context, it is preferred to remove these edges, because each of them forms a simple cycle consisting of the edge only, and consequently is not in the MST due to the cycle property (Theorem 4.1). To keep track of the relation between edges in the original and the contracted graph, each edge in the contracted graph must have a reference to the corresponding edge (and its endpoints) in the original graph.

The MST algorithms described in this thesis, including the optimal algorithm, uses graph contraction, so this section is dedicated to a description of contraction methods. First, a definition of notation.

Definition Let C be a subgraph of G. The graph derived from G by contracting each of the connected components in C is denoted $G \setminus C$.

7.1 Generic MST algorithm

Many MST algorithms operate by iteratively detecting a MST edge (for instance using the cut property in Theorem 4.2), connecting two distinct clusters until all MST edges are found. Here, a cluster is a set of vertices and all clusters are disjoint. Initially we define a cluster for each vertex. When a new MST edge is detected, it is added to the set of MST edges, and the clusters of the two endpoints are merged into one cluster, as they together form a connected component of MST edges. Recall that we assume that the input graph G is connected, and thus the MST of G has exactly $|V(G)| - 1$ edges. A generic algorithm showing this scheme is given in Algorithm 7.1. Additionally some MST algorithms (For example Borůvka's algorithm in Chapter 8 and the "Dense Case" algorithm in Chapter 10) operate in steps or passes, where each pass loosely speaking processes every cluster once.

In Algorithm 7.1 this corresponds to an extra inner loop, that runs until all clusters are processed. In this way, one execution of the outer loop corresponds to a single pass.

Algorithm 7.1: Generic MST algorithm

```
foreach vertex $v$ of $G$ do Define cluster $C(v) \leftarrow \{v\}$
$T \leftarrow$ all vertices of $G$
while $|E(T)| < |V(G)| - 1$ do
    Find some MST edge $(a, b)$, where $C(a) \neq C(b)$
    Add $(a, b)$ to $T$
    Merge clusters $C(a)$ and $C(b)$ into one cluster
return $T$
```

The condition $C(a) \neq C(b)$ is crucial due to the MST cycle property (Theorem 4.1). One drawback of this method, is that the graph has the same size in all passes, even though all edges (a, b) for which $C(a) = C(b)$ can not be part of the MST due to the cycle property. Therefore we wish to remove these edges from consideration. One way to do this, is to contract every component induced by MST trees, and hence clusters, in the end of each pass. This way, after the contraction each new vertex corresponds to a connected component before the contraction. Consequently, before each pass we can "reset" the clusters, such that we again define a cluster for each new vertex.

7.2 Simple contraction procedure

With contraction in the end of each pass, Borůvka's algorithm and the "Dense Case" algorithm have no need to explicitly define and merge distinct clusters. Additionally the algorithms can restrict them self to only mark a MST edge without adding it to T once it is detected. This will also make it easier to check if a particular edge is within the MST. Therefore we will describe contraction for graphs with no clusters associated. The contraction described first in this chapter can be achieved by:

- Detect components connected by MST-marked edges. Assign to each vertex the component it belongs to. This is a delayed cluster detection.
- Build a new contracted graph G' with: 1) One vertex for each component. 2) One edge for each edge connecting different components[1]. Each new edge must have a reference to the edge in the original graph associated.
- Add the MST-marked edges to T. This is possible to do simultaneous with one of the other two steps.

So in the end of each pass we contract the graph into a smaller graph, where each component connected by MST-marked edges (equivalent to a cluster in the original generic algorithm) is contracted to a single vertex. Furthermore some edges guaranteed not to be in the MST are removed, namely all edges with both endpoints in the same component.

[1]Will be improved later.

Depth-First Search and detection of connected components

A graph *traversal* is a method for examining all vertices and edges in a graph. A *Depth-First Search (DFS)* of a graph is a standard traversal method. Here, we will give a simplified version of DFS. Initially all vertices are marked as “unexplored”. Let the DFS algorithm be a recursive function taking a “current vertex”, say v, as parameter. A recursion starts by marking v as explored. Then for each incident edge to v, say e, it checks if the opposite endpoint of e, say u, is unexplored. If so, it calls DFS recursively on u. When all incident edges are processed, it returns. It is clear that all vertices in a connected component will be visited by this traversal.

The DFS traversal can easily be extended to detect connected components in a graph, and associate a unique component number to each vertex: Initially, mark all vertices as unexplored and set a global component number variable $i = 0$. Then repeatedly find an unexplored vertex in the graph, say v, call extended DFS on v, and increase i by one. For each extended DFS recursion, the algorithm associates the component number i with the current vertex. An unexplored vertex can efficiently be found be maintaining a reference to the first unexplored vertex in the list of vertices.

Contraction algorithm

Besides graph contraction, we can easily generalise the generic algorithm to a MSF (as opposed to MST) algorithm by changing the loop condition. When the MST of a connected component is found, the component is contracted to a single vertex. Hence, the graph has an empty edge set when the MST of each connected component is found. So the new loop condition is that the current graph G has a nonempty edge set.

Let `assignComponentNumbers(`G`)` be a modified version of the extended DFS graph traversal described above, that only processes MSF marked edges in the graph G. Let $C(v)$ denote the component number assigned to vertex v. That is, $C(v)$ is the MSF connected component that v belongs to.

The contraction step is formally defined in Algorithm 7.2. The modified generic MSF algorithm with passes is defined in Algorithm 7.3.

Algorithm 7.2: `contract(`G`,`T`)` - Simple contraction algorithm

```
assignComponentNumbers ($G$)
$G' \leftarrow$ new graph with one vertex $v_i$ for each connected component in $G$
forall edges $(a,b)$ in $E(G)$ do
    if $(a,b)$ is marked as a MSF edge then
        Add the original edge of $(a,b)$ to $T$
    else if $C(a) \neq C(b)$ then
        Create a new edge $e \leftarrow (v_{C(a)}, v_{C(b)})$ with the weight $w((a,b))$.
        Add $e$ to $G'$ with a reference to the original edge of $(a,b)$.
return ($G'$,$T$)
```

Algorithm 7.3: Generic MSF algorithm with contraction

```
$T \leftarrow$ all vertices of $G$
while $|E(G)| > 0$ do
    while pass not done do
        Find some MSF edge $(a, b)$
        Mark $(a, b)$ as a MSF edge
    $(G, T) \leftarrow$ contract($G, T$)
return $T$
```

Running time

The first phase of `contract(G,T)` is a modified DFS of G, which takes time $O(m)$. The second phase builds a new graph of maximum n vertices. The time is $O(n)$. The third phase visits all edges and do constant work in each iteration, so the time is $O(m)$. Thus the total running time of `contract` is $O(m)$ where m is the number of edges in the input graph.

7.3 Better edge weeding

The graph G' returned from the simple `contract` function does not have any self loops, since edges with endpoints in the same component are not added to G'. But G' can clearly have parallel edges, that is G' can be a multi graph. Consider a multiset of parallel edges between two particular vertices. Each pair of edges forms a cycle in the contracted graph. Consequently, due to the cycle property (Theorem 4.1), all edges in the multiset, except the lightest, is not a part of the MST. Therefore we can remove all these edges to reduce the number of input edges to the next pass even further. Given that one pass in the generic algorithm takes $O(m)$ time, this process must take $O(m)$ time to avoid breaking the total worst case running time.

One solution to achieve this improvement is given by Mareš [Mar04]. We will change the loop of `contract(G)` in Algorithm 7.2 to:

1. Let $n' = |V(G')|$. That is the number of connected components found. Initialise n' buckets, say $B = (b_1, \ldots, b_{n'})$. Then for each edge in $E(G)$, let i be the maximum endpoint component number, and insert the edge into bucket b_i. Like in the simple version, add MST marked edges to T and throw away other edges with endpoints in the same component in this process. Time: $O(|E(G)| + |V(G')|) = O(|E(G)|)$.

2. Initialise n' new buckets of linked lists, say $B' = (b'_1, \ldots, b'_{n'})$. Then for increasing i, take each edge in bucket b_i: Let j be the minimum endpoint component number of the edge, and put the edge into bucket b'_j. For each bucket in B', this process will bring together the edges with the same maximum component number. That is, parallel edges is brought together. In total that is radix sorting by endpoint component numbers of edges with endpoints in distinct components. Time: $O(|E(G)| + |V(G')|) = O(|E(G)|)$.

3. For each bucket of B' (edges with the same minimum component number): Remove unnecessary parallel edges. That is, remove all edges except the lightest to each neighbour component. Time: $O\left(|E(G)| + |V(G')|\right) = O\left(|E(G)|\right)$.

4. Build a new graph G' with: 1) One vertex for each connected component. 2) One edge for each remaining edge in the buckets of B'. Each new edge must have a reference to the edge in the original graph.
Time: $O\left(|E(G')| + |V(G')|\right) = O\left(|E(G')|\right)$.

Since $|E(G')| \leq |E(G)|$, this process takes time $O\left(|E(G)|\right) = O\left(m\right)$ which was the goal. For general graphs this process does not remove more edges than the first simple contraction algorithm in the worst case.

Contraction implementation

In this thesis, contraction is implemented with parallel edge weeding. So otherwise stated explicitly, every call to `contract` in the succeeding sections of this thesis refers to this improved edge weeding algorithm. Thus the graph returned from this call is guaranteed to have no parallel or self-looping edges, that is a simple graph. Additionally, if the input graph is connected, then the output graph is a simple connected graph.

At a point the optimal algorithm marks some edges as MSF edges, and other edges as *removed* before a call to `contract`. It is an obvious invariant that an edge can not both be marked as removed and as an MST edge. It is easy to extend the contraction algorithm to also remove edges marked as removed: It requires an extra conditional construction in the first edge iteration, throwing away edges marked as removed.

8 Borůvka's algorithm

Borůvka's algorithm proceeds in a sequence of "*Borůvka steps*" until all MST edges are found. In each step the algorithm finds the lightest edge incident to each vertex, and adds it to the set of MST edges. When a step is done, it contracts each component induced by MST edges into a single vertex. Hence, it easy to see that the algorithm also can find the MSF of an unconnected graph. Borůvka's algorithm is described in Algorithm 8.1.

Algorithm 8.1: Borůvka's algorithm

```
$T \leftarrow$ trivial forest with all vertices of $G$
while $|E(G)| > 0$ do
    /* Borůvka step */
    forall vertices $v \in G$ do
        $e \leftarrow$ the lightest edge incident to $v$
        Mark $e$ as an MST edge
    $(G, T) \leftarrow$ contract$(G, T)$
return $T$
```

For a graph with distinct edge weights, there is a unique MST, and the proof of Borůvka's algorithm is trivial due to the cut property (Theorem 4.2): The lightest edge connecting a vertex to the rest of the graph is chosen to be in the MST. If the graph has edges with equal weights, there is a risk that the algorithm will mark an edge which will cause a cycle of "MST edges". A simple way to fix this problem is to "be on the safe side" and prefer marked minimum weight edges over non-marked edges, if there exists multiple edges with minimum weight. This is equivalent to avoid marking an edge if there exists multiple edges with minimum weight incident to the vertex, and some of them are already marked.

8.1 Running time

Let n' and m' be the number of vertices and edges, respectively, in the graph in the beginning of a Borůvka step. Notice that $n' \leq n$ and $m' \leq m$. Each step visits each edge two times, once for each endpoint. The algorithm utilises `contract` once in each step. Thus the running time for a step is $O(m')$ which is $O(m)$.

In the worst case only $n'/2$ edges are marked as MST edges in a step. Therefore the number of vertices (and edges) are reduced by at least $n'/2$ in each call to `contract`. Thus the total running time for general graphs, ignoring that some edges are removed between the steps, is $O(m \log n)$. Because the number of vertices are reduced by at least $n'/2$ in each call to `contract`, the number of vertices in the beginning of step $i \geq 0$ is at most $n/2^i$ and therefore the number of edges is at most the number of edges in the complete graph of $n/2^i$ vertices, namely $\leq \left(n/2^i\right)^2$. The sum $\sum_{i=0}^{\infty} \left(n/2^i\right)^2$ is dominated by the n^2 term. Thus the running time for general graphs is $O\left(n^2\right)$. Consequently the total running time for general graphs using Borůvka with contraction is $O\left(\min\left\{m \log n, n^2\right\}\right)$. So if m is $O\left(n^2/\log n\right)$, corresponding to the density m/n is $O\left(n/\log n\right)$ (sparse), the running time is $O\left(m \log n\right)$. If m is $\Omega\left(n^2/\log n\right)$, corresponding to the density m/n is $\Omega\left(n/\log n\right)$ (dense), the running time is $O\left(n^2\right)$.

With reference to Theorem 3.1, simple planar graphs has the property that $m \leq 3n - 6$, which implies that m is $O\left(n\right)$. The graph after a single Borůvka step is also a simple planer graph without parallel edges, due to the properties of `contract`. The number of edges m_i in the beginning of step i is therefore $m_i \leq 3n/2^i - 6$. This gives a total running time order of

$$\sum_i m_i \leq \sum_i 3\frac{n}{2^i} - 6 \leq 3n \sum_i \frac{1}{2^i} \leq 6n \ .$$

This shows that the total running time for Borůvka's algorithm for planer graphs is linear in number of vertices and thereby edges in the graph. This was not possible with the first simple version of `contract` (Algorithm 7.2), since it does not remove parallel edges. That is, it does not preserve planarity. From this example, it is easy to see that Borůvka's algorithm runs in linear time for very sparse graphs where m is $O\left(n\right)$.

9 The Dijkstra-Jarník-Prim (DJP) algorithm

The DJP algorithm uses the greedy method to grow a tree of MST edges. It begins with a trivial tree T, with one arbitrary "root" vertex. Then, in each step, it finds the lightest edge $e = (u, v)$ where $u \in T$ and $v \notin T$. This edge connects a vertex u in T to the neighbouring vertex v outside T. The tree T is then augmented with the vertex v and the edge e. The process continues until all vertices of G are in T. The resulting tree T is a MST of G. It is easy to prove that this method finds a MST due to the cut property (Theorem 4.2): The algorithm always chooses the lightest edge connecting two disjoint sets of vertices in G, namely vertices inside and outside the tree T, respectively.

9.1 Implementation

In the DJP algorithm each vertex v has three extra info fields associated. The first is a state, explored or unexplored. The second is the best current edge for connecting T to v. The third is the weight of the best current edge for connecting T to v. This field is called the best current distance from T to v. The state approach is an easy way to tell for a vertex v, if $v \in T$ (explored) or $v \notin T$ (unexplored).

The algorithm uses a priority queue of unexplored vertices, with their distance to T as keys. Hence the minimum element in the queue, will always be the vertex with the lightest edge connecting T to a neighbouring vertex outside T. This queue is initially empty. All vertices are initialised to the unexplored state and with distance ∞ to T. A distance of ∞ intuitively means that no edges incident to v have been explored yet, and hence v has no best current edge yet. An arbitrary vertex is chosen to be the root vertex, v_0, its distance is set to 0 and it is inserted into the heap.

In each step, the vertex v with minimum distance to T is deleted from the heap and marked as explored. Unless v is the root vertex, the best current edge of v is added to the set of MST edges. When a new vertex v is explored, the algorithm visits all incident edges $e = (v, u)$. If u is explored, it is already inside T and nothing is done. If the current best distance to u is ∞, then e is the first edge connecting u to T. Therefore u is inserted with e as the best current edge. Otherwise, if the current best distance to u is greater than $w(e)$, this edge improves the connection from T to u, and hence e is set as the best current edge from u. Thus the distance of u will also be decreased to $w(e)$.

It is easy to verify that if a vertex is inside T, then it will never be inserted into the queue again. Thus we can avoid an explicit state variable and instead set the distance to $-\infty$ to indicate the vertex is inside T.

See Algorithm 9.1 for a formal description of the DJP algorithm. It is easy to see at any time that T is the set of MST-marked edges.

Algorithm 9.1: DJP MST algorithm

```
Initialise priority queue
forall vertices $v \in V(G)$ do key($v$) $\leftarrow \infty$
$v_0 \leftarrow$ any vertex in $V(G)$
key($v_0$) $\leftarrow 0$
insert($v_0$)
while heap is not empty do
    $v \leftarrow$ deleteMin
    key($v$) $\leftarrow -\infty$ /* Do not connect to this vertex */
    if $v \neq v_0$ then mark e($v$) as an MST edge
    forall edges incident to $v$: $(u, v)$ do
        if key($u$) $> -\infty$ then /* $u$ is outside the tree */
            if $w((u,v)) <$ key($u$) then
                e($u$) $\leftarrow (u, v)$
            if key($u$) $= \infty$ then
                key($u$) $\leftarrow w((u,v))$
                insert($u$)
            else if $w((u,v)) <$ key($u$) then
                decreaseKey of $u$ to $w((u,v))$
```

9.2 Running time

The initialisation phase takes $O(n)$ time. The running time of the while loop is dominated by the heap operations. Each vertex is inserted and deleted once. The key of a vertex is decreased at most once for each edge in the graph, that is at most m. By using a binary heap as priority queue, the running time is $O(n \log n + m \log n) = O(m \log n)$. After the discovery of Fibonacci heaps [FT87], the running time was reduced to $O(n \log n + n + m) = O(n \log n + m)$, due to the constant amortised running time of the insert and decreaseKey operations. So if m is $O(n \log n)$ corresponding to the density m/n is $O(\log n)$ (sparse), the running time is $O(n \log n)$. Otherwise, that is, if m is $\Omega(n \log n)$ corresponding to the density m/n is $\Omega(\log n)$ (dense), the running time is $O(m)$.

9.3 Alternative implementation

We will present a modified implementation for the DJP algorithm, that opposed to Algorithm 9.1 requires no decreaseKey operation. This is relevant for the Partition procedure presented in Section 14.2. With this implementation, the priority queue stores edges with their weights as keys. Consequently we do not assign keys to vertices in Algorithm 9.1. Instead of initially inserting v_0 into the queue, we just add v_0 to T and insert all of its incident edges.

Every time a vertex, say v, is explored and the tree T is augmented by v and an edge connecting T to v, we insert all edges incident to v into the queue[1]. That is a replacement of the inner for-loop body in Algorithm 9.1 by an insertion. Hence, the edge of minimum weight in the queue is always either 1) the lightest edge connecting a vertex in T to a neighbouring vertex outside T, or 2) some edge with both endpoints in T. So we change the `deleteMin` step of Algorithm 9.1, to repeatedly call `deleteMin` until the edge of minimum weight is an edge connecting T to a neighbouring vertex v outside T. Then v is marked as explored, and T is augmented with this edge and neighbouring vertex v.

Every edge is clearly inserted twice, and deleted at most twice. Thus the running time using a Fibonacci heap, is $O(m \log m + m) = O(m \log m) = O(m \log n)$, as m is $O(n^2)$. Consequently, this implementation is not suitable using a Fibonacci heap as priority queue.

9.4 A hybrid algorithm

For sparse graphs, the running time of DJP and Borůvka is $O(n \log n)$ and $O(m \log n)$, respectively. A trivial way to get a better running time, is to combine these two MST algorithms the following way: Firstly, run $O(\log \log n)$ Borůvka steps. The contracted graph G' after these steps will have

$$O\left(\frac{n}{2^{\log \log n}}\right) = O\left(\frac{n}{\log n}\right)$$

vertices and at most m edges. The running time of the Borůvka steps is $O(m \log \log n)$. Afterwards, run the DJP algorithm on G'. Using Fibonacci heaps, the running time is

$$O\left(\frac{n}{\log n} \log n + m\right) = O(n + m) = O(m) \ .$$

The running time is dominated by the Borůvka steps, so the total running time of the hybrid algorithm is $O(m \log \log n)$. For very sparse graphs[2] where m/n is $O(1)$, this algorithm runs in linear time $O(m)$, as Borůvka's algorithm runs in linear time. Similarly, for dense graphs where m/n is $\Omega(n/\log \log n)$, this algorithm runs in time $O(n^2)$, as Borůvka's algorithm runs in $O(n^2)$ time for dense graphs.

[1]Can be optimised by only inserting edges to unexplored vertices.
[2]Including planar graphs.

Part III

Advanced building blocks

10 The "Dense Case" algorithm

Briefly, to make the optimal MST algorithm work, it must be able to find the MST of a "sufficiently dense" graph in linear time. The density lower bound of $\Omega(\log n)$ in the original DJP algorithm (Chapter 9) is too high for this purpose. What sufficiently dense means will become clear in the end of this chapter. There exists various deterministic algorithms to achieve a linear running time for dense graphs, but the simplest is by [FT87], where the Fibonacci heap was presented. As stated in Section 9.2, the discovery of Fibonacci heaps reduced the running time of the DJP algorithm to $O(n \log n + m)$. A even more important result was a MST algorithm with complexity $O((\log^*(n) - \log^*(m/n))\, m)$, which is $O(m)$ for "sufficiently dense" graphs. Therefore this section will concentrate on describing this particular "Dense Case" algorithm of [FT87].

The Dense Case algorithm is basically a modified version of the Dijsktra-Jarník-Prim (DJP) algorithm. This version utilises Fibonacci heaps [FT87] as the priority queue. The main idea is to bound the maximum number of neighbouring vertices in the heap to a certain value k. We will deduce a function for k in Section 10.2. When a vertex is extracted from the priority queue, it is marked dead, indicating it is a part of a grown tree. If the heap exceeds the bound k, becomes empty, or the tree connects to a previously grown tree, we repeat the DJP algorithm with a live vertex as the root vertex. This continues until all vertices are marked dead, that is, all vertices is contained in a tree. When all vertices are marked dead, all connected components induced by MST edges are contracted. After that, the pass is done. That is, every grown tree is a connected component, which is contracted into a single vertex. The algorithm repeatedly runs such a pass until the full set of MST edges is found, which is equivalent to when the graph is contracted into a graph without any edges.

10.1 Implementation

As stated, the DenseCase algorithm uses a modified version of the DJP algorithm. Let `growTree(`v`)` be a embedded version of DJP given in Algorithm 9.1 with the following small modifications: Do not initialise the priority queue nor initialise the key of each vertex to ∞. Use the *live* vertex v as the root v_0. When a dead vertex is extracted from the queue, stop the execution, otherwise mark it dead. Before inserting a vertex into the heap, stop the execution if the heap is full, that is if it has size k. Thus `growTree` stops if it has connected to another tree, the heap is overflowed, or the heap is empty. The "Dense Case" algorithm is shown in Algorithm 10.1.

Algorithm 10.1: Dense Case MST algorithm

```
$T \leftarrow$ trivial forest with all vertices of $G$
while $E(G) > 0$ do
    Calculate $k$ for this pass /* To be chosen later in Section 10.2 */
    Initialise Fibonacci Heap with bound $k$
    forall vertices $v \in V(G)$ do
        Mark $v$ live
        key($v$) = $\infty$
    while there is a live vertex $v$ do
        growTree ($v$)
        Empty the heap
        forall inserted vertices $v$ do  key($v$) $\leftarrow \infty$
    $(G, T) \leftarrow$ contract($G, T$)
return $T$
```

The outer while-loop is the “pass loop” which runs until the graph is contracted into a graph without any edges. The inner while-loop runs until all vertices in the current graph are in a tree. Hence, it is easy to see that this algorithm also can find the MSF of an unconnected graph. Recall that a key of ∞ indicates that the vertex is outside the current tree. So in the end of a growing step, the keys of all inserted vertices are reset to ∞ because they have a finite key after being inserted.

10.2 Running time

Even though this algorithm also works for unconnected graphs, we will only study the running time for connected graphs. With reference to Section 4.2, it is possible to detect the connected components of an unconnected graph in linear time, so for the analysis we can assume the input graph is connected.

Using the following facts about the algorithm, we can bound the number of heap operations per pass:

- Only vertices adjacent to live vertices are eventually inserted into the heap or have their keys decreased.
- Each vertex is marked dead when it is deleted from the heap.
- The first vertex deleted in `growTree` is alive. This vertex is v_0.

Let $t = |V(G)|$, when a pass begins. That is the number of vertices in the contracted graph. The number of insert and decreaseKey operations is at most $2m$, namely at most once for each endpoint. That is $O(m)$ heap operations with amortised constant running time. The total number of deleteMin operations is the number of live vertices deleted plus the number of dead vertices deleted. Due to the second fact, the number of live vertices deleted is t. Once a dead vertex is deleted, `growTree` stops and starts again, so the next vertex to be deleted is live. This means the number of dead vertices deleted is at most t. In total that is $O(t)$ deleteMin operations on a heap of maximum size k. Besides the heap operations, the initialisation of a pass takes $O(t)$ time and the contraction step takes $O(m)$ time. The bookkeeping of inserted vertices is done by adding the vertex to a list when inserting it. The cost of emptying the heap and resetting the keys to ∞ are charged to the insert operations. The bookkeeping of finding a live root vertex is done by maintaining a reference to the first live vertex in the vertex list $V(G)$. When we need to find a new live vertex, we iterate forward through the vertex list until a live vertex is found. This sums up to $O(t)$ time per pass. So if the Fibonacci heap bound is k, then the total time for a complete pass is $O(t\log(k) + m + t) = O(t\log(k) + m)$.

Intuitively smaller values of k will reduce the running time per pass, but raise the number of passes. Larger values will reduce the number of passes, but raise the running time per pass. Values $k < \lceil m/t \rceil$ will not work in general, because trees will eventually not grow due to heap overflow in the first iteration of each `growTree`. Values $k \geq t$ will make a pass the final pass, because there is room for all trees in the heap. This value will make a pass identical to the original DJP algorithm. So if $k \geq n$ in the first pass, the complete algorithm is identical to DJP.

In [FT87], they chose the following function for k: For a pass with t input vertices, set $k = 2^{2m/t}$, where m is the original number of edges. With this choice of k each pass takes $O\left(t \log 2^{2m/t} + m\right) = O(m)$ time. So each pass takes linear time in the number of edges in the original graph. Because the number of vertices decreases from pass to pass, the value of k will increase from pass to pass. It remains to deduce an upper bound of how many passes are needed to get from $t = n$ to the worst case $t = 1$, if the input graph is connected.

A new tree T stops growing because of either:

- The heap size bound of k adjacent vertices is violated, which is equivalent to that T has at least k adjacent vertices outside T. Thus T has at least k edges with at least one endpoint in T, namely at least one edge for each adjacent vertex.
- The heap is empty, which is equivalent to that T can not grow bigger.
- T is connected to another tree T' (in form of a dead vertex). Here, T' is a joint tree of previously connected grown trees. This option is clearly not the case when the first tree of a pass stops growing. Therefore T' has at least k edges with at least one endpoint in T', and so has the resulting joint tree of $T \cup T'$.

From here, let a tree T be a composite tree of one or more connected trees grown by `growTree`. If a pass results with a single tree of MST edges, it is the last pass as it is contracted to a single vertex. Otherwise, a pass ends with that each tree T has at least k edges from G with at least one endpoint in T.

Let t be the number of vertices and $m' \leq m$ be the number of edges when a pass begins. As stated above, k is the minimum number of edge endpoints per tree T. The total number of edge endpoints is $2m'$ when a pass begins. The number of trees after a pass, and hence the number of vertices in the subsequent pass, is therefore $t' \leq 2m'/k$. The heap size bound for the subsequent pass is $k' = 2^{2m/t'}$. As $t' \leq 2m'/k$ and $m' \leq m$, we have that $2m/t' \geq 2mk/(2m') \geq k$, so $k' \geq 2^k$. In other words, k is boosted exponential between each pass.

The initial k in the first pass is $2^{2m/n}$ as the first $t = n$. When $k \geq n$, the first tree will not stop growing before it contains all vertices, because there is room for all of them in the heap. This is what happens in the original DJP algorithm, which has no heap size bound. Consequently when $k \geq n$, it is the last pass. So one way to analyse the maximum number of passes needed, is to find the minimum number of times, say i, we have to boost the initial k by 2^k before $k \geq n$. The number i is equal to the number of times the logarithm function must iteratively be applied to n before the result is less than or equal to the initial $k = 2^{2m/n}$. Formally the maximum number of passes needed is bounded by $1 + \min\left\{i \mid \log^{(i)}(n) \leq 2^{2m/n}\right\}$. As $\log^{(1)}\left(2^{2m/n}\right) = 2m/n$, the maximum number of passes needed is $O(1) + \min\left\{i \mid \log^{(i)}(n) \leq 2m/n\right\} = O(1) + \min\left\{i \mid \log^{(i)}(n) \leq m/n\right\}$. In [FT87], they define $\beta(m,n) = \min\left\{i \mid \log^{(i)}(n) \leq m/n\right\}$, so the maximum number of passes is bounded by

$$\beta(m,n) + O(1)\ .$$

As stated earlier, each pass takes $O(m)$ time, so the total running time is

$$O(m\beta(m,n))\ .$$

It is easy to see that $\beta(m,n)$ also can be expressed in terms of the more well-known "iterated logarithm" or "log-star" function. Intuitively $\log^*(n)$ is loosely the same as the minimum i plus "the rest" which is $\log^*(m/n) - O(1)$, so the expression for the minimum i becomes

$$\beta(m,n) = \log^*(n) - \log^*(m/n)\ .$$

Worst case and linear running times

It becomes clear from $\beta(m,n)$ that the number of passes does not depend directly on neither m or n, but instead of the composite edge-to-vertex ratio (graph density) m/n. We assume the graph is simple and connected. If $m < n$, then $m = n - 1$ and the graph is a tree, so there is no need to run this algorithm. Generally speaking a high density gives a low worst case running time and vice versa. The worst case is when $m = n \Rightarrow m/n = 1$, so $\beta(m,n) = \log^*(n)$. The best case is a complete graph, that is when m is $O\left(n^2\right) \Rightarrow m/n = O(n)$, so $\beta(m,n) = O(1)$. This shows that the worst case running time for the algorithm is $O(m\log^*(n))$. On the other hand it runs in linear time $O(m)$ for sufficiently dense graphs, which is the interesting property for the optimal MST algorithm.

So what is a sufficiently dense graph? As shown above $m/n = O(n)$ is the upper bound for density. The lower bound of density depends on the constant, say c, number of passes wanted. To limit i in $\beta(m,n)$ to the constant c, set $m/n = \log^{(c)}(n)$. This gives $\beta(m,n) = \min\left\{i \mid \log^{(i)}(n) \leq \log^{(c)}(n)\right\} = c$. This property is exploited in the optimal MST algorithm in Section 15.1, where the constant is $c = 3$, and consequently the number of passes is bounded by the constant 3.

10.3 Linear running time for the optimal algorithm

We will state a theorem about the Dense Case algorithm, which will be used by the optimal MST algorithm.

Theorem 10.1. *Consider a simple connected graph G of n vertices and m edges, where $m \geq n$. Then consider a graph G' of $n' \leq n/\log^{(3)}(n)$ vertices and $m' \leq m$ edges derived by contracting G. Given that the heap size bound k is calculated as a function of m (as opposed to m'), then the running time of the Dense Case algorithm for G' is $O(m)$.*

Proof. As $k = 2^{2m/t}$, each pass takes time $O\left(t \log 2^{2m/t} + m'\right) = O(m + m') = O(m)$. The initial k in the first Dense Case pass is $k = 2^{2m/n'}$ and when $k \geq n'$ it is the last pass. With the same arguments as in Section 10.2, the maximum number of Dense Case passes is

$$O(1) + \beta\left(m, n'\right) = O(1) + \min\left\{i \mid \log^{(i)}\left(n'\right) \leq m/n'\right\} .$$

Raising the left hand side of the inequality will increase i, and so will lowering the right hand side. So, because $n' \leq n/\log^{(3)}(n)$ and $m/n \geq 1$, we can conclude

$$\begin{aligned}
\beta\left(m, n'\right) &= \min\left\{i \mid \log^{(i)}\left(n'\right) \leq m/n'\right\} \\
&\leq \min\left\{i \mid \log^{(i)}\left(\frac{n}{\log^{(3)}(n)}\right) \leq \frac{m}{n}\log^{(3)}(n)\right\} \\
&\leq \min\left\{i \mid \log^{(i)}(n) \leq \log^{(3)}(n)\right\} \\
&= 3 \\
&= O(1) .
\end{aligned}$$

Consequently, the number of passes is bounded by a constant, and thus the running time is $O(m)$. □

11 MST decision trees

Briefly, to make the optimal MST algorithm run in optimal time, it must be able to determine the MST of small graphs relative to the input graph with an optimal number of edge-weight comparisons. The actual maximum size of the small graphs will be deduced in the analysis of Section 11.5. For a fixed graph G, this can be achieved by a hardwired optimal MST decision tree for G. An optimal decision tree for G is an "algorithm" that is hardwired to G and computes the MST of G for any given permutation of edge-weights with an optimal number of edge-weight comparisons. Before we can make use of optimal decision trees, we must build them. Intuitively, for a given number of vertices there exists, loosely speaking, many distinct graphs, and for each graph there exists, loosely speaking, many edge-weight permutations. Hence, the time for building an optimal decision tree for all graphs with a given number of vertices is relatively high. Consequently, we can only afford to build optimal decision trees for relatively small graphs, and thus we can assume that the size of the graphs described in this chapter is relatively small. Similarly, the number of decision trees is relatively small.
Before we describe decision trees and how to build them, we must show how to distinguish graphs.

11.1 Graphs

As stated in Chapter 3, a graph consists of a set of vertices, V, and a set of edges, E. Let n and m be the number of vertices and edges in a graph, respectively. In the context of input graphs to MST decision trees, we assume without loss of generality that a vertex is a unique integer, such that $V = \{1, \ldots, n\}$. The set of edges are distinct unordered pairs of vertices. A graph is uniquely defined by its edge set, ignoring edge weights. The elements in a set have no specific order, and thus many sequences of edges evaluates to the same edge set. For instance, $\{(1, 2), (2, 3)\} = \{(2, 3), (1, 2)\}$. To make the detection of a graph efficient, we need to be a little more restrictive with respect to edges. We require that edges are ordered lexicographically by their endpoints, for instance by first the smallest endpoint (vertex number), then by the highest endpoint. Let U be the universe of all possible edges in a graph of n vertices, that is the edge set in the complete graph. Hence, there exist $2^{|U|}$ distinct edge sets, and thus also graphs[1] with n vertices. Because edges are ordered elements, each of them, say e, maps to a distinct integer, say $f(e)$, in the interval $[0\ ,\ |U| - 1]$. It is easy to convert an arbitrary edge set, say E, into an array of sorted edges, say S, by radix sorting:

[1]Inclusive unconnected graphs.

1) Create n buckets, say $B = (b_1, \ldots, b_n)$. Then for each edge in E, let i be the highest endpoint, and insert the edge into bucket b_i. 2) Create n new buckets, say $B' = (b', \ldots, b'_n)$. Then for increasing i, take each edge in bucket b'_i: Let j be the smallest endpoint of the edge, and put the edge into bucket b'_j. 3) Initialise the array S of size m. Then for increasing j, append the edges of bucket b'_j to S. When done, S is a *sorted array* of edges $S = (e_1, \ldots, e_m)$, where $f(e_i) < f(e_{i+1})$. Consequently, each edge of a specific graph has a fixed position in this array.

It is easy to verify that the radix sorting process takes time $O(n + m) = O(m)$, which does not exceed the time for building the graph. A graph is uniquely represented by its sorted array of edges, S. Let s_i denote the i'th entry of S. A sorted array S is isomorphic to a unique number among all graphs with n vertices and m edges:

$$\sum_{i=0}^{m-1} |U|^i \cdot f(s_i)$$

It takes linear time in the number of edges to calculate this number. Hence, we can represent each possible graph with n vertices and m edges by a unique number. Consequently, we can represent any graph by a unique number. From this point, we will assume that all edge sequences are sorted.

Let $(w(e_1), \ldots, w(e_m))$ be the distinct weights of edges in a graph. The number of edge-weight permutations for a graph with m edges is $m!$. Without loss of generality, assume that the m distinct edge weights are $1, \ldots, m$. For example, a graph with $m = 3$ edges has $3! = 6$ edge-weight permutations, which are

$$(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)$$

We have showed how to sort the edges of a graph in such way that each of them has a fixed position in an array. We have also showed how to convert a graph (ignoring edge-weights) to a unique number representing the graph, and thus how to distinguish graphs. Both computations takes linear time in the size of the graph. It remains to show how to find the MST of a graph for any possible edge-weight permutation with an optimal number of edge-weight comparisons.

11.2 MST decision tree

A MST decision tree is a rooted binary tree, which is hardwired to a graph (ignoring its edge weights). Because the edges are sorted, we can access any particular edge in constant time. Each internal node of a tree has an edge-weight comparison associated. The data associated to each internal node is an ordered pair of edges, say (e_1, e_2). The edge-weight comparison at an internal node is $w(e_1) < w(e_2)$. One of the children of an internal node, say the left, represents that the comparison is false, and the other, say the right, represents that the comparison is true. Each leaf of a MST decision tree has a set of edges associated. If the decision tree is *correct*, then each of these sets is the solution to the MST problem for a graph with the edge-weight permutations on the root-to-leaf path. A decision tree is *optimal* if it is correct and there exists no correct decision trees with lesser depth. The depth of the optimal decision tree is the worst case number edge-weight comparisons needed to deduce a MST from the particular graph. See Figure 11.1 for an example of a graph with its hardwired optimal decision tree.

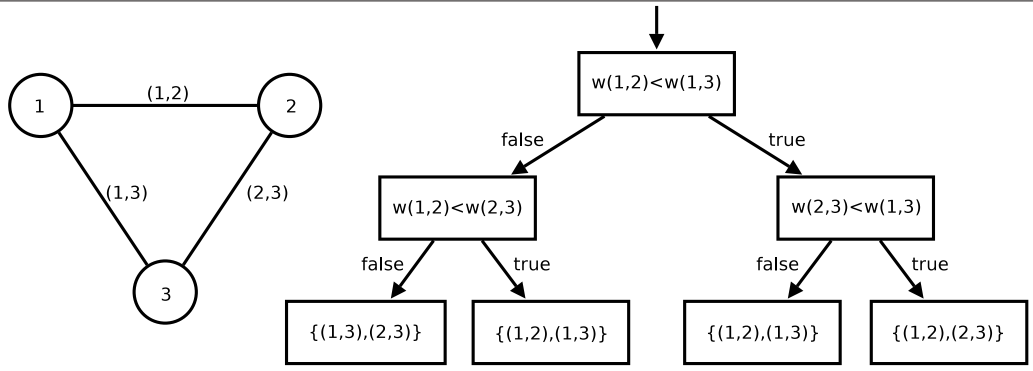


Figure 11.1: A graph with $n = 3$ vertices defined by $((1,2),(1,3),(2,3))$ and its hardwired optimal decision tree.

Notice that for graphs with 1 or 2 vertices, the MST problem has a trivial solution and thus does not need a decision tree to find a solution. A simple graph with $n = 1$, has no edges, and thus the set of MST edges is the empty set. A simple graph with $n = 2$ has either zero or one edge. We assume the graph is connected, so it has one edge, and thus the set of MST edges is the set consisting of this edge.

11.3 Brute force searching procedure

We will describe a procedure for finding optimal decision trees for all graphs with maximum r vertices. The running time of the procedure is deduced in the subsequent section. Due to the trivial solutions just described, we do not need to find decision trees for $r = 1$ and $r = 2$. Hence, in the analysis, we can assume that the number of vertices is $r \geq 3$. We will find an optimal decision tree for each graph with at most r vertices by brute force searching. The main idea is to generate all possible decision trees of a particular size, and then for each graph find an optimal tree among these.

Firstly, calculate an upper bound on the optimal decision tree depth for a graph with r vertices. The depth corresponds an upper bound on the number of necessary edge-weight comparisons. We will deduce such an upper bound in the analysis of Section 11.4. For a graph with r vertices, there is an upper bound on the number of possible edge-pairs, which also is the number of possibilities for an internal node. Thus, we can also deduce an upper bound on the number of distinct decision trees. So secondly, generate all possible decision trees of the calculated depth.

Then, for each possible graph, G, with at most r vertices:

1. Generate all possible edge-weight permutations for G.
2. Solve the MST problem for G with each edge-weight permutation using a standard MST algorithm, such as DJP, and associate the MST solution with the permutation.

3. Initialise a current optimal decision tree reference $T = \perp$ and a current optimal depth variable, $d = \infty$, to indicate that we have not yet found a correct decision tree for G.

4. For each of the original decision trees (without the modifications made for previous graphs):

   Place all edge-weight permutations with their MST solutions on the root node. Then, for each tree depth, from root to leaf, do the following for each node:

   > Check if all MST solutions are equal. If so, then save the MST at this node, and stop the process on this root-to-leaf path. That is, do not proceed in the subtree of this node in the subsequent depth steps. Otherwise, then for each permutation: Perform the edge-weight comparison associated with the node, and move the permutation with its MST solution to the child corresponding the comparison result.

   Because we check all possible decision trees, some of the comparisons will refer to an edge which does not exist in G. In that case, reject the current decision tree and proceed to the next.

   If at some depth, the process stops because the MST solution(s) are equal for each path, then the current decision tree is correct. If the depth is less than d, then update T to the current tree and d to the depth. The previous optimal decision tree is thus rejected for G, as we have found one with lesser depth. Otherwise, if the depth is not lesser than d, then reject the current tree, because one with lesser depth is already found.

   Otherwise, at some point the comparisons reach a leaf node where the edge-weight permutations still not agree on the MST. In that case, the decision tree is not correct, so we can reject it for G.

   Alternatively, we can reject the current decision tree when the depth exceeds the current optimal depth d.

5. When we have checked all trees, the tree T with depth d is an optimal decision tree for G. To reduce the size of T, we remove the subtrees of nodes where we have saved a MST, such that these nodes become leaf nodes. As we have checked all possible decision trees, we are sure to find a correct and optimal decision tree for G. Hence, we hardwire T to G.

11.4 Running time

As stated in Section 11.1, the number of graphs with r vertices is $2^{|U|}$, where U is the set of edges in the complete graph. The number of edges in the complete graph of r vertices is $\left(r^2 - r\right)/2 < r^2$. Hence, there exists at most

$$g = 2^{r^2}$$

graphs with r vertices. It is clear that this bound dominates the sum of number of graphs with up to r vertices: $\sum_{i=1}^{r} 2^{i^2} \leq 2 \cdot 2^{r^2}$.

The DJP algorithm (see Chapter 9) uses at most one edge-weight comparison per edge endpoint, so the maximum number of edge-weight comparisons is $r^2 - r < r^2$.

Consequently, this is an upper bound on the depth of an optimal decision tree. The number of nodes in a binary tree of height h is $2^h - 1$, hence the tree has $< 2^{r^2}$ internal nodes. Each internal node must identify an ordered pair of edges, say (e_1, e_2). As there is at most r^2 edges, there is at most $r^2 \cdot r^2 = r^4$ ordered pairs of edges, hence there is at most r^4 possibilities for each internal node. As there is $< 2^{r^2}$ internal nodes and $< r^4$ possibilities for each internal node, there exists at most

$$T = \left(r^4\right)^{2^{\left(r^2\right)}} = r^{4 \cdot 2^{\left(r^2\right)}} = r^{2^2 \cdot 2^{\left(r^2\right)}} = r^{2^{\left(r^2+2\right)}}$$

distinct decision trees for graphs with maximum r vertices. As each tree has less than 2^{r^2} internal nodes, it takes time

$$t_{\text{create}} = O\left(2^{r^2} T\right) = O\left(gT\right)$$

to create all decision trees.

As there is at most $\left(r^2 - r\right)/2 < r^2 - 2$ edges in a graph of $r \geq 3$ vertices, there exists at most $\left(r^2 - 2\right)!$ edge weight permutations per graph. As the number of edges in a graph is $O\left(r^2\right)$, there exists various deterministic MST algorithms, such as the DJP algorithm, with running time $O\left(r^4\right)$. Hence, it takes time

$$t_{\text{djp}} = O\left(\left(r^2 - 2\right)! \cdot r^4\right) = O\left(\left(r^2 - 2\right)! \cdot r^2 \cdot r^2\right) = O\left(\left(r^2\right)!\right)$$

to find the correct MST of each edge-weight permutation per graph.

Assuming the MST edge sets are sorted, we can check two sets for equality in $O\left(r\right)$ time. Each decision tree has depth $< r^2$. Hence, it takes time

$$t_{\text{check}} = O\left(\left(r^2 - 2\right)! \cdot r \cdot r^2\right) = O\left(\left(r^2\right)!\right)$$

to check if a decision tree is correct.

Summing up, the total running time for finding an optimal decision tree for each graph of at most r vertices is:

$$O\left(t_{\text{create}} + g\left(t_{\text{djp}} + T t_{\text{check}}\right)\right) .$$

We know that $t_{\text{djp}} = t_{\text{check}}$, and $t_{\text{create}} = O\left(gT\right)$, so the time is

$$O\left(gTt_{\text{check}}\right) = O\left(2^{r^2} \cdot r^{2^{\left(r^2+2\right)}} \cdot \left(r^2\right)!\right) .$$

Simplifying the running time

As $x! < x^x$ for any integer value $x \geq 2$, an upper bound on the time for checking a tree is

$$t_{\text{check}} = \left(r^2\right)! < \left(r^2\right)^{r^2} = r^{2r^2} < r^{2^{\left(r^2+2\right)}} .$$

Similarly, an upper bound on the number of graphs is

$$g = 2^{r^2} < r^{2^{\left(r^2\right)}} < r^{2^{\left(r^2+2\right)}} .$$

Consequently we have a running time which is order $r^{2^{\left(r^2+2\right)}}$ cubed. Thus, the running time is order

$$\begin{aligned}
\left(r^{2^{\left(r^2+2\right)}}\right)^3 &< \left(r^{2^{\left(r^2+2\right)}}\right)^4 \\
&= r^{4\cdot 2^{\left(r^2+2\right)}} \\
&= r^{2^{\left(r^2+4\right)}} \\
&= \left(2^{2^{\log^{(2)}(r)}}\right)^{2^{\left(r^2+4\right)}} \\
&= 2^{2^{\left(r^2+4\right)}\cdot 2^{\left(\log^{(2)}(r)\right)}} \\
&= 2^{2^{\left(r^2+\log^{(2)}(r)+4\right)}} .
\end{aligned}$$

Notice that

$$\lim_{r\to\infty} \frac{\log^{(2)}(r)+4}{r} = 0$$

which implies that $\log^{(2)}(r)+4$ is $o(r)$. Consequently the time is

$$O\left(2^{2^{\left(r^2+o(r)\right)}}\right) .$$

11.5 Maximum partition size for the optimal MST algorithm

Briefly, the optimal MST algorithm takes a graph with n vertices and m edges, of which it divides into small partitions (subgraphs). For each partition, it uses a hardwired optimal decision tree to find its MST. Hence, the algorithm must find optimal decision trees for all graphs (that is partitions) of a certain number of vertices, r, which is the maximum partition size. It remains to find the value r, such that the optimal decision trees can be precomputed in linear time of the input graph size.

Theorem 11.1. *For partitions with maximum $r = \lceil \log^{(3)}(n) \rceil$ vertices, the decision trees for the optimal MST algorithm can be precomputed in $o(n)$ time.*

Proof. First, notice that

$$\lim_{r\to\infty} \frac{\left(\log^{(3)}(r)\right)^2}{\log^{(2)}(r)} = \lim_{r\to\infty} \frac{\log^{(3)}(r)}{\log^{(2)}(r)} = 0 ,$$

which implies that $\left(\log^{(3)}(r)\right)^2$ is $o\left(\log^{(2)}(r)\right)$ and that $\log^{(3)}(r)$ is $o\left(\log^{(2)}(r)\right)$.

Setting $r = \log^{(3)}(n)$ gives running time order

$$\begin{aligned} 2^{2^{\left(\left(\log^{(3)}(n)\right)^2 + o\left(\log^{(3)}(n)\right)\right)}} &= 2^{2^{\left(o\left(\log^{(2)}(n)\right) + o\left(\log^{(2)}(n)\right)\right)}} \\ &= 2^{2^{o\left(\log^{(2)}(n)\right)}} \\ &= 2^{o(\log n)} \\ &= o(n) \ , \end{aligned}$$

which proves the theorem. □

Notice that we can set r as high as $\sqrt{\log^{(2)}(n)} - 1$ and achieve the same running time, but it provides no benefit to the running time of the optimal MST algorithm.

Impractical method

The number of atoms in the universe is approximately 2^{265}, so with reference to Table 11.1, for practical values of n, the partition size will never exceed $r = 4$ vertices. As stated previously, it is trivial to find the MST of a graph with $r = 1$ or $r = 2$. For any connected graph, if the number of edges is $r - 1$, then the MST is the set of all edges. So with regard to graphs with $r = 3$, only one exists where the number of edges exceeds $r - 1$. This graph has the edge set which is a cycle of three edges. See the graph in Figure 11.1 for an illustration. Hence, with reference to the cycle property in Theorem 4.1, the MST of this graph is the two lightest edges, or equivalently all edges, except the heaviest. So we can find the MST by scanning the edges, which takes linear time.

Generally, all graphs with up to $r = 4$ vertices are planar graphs, so Borůvka's algorithm (Chapter 8) can find the MST of all such graphs in linear time of the number of edges. But as r in practice is bounded by a small constant, any MST algorithm can deduce the MST in constant time. Consequently, the decision trees are needless for all realistic input graphs to the optimal MST algorithm, as we can run Borůvka's algorithm instead and achieve a linear running time.

For that reason, we have not implemented decision trees for the optimal MST algorithm. Instead, we use one of the methods described above depending on the graph size, as we are unable to generate test graphs with $n > 2^{65536}$, or $n > 2^{256}$ for that matter.

n, power of 2	$]2^2\ ,\ 2^4]$	$]2^4\ ,\ 2^{16}]$	$]2^{16}\ ,\ 2^{256}]$	$]2^{256}\ ,\ 2^{65536}]$
n, tower of 2's	$\left]2^{2^{2^0}}\ ,\ 2^{2^{2^1}}\right]$	$\left]2^{2^{2^1}}\ ,\ 2^{2^{2^2}}\right]$	$\left]2^{2^{2^2}}\ ,\ 2^{2^{2^3}}\right]$	$\left]2^{2^{2^3}}\ ,\ 2^{2^{2^4}}\right]$
$r = \left\lceil \log^{(3)}(n) \right\rceil$	1	2	3	4

Table 11.1: Values of $\log^{(3)}(n)$.

12 The soft heap

The main data structure utilised by the optimal MST algorithm is the *soft heap* by Chazelle [Cha00b]. Chazelle used soft heaps for the MST algorithm presented in [Cha00a], which at present[1] provides the best upper bound on the MST complexity. Recently the implementation and in particular its analysis has been simplified by [KZ09]. We will give a full description of Chazelle's original version of the heap. As the name suggests it is a heap based priority queue. The soft heap may corrupt some items by artificially raising their keys in the heap. The data structure must be initialised with an error rate parameter ε, where $0 < \varepsilon < 1/2$. The error rate dictates an upper bound on the number of corrupted items in the priority queue. The following theorem states the main properties of the soft heap. The proof of Theorem 12.1 follows in Section 12.8 and Section 12.9.

Theorem 12.1. *Consider a soft heap with fixed error rate $0 < \varepsilon < 1/2$ and no prior data. The amortised running time of each heap operation is constant, except for insert which takes $O(\log 1/\varepsilon)$ time. For a mixed sequence of heap operations that involves n inserts, the heap never contains more than εn corrupted elements.*

Notice, that as opposed to most data structures, n is not the current heap size, but the number of insertions. Consequently, it is theoretically possible, that all elements in the queue are corrupted at a given time. As a simple example, consider n inserts succeeded by $(1-\varepsilon)n$ deletes or deleteMins on a empty heap. Now, the heap will have size εn which is the same as the upper bound of corrupted elements.

12.1 Introduction

Like the Fibonacci heap [FT87], a soft heap is a sequence of binomial trees, that are allowed to be modified by removing subtrees. We will start by describing pure unmodified binomial trees. A binomial tree is a rooted tree, where each node has a nonnegative integer rank associated. A node of rank k has exactly k children. The children of a node with rank k have ranks $0, \ldots, k-1$. The rank of an entire tree is defined as the rank of its root node. The binomial tree of rank k has exactly 2^k nodes. Hence, the basic binomial tree of rank 0 is the trivial tree with one node. The binomial tree of rank $k > 0$ is formed by linking two binomial trees of rank $k-1$, such that the root node of one tree becomes

[1]This thesis is written in 2008.

a new child of the root node of the other tree. It is easy to verify by induction that the combination of two trees of rank $k-1$ has 2^k nodes.

From this point, we will refer the to modified binomial trees in a soft heap as *soft queues*. Unlike the Fibonacci heap, the soft heap only allows distinct soft queue ranks, that is zero or one soft queue of each rank. Recall that a soft queue is a binomial tree where some subtrees are removed. Let the *master tree* of a soft queue be the binomial tree from where the soft queue is derived from by removing subtrees. The rank of a soft queue node is defined as its rank in its master tree. That is the number of children in the master tree before removing children. Therefore the rank of a soft queue node is greater than or equal to its number of children. The soft heap does not allow an arbitrary number of children of a root node to disappear. Actually, the following invariant is maintained on the number of children, $\deg(v)$, of a soft queue root node v:

$$\lfloor \operatorname{rank}(v)/2 \rfloor \leq \deg(v) \leq \operatorname{rank}(v) \ .$$

Unlike normal heap structures, a soft queue node v may store more than one item. Actually, it may store an entire list of items without any explicit upper bound on the list size. However, in Section 12.8 we will show that an important implicit upper bound exists on the list size. Each soft queue node v has associated a "common key" of all items in its item list. This common value is called the "*ckey*" of v. The `ckey` of v is defined as an upper bound of keys in v's item list. That is if no items are deleted from the list, then the `ckey` is the maximum key among all item keys. A soft queue is heap ordered with respect to the `ckeys`. That is the `ckey` of a node is always less than or equal the `ckey` of all its child nodes. Consequently, all items in a list, except those with the key equal to the `ckey`, are *corrupted*, as they may not follow the heap order, and really should stay closer to the root node. By this definition of a corrupted item, it is easy to see that some items in a list are corrupted, if the items have two or more distinct keys. It is even easier to see that any item list of size 1 has no corrupted items, as the key of the item is equal to the `ckey`. When a soft heap is created, it calculates a value, r, which is a function of the fixed error rate ε. During the life time of a soft heap, it keeps the invariant that all corrupted items are stored at nodes with rank strictly greater than r. This is achieved by disallowing item lists with multiple items at nodes with rank $\leq r$. Consequently, items stored in the bottom a soft queue are not corrupted until they travel toward the root.

12.2 Data structure

Queue structure

As stated above, each node has a `rank`, a `ckey`, and a item list associated. We should be able to concatenate two item lists in constant time, so each item list must be a linked list. Therefore, each queue node must have a reference to the head and the tail node of its item list. We refer to these references as `il` and `il_tail`, respectively. Additionally a node should have references to its children. For a node v, this requires up to $k = \operatorname{rank}(v)$ references. In [Cha00b], Chazelle suggests to represent degree-k nodes as a sequence of degree-2 nodes, where each node has a `child` and a `next` reference. Until the description of the implementation in Section 12.10, we will view soft queue nodes as degree-k nodes to make the correspondence to binomial tree nodes clearer. So we assume that each queue node has a linked list of children.

Thus a queue node has the following fields:
`(ckey, rank, il, il_tail, children)`.

Top structure

The top structure of the soft heap is a double linked list called the *head list*. From this point, we will refer to the head nodes as heads. Let m be the number of soft queues. The head list has m heads, $h_1, \ldots, h_m$, where each head h_i consists of two references. The first reference, `queue`, of head h_i points to the root of a distinct soft queue r_i. We require that $\text{rank}(r_1) < \cdots < \text{rank}(r_m)$, so the queue ranks are distinct. The other reference, `suffix_min`, of head h_i points to the head h_j, with minimum `ckey` among all r_j's, where $j \geq i$. In other words, it points to the head node with minimum `ckey` among h_i and its successors. Thus `suffix_min` is a forward pointer in the head list. Additionally, each head h_i has a `rank` field associated. When a soft heap is initialised, it creates a static header and tail virtual head node, links them, and keeps fixed references to them. Therefore, the complete list of head nodes becomes $h_0, \ldots, h_{m+1}$. The header node is used as a fixed start node of the list. The rank of the tail node is initially set to ∞. The rank of the header node is unimportant, and rank of any other head node h_i always reflects the rank of the root of its queue: $\text{rank}(r_i)$.

Thus a head node has the following fields:
`(queue, prev, next, suffix_min, rank)`.

Figure 12.1 shows a snapshot of a soft heap instance.

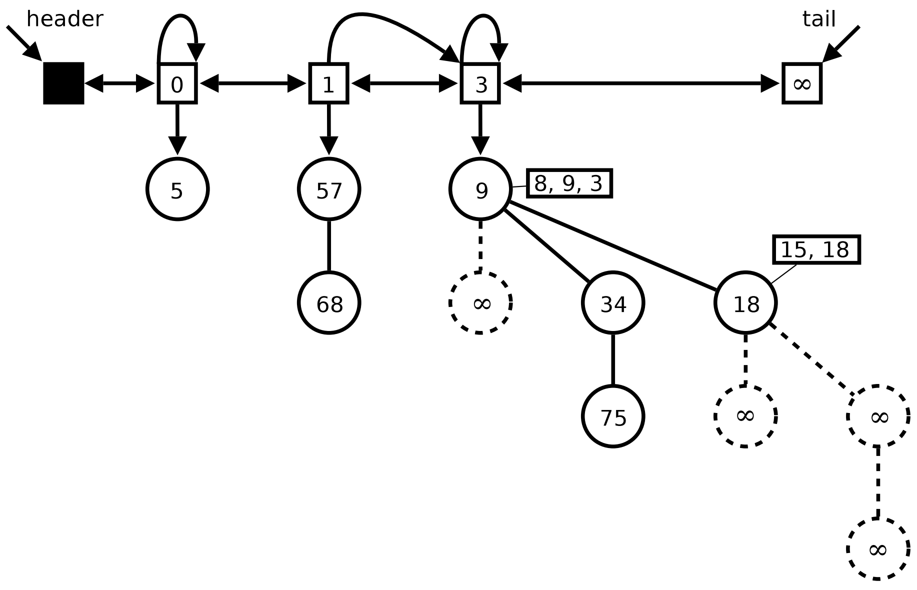


Figure 12.1: Soft heap data structure with three soft queues. The number inside a squared head is the queue rank. The number inside a queue node is the `ckey`. The item lists with size > 1 is shown next the queue node. Removed subtrees are shown with dashed lines.

12.3 Supporting heap functions

Queue melding

The purpose of the function `meld` is to meld a soft queue, `q`, into the soft heap. Let k be the rank of `q`. We meld a queue into the heap by first finding the smallest i such that rank $(h_i) \geq k$ by walking forward in the head list starting from the successor of the header node (that is h_1). As the rank of the tail is ∞, such head node always exists. If rank $(h_i) > k$, we create a new head node for `q`, and insert it right before h_i. Otherwise, if rank $(h_i) = k$, we have a conflict because the two queue ranks are equal, but we disallow equal queue ranks. Hence we remove h_i from the list, and link the two conflicting queues into one of rank $k + 1$, by making the root node with the largest `ckey` a new child of the other root. Thus, the heap order of the resulting queue is maintained. Now we stand with a new queue of rank $k + 1$. If rank $(h_{i+1}) = k + 1$, then a new conflict arises, and we remove the head node and link again. This process continues until there are no more conflicts, and so we insert a new head node with `queue` pointing to our new queue.

After this process, some of the `suffix_min` pointers between h_1 and the new head node may point to the wrong head nodes, maybe even dismantled head nodes. This problem is handled by a procedure called `fix_minlist`, which is described in the subsequent section. The updating of pointers is achieved by calling `fix_minlist` with the new head node as parameter.

Updating `suffix_min` pointers

The procedure `fix_minlist` takes a head node, h, as parameter, and walks backwards in the head list starting from h, while it updates the `suffix_min` pointers. As these are forward pointers, it is assumed, that there is no invalid pointers after h. The updating takes place by maintaining a pointer to the succeeding head containing the minimum `ckey` encountered so far. This pointer is initialised to the `suffix_min` pointer of h's successor head, as this must point to the head with minimum `ckey` among h's successors. A special case arises when the successor head is the tail node, where the pointer is initialised to h instead, as it has no real successors.

12.4 Heap operations

Finding corrupted items

The task of finding corrupted items in the heap is quite simple. We traverse all soft queues nodes with rank $> r$, and for each item list, we report all items where the key differs from the `ckey` of the queue node. This can be achieved by a Depth-First-Search of each queue down to and including nodes with rank $r + 1$.

Insert

With the melding procedure, it is easy to implement the `insert` method. Let e be the new key. We create a simple soft queue with one uncorrupted node v, with item list $\{e\}$, and meld the new queue into the heap. That is, the rank of v is 0, the child list is empty, and the `ckey` is e.

Notice that so far, the inserting, melding, and hence the entire data structure does not differ from binomial heaps, as each node stores exactly one item and no subtrees are removed. Consequently, starting with an empty soft heap and only inserting items, we will have a series of pure heap ordered binomial trees, and no items will be corrupted. This is the same picture as with the Fibonacci heap without any `decreaseKey` nor `delete` operations, right after a `deleteMin` operation, which cleans up and creates a binomial heap.

Introduction to deleteMin

It is trivial to implement a size counter for the soft heap, so it is easy to detect if the heap is empty, and report back if it is. The counter must be increased by one for each insertion, and decreased by one for each `deleteMin` or `delete` on a nonempty heap. Therefore, in the description of `deleteMin`, we assume that the priority queue is nonempty.

The `suffix_min` pointer of the first real head node points to the head node containing the soft queue with the minimum `ckey` among all queues, and hence in the entire heap. Therefore it is easy to locate where the minimum `ckey` "*should*" be. The problem is that the item list of the root node may be empty because we are lazy when we delete an item. This leads to the simple procedure if the item list is nonempty: We simply take the first item in the list (corrupted or not), delete it from the list, and return it. Hence at some point, the list becomes empty.

If the item list of the root is empty, we must refill it with items from nodes deeper in the queue. This leads to the `sift` function, which is the heart of the soft heap and is what distinguishes it from other heap types.

12.5 Sifting

To make the description of the advanced soft heap sifting clearer, let us describe the simpler sifting procedure of a pure binomial queue. To make the procedure as close as possible to that of the soft heap, we do not want to remove the root node from the binomial tree. Instead, we set the key of the root node to ∞ to indicate the item is deleted, and afterwards we sift the new minimum item to the root. Naturally this method preserves the tree size, but it violates the heap order of the tree after setting the root key to ∞.

Notice that the rank of a tree (or subtree) and its root is equal, so we will use the two rank terms interchangeably.

Sifting in a binomial queue

Let $\mathrm{ckey}(v)$ be the key of a binomial node v. Let r be the empty root node and let $k = \mathrm{rank}(r)$. Then we set $\mathrm{ckey}(r) = \infty$ such that it will travel to the bottom of the tree when we reestablish the heap order. Now, one of the child nodes of r must have the minimum key among all keys in the tree, as they are all roots, and thus each of them contains the minimum key of its subtree. Hence, all child nodes are now candidates to become the new root. Then we unlink all child nodes from r, so $\mathrm{rank}(r) = 0$. The old child nodes, or equivalently subtrees, now form k binomial trees with ranks $0, \ldots, k-1$.

The sifting procedure maintains a *current tree*, which it repeatedly relinks with the unlinked subtrees from the initial root. Let the *current root* be the root node of the current tree. Initially, let the single-node tree r be the current tree. Then we repeatedly relink the current tree with the old subtrees in increasing rank order, $0 \ldots k-1$, such that the heap order is preserved. That is to let the root with largest key become a new child of the other root. After each linking, let the newly linked tree be the current tree. When linking, the ranks should be updated, such that the rank of the current root node is increased by one, corresponding to its new number of children. The rank of the current tree is clearly also the same as one plus the rank of the old subtree just linked, because two trees of this rank were linked. When describing soft heap sifting, the latter definition of rank of the current tree (root) will be used. Similarly, the rank of the new child node is the same as the rank of the old subtree just linked. The current tree after linking the last two trees of rank $k-1$ has rank k, and forms the new final binomial tree that preserves the heap order again.

Sifting in a soft queue

The sift procedure of the soft heap is very similar to that of ordinary binomial trees, except two important differences.

Firstly, by the definition of soft queues, some nodes, and thus also the current root node, may be missing some children. Consequently, some child ranks among $0 \ldots k-1$ may not exist. Recall that the rank of a soft queue node is *not* the current number of children, but the number of children in the master tree. Therefore, it does not make sense to increase the rank of the current root by one after each linking, as the rank when the relinking is complete, may differ from the rank in the master tree. Consequently, we need to save the initial root rank before relinking and assign it back to the final root node when the relinking is complete. Similarly, if the current root is relinked such that it becomes a child of an old subtree root, then we need to assign it the rank of the old subtree root. Despite of the rank complications of soft heap relinking, the current tree relinks with all (existing) old subtrees in increasing rank order. As the rank of the current root does not follow the number of children, let us define the *current root rank* as one plus the rank of the old subtree just relinked.

Secondly, the most important difference. Under some conditions, `sift` may call itself after a relinking step, that is a recursion of the `sift` function. Such call saves the item list of the current root, sets its ckey to ∞, and brings further items to the root, which will be merged with the saved item list in the end of the call. As stated above, each linking step has a current root rank as with binomial trees. At the time when a recursive call occurs, some subtrees might not be relinked yet. Hence the current root has only child ranks up to the rank of the child just linked. Consequently, the recursive call relinks subtrees of the current root up to this particular rank. That is a repetition of the relinkings up to the particular rank. The original call to `sift` without any recursive calls, will not bring item lists together at the current root, because its item list were empty at the time when `sift` were called from `deleteMin`. A recursive call to `sift` will clearly bring some items together in the same item list, as the current root node this time has a nonempty item list, which we make unavailable by setting its `ckey` to ∞, indicating that it is ready to be removed. In other words, setting the `ckey` of a nonempty node to ∞ makes the number of available nodes smaller than the number of items, and thus some items must share nodes (item lists) to remain.

The main condition for recursion is that the current root rank is $> r$. This is only possible after the relinking of two trees where the old subtree had rank $\geq r$. This condition ensures that no corrupted items will appear at nodes with rank $\leq r$, as item lists are not merged at nodes with these ranks. In addition, one (or both) of two conditions must hold: 1) The current root rank must be odd, or 2) the current root must have lost the child with the current root rank. The latter condition is similar to that the next subtree (in the sequence of existing subtrees with increasing rank) has a rank that is strictly greater than the current root rank.

Implementation of sift

To make the `sift` function uniform, while supporting both the original non-corrupting and the recursive corrupting calls, we can implement it as follows.

Every execution of `sift` must have a reference to the root of the current tree. Thus, we let the function take a reference to the current root as a parameter. Likewise, we let it return a reference to the current root, such that the caller can update its own reference when the control returns.

Firstly, save the item list and rank of the current root in local variables, then empty its item list and assign ∞ to its `ckey`. Secondly, save the child list, or equivalently subtree list, in yet another temporary variable, and unlink all children from the current root. Now we are ready to relink all children to preserve the heap order. So we iterate through the old subtree list in increasing rank order. For each subtree, (`subtree`), we check if $\text{ckey}\,(\texttt{root}) < \text{ckey}\,(\texttt{subtree})$, where `root` is the current root. If this is the case, we make the node pointed to by `subtree` a new child of the node pointed to by `root`. Otherwise, we need to swap the nodes, by making the node pointed to by `root` a new child of the node pointed to by `subtree`, and then update the rank of the new child (that is the node pointed to by `root`) to the rank of the old subtree. Lastly, we update the current root pointer, `root`, to the node pointed to by `subtree`. There is one important exception to the latter case, namely if $\text{ckey}\,(\texttt{root}) = \infty$. In that case we do not link the two trees, so the node with infinite `ckey` disappears from the structure and is thus regarded as removed. This is the point where the node rank and number of children can be made to differ.

Then we calculate the current root rank, which is $\text{rank}\,(\texttt{root}) = \text{rank}\,(\texttt{subtree}) + 1$. In a pure binomial queue this is the target root rank for the relinking step. Then we check if the recursive condition holds. If so, we call `sift` recursively with `root` as parameter. This call will once again temporary save the item list of the current root, setting its `ckey` to ∞, and relink the current children. As we will see in a moment, this call will finally concatenate the saved item list with that of the resulting root of the call.

When we are done iterating through the subtree list, we assign the initial root rank to the resulting root, and concatenate the item list of the current root with the saved item list. Concatenation of item lists in general also requires an update of the corresponding `ckey`. We always sift the node with smallest `ckey` to the root, so the items sifted to the root the second time, that is in the recursive call, has a larger `ckey` than the saved item list. Hence, we let the `ckey` of the current root remain unchanged. The item list merging is what Chazelle refers to as the "car pooling" of priority queues, because several items (respectively, people) share a common node (respectively, car) towards the root (respectively, job).

To detect if the current root node has lost the child of current root rank, we check if the subtree of the next linking step has a rank strictly greater than the current root rank. To make this work in pseudo code, we add a virtual subtree with the same rank as the initial root of the call to the list of subtrees, such that a subtree always has a next subtree. Obviously, we must only iterate over the real subtrees.

See a formal description of the `sift` function in Algorithm 12.1, and a sifting example in Figure 12.2, which shows four linking states around a recursive call to `sift`.

Algorithm 12.1: `sift( root )` - soft heap sifting

```
rootRank ← rank (root)
L ← itemList(root)
itemList(root) ← ∅
ckey(root) ← ∞
subtrees ← all root's children ∪ virtual tree with rank ← rootRank
Unlink all children from root
foreach subtree in subtrees in increasing rank order up to rootRank − 1 do
    if ckey(subtree) < ckey(root) then
        if ckey(root) is finite then
            Add root as new child to subtree
            rank(root) ← rank(subtree) /* Here, root is temporary not a root */
        root ← subtree
    else
        Add subtree as new child to root
    rank (root) ← rank (subtree) + 1
    nextSubtree ← the next subtree in the rank ordered subtrees sequence
    if rank (root) > r and (rank (root) is odd or rank(nextSubtree) > rank (root))
    then
        root ← sift(root)
rank (root) ← rootRank
itemList(root) ← itemList(root) ∪ L
return root
```

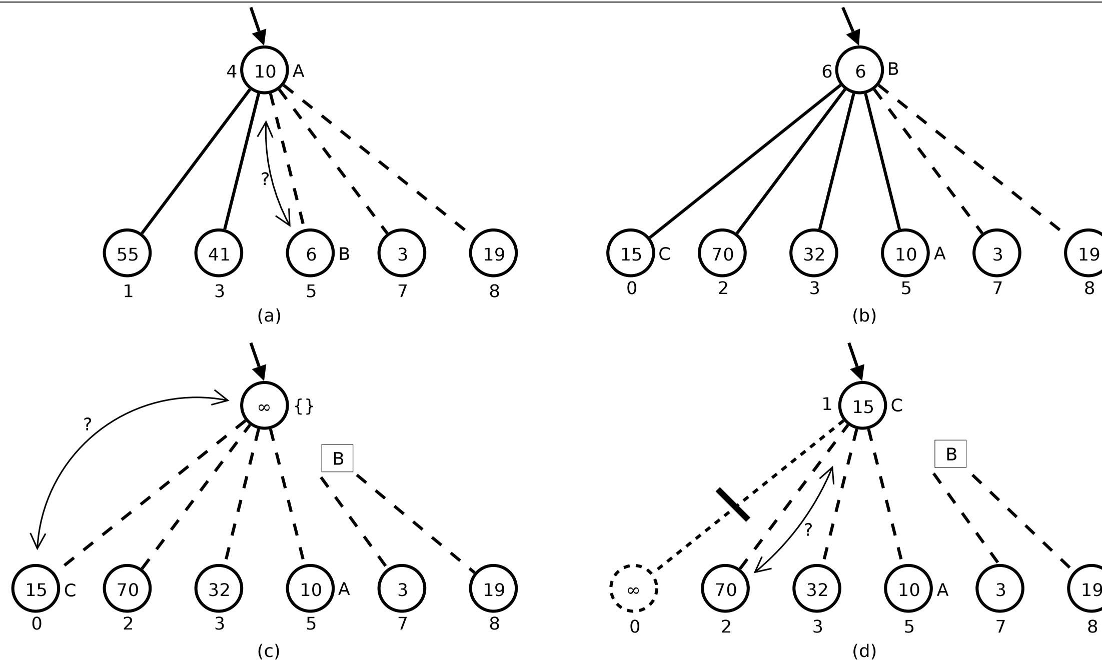


Figure 12.2: Example of sifting, starting in a relinking iteration. Only the current root and subtree roots are shown. Unlinked subtrees are shown with dashed lines to the root. The number inside a node is the `ckey`. The number below a subtree root is the rank. The number next to the root is the current root rank. The letter next to some nodes is the item list. a) 10 is just linked with 41. 10 and 6 is about to be linked. b) 10 and 6 linked. The root has lost the child of current rank 6, so we call recursive. c) Recursive call: The current unlinked subtrees (3 and 19) are unknown to the recursive call. The item list (B) at the root is saved, illustrated by the new box. Then the item list at the root is emptied, the `ckey` is set to ∞, and the current children are unlinked. The root and 15 is about to be linked. d) 15 and ∞ not linked, but ∞ is removed. The next linking step will involve 15 and 70. Assuming no further recursive calls occur, this recursive call will end by swapping the root (15) and 10, making 10 the new root, and merging the items lists A and B.

12.6 DeleteMin continued

As stated in the introduction to `deleteMin`, the advanced part occurs when the item list of the root with minimum `ckey` is empty. In that case, we first check if the rank invariant is violated. If so, it is a consequence of previously sifting, as it is the only function that removes subtrees. Then we simply dismantle the root node, by removing its head node from the head list. This may induce invalid `suffix_min` pointers between the previous head node and the first head node, so we call `fix_minlist` on the previous head node to update them. Afterwards we meld the remaining children back into the heap. The advantage of postponing the rank invariant detection until the root node is empty, is precisely that it is empty. Otherwise, we would stay with a nonempty item list which does not belong to any queue node, so what would we do with it? Conversely, if the rank invariant holds, we call `sift` to move items towards the root of the queue. Recall that the root of the queue node may change during this call, and most likely does. Therefore we update the `queue` root pointer of the head to that returned by `sift`. If the entire queue

Algorithm 12.2: `deleteMin()` - soft heap deleteMin

```
h ← suffix_min of the first head
root ← queue(h)
while itemList(root) is empty do
    if deg(root) < $\lfloor$rank(root)/2$\rfloor$ then
        Remove h from the head list
        fix_minlist(prev(h))
        foreach child of root do
            meld(child)
    else
        root ← sift(root)
        if ckey(root) = $\infty$ then
            Remove h from the head list
            h ← prev(h)
        fix_minlist(h)
    h ← suffix_min of the first head
    root ← queue(h)
min ← first item in itemList(root)
Remove first item in itemList(root)
return min
```

was empty, then the `ckey` of the root will be infinite after the call. If so, we dismantle the node by removing its head node from the head list. Afterwards we call `fix_minlist` to restore `suffix_min` pointers. If the head node was removed, we call it with the previous head node as parameter, otherwise with the head node itself.

After the `meld` or `sift` action, we once again find the root with the minimum `ckey` among all roots by following the `suffix_min` pointer of the first real head node. If the item list of this root is also empty, then we repeat the procedure until we find a nonempty root. When the item list is nonempty, we perform the simple action of extracting the first item, and returning it.

See a formal description of the `deleteMin` operation in Algorithm 12.2.

A findMin operation

A `findMin` operation follows immediately from the `deleteMin` operation. The only difference between the two is the deletion of the first item in the item list.

12.7 The r function

To achieve the desired error rate and running time, we set the value of r to the following function:

$$r \overset{\text{def}}{=} 2 + 2\left\lceil \log \frac{1}{\varepsilon} \right\rceil . \tag{12.1}$$

The proof that this function achieves the desired error rate and running times will follow in the analysis of the succeeding sections. It is important to notice that r is always an even integer.

12.8 Corrupted items analysis

Lemma 12.2. *Let v be a node in a soft heap. The size of v's item list is bounded by*

$$\max\left\{1, 2^{\lceil(\mathrm{rank}(v))/2\rceil - r/2}\right\} .$$

Proof. If the lemma holds, then it easy to verify that nodes v with $\mathrm{rank}(v) \leq r$ have maximum list size of 1, as opposed to nodes with $\mathrm{rank}(v) > r$, which have a greater bound.

Before the first call to `sift`, all queues are still pure binomial trees, and thus all item lists have size 1. Hence, the lemma holds for the base case. If `sift` does not call itself, then some item list is sifted (relinked) to a node of higher ranking, and thus the lemma still holds, because this will only raise the value of the second parameter. Otherwise, that is if sift calls itself, then the item list at the root becomes the concatenation of two item lists, which both are derived from child nodes of particular ranks. Thus, we must find an upper bound on the child node item list size. The rank of all children is strictly less than the current root rank. Merging happens only if the current root rank is $> r$, and if either it is odd, or no old subtree exists with rank of the current root rank. A recursive call will just repeat the procedure of sifting items to the current root. Hence, the following inductive bounds on the maximum child node list size, is the same for the saved item list and the one brought to the root by the recursive call.

The induction assumption is that the inequality holds for the children. We have two inductive cases:

- The current root rank is odd: A lower bound on the real root rank is the current root rank, so let $\mathrm{rank}(v)$ be the current root rank. The rank of a child is at most $\mathrm{rank}(v) - 1$. Due to the main recursion condition, we know that $\mathrm{rank}(v) \geq r + 1$, so

  $$2^{\lceil(\mathrm{rank}(v)-1)/2\rceil - r/2} \geq 2^{\lceil r/2\rceil - r/2} \geq 2^0 = 1 .$$

  Consequently, we can eliminate the max term, and stick to its last parameter in the following derivation. By the induction assumption and that $\mathrm{rank}(v)$ is odd[2], the size of an item list at a child of v is at most

  $$\max\left\{1, 2^{\lceil(\mathrm{rank}(v)-1)/2\rceil - r/2}\right\} = 2^{\lceil(\mathrm{rank}(v)-1)/2\rceil - r/2} = 2^{\lceil\mathrm{rank}(v)/2\rceil - 1 - r/2} .$$

- The current root has no child with current root rank and this rank is even: If a node has lost a child, then the node itself must have a rank greater than the rank of this child. Hence a lower bound on the real root rank is the current root rank plus one,

[2]Consequently $\mathrm{rank}(v) - 1$ is even.

so let $\operatorname{rank}(v)$ be the current root rank plus one. Thus, the rank of a child is at most $\operatorname{rank}(v) - 2$. Due to the main recursion condition, we know that $\operatorname{rank}(v) \geq r + 2$, so

$$2^{\lceil (\operatorname{rank}(v)-2)/2 \rceil - r/2} \geq 2^{\lceil r/2 \rceil - r/2} \geq 2^0 = 1 \ .$$

Consequently, we can eliminate the max term, and stick to its last parameter in the following derivation. By the induction assumption the size of an item list at a child of v is at most

$$\max\left\{1, 2^{\lceil (\operatorname{rank}(v)-2)/2 \rceil - r/2}\right\} = 2^{\lceil (\operatorname{rank}(v)-2)/2 \rceil - r/2} = 2^{\lceil \operatorname{rank}(v)/2 \rceil - 1 - r/2} \ .$$

The analysis of the case where the current root rank is odd and the current root has lost a child with this rank is irrelevant, because it is covered by the first case.

It turns out that the two upper bounds on child item list sizes are equal. The concatenation of two lists with this size (the saved and the one from the recursive call) has size at most

$$2 \cdot 2^{\lceil \operatorname{rank}(v)/2 \rceil - 1 - r/2} = 2^{\lceil \operatorname{rank}(v)/2 \rceil - r/2} \ ,$$

which proves the lemma. □

Lemma 12.3. *Let S be the node set of a binomial tree. Then*

$$\sum_{v \in S} 2^{\operatorname{rank}(v)/2} \leq 4\,|S| \ .$$

Proof. Recall that a binomial tree of rank k has 2^k nodes. Hence, the rank of a tree with node set S is $\log|S|$. Besides the root node of rank $\log|S|$, for each rank $k \in [0\ ,\ \log|S| - 1]$, a binomial tree has exactly $|S|/2^{k+1}$ nodes. Hence a tree has $|S|/2$ leafs (rank 0 nodes) and $|S|/2^{k+1}$ nodes with rank strictly greater than k including the root.

We know the number of nodes with each possible rank in a tree of $|S|$ nodes, so we can rewrite the sum:

$$\begin{aligned}
\sum_{v \in S} 2^{\operatorname{rank}(v)/2} &= \sum_{i=0}^{\log|S|-1} \frac{|S|}{2^{i+1}} \cdot 2^{i/2} + 1 \cdot 2^{\log|S|/2} \\
&= |S| \sum_{i=0}^{\log|S|-1} \frac{2^{i/2}}{2^{i+1}} + |S|^{1/2} \\
&\leq |S| \sum_{i=0}^{\infty} \frac{1}{2^{i/2+1}} + |S|^{1/2} \\
&= |S| \left(\sum_{i=1}^{\infty} \frac{1}{2^i} + \sum_{i=1}^{\infty} \frac{1}{2^{i+1/2}} \right) + |S|^{1/2} \\
&\leq |S|\,(1+1) + |S|^{1/2} \\
&\leq 4\,|S| \ ,
\end{aligned}$$

which proves the lemma. □

Lemma 12.4. *The soft heap contains at most $n/2^{r-3}$ corrupted items at any given time.*

Proof. Let q be a soft queue, and let q' be the corresponding master tree. Let R be the set of nodes with rank $> r$ in q, and let R' be the set of nodes with rank $> r$ in q'. As q is derived from q' by only removing subtrees, q' can not contain more nodes than q with any rank, hence $|R| \le |R'|$. We know from the proof of Lemma 12.3 that $|R'|$ is $1/2^{r+1}$ of the queue size. The total size of all queues is at most n. Thus, the sum of nodes with rank $> r$ in all queues is

$$\sum_{q'} |R'_{q'}| \le \frac{n}{2^{r+1}} \le \frac{n}{2^r} . \tag{12.2}$$

From Lemma 12.2 we know that the maximum list size at a node v with rank $(v) > r$ is

$$2^{\lceil \text{rank}(v)/2 \rceil - r/2} \le 2 \cdot 2^{(\text{rank}(v)-r-1)/2} .$$

Consequently, there is at most $2 \cdot 2^{(\text{rank}(v)-r-1)/2}$ corrupted items at nodes with rank $(v) > r$, and 0 in nodes with rank $(v) \le r$. An upper bound on the total number of corrupted items, is the sum of maximum list sizes at nodes with rank $> r$ in the master trees:

$$\sum_{q'} \sum_{v \in R'_{q'}} 2 \cdot 2^{(\text{rank}(v)-r-1)/2} = 2 \sum_{q'} \sum_{v \in R'_{q'}} 2^{(\text{rank}(v)-r-1)/2} . \tag{12.3}$$

Let $S_{q'}$ be the nodes of $R'_{q'}$. With reference to Figure 12.3, $S_{q'}$ forms a binomial tree, where the rank of a node $v \in R'_{q'}$ becomes rank $(v) - r - 1$ in $S_{q'}$.

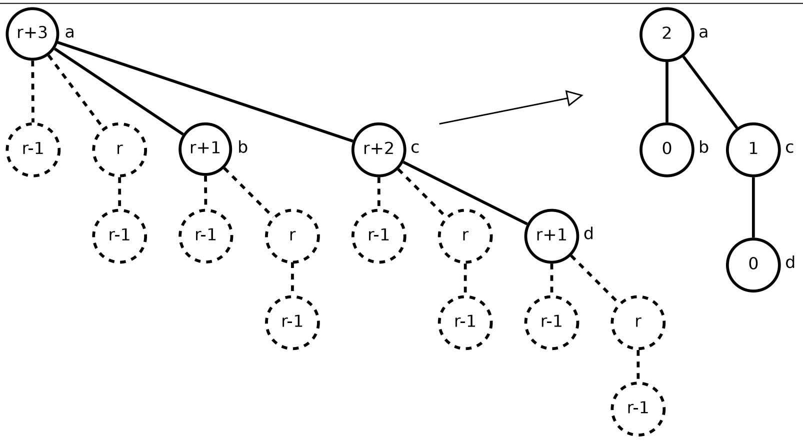


Figure 12.3: To the left: A binomial tree, q', with node ranks. The nodes in $R'_{q'}$ have solid lines. To the right: The binomial tree formed by nodes in $S_{q'}$ with node ranks.

Hence, by Lemma 12.3 we can bound Equation 12.3 to

$$2 \sum_{q'} \sum_{v \in S_{q'}} 2^{(\text{rank}(v))/2} \le 2 \sum_{q'} 4 \left|S_{q'}\right| = 8 \sum_{q'} \left|S_{q'}\right| = 8 \sum_{q'} \left|R'_{q'}\right| .$$

From Equation 12.2 we have an upper bound on the sum of nodes with rank $> r$ in all queues, so an upper bound on the number of corrupted items is

$$8 \sum_{q'} \left|R'_{q'}\right| \le 8 \frac{n}{2^r} = \frac{n}{2^{r-3}} ,$$

which completes the proof. $\square$

Theorem 12.5. *The soft heap contains at most εn corrupted items at any given time.*

Proof. Inserting the value of r from Equation 12.1 in the bound from Lemma 12.4 gives

$$\begin{aligned}
\frac{n}{2^{r-3}} &= \frac{n}{2^{(2+2\lceil \log 1/\varepsilon \rceil)-3}} \\
&= \frac{n}{2^{2\lceil -\log \varepsilon \rceil -1}} \\
&= 2^{1-2\lceil -\log \varepsilon \rceil} n \\
&= 2^{1+2\lfloor \log \varepsilon \rfloor} n && (\text{as } \lceil -x \rceil = -\lfloor x \rfloor \text{ for any } x) \\
&\leq 2^{1+2\log \varepsilon} n \\
&= 2\varepsilon^2 n \\
&\leq \varepsilon n \ , && (\text{as } 0 < \varepsilon < 1/2)
\end{aligned}$$

which completes the proof, and partially proves Theorem 12.1 on page 43. □

12.9 Running times analysis

Finding corrupted items

The running time for finding corrupted items is clearly proportional to the number of items in the heap. This number it naturally less than or equal to n. So given that we only search for corrupted items a constant number of times in the same heap, the cost can be charged to the insertions.

Initial observations of insert and deleteMin

Except for the melding, it is easy to verify that the `insert` operation runs in constant time, as it only creates one new simple queue of one node. With regard to the `deleteMin` operation, it is also easy to verify that, except for the loop, it runs in constant time, as it takes constant time to follow pointers and extract an item from a linked list. The running time of the loop is dominated by `meld`, `sift`, and `fix_minlist`. Hence we need to analyse the running time for these functions.

Correlation between deleteMin and findMin

The two operations basically do the same work except for deleting the minimum element, so `findMin` runs in constant time except for the loop. It is easy to verify that the total number of meldings and siftings induced by the two operations depends only on the deleteMins. Consequently, we can charge the running time of `findMin` to `deleteMin`, and we can restrict ourself to analyse the running time of `deleteMin`.

Updating of `suffix_min` pointers

The cost of walking in the head list from a head node of rank k to the header node, while updating `suffix_min` pointers, is clearly $O(k+1)$. Hence, the running time of `fix_minlist` is $O(k+1)$, where k is the rank of the starting head.

Meldings

Like binomial trees, we assign one cyber-dollar to each queue. The cost of melding a queue q into the heap is $\text{rank}(q)+1$ cyber-dollars. The first $\text{rank}(q)$ cyber-dollars are used to find the position for the new queue, that is the minimum i such that $\text{rank}(h_i) \geq \text{rank}(q)$. The last cyber-dollar is assigned to the new queue. If there are two queues of the same rank, we release their two cyber-dollars, spend one for the constant linking cost, and assign the other to the new queue. The cost of the succeeding call to `fix_minlist` does not exceed the cost of finding the position and the linking, because it visits no more head nodes than `meld`. As a matter of fact some of them might be removed in the linking steps. Hence, a general amortised running time of queue `meld` is $O(\text{rank}(q)+1)$.

So with regard to the `insert` operation, the amortised cost is constant because it calls `meld` with a queue of rank 0.

Melds induced by deleteMin

When `deleteMin` detects an empty queue root node v, where the number of children violates the rank invariant ($< \lfloor \text{rank}(v)/2 \rfloor$ children), it remelds the remaining children into the heap, and dismantles the root. At the time when the children existed, their ranks were $0, \ldots, \text{rank}(v)-1$. When at least $\lfloor \text{rank}(v)/2 \rfloor + 1$ children are lost, there must be least one of these which had rank $\geq \lceil \text{rank}(v)/2 \rceil$. In the master tree, this particular lost child was a root in a subtree of at least $2^{\lceil \text{rank}(v)/2 \rceil}$ nodes. The cost of remelding the children back into the heap, is the sum of their ranks plus the number of children. Thus the worst case of remaining children is those with highest ranks. Hence, the total sum of ranks of the remaining children is at most

$$\sum_{k=\lceil \text{rank}(v)/2 \rceil+1}^{\text{rank}(v)-1} k \leq \sum_{k=0}^{\text{rank}(v)} k = \frac{\text{rank}(v)^2 + \text{rank}(v)}{2}$$

which is $O\left(\text{rank}(v)^2\right)$. In addition comes the number of children, which is clearly dominated by this sum.

The number of removed subtree nodes is asymptotic greater than worst case remelding cost. In other words, the worst case remelding sum is $O\left(2^{\lceil \text{rank}(v)/2 \rceil}\right)$. As a matter of fact the two functions differ by at most 3 in the interval where the sum of ranks dominates. This implies that the remelding cost is bounded by a constant for each previously removed node in the subtree. Consequently, we can charge the melds induced by `deleteMin` to these nodes, as they will not be charged again. In other words, the remelding has already been paid for by the original insertions of these removed nodes.

In addition to the cost of the actual melding, comes the cost of the preceding call to `fix_minlist` in `deleteMin`. The cost of this call is the rank of the dismantled root node. The rank of the highest ranked child is one less, so the cost of `fix_minlist` has already been paid for.

We have shown that the total cost of melding is charged to the `insert` operation, which is of constant cost per item. Hence the total running time of `meld` after n insertions is $O(n)$.

Sifting

It is clear that, except for the recursive call, each iteration of the loop takes constant time. If a recursive call takes constant time, we charge it to the current iteration. The time for saving the child list and unlinking children is charged to the very same children in the iteration. The remaining parts of `sift` clearly run in constant time. So we want to find the total number of iterations, or equivalently the total number of relinkings.

Let C be the total number of recursive calls to `sift`, which do not take constant time. This is equivalent to the number of times the recursion condition holds and the recursive call does not take constant time. Consider a sequence of consecutive current root ranks $> r$ encountered in a call to `sift`. Recall that this sequence is strictly increasing. Then consider a subsequence of size two, and let k be the first rank. If k is odd, then the recursion condition holds. Conversely, if k is even, but the second current root rank is $k+1$, then the second rank is odd. If the second current root rank is $> k+1$, then the recursion condition held at the first current root rank. This is because after the first linking, the next subtree (that is the second in the subsequence) had rank $> k+1$, and thus the next child were missing. Thus in any such subsequence of size two, at least one recursive call must occur.

The maximum number of linkings starting from a subtree of rank $< r$ to a subtree of rank r is $r+1$, as the smallest possible rank is 0 and `sift` does not recurse when the iteration rank is $\leq r$. We have just shown that two consecutive iteration ranks $> r$ implies at least one recursion call. Hence for each current root rank sequence of size $r + O(1)$, or equivalently $O(r)$, at least one recursive call occurs. This implies there is at most one sequence of size $O(r)$ per recursive call. A linking of $O(r)$ subtrees takes $O(r)$ time. Hence the running time is $O(rC)$.

If a root node has no subtrees, because their `ckeys` were set to ∞, then a recursive call to `sift` takes constant time. Conversely, if a root node has at least one subtree, then a recursive call induces a concatenation of two nonempty item lists. Beginning with n item lists, each with a single item, the maximum number of possible merges is $n-1$. Hence, the number of non-constant time recursive calls is $C \leq n$. Consequently, the total running time for sifting after n insertions is $O(rn)$.

In addition to the cost of the actual sifting, comes the cost of the succeeding call to `fix_minlist` in `deleteMin`. The cost of this call is at most the rank the root node. Sifting is only performed if the rank invariant holds. Hence, the root had at least $\lfloor \text{rank}(v)/2 \rfloor$ children before the sifting, which implies that `sift` has performed at least as many linkings. Consequently, the cost of the call to `fix_minlist` is bounded by a constant for each of these relinkings. In other words, the call has already been paid for by the relinkings.

Insert and deleteMin

Summing up, the amortised running time of n inserts without other heap operations is $O(n)$, or equivalently $O(1)$ per insertion. This is not very interesting, so let us calculate the general running time.

Theorem 12.6. *Consider a soft heap with fixed error rate $0 < \varepsilon < 1/2$ and no prior data. The amortised running time of `insert` is $O(\log 1/\varepsilon)$. The amortised running time of `deleteMin` is $O(1)$.*

Proof. As $\varepsilon < 1/2$, inserting the value of r in the running time for `sift` after n inserts, gives

$$O(rn) = O((2 + 2\lceil \log 1/\varepsilon \rceil) n) = O((\log 1/\varepsilon) n) \ .$$

The total running time for `meld` after n inserts is $O(n)$, so `sift` dominates the total running time after n inserts. Hence the amortised cost for each inserted item during its lifetime is $O(\log 1/\varepsilon)$, which we charge to the `insert` operation. Consequently, the amortised running time of `deleteMin` is $O(1)$, because the `insert` operation has paid for the deleted item during its lifetime. This theorem partially proves Theorem 12.1 on page 43. □

12.10 Implementation

As stated in Section 12.2, Chazelle suggests to represent degree-k nodes as a sequence of degree-2 nodes, with a `next` and a `child` pointer, respectively. This way to view and represent (modified) binomial tree structures is complicated, as the correspondence to the standard way of viewing binomial trees is not always very intuitive. In particular, the correspondence is not very clear for the `sift` function. Nevertheless, Chazelle's paper [Cha00b] includes source code for most soft heap functions, where degree-k nodes are represented as a sequence of degree-2 nodes. This source code is used as base for the implementation of the soft heap. We will briefly describe the implementation.

If a node has no children, that is a leaf node, then both pointers are `null`. Otherwise `child` points to a child node, and `next` points to the next degree-2 node in the sequence.

A trivial rank-0 tree is represented as a single node with both pointers set to `null`. When two soft queues of rank $k-1$ are linked, one of the root nodes becomes the parent (top) of the other (bottom), and hence the rank of the top node is increased by one. In Chazelle's implementation, a new degree-2 node is created with rank k. This new node shows the parent-child correspondence between the two old roots, and are now considered as the root node of the resulting tree. The `child` pointer of the node is set to the bottom node and the `next` pointer is set to the top node. The rank of the `next` node remains unchanged and thus becomes a holder for a child of the root node. The new node is assigned the `ckey` and item list of the `next` node. A linking example is given in Figure 12.4. If we want to unlink the two queues, then we just remove the this node, and afterwards the `child` and `next` nodes are root nodes of two individual soft queues again.

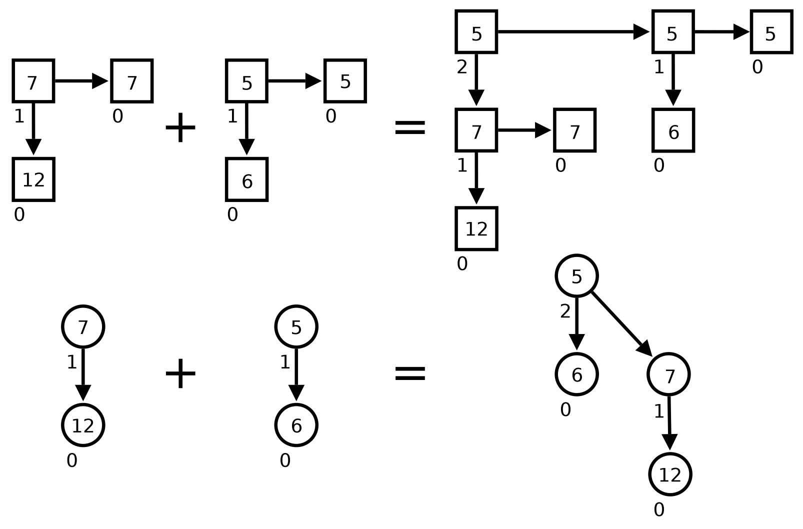


Figure 12.4: Linking of two binomial queues with rank 1. The number inside a node is the ckey. The number below a node is the rank. Top: Chazelle's representation. Bottom: Standard view.

Sift

In Chazelle's implementation [Cha00b], there is no explicit subtree unlinking and relinking loop in the `sift` operation. Instead, `sift` is defined as a recursive constant time function, which repeatedly calls itself with the `next` node as current root node.

The execution of `sift` begins by clearing the item list of the current node. If the node is a leaf (both pointers are `null`), then its `ckey` is set to ∞ and `sift` returns. As we only follow `next` pointers, this node is actually not a leaf in the soft queue, but the last in a sequence of many degree-2 nodes representing the root. The leaf procedure corresponds to the part of our `sift` function before the loop.

If the node is not a leaf, then it calls recursively with `next` as parameter. When the control returns from a recursive call, it checks if ckey (`next`) > ckey (`child`). If so, it exchanges the `next` and `child` pointers to preserve the heap order. The item list of the current root node is by now stored at `next`. Hence, the item list of `next` is saved in a local variable L. If the recursive conditions holds, then `sift` is called again with `next` as parameter. Afterwards, the heap order check is performed again, and the item list of `next` is concatenated with that of L. This relinking procedure corresponds to one linking iteration in our version of `sift`. The heap order check after a potential second recursive call, is performed in the last linking step before the call returns in our version of `sift`.

The execution of `sift` ends by assigning L to the item list of the current node and cleaning up nodes with rank ∞: It checks if ckey (`child`) $= \infty$ and ckey (`next`) $= \infty$. If so, then both references are set to `null`, to indicate that the node has no children. Otherwise, if ckey (`child`) $= \infty$ and ckey (`next`) $\neq \infty$, then the `child` pointer is set to the `child` pointer of `next`, and the `next` pointer is set to the `next` pointer of `next`. The list assigning corresponds to the part of our version of `sift` right after the loop. The cleanup procedure corresponds to our version `sift` where we check if ckey (`root`) $\neq \infty$.

With this implementation the recursion condition is as follows: The rank of the current node must be $> r$ and either 1) the current node rank is odd, or 2) the rank of `child` is strictly less than the current root rank minus one.

12.11 Additional heap operations

The optimal MST algorithm does not need to utilise soft heap melding[3] nor deletion. Anyway, we will give a brief description of them here. If the soft heap had a `decreaseKey` operation, then the algorithm could utilise it. It follows why this operation is hard to implement.

Melding of two soft heaps

We have already described a function for melding a single soft queue into a soft heap, so melding of two soft heaps is trivial: Iterate forward through both head lists simultaneously, while moving queues from the heap of smaller rank into the heap of larger rank. If both heaps have a queue of equal rank, say k, then a conflict arises, and we link them resulting in a queue of rank $k+1$. If one heap contains a queue of rank $k+1$, then we link again. If both heaps contain a queue of rank $k+1$, then we have three queues with equal rank. If so, we leave the queue from the heap of larger rank in the head list, and link the queue from the heap of smaller rank with the third queue. With this method, we always temporary have maximum three queues of equal rank. The resulting soft heap of larger rank will at most increase its rank by one. When the process stops, we call `fix_minlist` with the newest head node as parameter.

The time for all linkings is clearly amortised constant, as each queue has a cyber-dollar to pay for linkings. But the time for iterating through the head lists is order the rank of the heap with smallest rank. This can easily be fixed, by increasing the potential function of a soft heap from the number of queues, say m, to $m + k$, where k is the heap rank. Then the $m + k$ cyber-dollars of the heap with smaller rank (plus a constant if the rank of the heap of larger rank increases by one) will pay for the melding. As $k \leq n$, the changing of the potential function requires the `insert` operation to pay two cyber-dollars, which is still a constant. Hence, the heap meld operation runs in amortised constant time.

Delete

Due to the nature of heap ordered trees, an efficient implementation of the `delete` operation requires a reference to the item list node where the item is stored. Consequently, the `insert` operation must return the item list node, where the new item is stored. When the item list node is located, we remove it from the item list, which must be a double linked list to support removes in constant time. Alternatively, we can lazily mark the item as deleted. The delete operation may result in an empty list or a list where the `ckey` does not reflect any of the item list keys. If the item list becomes empty, the heap node becomes

[3]Melding of two soft heaps, as opposed to melding of a queue into a soft heap.

an unused node, but it will not affect the heap running times, because they depend on the number of insertions. If the `ckey` does not reflect any item keys, it is still an upper bound on the keys, which were already corrupted. As a matter of fact, both scenarios are also possible for a root node after a `deleteMin` operation. The operation clearly runs in constant time.

DecreaseKey

Like `delete`, an efficient implementation of the `decreaseKey` operation requires a reference to the item list node where the item is stored. For a corrupted item list, it is straightforward to decrease the key of the item, as the `ckey` of the heap node is an upper bound on the item keys. The problem arises for the single item uncorrupted item lists, and in particularly those with rank $\leq r$. Decreasing the key of an item will corrupt the item, unless the `ckey` is decreased similarly. Decreasing the `ckey` of the queue node requires references from each item list node to its queue node. Consequently, these must be updated each time we merge two item lists. A new problem arises if the decreased `ckey` is less than the `ckey` of its parent. Hence, we need to reconfigure the queue to preserve the heap order. This requires a reference from each queue node to its parent, as the affected node must move upwards in the heap. In sum, this method requires a huge modification of the data structure, and a lot of time.

Another, and simpler method, is to let `decreaseKey` call `delete` followed by `insert`. This is a standard simple artifice to implement the `decreaseKey` priority queue operation. With soft heaps this method has some consequences. Let m denote the number of `decreaseKey` operations performed, and still let n denote the number of “real” `insert` operations performed. Firstly, the total number of insertions gets increased for each `decreaseKey` operation, resulting with the upper bound of $\varepsilon\,(n+m)$ corrupted items in the heap at any given time. Secondly, the total running time of `meld` and `sift`, respectively, will become $O\,(n+m)$ and $O\,(r\,(n+m))$, respectively. Consequently, we can not easily deduce a nice amortised running time for the `insert` operation, like we could without `decreaseKey`.

Whichever method we use, it totally breaks apart our analysis of the soft heap, and it makes the new analysis even more complicated. This may be the reason, that Chazelle does not describe a soft heap `decreaseKey` operation in his paper [Cha00b].

The optimal MST algorithm uses a modified version of the DJP algorithm in Chapter 9, which can be implemented with or without utilising the `decreaseKey` priority queue operation. To minimise the running time, this part of the algorithm utilises a soft heap. Due to the various complications just described, we have not implemented a soft heap `decreaseKey` operation, and thus the algorithm performs the modified DJP algorithm without it.

Part IV

The optimal MST algorithm

13 Graph representation

Before describing the actual optimal MST algorithm, we will give a brief overview of how we have implemented our graphs. Chapter 16 gives a description of how the actual algorithm is implemented.

As we will describe here, graphs, vertices, and edges are implemented as classes.

A graph object has the following fields associated:
`( V, E, incidentEdges, n, m )`.
The `V` field is an array of vertices in the graph, and the `E` field is an array of edges in the graph. The `incidentEdges` field is an array of indices in `E`. Each vertex has a pointer associated to a position in this array, which serves as the incidence array for the vertex. Thus, the array is a composite incidence array for all vertices in the graph. The `n` and `m` fields are the number of vertices n and number of edges m, respectively.

A vertex object has the following fields associated:
`( edges, component, degree )`.
The `edges` field is a pointer to a position in the `incidentEdges` array of its graph. The `component` number are used when detecting connected components in a graph. The `degree` field is the degree of the vertex, and thus the number of incident edge indices belonging to the vertex in `incidentEdges`.

An edge object has the following fields associated:
`( a, b, w, inMST, isRemoved, original )`.
The `a` and `b` fields are the indices of its two endpoints in the `V` array of its graph. The `w` field is the weight of the edge. The `inMST` and `isRemoved` fields are two boolean values telling if the edge has been marked as a MST edge or as removed, respectively. The `original` field is a pointer to an edge. If the graph is derived from another graph by contraction, then this is a pointer to the corresponding edge in the original graph. Otherwise, it is a pointer to the edge itself.

Thus, if we have graph object, then we can access all its vertices and edges through `V` and `E`, respectively. The fields `n` and `m` store the size of these arrays. If we have a vertex object `v` together with its graph object `G`, then we can access `v`'s incident edges through `G.E[ v.edges[` $0,\dots,$`v.degree`-1 `] ]`. If we have an edge object `e` together with its graph object `G`, then we can access `e`'s endpoint vertices through `G.V[ e.a ]` and `G.V[ e.b ]`. Each such access clearly takes constant time.

The array implementation of $V(G)$, $E(G)$ and the composite incidence list, are clearly inflexible, as we can not remove (add) an edge or vertex from (to) a graph, without rebuilding the graph. But this is not a major issue in the context of the optimal MST algorithm, because it only needs to alter a graph when contracting it. Contraction takes linear time in the size of the input graph, so we can afford to build a new (and smaller) graph.

Assuming we know the graph size when initialising the graph, the array implementation clearly requires less space (a constant factor) than a linked list implementation. Furthermore, an array implementation requires fewer memory allocations and deallocations, while data in an array are guaranteed to stay close in memory, as opposed to flexible linked lists with many small memory allocations from (more or less) arbitrary locations in memory.

One drawback when contracting is that we need to have two graphs in memory simultaneously. But the input graph to a MST algorithm must stay in memory anyway if we need to access it afterwards, for example to present the MST of the graph, or as input to another MST algorithm during a running time experiment. However, after a contraction step we can remove the input graph if it is unnecessary.

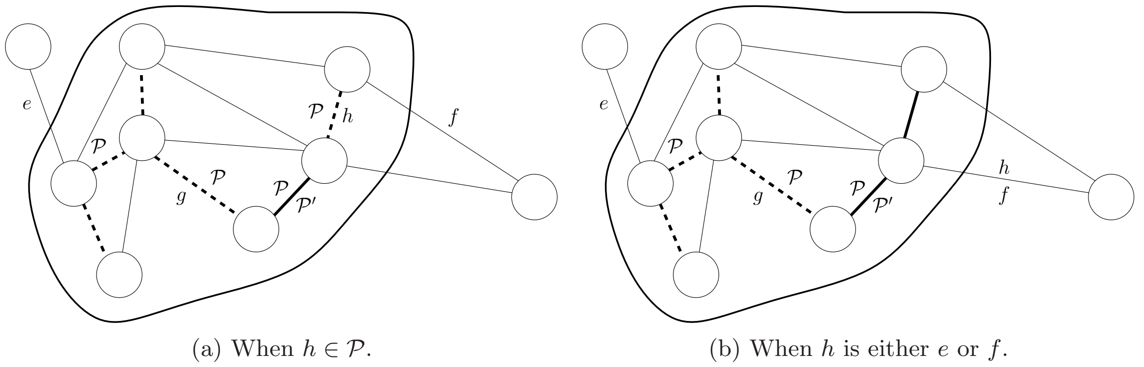
14 Key lemma and procedure

In this chapter we will present a key lemma and deduce a theorem, which is utilised by the optimal MST algorithm. Additionally, in Section 14.2 and Section 14.3, we will present the key procedure of the optimal MST algorithm and state its properties as a lemma. The optimal algorithm is finally presented in Chapter 15.

14.1 Key lemma

Before we state the key lemma for the optimal algorithm, we need to state a lemma about the DJP algorithm in Chapter 9.

Lemma 14.1. *Let T be the tree formed after the execution of some number of steps of the DJP algorithm. Let e and f be two arbitrary edges, each with exactly one endpoint in T. Let g be the heaviest edge on the path from e to f in T. Then, $w(g) \leq \max\{w(e), w(f)\}$.*



(a) When $h \in \mathcal{P}$. (b) When h is either e or f.

Figure 14.1: Proof of Lemma 14.1. Edges in T are fat. Edges in T' are solid. Edges in $(T - T')$ are dashed.

Proof. Let $\mathcal{P} \subseteq T$ be the path in T from e to f. Assume the contrary, namely that g is the heaviest edge in $\mathcal{P} \cup \{e, f\}$. Consider the step where g is selected by DJP and let T' be the present portion of the tree T. Let $\mathcal{P}'$ be the present portion of $\mathcal{P}$, that is $\mathcal{P}' = T' \cap \mathcal{P}$. At this point there is exactly two edges in the set $(\mathcal{P} - \mathcal{P}') \cup \{e, f\}$ that are eligible to be selected. One of these is g in one end of $\mathcal{P}'$. Let h be the other edge, which is in the opposite end of $\mathcal{P}'$. If $h \in \mathcal{P}$, then by the definition of g, it must be lighter than g. If h

is either e or f, then by our assumption, it must be lighter than g. In both cases g could not be selected, which is a contradiction. See Figure 14.1 for an illustration. □

The optimal MST algorithm finds a tree of some MST edges in a *corrupted* graph, where some edge weights have been raised due the use of the soft heap. In particular the algorithm is using some number of steps of the DJP algorithm to find this tree. We will state a formal definition regarding these corrupted edges.

Definition Let G be a graph, and let M be a subset of the edges in $E(G)$. A graph derived from G by raising the weight of each edge in M by an arbitrary amount is denoted $G \Uparrow M$. The edges in M are called *corrupted*.

Notice the important point, that the real weight of edges in M are not raised, it is only their keys in the soft heap. Thus the graph itself will not be altered by the soft heap.

Definition Let C be a subgraph, and let M be a set of edges. The subset of edges in M with exactly one endpoint in C is denoted M_C.

Definition Let G be a graph and M_C be a set of edges. The graph derived from G by removing edges in M_C is denoted $G - M_C$.

Definition Consider an execution of some steps of the DJP algorithm and let T be the resulting tree formed. Let C be the subgraph of G induced by T. Then C is said to be *DJP-contractible* with respect to G.

We are now ready to state the key lemma:

Lemma 14.2. *Let M be a subset of the edges in a graph G. If C is a subgraph of G that is DJP-contractible with respect to $G \Uparrow M$, then* $\mathrm{MSF}(G)$ *is a subset of*

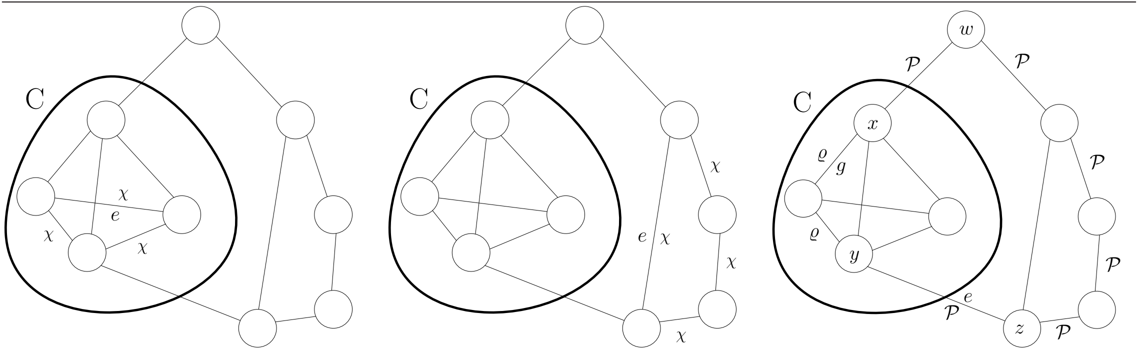
$$\mathrm{MSF}(C) \cup \mathrm{MSF}(G \setminus C - M_C) \cup M_C \ .$$

Proof. We will prove for any edge $e \in G$, where $e \notin \mathrm{MSF}(C) \cup \mathrm{MSF}(G \setminus C - M_C) \cup M_C$ implies that $e \notin \mathrm{MSF}(G)$. Referring to Theorem 4.3, if an edge $e \notin \mathrm{MSF}(G')$ for any graph G', then it is the heaviest edge in some cycle in G'. Notice that the edge sets $E(C)$, $E(G \setminus C - M_C)$, and M_C are pairwise disjoint and their union is $E(G)$. Let $H = G \setminus C - M_C$. Hence we need to show for edges $e \in C - \mathrm{MSF}(C)$ and edges $e \in H - \mathrm{MSF}(H)$, that $e \notin \mathrm{MSF}(G)$.

If $e \in C - \mathrm{MSF}(C)$, then e is the heaviest edge in a cycle χ in C, and so it is in G. Hence $e \notin \mathrm{MSF}(G)$. See Figure 14.2a.

If $e \in H - \mathrm{MSF}(H)$, then e is the heaviest edge in a cycle χ in H. We have two cases. The simplest case is when χ *does not* involve the vertex derived by contracting the subgraph C. Then the same cycle exists in G, and thus $e \notin \mathrm{MSF}(G)$. See Figure 14.2b.

Otherwise, χ *does* involve the vertex derived by contracting the subgraph C. The cycle χ in H forms a path $\mathcal{P}$ in G. The end edges of $\mathcal{P}$, say (x, w) and (y, z), have exactly one endpoint in C. As H does not include corrupted edges with one endpoint in C, the G-weight of (x, w) and (y, z) is the same as their $(G \Uparrow M)$-weight. Let T be the spanning tree of $C \Uparrow M$ derived by the DJP algorithm. Let ϱ be the path in T connecting (x, w) and (y, z), and let g be the heaviest edge in ϱ. Notice that $\mathcal{P} \cup \varrho$ forms a cycle in G. See Figure 14.2c. By Lemma 14.1 one of (x, w) and (y, z) is heavier than the $(G \Uparrow M)$-weight of g. The G-weight of an edge is at most its $(G \Uparrow M)$-weight, so one of (x, w) and (y, z) is heavier than g. As $(x, w), (y, z) \in \mathcal{P}$, and e is the heaviest edge in $\mathcal{P}$, e is also the heaviest edge in $\mathcal{P} \cup \varrho$, which is a cycle in G. Hence $e \notin \mathrm{MSF}(G)$.



(a) Cycle χ in the subgraph C.

(b) Cycle χ in H, that does not involve the vertex derived by contracting the subgraph C.

(c) Cycle $\chi = \mathcal{P}$ in H, that does involve the vertex derived by contracting the subgraph C.

Figure 14.2: Proof of Lemma 14.2.

□

Before stating the needed theorem, we will show the key lemma applied twice to make it more clear.

Let the subgraph C_1 be DJP-contractible with respect to $G \Uparrow M_1$. Consider the contracted graph $G \setminus C_1 - M_{C_1}$, which is a graph derived by contracting connected components and by removing some of the edges with one endpoint in a connected component. Let the subgraph C_2 be DJP-contractible with respect to $(G \setminus C_1 - M_{C_1}) \Uparrow M_2$. By applying Lemma 14.2 again, $\mathrm{MSF}(G \setminus C_1 - M_{C_1})$ is a subset of

$$\mathrm{MSF}(C_2) \cup \mathrm{MSF}((G \setminus C_1 - M_{C_1}) \setminus C_2 - M_{C_2}) \cup M_{C_2} .$$

As edges in M_{C_1} are removed before C_2 is built, it is easy to see for the connected components C_1 and C_2, that $(G \setminus C_1 - M_{C_1}) \setminus C_2 = G \setminus (C_1 \cup C_2) - M_{C_1}$ so the above superset is the same as

$$\mathrm{MSF}(C_2) \cup \mathrm{MSF}(G \setminus (C_1 \cup C_2) - (M_{C_1} \cup M_{C_2})) \cup M_{C_2} ,$$

and so

$$\begin{aligned} \mathrm{MSF}(G) &\subseteq \mathrm{MSF}(C_1) \cup \mathrm{MSF}(G \setminus C_1 - M_{C_1}) \cup M_{C_1} \\ &\subseteq \mathrm{MSF}(C_1) \cup \mathrm{MSF}(C_2) \cup \mathrm{MSF}(G \setminus (C_1 \cup C_2) - (M_{C_1} \cup M_{C_2})) \cup M_{C_1} \cup M_{C_2} . \end{aligned}$$

Theorem 14.3. *Let the subgraphs C_1, C_2, C_3, ... , C_i be DJP-contractible with respect to $G \Uparrow M_1$, $(G \setminus C_1 - M_{C_1}) \Uparrow M_2$, $(G \setminus (C_1 \cup C_2) - (M_{C_1} \cup M_{C_2})) \Uparrow M_3$, ... , $\left(G \setminus \bigcup_{j=1}^{i-1} C_j - \bigcup_{j=1}^{i-1} M_{C_j}\right) \Uparrow M_i$, respectively, where M_j is the set of corrupted edges when growing C_j and M_{C_j} is the set of edges in M_j with one endpoint in C_j. That is C_j is DJP-contractible with respect to a graph derived from G after several rounds of contractions and edge deletions. Then* MSF (G) *is a subset of*

$$\bigcup_{j=1}^{i} \mathrm{MSF}\,(C_j) \cup \mathrm{MSF}\left(G \setminus \bigcup_{j=1}^{i} C_j - \bigcup_{j=1}^{i} M_{C_j}\right) \cup \bigcup_{j=1}^{i} M_{C_j}\ .$$

Proof. Apply Lemma 14.2 repeatedly for the DJP-contractible subgraphs $C_1, C_1, C_2, \ldots, C_i$. □

The optimal MST algorithm will repeatedly grow DJP-contractible subgraphs C_j starting from a non-contracted vertex, using the Partition procedure described in the subsequent section.

14.2 The Partition procedure

The purpose of the Partition procedure is to repeatedly find relatively small DJP-contractible subgraphs $C_1, \ldots, C_k$, such that all vertices of the graph will be in at least one subgraph. We will use the terms partition, component and subgraph interchangeably. The main part of the procedure operates like a modified version of the Dense Case algorithm (Chapter 10), but with two main differences. Firstly, to minimise the running time, the DJP trees is grown using a soft heap instead of an ordinary non-corrupting priority queue. Additionally, in the absence of a decreaseKey operation on the soft heap, the priority queue will store edges instead of pairs of vertices their and best edges. This version of the DJP algorithm is described in Section 9.3. Secondly, the priority queue has no size bound, but a tree will stop growing if it reaches some maximum size of vertices. Due to the properties of a soft heap, some of the edges in $C_1, \ldots, C_k$ are corrupted. Besides the input graph G, the procedure takes as input a `maxsize` and an ε parameter. The procedure repeatedly grows DJP-contractible partitions using a fresh soft heap with error parameter ε. A tree stops growing when the partition reaches `maxsize` vertices or is connected to a previously grown partition. Instead of explicitly contracting the graph to $G \setminus C_i - M_{C_i}$ when C_i is built, the procedure marks all vertices in C_i dead. This way the procedure can easily detect if succeeding C_i are connected to an old, and hence contracted C_i. The actual contraction of connected components is done in a subsequent step to the procedure. Furthermore all corrupted edges with one endpoint in C_i, that is M_{C_i}, are removed from the graph after C_i is built. The Partition procedure is formally described in Algorithm 14.1.

The first component C_1 will grow to `maxsize` vertices. Any other component will grow to at most `maxsize` vertices. It is clear that if a C_i does not reach `maxsize` vertices, then it has connected itself to an existing component, that is a dead vertex. If a component C_i connects to a dead vertex, then C_i will clearly have the vertex in common with one or more previously built C_i. Let a *conglomerate* be a collection of C_j connected by common vertices, such that a conglomerate is a connected component of partitions in G. A conglomerate

Algorithm 14.1: `partition(` G`, maxsize,` ε `)` - Partition procedure

```
Input  : Connected graph $G$, partition maxsize and soft heap error rate $\varepsilon$.
Output: Partitions $(C_i)$ of $G$ with at most maxsize vertices each and the corrupted
         edges $M_C$.
Mark all vertices live
$C \leftarrow M_C \leftarrow \emptyset$
$i \leftarrow 0$
while there is a live vertex do
    $i \leftarrow i + 1$
    $V_i \leftarrow \{v\}$, where $v$ is any live vertex
    Initialise soft heap Q with error rate $\varepsilon$
    Insert $v$'s edges into Q
    while all vertices in $V_i$ are alive and $|V_i| <$ maxsize do
        repeat
            $(x, y) \leftarrow$ Q.deleteMin
            Assume $x \in V_i$
        until $y \notin V_i$
        $V_i \leftarrow V_i \cup \{y\}$
        if $y$ is live then Insert $y$'s edges into Q
    Mark all vertices in $V_i$ dead
    $M_{V_i} \leftarrow$ the corrupted edges in Q with one endpoint in $V_i$
    $M_C \leftarrow M_C \cup M_{V_i}$
    $G \leftarrow G - M_{V_i}$
    Empty the soft heap Q
    $C_i \leftarrow$ the subgraph of $G$ induced by $V_i$
    $C \leftarrow C \cup \{C_i\}$
return $(M_C, C)$
```

has at least `maxsize` vertices, as the first grown component of each conglomerate reaches `maxsize` vertices.

14.3 Partition running time and implementation

It is easy to verify that each edge is inserted into the heap no more than twice, namely once for each endpoint. So the number of `insert` operations is a most $2m$. The total number of `deleteMin` operations in the inner loop is no more than the number of inserts. The cost of finding corrupted edges and emptying the heap are charged to the `insert` operations which created it.

It seems to that the Partition procedure has a lot of set operations, which are relatively complex and slow. All union operations in the procedure are on the form $A \leftarrow A \cup B$, in other words they are an addition to the set A. It will become clear in a moment, that in practice the operands (A and B) of all the union operations are disjoint. Consequently V_i, M_C and C can be implemented as simple linked lists or array lists with an list append operation instead of an set union operation.

The statement $G \leftarrow G - M_{V_i}$ removes edges in M_{V_i} from the graph G. This suggests either building a new graph or modifying the existing graph. Building a new graph from scratch for each C_i will take $O(m)$ time, which is too slow. Modifying the existing graph in $O(1)$ time per edge in M_{V_i} requires a more complicated graph structure, which also requires a greater constant in the graph space requirement. The modified structures are double linked incidence lists and a references from each edge to its position in the incidence list of each endpoint. A third solution, which is implemented, is to raise a `isRemoved` flag for all edges in M_{V_i}. Afterwards, edges with this flag raised, will not be taken into account when iterating over edges. Thus removed edges will not be inserted into the heap nor be present in any partition C_i afterwards. Once an edge is removed from the graph, it can not be corrupted again, as it will not be inserted into the heap again.

It is only edges incident to vertices in V_i that are inserted into the heap. As a result the heap can contain two copies of the same edge, but only if both endpoints are in V_i. None of these will appear in M_{V_i} due to its definition, namely vertices with one endpoint in V_i. Consequently M_{V_i} and M_C can be implemented as linked lists. It is clear from the DJP algorithm that the vertex y is outside the current partition V_i when it is added to the set, and thus V_i can be implemented as a linked list. It is also clear that all partitions (C_i) are distinct, so C can also be implemented as a linked list.

The procedure explicitly requires that we must be able to detect if an edge has only one endpoint in V_i, and if so, which endpoint is inside and outside, respectively. This could be implemented using a union-find set data structure for V_i. But it is simply implemented as an extra `isInVi` flag for each vertex. When the vertex y is added to the V_i list, the `isInVi` flag are also raised for y. In the end of the outer while loop these flags are lowered again. When we check if an edge has exactly one endpoint in V_i, we check if exactly one of the endpoint flags are raised.

The pseudo code of the Partition procedure builds the partition C_i, which is the subgraph of G induced by V_i. That is C_i is the subgraph of G which has the vertices in V_i and the (non-removed) edges connecting them. An important observation is that the edges in M_{V_i} are removed from G after V_i is built, but before C_i is built. But because M_{V_i} by definition is the set of corrupted edges with *one* endpoint in V_i, none of these are connecting vertices in V_i. Consequently it is possible to build the list of edges in C_i once they are detected during the insert iterations over incident edges. Let E_i be this list of edges, just like V_i is the list of vertices. Once the procedure extends the partition with a new vertex y, which is live, it inserts all y's incident edges into the heap. This insertion procedure is extended to check if the opposite endpoint of each edge is in V_i. If this is the case, the corresponding edge is appended to E_i. It is also possible to perform this check (without inserting the edges) on each incident edge to a dead, and hence the last, vertex y. This way E_i will contain all edges induced by V_i when building C_i. But in the worst case this will require a check on the same edge $O(n)$ (number of partitions) times. This will happen if all partitions, except the first, connects to the same dead vertex. Another solution to find C_i-edges incident to the dead vertex, is to iterate over all live vertices in V_i. That is all vertices in V_i before the dead y is appended. For each vertex, perform a check on each incident edge, if it connects to y. If this is the case, the corresponding edge is appended to E_i. So while a vertex is alive, its incident edges are visited at least once (the insertion) and at most twice (if connected to a dead vertex). Once a vertex is dead, its incident edges will not be visited anymore. So in total each edge are visited at least twice and at most four times, which is a constant upper bound.

The time to create a subgraph C_i is $O(|E_i| + |V_i|)$ and is charged to the creation of the lists, which again is charged to the heap operations. The time to add corrupted edges to M_C is proportional to the size of M_{V_i} which is no more than the number of insertions, and is charged to the insertions.

As all operations are charged to the heap operations, the total running time depends on the number of heap operations. Each edge is visited and inserted into the heap a constant number of times, and all other heap operations have constant running time per edge and is charged to the insertions. Thus the number of inserts dominate the running time.

The amortised running time of soft heap insert is $O(\log 1/\varepsilon)$, and the number of inserts is bounded by $2m$, so the total running time of Partition is $O(m \log 1/\varepsilon)$.

By Theorem 12.1, the number of corrupted edges in the soft heap are bounded by the number of insertions scaled by ε. The corrupted edges with endpoints in distinct C_i are a subset of the corrupted edges in the soft heap, so the total sum of edges in each M_{V_i} is $|M_C| \leq 2\varepsilon m$. Thus we can state the following lemma:

Lemma 14.4. *Given a connected graph G, an error parameter $0 < \varepsilon < 1/2$, and a* `maxsize`*, the Partition procedure finds a corrupted edge set M_C and subgraphs $C_1, \ldots, C_k$ which are edge-disjoint, in time $O(m \log 1/\varepsilon)$, while satisfying the following conditions:*

- *For all $v \in V(G)$, there is some i such that $v \in V(C_i)$.*
- *For all $C_i \in C$, $|V(C_i)| \leq$* `maxsize`*.*
- *For each conglomerate $P \in \bigcup_i C_i$, $|V(P)| \geq$* `maxsize`*.*
- $|M_C| \leq 2\varepsilon \cdot |E(G)|$.
- $\mathrm{MSF}(G) \subseteq \bigcup_i C_i \cup \mathrm{MSF}(G \setminus \bigcup_i C_i - M_C) \cup M_C$ *(By Theorem 14.3).*

15 The optimal MST algorithm

At this point we have studied all the necessary building blocks for the optimal MST algorithm. With these algorithms, the optimal MST algorithm is relatively simple. The necessary building blocks are: Graph contraction (Chapter 7), Borůvka's MST algorithm (Chapter 8), the "Dense Case" MST algorithm (Chapter 10), optimal MST decision trees (Chapter 11), and the Partition procedure (Section 14.2).

Section 15.1 describes the algorithm and Section 15.2 shows a visual example of the algorithm. Finally we analyse the running time of the algorithm in Sections 15.3–15.6.

15.1 The algorithm

Brief overview

Initially the algorithm precomputes optimal decision trees for very small graphs relative to the connected input graph.

Afterwards, the algorithm removes some edges guaranteed not to be in the MST, by detecting a superset of the final MST edges. These are detected by growing very small partitions using the Partition procedure, finding the MST of each partition with an optimal decision tree, and running the Dense Case algorithm on the graph derived from contracting the conglomerates induced by the partitions. Then it finds a subset of the real MST edges among this superset using two steps of Borůvka's algorithm. Borůvka's algorithm contracts the graph along the real MST edges found, so we repeat the algorithm recursively for the contracted graph. The recursion stops when the input graph is the trivial graph of one vertex and no edges.

Detailed description

Here follows a detailed description of the algorithm with supporting pseudo code in Algorithm 15.1. The subsequent Section 15.2 shows a visual example of the algorithm.

Initialisation and precomputing Build optimal MST decision trees for all graphs with at most $\lceil \log^{(3)}(n) \rceil$ vertices, using the brute force searching procedure described in Section 11.3. Here, n is the number of vertices in the connected input graph. Then proceed to the actual recursive algorithm on the input graph. The set of edges returned from the algorithm forms the MST edges of the input graph.

Recursive algorithm input Let G be the connected input graph for the current recursion, and let $n = |V(G)|$, $m = |E(G)|$. If G is the trivial graph of one vertex and no edges, then all MST edges are already found and we just return an empty edge set. Otherwise, we calculate the maximum partition size $r = \lceil \log^{(3)}(n) \rceil$ for this recursion, and proceed.

Partition Grow DJP-contractible subgraphs C_i of maximum number of vertices r using the Partition procedure (Section 14.2). Let k be the number of partitions grown. The partitions are grown using a soft heap with error rate $0 < \varepsilon < 1/2$, which corrupts some, but no more than $2\varepsilon m$ of the inserted keys (edge weights). The actual error rate will be chosen in Section 15.3. The corrupted edges with endpoints in different components, C_i and C_j where $i \neq j$, are marked as removed in G and are finally returned as M. For the pseudo code, a list of the grown partitions $C = (C_1, \ldots, C_k)$ is also returned.
This sums to calling `Partition(` G`,` r`,` ε `)`.

Decision trees For each partition C_i returned from `Partition`: Identify the MST edges with an optimal number of edge comparisons using an optimal MST decision tree for C_i. Let F_i be the MST deduced from the decision tree. The edges in F_i are marked as MST edges in G. For the pseudo code, let `DecisionTree(`C`)` denote a function that takes a list of graphs, C (here, subgraphs), and for each graph $C_i \in C$ finds the set of MST edges, F_i. For the pseudo code, the resulting MST edge sets $F_1, \ldots, F_k$ are returned as a list F.

Contract graph As stated in Lemma 14.4, each C_i is a part of a conglomerate (connected component) of at least $\log^{(3)}(n)$ vertices. Hence, the MST of each C_i, that is F_i, is a part of the same connected component of at least $\log^{(3)}(n)$ vertices. The decision tree step has marked edges in $F_1 \cup \cdots \cup F_k$ as MST edges and the Partition procedure has marked edges in M as removed. So call `contract(`G`)` with the input graph as parameter, to remove edges which are marked as removed and to contract the remaining graph along the detected MST edges. As each MST connected component has at least $\log^{(3)}(n)$ vertices, the resulting contracted graph has at most $n/\log^{(3)}(n)$ vertices. Let $G_a = G \setminus (F_1 \cup \cdots \cup F_k) - M$ denote the resulting contracted graph.

Dense Case Let $n_a = |V(G_a)|$ and let $m_a = |E(G_a)|$. According to Theorem 10.1, the "Dense Case" algorithm runs in linear time of m on a graph as G_a with $n_a \leq n/\log^{(3)}(n)$ vertices and $m_a \leq m$ edges. So we call `DenseCase(`G_a`,`m`)` to find the MSF of the contracted graph G_a in linear time of m. Let F_0 denote the resulting set of MSF edges.

Candidate edges As stated in Theorem 14.3, the MST of G is a subset of

$$\bigcup_{j=1}^{k} \mathrm{MSF}(C_j) \cup \mathrm{MSF}\left(G \setminus \bigcup_{j=1}^{k} C_j - \bigcup_{j=1}^{k} M_{C_j} \right) \cup \bigcup_{j=1}^{k} M_{C_j} .$$

The first union of MSF edges are those found by the `DecisionTree` step. The second set of MSF edges are those found by the `DenseCase` step. The third and last set of edges are the corrupted edges returned by `Partition`. Consequently, the edges in $E\,(F_0 \cup \cdots \cup F_k) \cup M$ are MST candidate edges for G. Notice that the MSF edges found so far only are candidate edges, and not detected as real MST edges. Let $G_b = F_0 \cup F_1 \cup \cdots \cup F_k \cup M$ denote the graph of all the original vertices in G and these candidate edges. In other words, G_b is a graph derived from G by eliminating edges that are guaranteed not to belong to the MST of G.

Borůvka steps Finally we identify some real MST edges from the set of candidate edges in G_b, and contract the graph along these edges. As stated in Chapter 8, a Borůvka step runs in linear time in the number of edges. Hence, two (a constant number of) Borůvka steps also run in linear time. For the pseudo code, let `Boruvka2` denote a function running at most two Borůvka steps. The function returns (T', G_c), where T' is the MST edges found, and G_c is the resulting contracted graph. Notice that the number of steps may be less than two if the input graph is contracted to a single vertex in the first step. The number of edges in G_b does not exceed m, so running two Borůvka steps takes linear time in the number of edges in G.

Recursion Lastly, we repeat the algorithm by calling recursively on the contracted graph G_c, starting from the "Recursive algorithm input" step. Let T be the MST edge set returned from this call. When the recursive call returns, we merge the set of real MST found in the Borůvka steps, T', with the real MST edges found in the recursive call, T, and return this set of MST edges. That is, return $T \cup T'$.

See a formal description of the optimal MST algorithm without the precomputing step in Algorithm 15.1.

Algorithm 15.1: `optimalMST`(G) - The optimal MST algorithm

```
Input  : Connected graph $G$.
Output: The MST of $G$.
if $E(G) = \emptyset$ then return $\emptyset$
$r \leftarrow \lceil \log^{(3)}(|V(G)|) \rceil$
$(M, C) \leftarrow$ Partition( $G$, $r$, $\varepsilon$ )
$F \leftarrow$ DecisionTree( $C$ )
Let $k \leftarrow |C|$ and $F = \{F_1, \ldots, F_k\}$
$G_a \leftarrow G \setminus (F_1 \cup \cdots \cup F_k) - M$
$F_0 \leftarrow$ DenseCase( $G_a$, $|E(G)|$ )
$G_b \leftarrow F_0 \cup F_1 \cup \cdots \cup F_k \cup M$
$(T', G_c) \leftarrow$ Boruvka2( $G_b$ )
$T \leftarrow$ OptimalMST( $G_c$ )
return $(T \cup T')$
```

The correctness of this algorithm follows directly from Theorem 14.3 and Lemma 14.4.

Minor details

Instead of compute decision trees every time the algorithm is invoked, we can save them at the first invocation, and then load them back in subsequent invocations, assuming that $\lceil \log^{(3)}(n) \rceil$ does not exceed the maximum number of vertices for the stored decision trees. Otherwise, we have to compute new decision trees and save them.

As always we assume the input graph G is connected. So if $m = n - 1$, then G is already a tree, and we can return the edges in $E(G)$ before the Partition step. Consequently, in the analysis we can assume that $n \leq m$. If no edges are removed in the `Partition` procedure, that is $|M| = 0$, then we have another special case: Firstly, if all components C_i are connected, then $F_1 \cup \cdots \cup F_k$ forms the MST of G, and we can optimise the algorithm by returning the edges in $E(F_1 \cup \cdots \cup F_k)$ after the `DecisionTree` step. Otherwise, then $F_0 \cup \cdots \cup F_k$ forms the MST of G, and we can optimise the algorithm by returning the edges in $E(F_0 \cup \cdots \cup F_k)$ after the `DenseCase` step.

15.2 Visual example

The subsequent two pages are dedicated to show a visual example of a last non-trivial recursion of the algorithm.

Figure 15.1 shows the input graph G.

Figure 15.2 shows the result of the Partition procedure with a *fictive* $r = 4$. The partitions (the C_i) grown are shown with dash-dotted lines. The removed edges in M are shown with dotted lines.

Figure 15.3 shows the result of `DecisionTree`. Only edges with both endpoints in the same C_i are shown. The C_i are again shown with dash-dotted lines. The MST edges found by optimal decision trees (the F_i) are shown with fat lines.

Figure 15.4 shows the graph G_a derived by removing edges in M and contracting connected MST components. In this example, weeding of parallel edges is omitted. That is $G_a = G \setminus (F_1 \cup \cdots \cup F_k) - M$. The super-vertices in G_a (the connected MST components in G) are shown with fat dash-dotted lines.

Figure 15.5 shows the MST edges found by `DenseCase`, that is F_0. These edges are shown with fat lines.

Figure 15.6 shows the graph G_b of MST candidate edges. That is $G_b = F_0 \cup F_1 \cup \cdots \cup F_k \cup M$. Edges originating from M and F_0 are annotated. The remaining edges originate from $F_1, \ldots, F_k$.

Figure 15.7a shows the MST edges found in the graph G_b by `Boruvka2`, that is T'. The MST edges are shown with fat lines. `Boruvka2` will contract this graph to one vertex, which will make the recursion stop in the subsequent recursive call.

Figure 15.7b shows the same MST edges as Figure 15.7a, but in context of the original graph G.

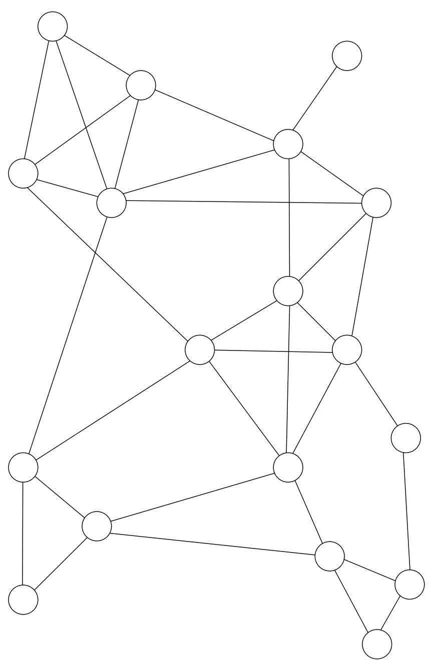


Figure 15.1: The input graph G.

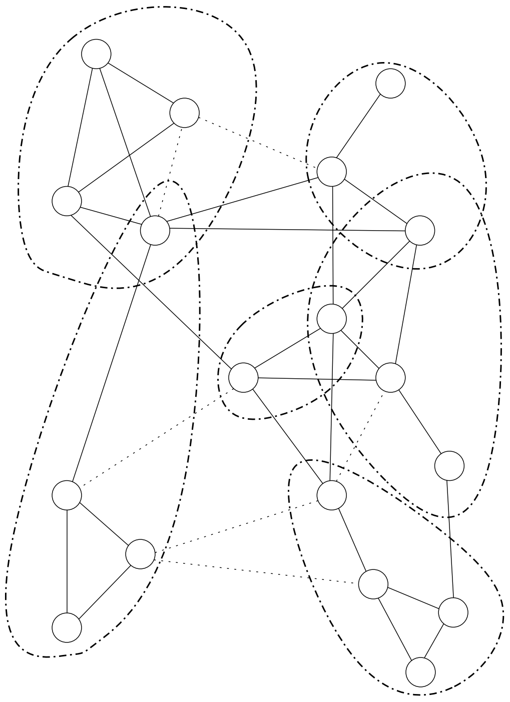


Figure 15.2: After Partition.

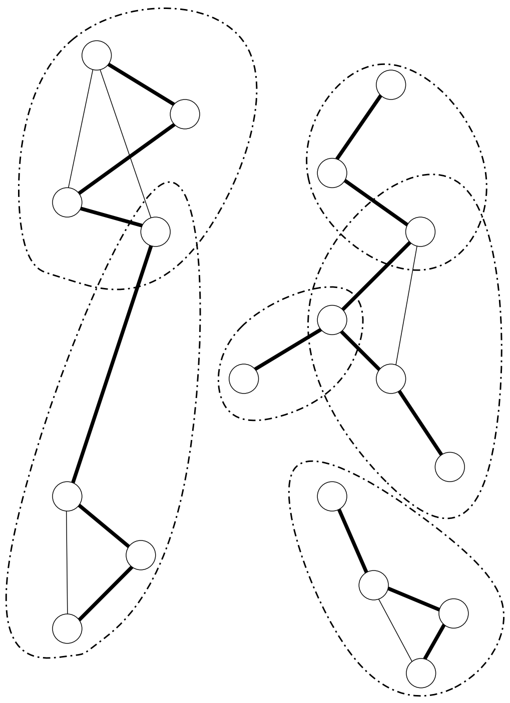


Figure 15.3: After DecisionTree.

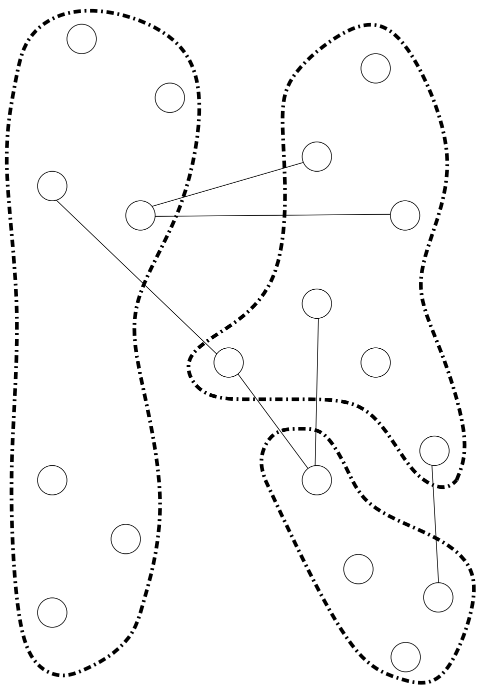


Figure 15.4: The contracted graph G_a.

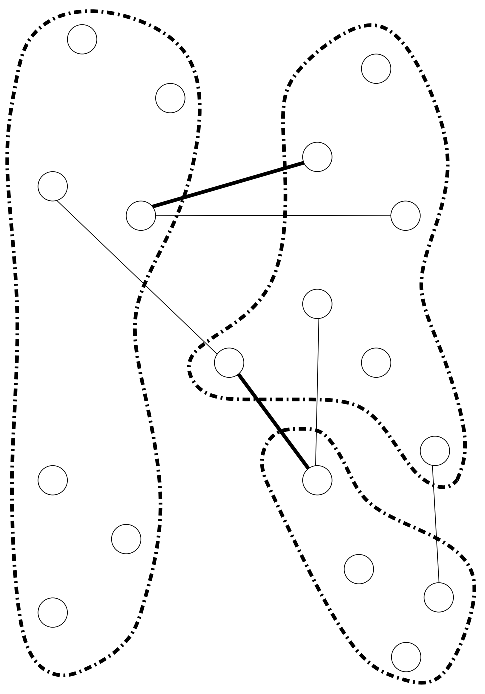


Figure 15.5: After DenseCase.

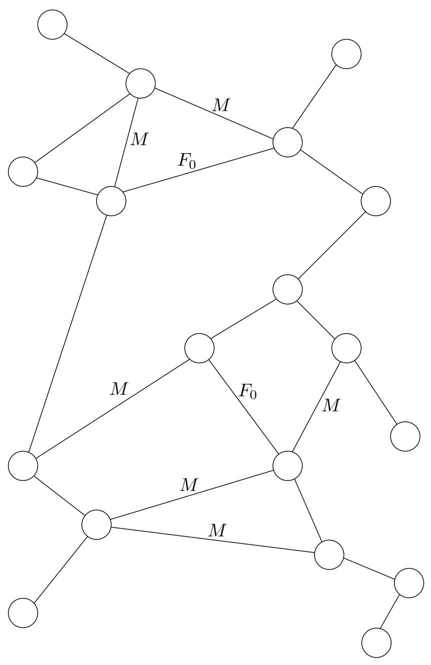
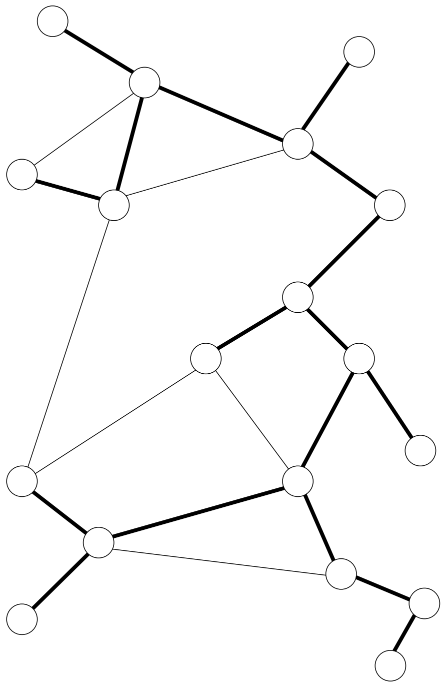
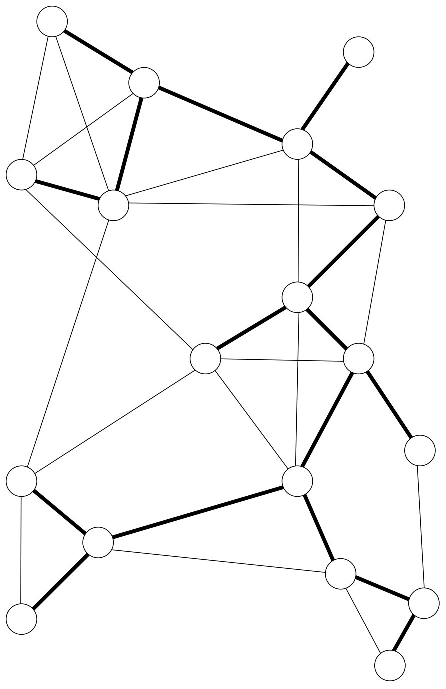


Figure 15.6: Candidate edges G_b.



(a) Candidate edges G_b context.



(b) Original graph G context.

Figure 15.7: MST edges T' found by Boruvka2.

15.3 Running time

According to Theorem 11.1, the optimal MST decision trees can be computed in linear time of the input graph for our value of r. Let us analyse one execution of `optimalMST`. According to Lemma 14.4, the running time of the Partition procedure is $O(m \log 1/\varepsilon)$, where m is the number of edges in G. Consequently, setting ε to any constant will give a linear running time of the Partition procedure. The running time of the decision tree step is unknown, but known to be optimal for each partition. According to Section 7.3, removal of edges and contraction of G to G_a takes linear time in m. As stated in the description, the running time of the Dense Case step is also linear due to the reduction of vertices from G to G_a. The construction of G_b also takes linear time, since the size of G_b is at most the size of G. As stated in the description, the running time of the two Borůvka steps is also linear. Summing up, if ε is constant, then except for the decision tree step, the running time of one execution without a recursive call is linear in the size of the input graph. So it remains to find a value for ε and analyse the decision tree complexity.

As stated in Lemma 14.4, the number of corrupted edges with endpoints in distinct partitions is $|M| \leq 2\varepsilon m$. Pettie and Ramachandran [PR02] choose the constant $\varepsilon = 1/8$, so $|M| \leq m/4$. If no edges are removed during the Partition procedure, then $|E(F_0 \cup \cdots \cup F_k)| = n - 1$. Otherwise, removal of edges may result in that the graph G_a is unconnected, resulting in fewer edges in the set. Hence the number of edges in the set is $< n$. Consequently, the number of edges in the graph G_b is

$$m_b \leq \frac{m}{4} + n \ .$$

The vertices in G_b are the same as in G, so $n_b = n$. According to Chapter 8, the number of vertices are reduced by at least a factor of 2 in each Borůvka step, so the number of vertices in the graph G_c is $n_c \leq n/2^2 = n/4$. As each Borůvka contraction step removes at least as many edges as vertices, the number of edges in the contracted graph G_c after two Borůvka steps is

$$m_c \leq m_b - \frac{n_b}{2} - \frac{n_b}{4} = m_b - \frac{3n_b}{4} = m_b - \frac{3n}{4} \ .$$

By inserting m_b and because $n \leq m$, we have

$$m_c \leq \left(\frac{m}{4} + n\right) - \frac{3n}{4} = \frac{m}{4} + \frac{n}{4} \leq \frac{m}{4} + \frac{m}{4} = \frac{m}{2} \ .$$

Consequently, the final graph G_c has $\leq n/4$ vertices and $\leq m/2$ edges. This implies a geometric reduction in both the number of edges and the number of vertices in the graph for the recursive call.

Definition Let $\mathcal{T}^*(H)$ be the optimal number of comparisons needed to determine the MST of a specific graph H. That is the depth of an optimal decision tree for H.

Definition Let $\mathcal{T}^*(m,n)$ be the optimal number of comparisons needed to determine the MST of any graph with n vertices and m edges. That is, the maximum decision tree height among the class of graphs with n vertices and m edges. Formally, that is

$$\mathcal{T}^*(m,n) = \max\{\mathcal{T}^*(H) \ : \ |V(H)| = n \wedge |E(H)| = m\} \ .$$

The function $\mathcal{T}^*$ is called the *decision tree complexity of MST*, because its value corresponds to the height of an optimal decision tree. It follows from the definition of $\mathcal{T}^*(m,n)$ that for any graph H:

$$\mathcal{T}^*(H) \leq \mathcal{T}^*(|V(H)|, |E(H)|) \ . \tag{15.1}$$

Let $T(m,n)$ be the running time of `optimalMST` on a graph G with n vertices and m edges. Notice that if the input graph G to `optimalMST` is connected, then G_c is also connected, and thus the graph of a recursive call is connected. Hence, the base case graph has one vertex and no edges, and $T(0,1)$ is equal to a constant. As the running time of `optimalMST` is linear in m except for the `DecisionTree` step and the recursive call, two constants $c_1, c_2 > 0$ exist, such that the running time is

$$T(m,n) \leq \sum_i c_1 \mathcal{T}^*(C_i) + T\left(\frac{m}{2}, \frac{n}{4}\right) + c_2 m \ . \tag{15.2}$$

It remains to show some properties of the $\mathcal{T}^*$ function to deduce a total running time of the algorithm. As we have a geometric reduction in the number of vertices and edges for each recursion, and the C_i are edge-disjoint, it is easy to see by Equation 15.1, that if $\mathcal{T}^*(m,n) = O(m)$, then $T(m,n) = O(m)$.

15.4 Decision tree analysis

First a definition, we will use for the rest of this section.

Definition Let $C_1, \ldots, C_k$ be the edge-disjoint subgraphs grown by the Partition procedure. Let

$$H = \bigcup_{i=1}^{k} C_i \ .$$

That is, H is the union of the grown components. In the example given in Section 15.2, H is the (unconnected) graph of vertices and edges shown in Figure 15.3.

Notice that H has the same vertices as G and a subset of the edges in G. That is $V(H) = V(G)$ and $E(H) \subseteq E(G)$.

Properties of the partitions

Lemma 15.1. *The structure of H dictates that*

$$\mathrm{MSF}(H) = \bigcup_{i=1}^{k} \mathrm{MSF}(C_i) \ .$$

Proof. According to Theorem 4.1 and Theorem 4.3, the heaviest edge in each simple cycle of H is *not* in $\mathrm{MSF}(H)$, and the remaining edges is in $\mathrm{MSF}(H)$. Consequently, if every simple cycle in H is contained in exactly one C_i, then the lemma holds. We will show that every simple cycle in H is contained in exactly one C_i.

It is clear from the Partition procedure that a component, C_i, shares at most one vertex with previously grown components $\bigcup_{j<i} C_j$. Let χ be a simple cycle in H, and let i be the largest index, such that C_i contains an edge in χ. Because C_i shares at most one vertex with $\bigcup_{j<i} C_j$, the simple cycle χ can only have edges in C_i, which proves the lemma. □

Decision tree lemmas

Definition Let an *intercomponent comparison* denote a decision tree comparison $w(e) < w(f)$, where $e \in C_i$, $f \in C_j$, and $i \neq j$.

Definition Let an *intra-C_i comparison* denote a decision tree comparison $w(e) < w(f)$, where $e, f \in C_i$.

Lemma 15.2. *There exists an optimal decision tree T for H, which makes no intercomponent comparisons.*

Proof. As $\mathrm{MSF}(H) = \bigcup_{i=1}^{k} \mathrm{MSF}(C_i)$, the decision tree T must determine the MSF of each C_i correctly. The MSF of each C_i can only be determined by performing intra-C_i comparisons. Equivalently, any intercomponent comparison will not give any information about the relative edge weights in the components, and thus no information about the MSF of the components. Consequently, a correct decision tree without intercomponent comparison nodes exists, and will have equal or lesser height than a decision tree containing intercomponent comparison nodes. Hence, there exists an optimal decision tree for H, which makes no intercomponent comparisons. □

Definition Pettie and Ramachandran [PR02] define a *canonical decision tree* for H the following way:
A canonical decision tree for H is defined by a decision tree where all intra-C_i comparisons precedes all intra-C_{i+1} comparisons. The canonical decision tree is formed so all subtrees containing intra-C_i comparisons are identical. That is, they have the same shape, and the same comparisons associated with corresponding nodes. See Figure 15.8 for an illustration of a canonical decision tree.

Lemma 15.3.
$$\mathcal{T}^*(H) = \sum_i \mathcal{T}^*(C_i) .$$

Proof. It is easy to verify that $\mathcal{T}^*(H) \leq \sum_i \mathcal{T}^*(C_i)$: For each i, let T_i be an optimal decision tree for C_i. We can construct a correct canonical decision tree for H by replacing

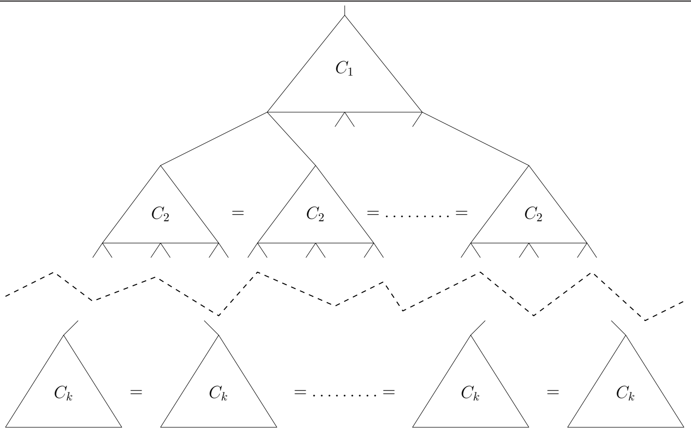


Figure 15.8: Canonical decision tree for H with k components.

each leaf of T_1 by T_2, and then replacing each leaf of the resulting tree by T_3 and so forth. In general that is a replacement of leafs in T_i by T_{i+1}. The MST edge set at each leaf of the resulting tree should be the union of the MST edge sets of the original trees on the root-to-leaf path. The resulting correct tree height is clearly the sum of T_i heights, $\sum_i \mathcal{T}^* (C_i)$, and thus $\mathcal{T}^* (H) \leq \sum_i \mathcal{T}^* (C_i)$ as there may exist a correct decision tree with lesser height. It remains to show, that there exists no optimal decision tree for H with height less than the sum of T_i heights.

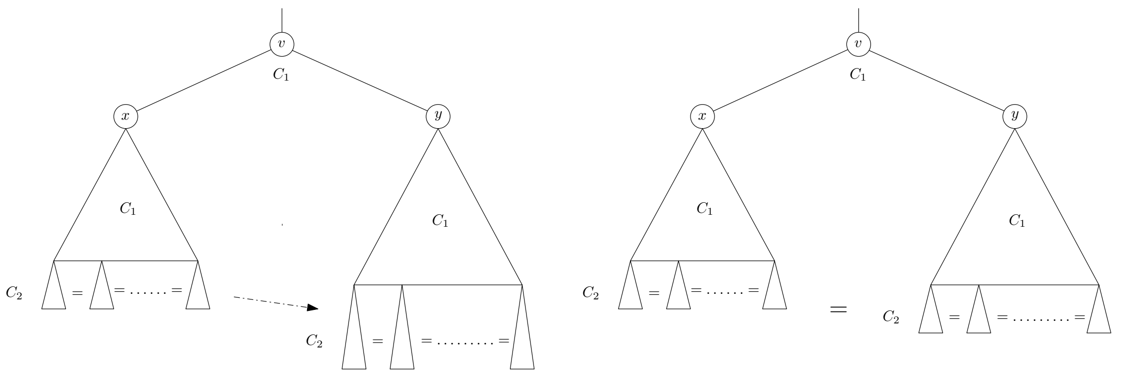
Let T be an optimal decision tree for H. As this tree is optimal, we assume with reference to Lemma 15.2, that T contains no intercomponent comparison nodes. We also know that T has height $\leq \sum_i \mathcal{T}^* (C_i)$. We will show that T can be transformed into a correct canonical decision tree T' without increasing the height. As T was optimal and we do not increase the height, the resulting tree T' will have the same height as $\sum_i \mathcal{T}^* (C_i)$ because T' must contain a path which is the concatenation of the longest path in an optimal decision tree for each of the C_i. It remains to show how to transform the tree correctly without increasing the height.

We will start by showing the transformation for the simple case, when there are only two components, C_1 and C_2. We start at the deepest level of T, and proceed against the root while transforming subtrees to the desired form, level by level. For each node v, we assume inductively that the two subtrees of v have been transformed into the desired form. That is for each of the two subtrees, the C_1 comparisons occur before C_2 comparisons and all C_2-subtrees are identical. It is easy to see that the induction assumption holds for the base case at the deepest level.

We have two cases for a node v with left child x and right child y at the current level:

- The node v is an intra-C_1 comparison node: We do not need to move v as it is a C_1 comparison node, and thus should stay in the top of the current structure. We assume inductively that all C_2-subtrees under the left and right child are identical, respectively. But the C_2-subtrees under the two children may not be identical. All C_2-subtrees must compute the same MST edge set for C_2. Hence, the subtree rooted at v can be transformed into the canonical form by replacing all C_2-subtrees by the

C_2-subtree with minimum height. See Figure 15.9 for an example of this procedure. This procedure will clearly make all C_2-subtrees of v identical and will clearly not increase the height of T'.



(a) The subtree rooted at v before the transformation.

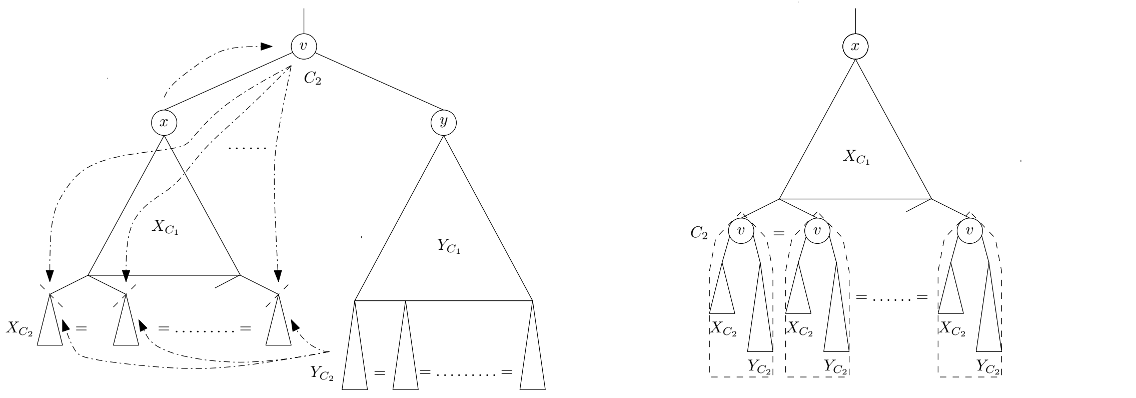
(b) The canonical subtree after the transformation.

Figure 15.9: The node v has an intra-C_1 comparison associated. In this example, the C_2-subtrees under x has minimum height. Hence, all C_2 subtrees must be replaced by such subtree.

- The node v is an intra-C_2 comparison node: We must move v as it is a C_2 comparison node, and thus should not stay at the top of the current structure. Let X_{C_1} denote the C_1-subtree rooted by x, and let Y_{C_1} denote the C_1-subtree rooted by y. Let X_{C_2} denote one of the isomorphic C_2-subtrees under x, and let Y_{C_2} denote one of the isomorphic C_2-subtrees under y. We know that the two C_1-subtrees X_{C_1} and Y_{C_1} must compute the same MST edge set for C_1. Hence, the subtree rooted at v can be transformed into the canonical form by keeping only one of these: Let Z denote the C_1-subtree among X_{C_1} and Y_{C_1} with minimum height, and let Z' denote the opposite subtree. Firstly, we replace v by the subtree Z. Then we unlink all the isomorphic C_2-subtrees from Z, and replace their roots by a new level of leaf nodes for Z. Then we copy the C_2-comparison at v to all these new leafs. For each such copy, we make X_{C_2} the left subtree of the leaf, and make Y_{C_2} the right subtree of the leaf. This will clearly make the C_2-subtrees under Z identical. See Figure 15.10 for an example of this procedure. The composite operation of moving v to a new level of leafs in Z does not change the height of T'. As the height of Z was less than or equal to the height of Z', copying an isomorphic C_2-subtree from Z' to Z does not increase the height of T'. Consequently, this procedure does not increase the net height of T'.

When the transformations are done at the level of the root node, we have showed by induction, that the resulting height of T' will not exceed the height of T. Consequently, the lemma holds for the simple case of two components.

Assume inductively that the result holds for $k-1 \geq 2$ components. We can easily extend the result to be valid for k components $C_1, \ldots, C_k$: Group the first $k-1$ components as C'_1, let C_k be C'_2, and use the above method for C'_1 and C'_2. This will make all $C'_2 = C_k$ subtrees identical and bring them to the bottom of the tree, without increasing the height. Then dismantle the C_k-subtrees, and proceed the transformation recursively for the remaining $k-1$ components.



(a) The subtree rooted at v before the transformation.

(b) The canonical subtree after the transformation. The new identical C_2-subtrees are shown with dashed lines.

Figure 15.10: The node v has an intra-C_2 comparison associated. In this example, the left subtree X_{C_1} of v has minimum height. So we must copy the comparison of v to each node in a new level of leaf nodes in X_{C_1}, and replace v by x. Additionally we must make X_{C_2} (respectively, Y_{C_2}) the left (respectively, right) child of each new leaf.

□

Derivation of equations

It is easy to see for any $n, m > 2$, that we can build a graph that has a simple cycle that includes every edge. Every edge in a cycle must participate in a least one edge-weight comparison, so

$$m/2 \leq \mathcal{T}^*(m,n) \quad \Leftrightarrow \quad m \leq 2\mathcal{T}^*(m,n) \ . \tag{15.3}$$

Adding isolated vertices to a graph will clearly not decrease the height of an optimal decision tree. Hence, for $n' \geq n$,

$$\mathcal{T}^*(m,n) \leq \mathcal{T}^*\left(m,n'\right) \ . \tag{15.4}$$

Similarly, adding edges to a graph will clearly not decrease the height of an optimal decision tree. Hence, for $m' \geq m$,

$$\mathcal{T}^*(m,n) \leq \mathcal{T}^*\left(m',n\right) \ . \tag{15.5}$$

Hence for $n' \geq n, m' \geq m$:

$$\mathcal{T}^*(m,n) \leq \mathcal{T}^*\left(m',n'\right) \ . \tag{15.6}$$

For any graph with $n \geq 1$, $m \geq 0$, we can make an exact copy, so the resulting (unconnected) graph has $2n$ vertices and $2m$ edges with two identical partitions. The decision tree complexity of this graph is clearly the double of the original graph. Due to Equation 15.6 we have

$$2\mathcal{T}^*(m,n) \leq \mathcal{T}^*(2m,2n) \quad \Leftrightarrow \quad \mathcal{T}^*(m,n) \leq \frac{1}{2}\mathcal{T}^*(2m,2n)\ . \tag{15.7}$$

Let $n_H = |V(H)|$, and $m_H = |E(H)|$. As $n = n_H$, $m \geq m_H$, and by Equation 15.1 and Equation 15.5, the result of Lemma 15.3 leads to

$$\sum_i \mathcal{T}^*(C_i) = \mathcal{T}^*(H) \leq \mathcal{T}^*(m_H, n_H) \leq \mathcal{T}^*(m,n)\ . \tag{15.8}$$

15.5 Deduction of the running time

We now have the needed equations to deduce the running time of the algorithm. From Equation 15.2 we have:

$$\begin{aligned} T(m,n) &\leq \sum_i c_1\mathcal{T}^*(C_i) + T\left(\frac{m}{2},\frac{n}{4}\right) + c_2 m \\ &\leq c_1\mathcal{T}^*(m,n) + T\left(\frac{m}{2},\frac{n}{4}\right) + c_2 m && \text{(Equation 15.8)}\ . \end{aligned}$$

There exists some function $f(m,n)$, so $T\left(\frac{m}{2},\frac{n}{4}\right) \leq f(m,n)\cdot\mathcal{T}^*\left(\frac{m}{2},\frac{n}{4}\right)$, and thus

$$\begin{aligned} T(m,n) &\leq c_1\mathcal{T}^*(m,n) + T\left(\frac{m}{2},\frac{n}{4}\right) + c_2 m \\ &\leq c_1\mathcal{T}^*(m,n) + f(m,n)\cdot\mathcal{T}^*\left(\frac{m}{2},\frac{n}{4}\right) + c_2 m \\ &\leq c_1\mathcal{T}^*(m,n) + f(m,n)\cdot\mathcal{T}^*\left(\frac{m}{2},\frac{n}{4}\right) + 2c_2\mathcal{T}^*(m,n) && \text{(Equation 15.3)} \\ &\leq c_1\mathcal{T}^*(m,n) + \frac{f(m,n)}{2}\cdot\mathcal{T}^*\left(m,\frac{n}{2}\right) + 2c_2\mathcal{T}^*(m,n) && \text{(Equation 15.7)} \\ &\leq c_1\mathcal{T}^*(m,n) + \frac{f(m,n)}{2}\cdot\mathcal{T}^*(m,n) + 2c_2\mathcal{T}^*(m,n) && \text{(Equation 15.4)} \\ &= \left(c_1 + \frac{f(m,n)}{2} + 2c_2\right)\mathcal{T}^*(m,n)\ . \end{aligned}$$

To complete the inequality, we must find a function for $f(m,n)$, such that $f(m,n) \geq c_1 + f(m,n)/2 + 2c_2$. Solving this inequality gives

$$\begin{aligned} f(m,n) &\geq c_1 + f(m,n)/2 + 2c_2 \Rightarrow \\ 2f(m,n) &\geq 2c_1 + f(m,n) + 4c_2 \Rightarrow \\ c = f(m,n) &\geq 2c_1 + 4c_2\ . \end{aligned}$$

Consequently, the function $f(m,n)$ is equal to a constant.

So, for any constant $c \geq 2c_1 + 4c_2$

$$\begin{aligned} T(m,n) &\leq c_1\mathcal{T}^*(m,n) + T\left(\frac{m}{2},\frac{n}{4}\right) + c_2 m \\ &\leq c\mathcal{T}^*(m,n) \end{aligned}$$

which gives us the main result of the algorithm.

Theorem 15.4. *Let $\mathcal{T}^*(m,n)$ be the decision tree complexity of the MST problem on graphs with n vertices and m edges. The algorithm* `optimalMST` *described in Algorithm 15.1 computes the MST of a connected graph with n vertices and m edges deterministically in $O(\mathcal{T}^*(m,n))$ time.*

15.6 Running time for practical graph instances

Theorem 15.4 states that the optimal algorithm runs in time $\mathcal{T}^*(m,n)$. The result of the theorem originates from Equation 15.2, which shows that the running time is

$$T(m,n) \leq \sum_i c_1 \mathcal{T}^*(C_i) + T\left(\frac{m}{2}, \frac{n}{4}\right) + c_2 m \; .$$

As stated in Section 11.5, the number of atoms in the universe is approximately 2^{265}, so in practice n is much smaller. Consequently, for practical graph instances $r \leq 4$, so $\mathcal{T}^*(C_i) = O(|E(C_i)|)$ for each C_i as Borůvka's algorithm runs in linear time for these graph sizes. As the C_i are edge disjoint and we have a geometric reduction of the graph size for each recursion, there exists a constant c, so Equation 15.2 evaluates to $T(m,n) \leq cm$. In other words, the running time of the optimal MST algorithm is $O(m)$ for practical graph instances.

16 Implementation

16.1 Implementation details

Graphs with vertices and edges are implemented as described in Chapter 13. The implementation of the building blocks of the optimal algorithm, except the decision trees, are briefly described in their respective chapters. We will not go into further details of these.

As stated in Section 11.5, for all practical number of vertices n in the input graph, the value of r is at most 3 or 4. Hence, in practice, the algorithm only needs to precompute optimal decision trees for graphs with at most 4 vertices, and grow partitions of at most 4 vertices. As stated in Section 11.5, we can find the MST of such graph in linear time of r with Borůvka's algorithm, or just by scanning the edges if $r \leq 3$. Consequently, we have not implemented the decision tree part of the algorithm, but instead we use the methods just described for the `DecisionTree` step of the algorithm.

Furthermore, in our implementation, we in fact do not have an explicit `DecisionTree` step in the main algorithm, and in the `Partition` procedure, we do not explicitly build the C list. Instead, we find the MST of each C_i with the above method, once it is built. The MST edges found in C_i (that is F_i in the description), are marked as MST edges in G, like edges in M are marked as removed. Consequently, the contraction of G to $G_a = G \setminus (F_1 \cup \cdots \cup F_k) - M$ is performed implicitly by a call to `contract(`G`)` (See Chapter 7) straight after the Partition procedure. With this method we avoid to build the C and F lists, and to store all C_i and F_i simultaneously, which saves memory.

With graphs implemented as described in Chapter 13, the implementation of `optimalMST` and other algorithms described in this thesis is relatively straightforward.

16.2 Practical information

The optimal MST algorithm and the other algorithms described in this thesis are implemented in the C++ language. The source code is available on a Compact Disc (CD) in Appendix C.

The files are grouped into multiple folders. The main algorithm `optimalMST` can be found in `optimalmst/OptimalMST.cpp`. Each sub algorithm has its own folder, for example Partition in the folder `partition/`. Each MST algorithm is implemented as a class, and they all have a common interface, namely the `findMSTedges()` method, defined in `graph/MSTFinder.hpp`. The attached source code contains no `main()` function, so it is not possible to compile it out-of-the-box. But given a `Graph` instance, say `g`, with distinct edge weights and an empty `EdgeArray` instance, say `mst`, we can find the MST by instantiate a `MSTFinder` object (for instance an `OptimalMST` object), say `finder`, and invoke `finder.findMSTedges(g,mst,false);` . The third parameter (`false`) disallows removal of `g` during the process.

The project has been compiled with GNU's C++ compiler (g++) version 4.2.4 on a machine with a 64-bit Intel(R) Core(TM)2 Duo 2x 2.13 GHz CPU, 2048 kB cache, and 6 GB of main memory. The operation system was Ubuntu 8.04 (Hardy) with 64-bit Linux kernel version 2.6.24-21-generic.

Part V

Experiments

17 Introduction and correctness

Introduction

All random numbers used in the following experiments originate from the website `http://random.org`, which provides daily updates of files with "truly random" bits. The randomness of the bits comes from atmospheric noise.

All tests instances are categorised into one of two sizes: For "small" instances, the tests have been performed six times for the same instance size. The resulting running time is calculated as the average running time of all six tests, except the minimum and maximum running time. That is, the average of the four tests closed to the median. For "large" instances, the tests have been performed three times for the same instance size. The resulting running time is calculated as the average running time of all three tests.

All experiments have been executed on the machine described in Section 16.2, where the project was compiled. Non-essential system services, such as graphical user interface services, were stopped before the experiments were executed.

Correctness

The correctness of the implementation of the optimal algorithm and the other MST algorithms described in this thesis has been verified, by for several graphs running all algorithms on the same graph, and testing equality for the resulting sets of MST edges.

Even though the algorithms are correct regarding the resulting set of MST edges, their running times might break the theoretical running times described in the thesis. For instance, an inner loop could carelessly be executed too often. Borůvka's algorithm, the DJP algorithm, and thus the Hybrid algorithm, as well as the contraction procedure and the main part of the optimal algorithm, are relatively simple to verify manually. Therefore, we have restricted our selves to plug in simple counters at nontrivial points in the source code for the Dense Case algorithm and the Partition procedure. The counters of the Partition procedure revealed more edge visits than expected during the detection of edges in a component. Specifically, these edges were visited from a dead vertex endpoint. This problem were fixed as described in Section 14.3.

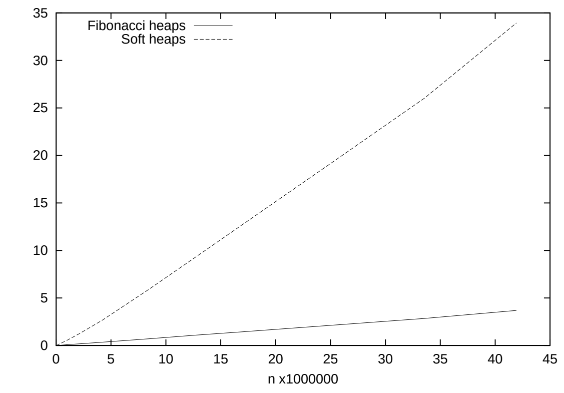
18 Running times of priority queues

As stated in Chapter 6, the amortised running time for the Fibonacci heap insert and deleteMin operations is $O(1)$ and $O(\log n)$, respectively. The amortised running time for soft heaps (Chapter 12) with a constant error rate, such as $\varepsilon = 1/8$ used in the Partition procedure, is $O(1)$ for both operations. Consequently, the running time for the Partition procedure is theoretically smaller using a soft heap. The Big-Oh notation might hide a big constant, so we will perform the following running time experiments regarding Fibonacci heaps and soft heaps with $\varepsilon = 1/8$:

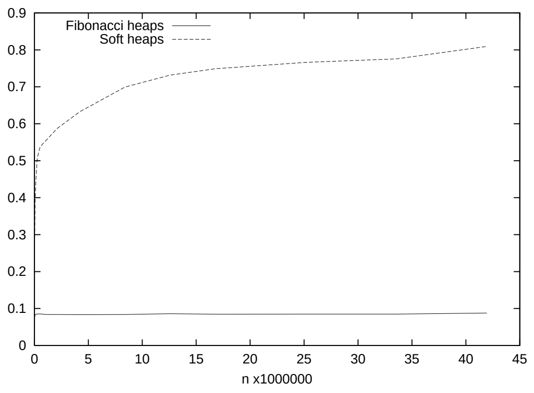
- Running time for n insertions of random keys. Due to constant running time of both heaps, we expect a linear result for both heaps. As the Fibonacci heap insert operation is simpler than the soft heap insert operation, we expect Fibonacci heaps to be fastest, that is a smaller Big-Oh constant. See the result in Figure 18.1.
- Running time for n deleteMins. As the theoretical running time for soft heaps is asymptotically smaller than for Fibonacci heaps, we expect soft heaps to be fastest and linear. See the result in Figure 18.2.
- Heap initialised with n (original) elements. Running time for n operations, each with the probability 0.5. The operation is either insertion of a random key, or deleteMin. This experiment is referred to as "mixed heap operations". As the theoretical running time for soft heaps is asymptotically smaller than for Fibonacci heaps, we expect soft heaps to be fastest and linear. See the result in Figure 18.3.

Additionally, we will test the running time for finding corrupted elements in a soft heap after the third test is complete, as well as count the average number of corrupted items. Here, we expect the running time to be linear. The result of this experiment is presented in Figure 18.4, where Figure 18.4b shows the ratio between the number of corrupted elements and original insertions. That is the number of corrupted items from Figure 18.4a divided by n. As the total number of insertions is $\geq n$, and expected to be $1.5n$, this is an upper bound on the actual ratio.

The results of the heap comparisons are as expected, except for some small irregularities to the right for some plots, which is caused by disk swapping for the large instances. Figures 18.1b–18.4b show the running time results, where the time is divided by n. These plots show for all heap experiments, except Fibonacci heap deleteMin and mixed, that the amortised time per element is constant since they all seem to converge to a constant. Regarding Fibonacci heap deleteMins and mixed in Figure 18.2b and Figure 18.3b, the running time does clearly not converge to a constant, which matches the $O(\log n)$ deleteMin running time.

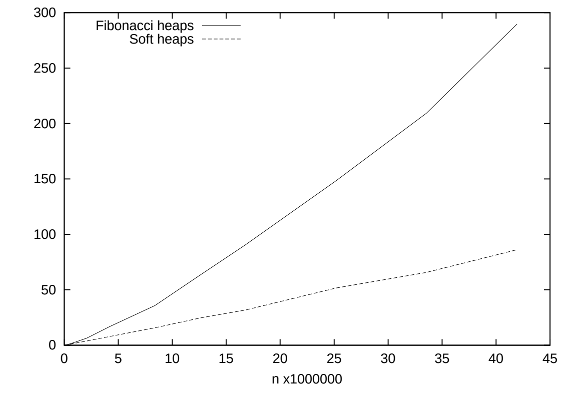


(a) Time in seconds.

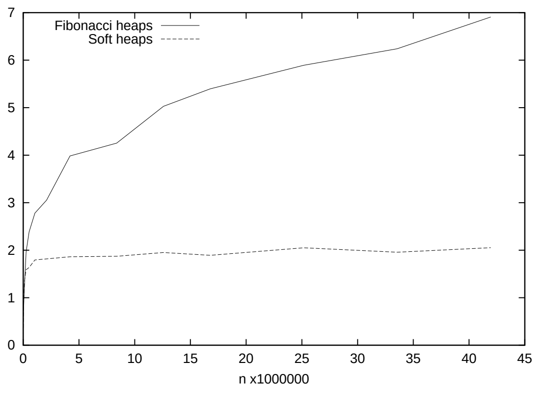


(b) Time divided by $10^{-6}n$.

Figure 18.1: Time for heap insertions.

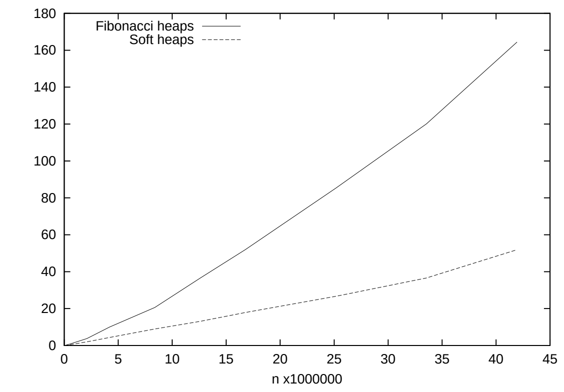


(a) Time in seconds.

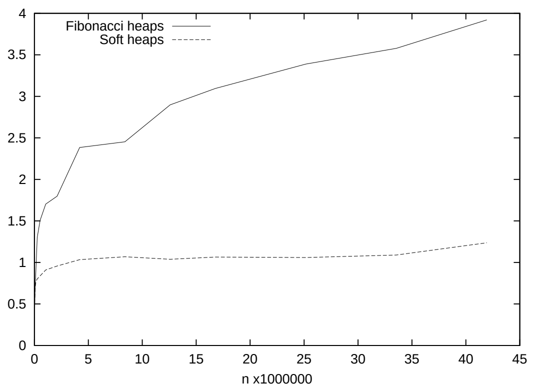


(b) Time divided by $10^{-6}n$.

Figure 18.2: Time for heap deleteMins

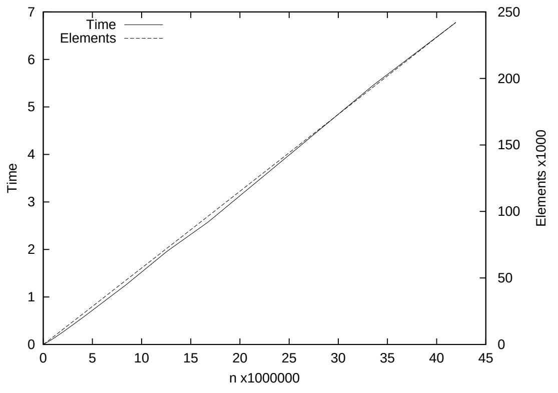


(a) Time in seconds.

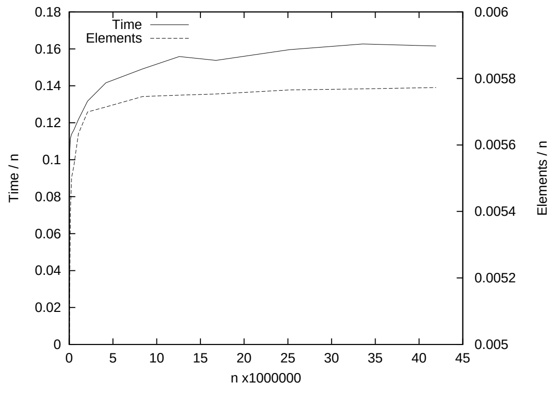


(b) Time divided by $10^{-6}n$.

Figure 18.3: Time for mixed heap operations.



(a) Time in seconds and the number of corrupted elements.



(b) Time divided by $10^{-6}n$ and number of corrupted elements divided by n.

Figure 18.4: Time for finding corrupted elements and number of corrupted elements.

As the soft heap error rate (maximum number of corrupted elements per insertion) is set to $\varepsilon = 1/8 = 0.125$, Figure 18.4b clearly shows for random keys, that the actual number of corrupted elements are significantly smaller than the upper bound guarantee. In fact the ratio seems be bounded by the constant $0.0058 < \varepsilon$.

Conclusion

As expected, for random keys and practical sizes of n, soft heaps are faster than Fibonacci heaps for non insertion-only operations. The drawback is of course that soft heaps corrupts some elements, but for random data the number of corrupted elements are significantly lower than the upper bound guarantee.

19 Running times of MST algorithms

19.1 MST algorithms on random graphs

Algorithm	Running time		
	Planar	Sparse	Dense
Optimal [PR02], Chapter 15	General: $O(\mathcal{T}^*(m,n))$ Practice: $O(m)$		
Borůvka [Bor26], Chapter 8	$O(m)$	$O(m \log n)$	$O(n^2)$
DJP [Jar30, FT87], Chapter 9	$O(n \log n)$		$O(m)$
Dense Case [FT87], Chapter 10	$O(m \log^*(n))$		$O(m)$
Borůvka + DJP, Section 9.4	$O(m)$	$O(m \log \log n)$	$O(n^2)$

Table 19.1: Worst case running times of five MST algorithms.

We will test the running time of the five MST algorithms presented in Table 19.1 for various instances of simple connected random graphs[1].

Notice that with reference to Section 15.6, the optimal algorithm is expected to run in linear time, $O(m)$, for the test graphs we are able to create. The optimal algorithm is clearly the most complicated among the five algorithms and may have a significant overhead. Therefore we expect the hidden Big-Oh constant of the optimal algorithm to be large compared to the other algorithms. The constant may be so large, that the other algorithms are faster for the graphs with sizes we are able to create. The DJP and Dense Case algorithms both run in linear time $O(m)$ for dense graphs. As their hidden Big-Oh constants are smaller, we expect these algorithms to run faster than the optimal algorithm for dense graphs. The same expectation applies to Borůvka's algorithm for graphs where m is $O(n)$.

[1]To be described in the subsequent subsection.

Random graph

A simple connected random graph of n vertices and m edges is built as follows:

1. Make a spanning tree of n vertices and $n-1$ edges: Create n vertices, and initially pick a random “root” vertex to be in the current tree. Then repeatedly augment the tree by connecting a random vertex inside the current tree with a random vertex outside the current tree with an edge. This step ensures the random graph is connected.
2. For the remaining $m-n+1$ edges: Repeatedly pick two random vertices not already connected directly by an edge, and connect these with an edge.

All edges are created with a random integer weight ≥ 1.

Test type 1: Constant density m/n, increasing n

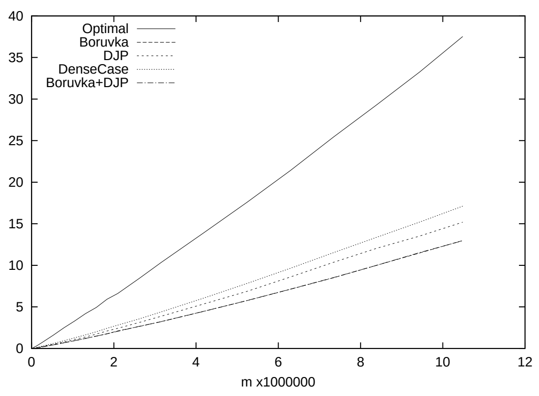
Let $m_{\max} = n\left(n-1\right)/2$, that is the number of edges in the complete graph K_n, which is $O\left(n^2\right)$. We will test the running time of the five algorithms for random graphs with the constant densities presented in Table 19.2 for increasing n.

$m = n$ (one cycle)	$m = 1.5n$ (very sparse)	$m = n \log\log n$
$m = n \log n$	$m = n\sqrt{n}$	$m = m_{\max}/\log n$
$m = m_{\max}/\log\log n$	$m = m_{\max}/1.5$ (very dense)	$m = m_{\max}$ (complete)

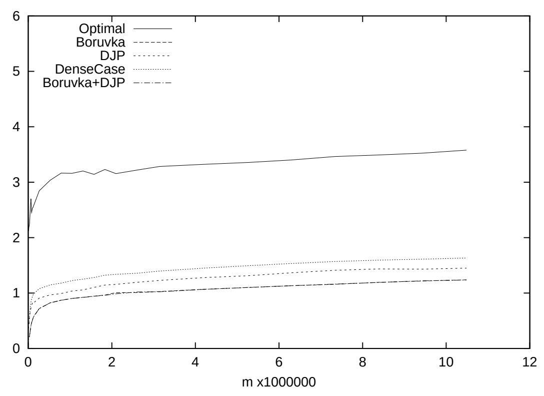
Table 19.2: Constant densities.

The test results are presented in Figures 19.1–19.4, as well as Figures B.1–B.5 in Appendix B. The left plots are the running times and the right plots are the running times divided by m, made to check for linearity. From the plots we can see that the algorithms can be grouped into three categories: 1) The DJP based (DJP and Dense Case). 2) The Borůvka based (Borůvka and Borůvka+DJP). 3) The optimal algorithm, based on both 1 and 2.

Except for the densities $m = n$ and $m = 1.5n$ (where m is $O\left(n\right)$) the picture is clear: The DJP algorithm is the fastest algorithm succeeded by the Dense Case algorithm. Both algorithms theoretically run in linear time for dense graphs. Theoretically, the Dense Case algorithm is faster than DJP when m is $O\left(n \log n\right)$, but this is not reflected by the plots. Intuitively, the hidden Big-Oh constant of the Dense Case algorithm is greater than for the DJP algorithm, because the Dense Case algorithm has some overhead, running DJP multiple times plus some additional work. For sparse graphs where m is $O\left(n\right)$, the running time ratio $O\left(\log n/\log^*\left(n\right)\right)$ between the two algorithms is relative small for our values of n, and it looks like it is smaller than the ratio between the hidden Big-Oh constants.

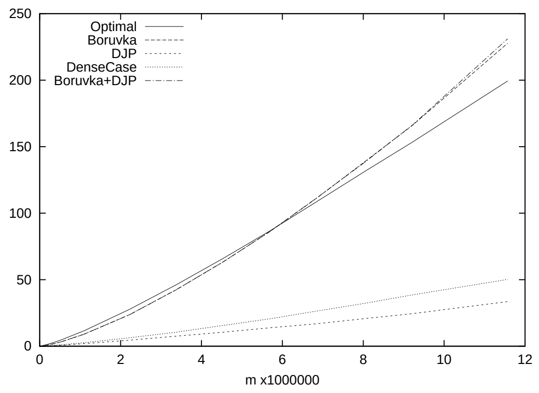


(a) Time in seconds.

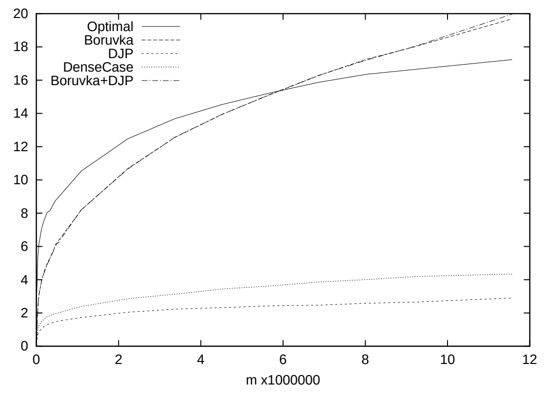


(b) Time divided by $10^{-6}m$.

Figure 19.1: Constant density, $m = n$.

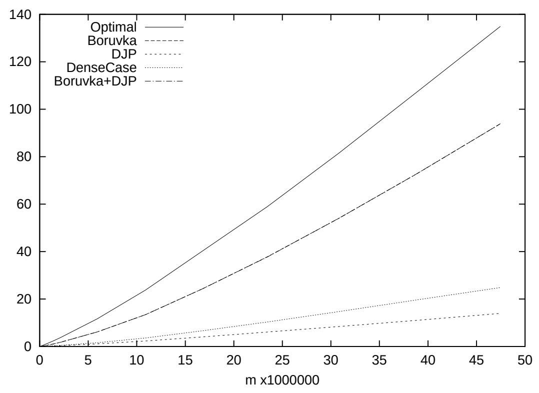


(a) Time in seconds.

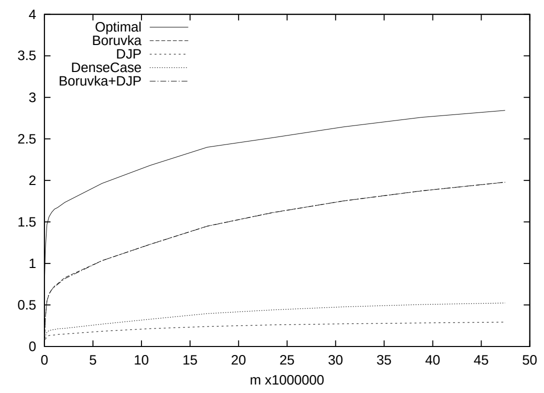


(b) Time divided by $10^{-6}m$.

Figure 19.2: Constant density, $m = n \log \log n$.

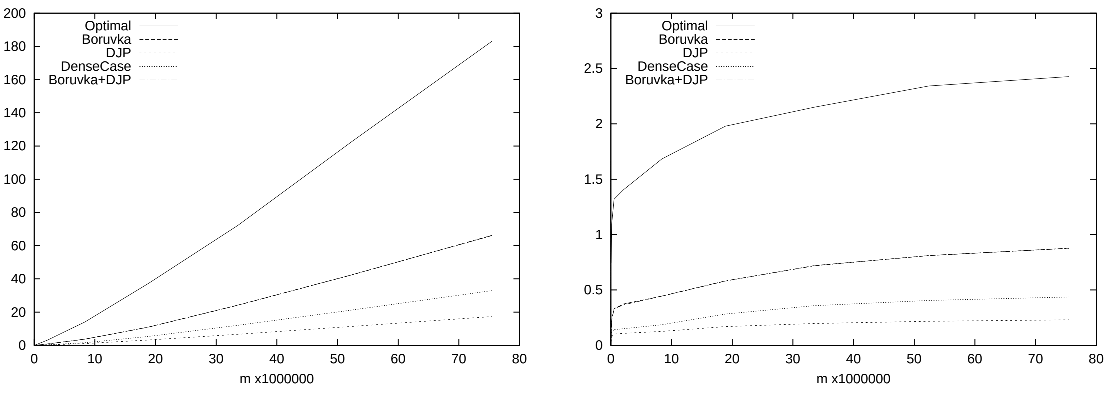


(a) Time in seconds.



(b) Time divided by $10^{-6}m$.

Figure 19.3: Constant density, $m = n\sqrt{n}$.



(a) Time in seconds. (b) Time divided by $10^{-6}m$.

Figure 19.4: Constant density, $m = m_{\max}$.

For all densities, the running times for Borůvka's algorithm and the Borůvka+DJP hybrid algorithm are almost indistinguishable. This can easily be explained by the first $\log\log n$ steps of Borůvka and Borůvka+DJP are identical. Let n' be the number of vertices in the contracted graph after the Borůvka steps. Output from the constant density experiments shows that the ratio between the worst number of vertices after the Borůvka steps and n', that is $(n/\log n)/n'$, is at least 33. This ratio is clearly greater than $\log n$ for our values of n, and thus the number of vertices are reduced by at least $\log^2 n$ in the Borůvka steps. This makes the input graph to DJP very small for our values of n, and thus the difference in running time between the original Borůvka algorithm and the Borůvka+DJP algorithm is insignificant. Furthermore the experiments also show that for some inputs graphs, Borůvka's algorithm needs $\leq \log\log n$ steps to determine the MST, and thus the DJP is not executed at all in the hybrid algorithm, which makes it identical to Borůvka's algorithm. The plot where $m = n$ shows that these algorithm are the fastest, which matches the linear running time of Borůvka's algorithm for very sparse graphs. But even for $m = 1.5n$ in Figure B.1, the DJP algorithm is slightly faster.

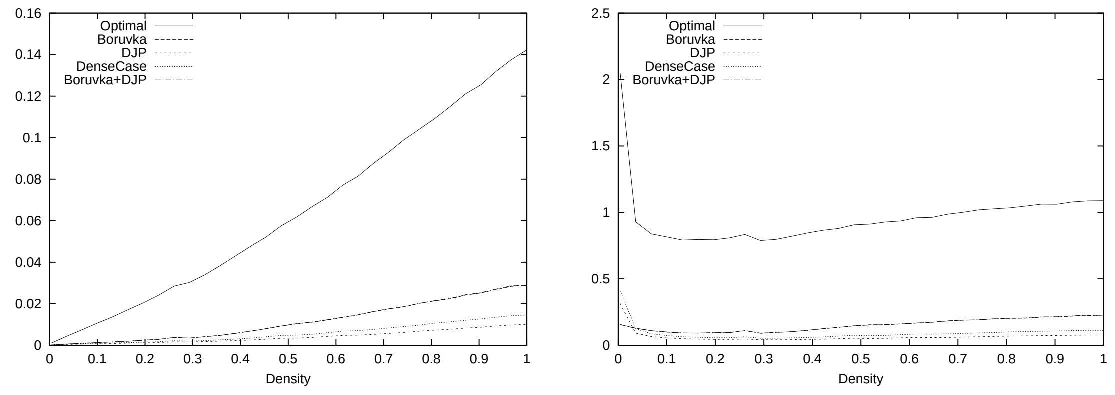
It is clear from all plots, except for $m = \log\log n$ and $m = \log n$, that the running time of the optimal algorithm grows significantly faster than the running time of any other algorithm. For our possible test values of n, m, the (relative large) hidden Big-Oh constant for the optimal algorithm seems to dominate the running time. Only for $m = \log\log n$ and $m = \log n$, the running time of the Borůvka based algorithms exceeds the running time for the optimal algorithm for large values of n.

Type type 2: Constant n, increasing density m/n

$n = 2^9 = 512$	$n = 2^{11} = 2048$	$n = 2^{13} = 8192$	$n = 10^4 = 10000$

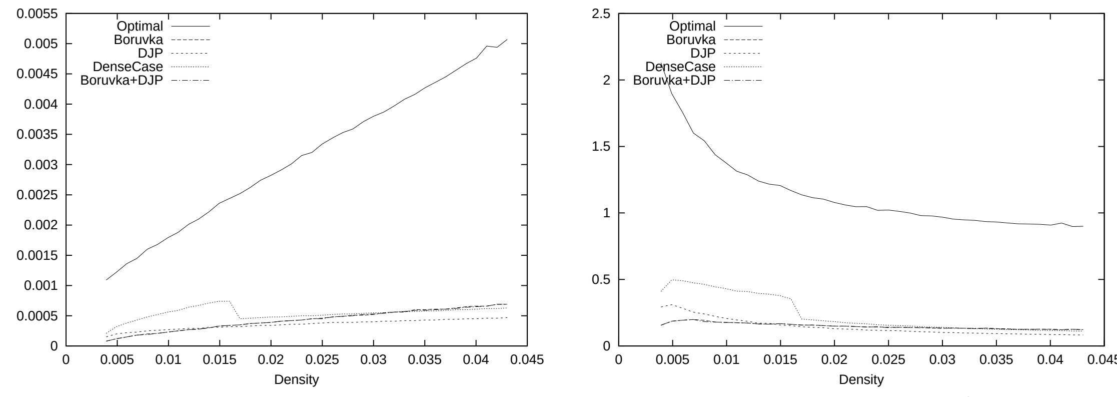
Table 19.3: Constant number of vertices.

We will test the running time for increasing densities (equivalently, m) for the constant values of n presented in Table 19.3. The relative low values of n is because higher values will cause disk swapping on the test machine for high densities. The results are presented in Figure 19.5, as well as Figures B.6–B.8 in Appendix B. Again, the left plots are the running times and the right plots are the running times divided by m, made to check for linearity. Notice the density values]0 , 1] on the x-axis, which is the density function $m/m_{\max}$. Because the interesting densities are very low (relatively sparse graphs), we have also performed these tests specially for sparse graphs. The results are presented in Figure 19.6 and Figure 19.7, as well as Figure B.9 and Figure B.10 in Appendix B. To verify the result that the optimal algorithm is faster than the Borůvka based algorithms for the densities $O(\log\log n)$ and $O(\log n)$ for large values of n, we have also performed the test for sparse graphs with large $n = 2^{21}$. The result is presented in Figure 19.8. For this value of n, we have $n \log\log n/m_{\max} \approx 4.2 \cdot 10^{-6}$.



(a) Time in seconds. (b) Time divided by $10^{-6}m$.

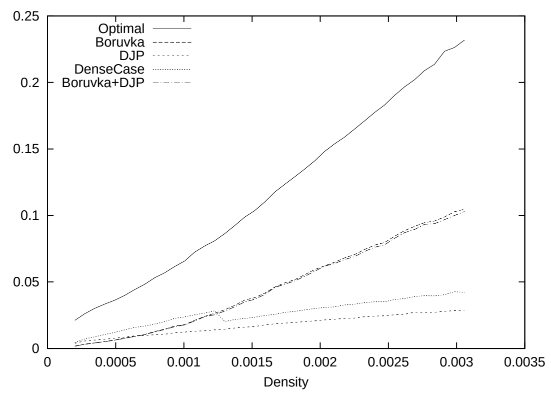
Figure 19.5: Constant $n = 2^9 = 512$.



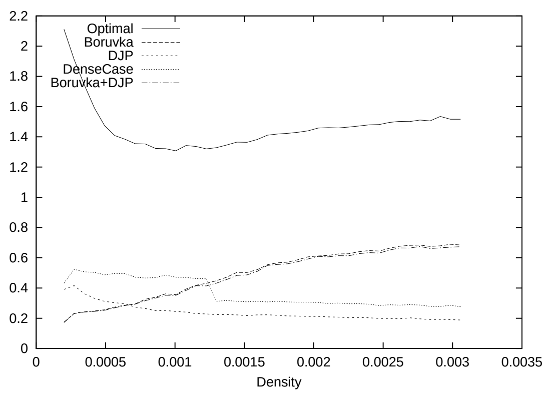
(a) Time in seconds. (b) Time divided by $10^{-6}m$.

Figure 19.6: Constant $n = 2^9 = 512$, sparse.

The plots show nothing notable compared to the plots for constant density: For almost all densities, DJP is the fastest algorithm succeeded by Dense Case, then comes Borůvka and the hybrid, and lastly the relative slow optimal algorithm. One exception is for very sparse graphs, where Borůvka and the hybrid is fastest. The relatively big drop(s) in running time for the Dense Case algorithm can be explained by a drop in the number of

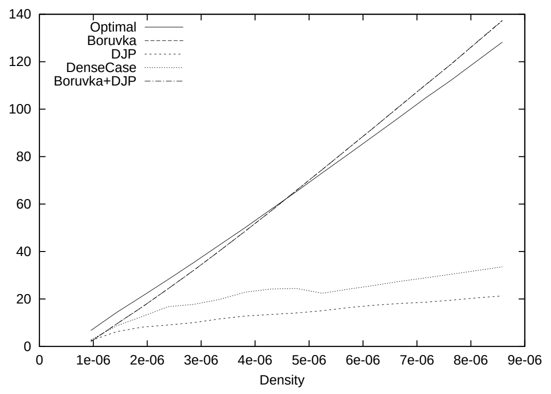


(a) Time in seconds.

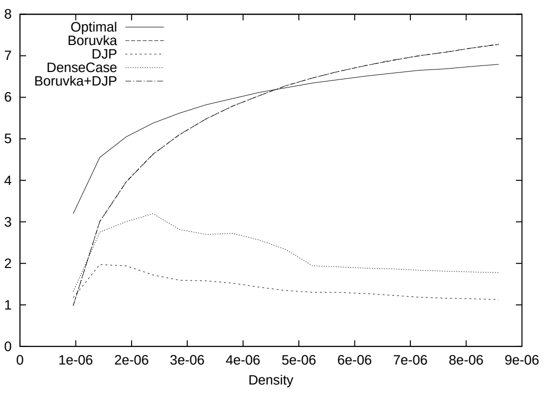


(b) Time divided by $10^{-6}m$.

Figure 19.7: Constant $n = 10^4 = 10000$, sparse.



(a) Time in seconds.



(b) Time divided by $10^{-6}m$.

Figure 19.8: Constant $n = 2^{21}$, sparse.

passes and therefore time consuming contractions. For low values of n we do not see the running time of Borůvka and the hybrid exceed the running time of the optimal algorithm for certain densities. But for $n = 2^{21}$, we see that Borůvka and the hybrid exceeds the running time for the optimal algorithm around the density $m/n = \log\log n$. According to the constant density experiments, even for this value of n, we expect the running times of these algorithms to drop below the running time for the optimal algorithm when the density increases, but the practical limits of the test machine did not allow us to perform this experiment.

19.2 MST experiment follow-ups

All tests, except where m is $O(n)$, show that the DJP algorithm is the fastest among the algorithms for random graphs. A simple observation is that all algorithms, except DJP, uses the contraction procedure. A program profile analysis shows that the average time used for graph contraction in all algorithms is about 40%. So graph contraction is causing a significant overhead for these algorithms. The Dense Case algorithm has a relatively low worst case number of passes ($O(\log^*(n))$) and thus performs a relatively low number of contractions. This can explain why for most densities, it is the fastest algorithm using contraction. For practical graph instances, it seems that the optimal algorithm only is able to beat some of the other algorithms for graphs with a narrow interval of densities, around $m/n = O(\log\log n)$ and $m/n = O(\log n)$. But here it only beats Borůvka's algorithm and the hybrid algorithm. Borůvka's algorithm is clearly the simplest algorithm among the algorithms. Due to its simplicity, it is easy to describe a family of worst case graphs for Borůvka's algorithm, which presumably will make the difference in running time to the optimal algorithm even bigger. We will describe such family of graphs in the subsequent section.

The optimal algorithm detects MST candidate edges from the input graph using the DJP scheme in two variants, starting by the Partition procedure followed by the Dense Case algorithm. Hence, designing worst case graphs for the DJP algorithm (or the Dense Case variant) will presumably also make the optimal algorithm run slower. The optimal algorithm also uses Borůvka steps to detect MST edges, but the edges are candidate edges which are only a subset of the edges from the input graph. Hence, designing a worst-case graph family for Borůvka's algorithm might give a better test result for the optimal algorithm on this family of graphs.

19.3 Worst case graph family for Borůvka's algorithm

We will design a graph which makes Borůvka's algorithm run the worst-case $\log n$ steps, while maximising the sum of edges in each step compared to the original number of edges m. Recall that Borůvka's algorithm reduces the number of vertices by at least factor 2 per step, and the number of edges by at least the number of reduced vertices.

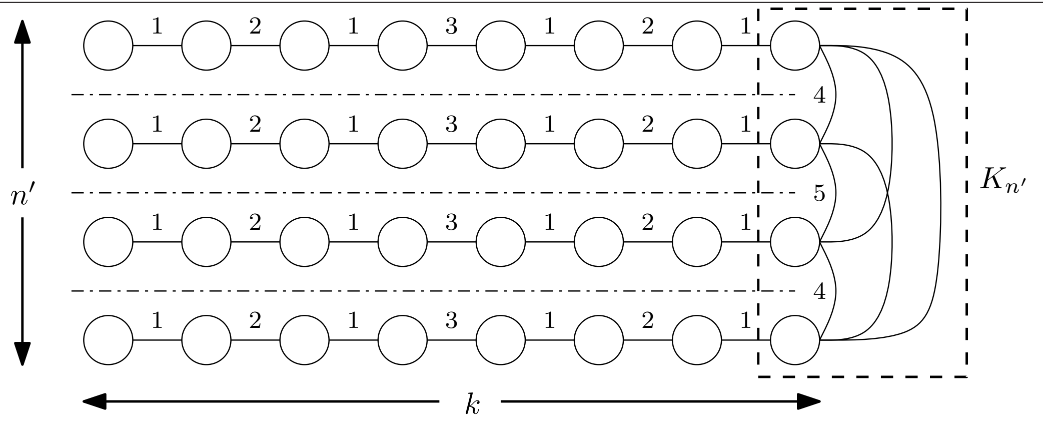


Figure 19.9: Borůvka worst case graph with $k = 8$ and $n' = 4$.

With reference to Figure 19.9, consider a connected sub graph of k vertices and $k-1$ edges formed as one simple path. We assume without loss of generality, that $k = 2^a$ for some integer $a \geq 1$. The edges in the sub graph are assigned weights $w = 1, 2, 3, \ldots, \log k = a$ the following way: For integers $b \geq 0$, assign weight w to the edges connecting vertex

$2^{w-1} + b \cdot 2^w$ and vertex $2^{w-1} + b \cdot 2^w + 1$ on the path. This will clearly make Borůvka's algorithm run $O(\log k)$ steps for such graph, because it will reduce the number of vertices by exactly 2 for each step. Then consider n' of these sub graphs of k vertices, and choose an arbitrary "master vertex" from each sub graph. Then connect all pairs of master vertices with an edge, resulting in a clique (complete sub graph) $K_{n'}$. We assume without loss of generality, that $n' = 2^{a'}$ for some integer a'. Then consider some simple path in the clique involving all n' master vertices, and assign edge weights $\log k + 1, \ldots, \log k + 1 + \log n'$ to these edges like we did for a sub graph path. Any other edge in the clique is assigned an arbitrary unique weight $> \log k + 1 + \log n'$.

This family of graphs will clearly make Borůvka's algorithm initially run $O(\log k)$ steps without contracting edges in the clique. After the $\log k$'th step, the contracted graph will have n' vertices, namely one for each sub graph. Additionally this graph is the complete graph of $O\left(n'^2\right)$ edges originating from the clique. Then the algorithm will run $O(\log n')$ steps where the clique is contracted. In total that is $O(\log k + \log n') = O(\log(kn')) = O(\log n)$ steps for this family of graphs.

It remains to find a k maximising the sum of edges in all steps compared to the original number of edges m. That is maximise $\sum_i m_i/m$, where m_i is the number of edges in the i'th step.

There are k vertices per n', summing to $n = O(kn')$ vertices in the original graph. There are $k-1$ or $O(k)$ edges per n', and $O\left(n'^2\right)$ edges in the clique, summing to $m = O\left(n'^2 + n'k\right)$ edges in the original graph. As the algorithm runs $O(\log k)$ steps before the clique starts to be is contracted, the sum of edges in the first $O(\log k)$ steps is $O\left(n'^2 \log k + n'k\right)$. As the clique by definition is a complete graph, the sum of edges in the subsequent $O(\log n')$ steps is $O\left(n'^2\right)$, which is dominated by the first sum.

Consequently, the ratio between the sum of edges in all steps and the original number of edges is

$$\frac{\sum_i m_i}{m} = O\left(\frac{n'^2 \log k + n'k}{n'^2 + n'k}\right) = O\left(\frac{n'(n' \log k)}{n'(n'+k)}\right) = O\left(\frac{n' \log k}{n'+k}\right) .$$

We can find the maximum point of this function by an extreme analysis. The slope of a function is zero at its extreme points. We can find where the slope is zero by finding the first derivative of the function, and calculate points with zero slope. As we must find a k maximising the function, we must find the first derivative of the function where n' is treated as a constant, and solve the equation where the first derivative equals zero.
The first derivative of $f(k) = \log k = \ln k / \ln 2$ is $f'(k) = 1/(k \ln 2)$, which is $O(1/k)$.

The first derivative of a function $f(k) = g(k)/h(k)$ is $\left(g'(k)h(k) - g(k)h'(k)\right)/h^2(k)$, so we have

$$\begin{aligned} 0 &= \frac{n'/k \cdot (n'+k) - n' \log k \cdot 1}{(n'+k)^2} \\ &= n'^2/k + n' - n' \log k \\ &= n'^2 + n'k - n'k \log k \\ &= n'\left(n' + (k - k \log k)\right) . \end{aligned}$$

Setting $n' = O(k \log k - k) = O(k \log k)$ maximises the function.

Consequently, for some k, the worst case number of vertices is $n = O(kn') = O\left(k^2 \log k\right)$, and the number of edges is $m = O\left(n'^2 + kn'\right) = O\left(k^2 \log^2 k + k^2 \log k\right) = O\left(k^2 \log^2 k\right)$. So the density is $m/n = O(\log k)$. There exists no non-recursive function for k, but $n = O\left(k^2 \log k\right) \Rightarrow k = O\left(\sqrt{n}/\sqrt{\log k}\right)$, so

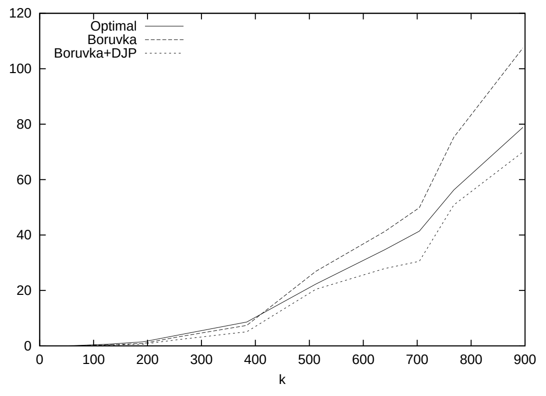
$$m/n = O\left(\log \sqrt{n} - \log \sqrt{\log k}\right) = O(\log n - \log \log k) \ .$$

As $k \leq n$ the density is bounded by $\Omega(\log \log n)$ and $O(\log n)$. These density bounds match the results in Figure 19.2 and Figure B.2, where the running time of Borůvka's algorithm exceeds the running time of the optimal algorithm.

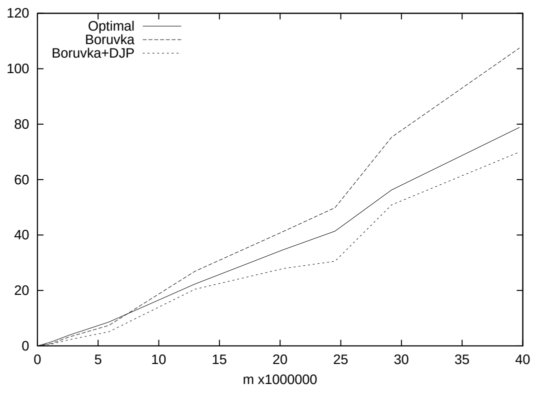
Results

We have tested the running time for this graph family for the optimal algorithm, Borůvka's algorithm, and the Borůvka+DJP hybrid algorithm. Apart from observing better running times for the optimal algorithm compared to Borůvka, we expect to observe a significant difference in the running times for Borůvka and Borůvka+DJP, because the hybrid only runs the first $\log \log n$ of the $\log n$ Borůvka steps.

Figure 19.10a shows the running time results for this graph family as a function of k. Figure 19.10b shows the same results, but as a function of m. Figure 19.11a shows the running time divided by m, and Figure 19.11b show the running time divided by $m \log n$, which is the expected running time for Borůvka's algorithm.



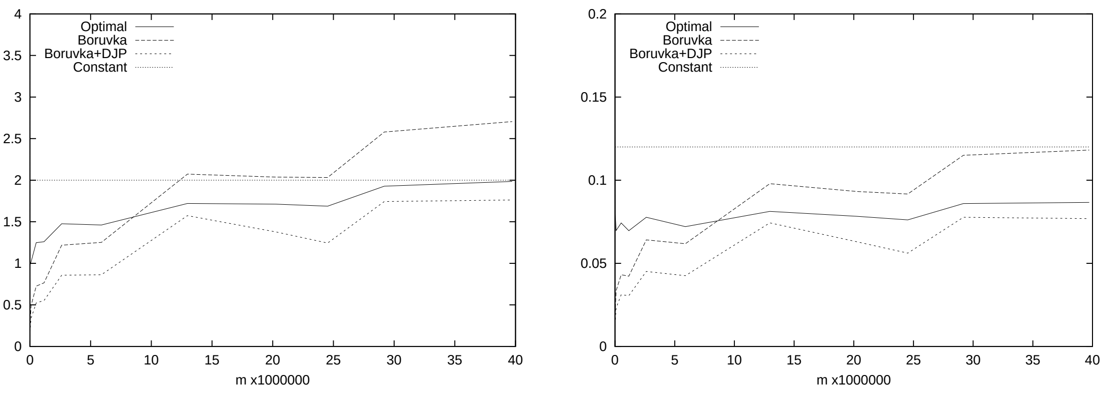
(a) Time in seconds as function of k.



(b) Time in seconds as function of m.

Figure 19.10: Running times for the Borůvka worst case graph.

As expected, the plots show that the optimal algorithm is significantly better than Borůvka's algorithm for this graph family. Here for $m > 8 \cdot 10^6$. The running time of the hybrid algorithm is also significantly better than for Borůvka's algorithm, and moreover as a side effect, it is also better than for the optimal algorithm. But compared to the density $m = n \log \log n$ of random graphs (Figure 19.2), this result is not better, because 1) the optimal algorithm is better for $m > 6 \cdot 10^6$ on random graphs, 2) the difference in running time seems to grow faster for random graphs. Furthermore, for this graph family the optimal algorithm is slower than the hybrid algorithm.



(a) Time divided by $10^{-6}m$.

(b) Time divided by $10^{-6}m \log n$.

Figure 19.11: Running times for the Borůvka worst case graph.

Consequently, this special worst case graph for Borůvka's algorithm does not make the optimal MST algorithm faster compared to Borůvka's algorithm. A simple explanation is that even though the Borůvka steps of the optimal algorithm only process a subset of the original edges of the worst case graph, then these are adequate to also make the optimal algorithm similar slower for practical instances.

19.4 Overall MST experiment results

For practical graph instances, we have shown that the optimal MST algorithm can be faster than Borůvka's algorithm for a narrow interval of graph densities. Even though this sub result is positive from the perspective of the optimal MST algorithm, for all practical graph instances, there exists MST algorithms which are significantly faster in practice. Among the tested algorithms, DJP is clearly the winning algorithm when it comes to running time in practice. However, with reference to Table 4.1 in Section 4.3, there may be faster MST algorithms for practical graph instances.

References

[AK07] Claus Andersen and Henrik Bitsch Kirk. Advanced algorithms - data structures 2007, projekt 1 - prioritetskøer. 2007.

[Bor26] O. Borůvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, 3:37–58, 1926. In Czech.

[Cha00a] Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000. Abstract only.

[Cha00b] Bernard Chazelle. The soft heap: An approximate priority queue with optimal error rate. *J. ACM*, 47(6):1012–1027, 2000.

[Flo64] Robert W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7(12):701, December 1964.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[Jar30] V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 6:57–63, 1930. In Czech.

[KKT95] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995. Abstract only.

[KZ09] Haim Kaplan and Uri Zwick. A simpler implementation and analysis of Chazelle's Soft Heaps. To appear, SODA, 2009.

[Mar04] Martin Mareš. Two linear time algorithms for MST on minor closed graph classes. *Archivum Mathematicum*, 40:315–320, 2004.

[PR02] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, 2002.

[Som58] D.M.Y. Sommerville. *An introduction to the geometry of N dimensions*. Dover Publ. Inc., New York, 1958. Euler's formula only.

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

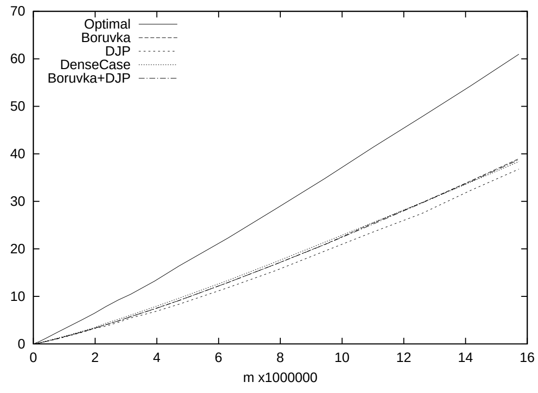
Appendix

A Index of terms used

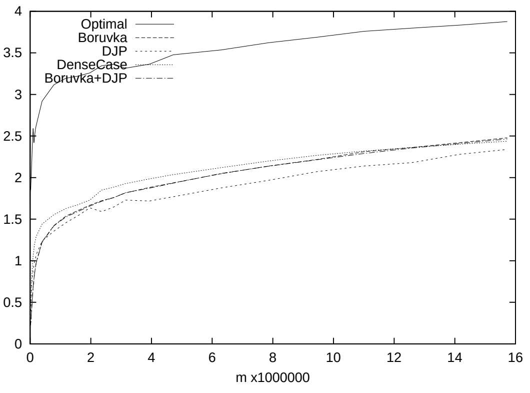
DFS	Depth-First Search.
DJP	Dijkstra-Jarník-Prim.
MSF	Minimum Spanning Forest.
MST	Minimum Spanning Tree.

B MST experiment plots

Running times for constant density m/n

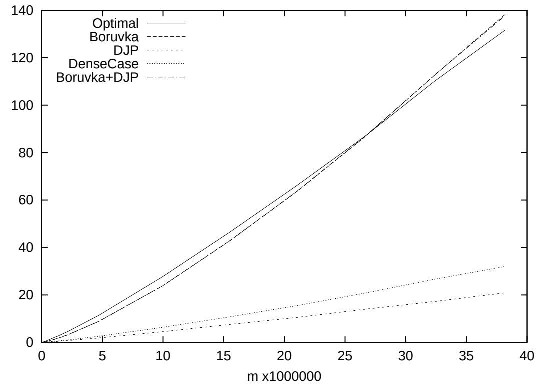


(a) Time in seconds.

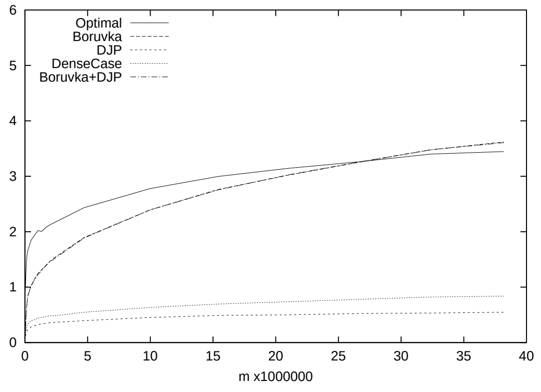


(b) Time divided by $10^{-6}m$.

Figure B.1: Constant density, $m = 1.5n$.

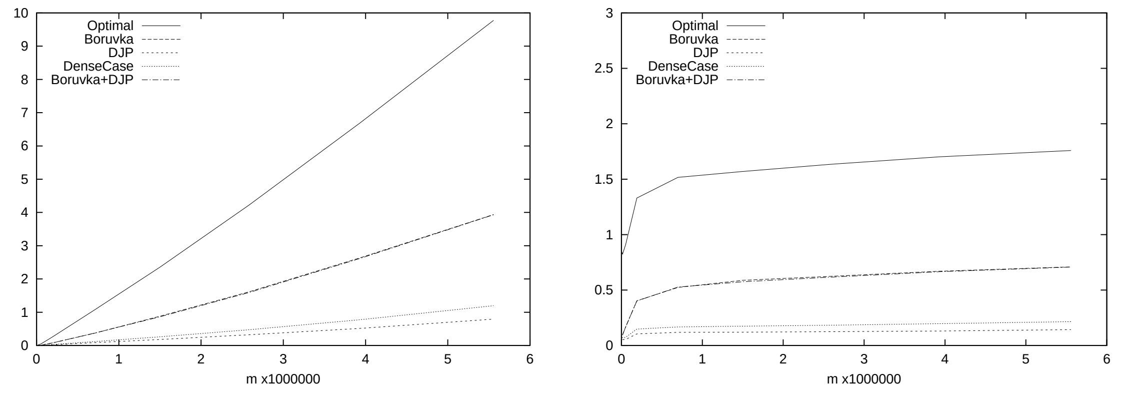


(a) Time in seconds.



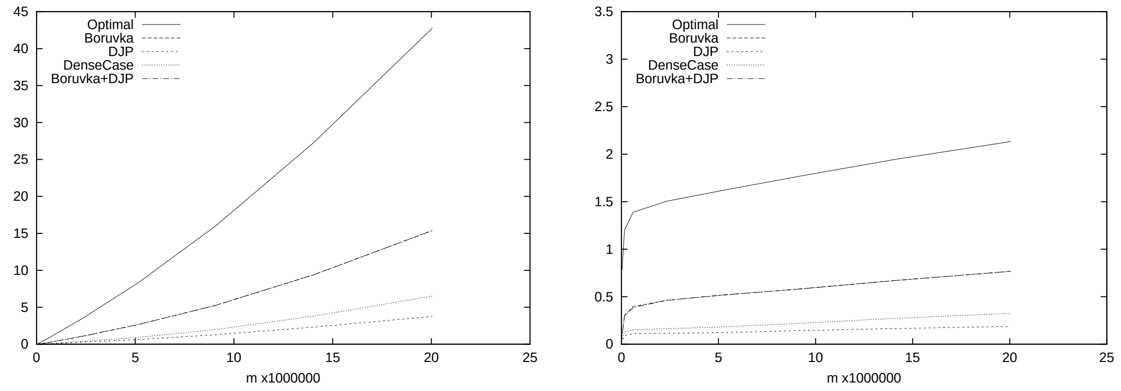
(b) Time divided by $10^{-6}m$.

Figure B.2: Constant density, $m = n \log n$.



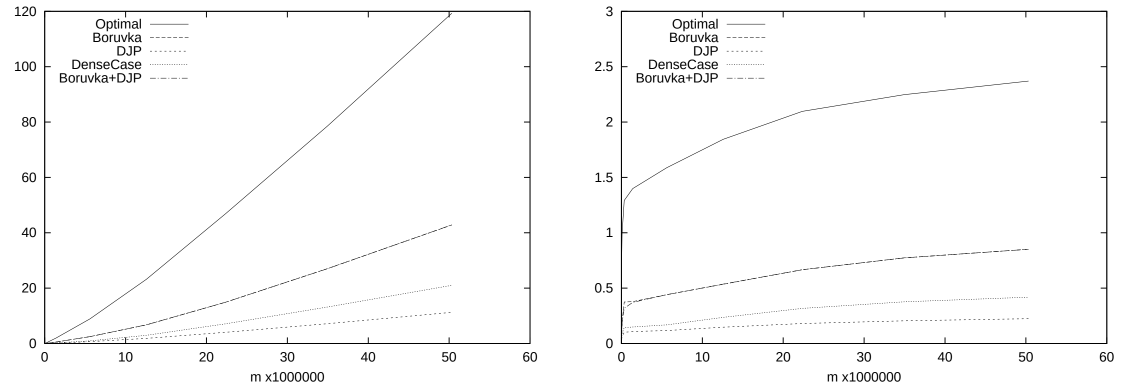
(a) Time in seconds. (b) Time divided by $10^{-6}m$.

Figure B.3: Constant density, $m = m_{\max}/\log n$.



(a) Time in seconds. (b) Time divided by $10^{-6}m$.

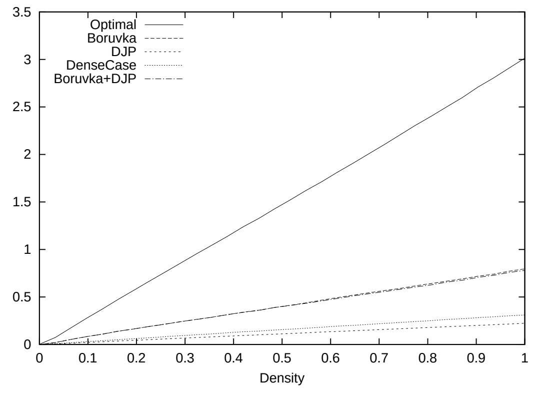
Figure B.4: Constant density, $m = m_{\max}/\log\log n$.



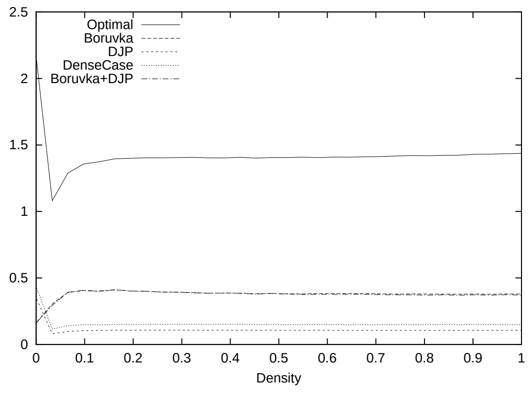
(a) Time in seconds. (b) Time divided by $10^{-6}m$.

Figure B.5: Constant density, $m = m_{\max}/1.5$.

Running times for constant n

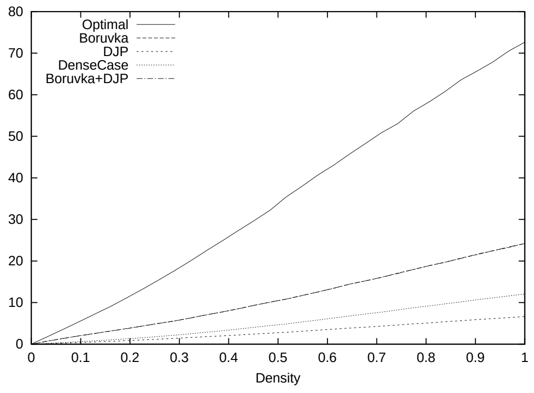


(a) Time in seconds.

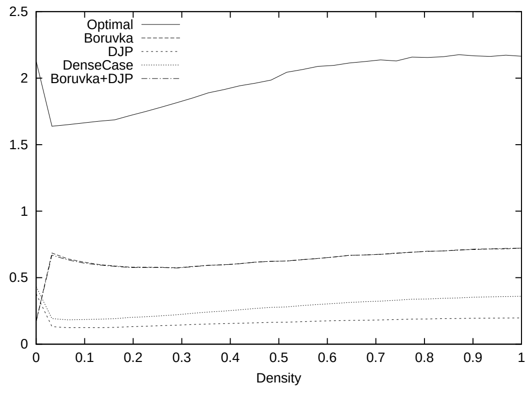


(b) Time divided by $10^{-6}m$.

Figure B.6: Constant $n = 2^{11} = 2048$.

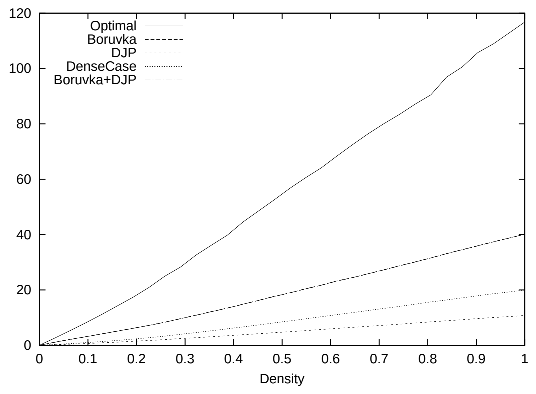


(a) Time in seconds.

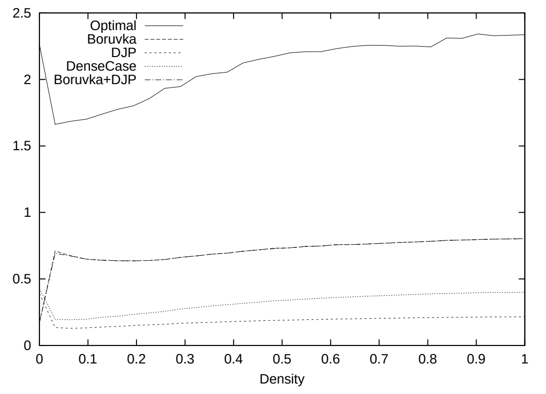


(b) Time divided by $10^{-6}m$.

Figure B.7: Constant $n = 2^{13} = 8192$.

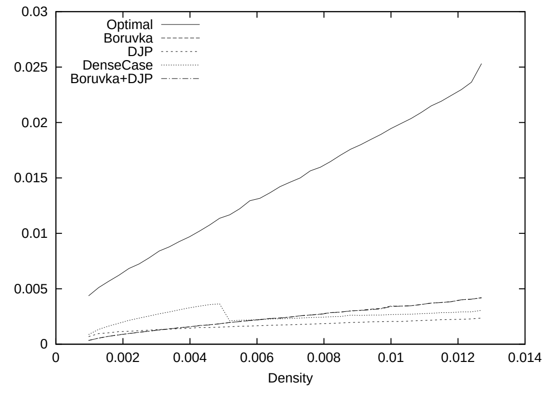


(a) Time in seconds.

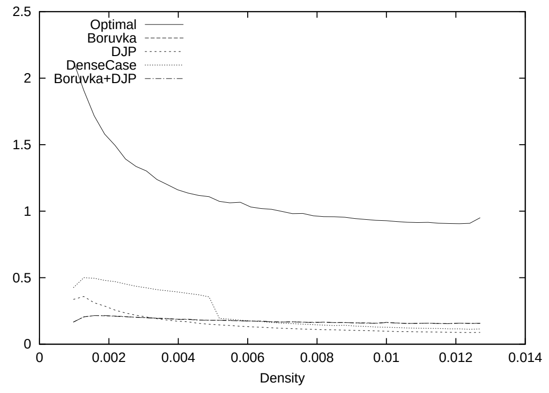


(b) Time divided by $10^{-6}m$.

Figure B.8: Constant $n = 10^4 = 10000$.

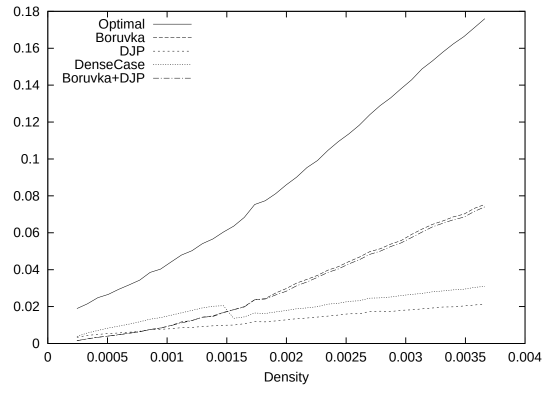


(a) Time in seconds.

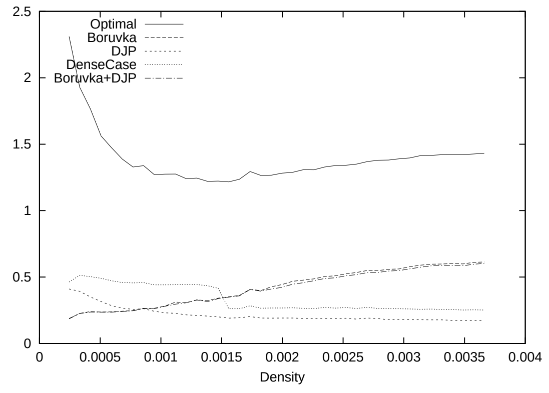


(b) Time divided by $10^{-6}m$.

Figure B.9: Constant $n = 2^{11} = 2048$, sparse.



(a) Time in seconds.



(b) Time divided by $10^{-6}m$.

Figure B.10: Constant $n = 2^{13} = 8192$, sparse.

C Source code

The source code produced for this thesis is attached in the plastic pocket on a Compact Disc (CD).