Charles University in Prague
Faculty of Mathematics and Physics

Doctoral Thesis

Graph Algorithms

Martin Mareš

Department of Applied Mathematics
Malostranské nám. 25
Prague, Czech Republic

Supervisor: Prof. RNDr. Jaroslav Nešetřil, DrSc.
Branch I4: Discrete Models and Algorithms

2008

I hereby declare that I have written this thesis on my own and using exclusively the cites sources. For any work in the thesis that has been co-published with other authors, I have the permission of them all to include this work in my thesis. I authorize the Charles University to lend this document to other institutions and individuals for academic or research purposes.

Martin Mareš
Prague, May 4th, 2008

Preface

This thesis tells the story of two well-established problems of algorithmic graph theory: the minimum spanning trees and ranks of permutations. At distance, both problems seem to be simple, boring and already solved, because we have polynomial-time algorithms for them since ages. But when we come closer and seek algorithms that are really efficient, the problems twirl and twist and withstand many a brave attempt at the optimum solution. They also reveal a vast and diverse landscape of a deep and beautiful theory. Still closer, this landscape turns out to be interwoven with the intricate details of various models of computation and even of arithmetics itself.

I have tried to cover all known important results on both problems and unite them in a single coherent theory. At many places, I have attempted to contribute my own little stones to this mosaic: several new results, simplifications of existing ones, and last, but not least filling in important details where the original authors have missed some.

When compared with the earlier surveys on the minimum spanning trees, most notably Graham and Hell [GH85] and Eisner [Eis97], this work adds many of the recent advances, the dynamic algorithms and also the relationship with computational models. No previous work covering the ranking problems in their entirety is known.

The early parts of this thesis also served as a basis for a course on graph algorithms which I was teaching at our faculty during years 2006 and 2007. They are included in the textbook [Mar07] which I have written for this course.

My original results

- The lower bound in Section 1.5. Not published yet.
- The tree isomorphism algorithm in Section 2.2. Not published yet.
- Both algorithms for minor-closed graph classes in Section 3.1. Published in [Mar04].
- The linear-time verification algorithm in Section 3.4 is a simplification of the algorithm of King [Kin97] and it corrects many omissions in the original paper. Not published yet.
- The ranking algorithms in Sections 7.1 to 7.3 are results of joint research with Milan Straka. Published in [MS07].
- The remaining sections of Chapter 7 contain unpublished original results.

Other minor contributions

- The flattening procedure in Section 2.2. Included in [Mar04].
- The unified view of vector computations in Section 2.4. Published in the textbook [Mar07]. The main ideas of this section were also included in the yearbook of the Czech Mathematical Olympiad [HMN+07].
- Slight simplifications of the soft heaps and their analysis in Section 4.1.
- The dynamic MST algorithm for graphs with limited edge weights in Section 5.4.

Acknowledgements

First of all, I would like to thank my supervisor, Jaroslav Nešetřil, for introducing me to the world of discrete mathematics and gently guiding my attempts to explore it with his deep insight. I am very grateful to all members of the Department of Applied Mathematics and the Institute for Theoretical Computer Science for the work environment which was friendly and highly inspiring. I cannot forget the participants of the department's seminars, who have listened to my talks and provided lots of important feedback. I also send my acknowledgements to the members of the Math department at ETH Zürich and of DIMACS at the Rutgers University (especially to János Komlós) where I spent several pleasant months working on what finally become a part of this thesis.

I also thank to my family for supporting me during the plentiful years of my study, to my girlfriend Anička for lots of patience when I was caught up by my work and hardly speaking at all, to all the polar bears of Kobylisy for their furry presence, and finally to our cats Minuta and Damián for their mastership in hiding my papers, which has frequently forced me to think of new ways of looking at problems when the old ones were impossible to find.

Notation

I have tried to stick to the usual notation except where it was too inconvenient. Most symbols are defined at the place where they are used for the first time. A complete index of symbols with pointers to their definitions is then available in Appendix A. This appendix also describes the formalism of multigraphs and of the Ackermann's function, both of which are not defined consistently in the common literature.

To avoid piling up too many symbols at places that speak about a single fixed graph, this graph is always called G, its set of vertices and edges are denoted by V and E respectively, and I also use n for the number of its vertices and m for the number of edges. At places where there could be a danger of confusion, more explicit notation is used instead.

So, my gentle reader, let us nestle deep in an ancient wing armchair. The saga of the graph algorithms begins ...

Table of contents

1. Minimum Spanning Trees

1.1. The Problem

The problem of finding a minimum spanning tree of a weighted graph is one of the best studied problems in the area of combinatorial optimization since its birth. Its colorful history (see [GH85] and [Nešˇ97] for the full account) begins in 1926 with the pioneering work of Borůvka [Bor26a][1], who studied primarily an Euclidean version of the problem related to planning of electrical transmission lines (see [Bor26b]), but gave an efficient algorithm for the general version of the problem. As it was well before the dawn of graph theory, the language of his paper was complicated, so we will better state the problem in contemporary terminology:

1.1.1. Problem. Given an undirected graph G with weights $w : E(G) \to \mathbb{R}$, find its minimum spanning tree, defined as follows:

1.1.2. Definition. For a given graph G with weights $w : E(G) \to \mathbb{R}$:

- A subgraph $H \subseteq G$ is called a *spanning subgraph* if $V(H) = V(G)$.
- A *spanning tree* of G is any spanning subgraph of G that is a tree.
- For any subgraph $H \subseteq G$ we define its *weight* $w(H) := \sum_{e \in E(H)} w(e)$. When comparing two weights, we will use the terms *lighter* and *heavier* in the obvious sense.
- A *minimum spanning tree (MST)* of G is a spanning tree T such that its weight $w(T)$ is the smallest possible among all the spanning trees of G.
- For a disconnected graph, a *(minimum) spanning forest (MSF)* is defined as a union of (minimum) spanning trees of its connected components.

Borůvka's work was further extended by Jarník [Jar30], again in mostly geometric setting. He has discovered another efficient algorithm. However, when computer science and graph theory started forming in the 1950's and the spanning tree problem was one of the central topics of the flourishing new disciplines, the previous work was not well known and the algorithms had to be rediscovered several times.

In the next 50 years, several significantly faster algorithms were discovered, ranging from the $\mathcal{O}(m\,\beta(m,n))$ time algorithm by Fredman and Tarjan [FT87], over algorithms with inverse-Ackermann type complexity by Chazelle [Cha00a] and Pettie [Pet99], to an algorithm by Pettie [PR02b] whose time complexity is provably optimal.

In the upcoming chapters, we will explore this colorful universe of MST algorithms. We will meet the canonical works of the classics, the clever ideas of their successors, various approaches to the problem including randomization and solving of important special cases. At several places, we will try to contribute our little stones to this mosaic.

[1] See [NMN01] for an English translation with commentary.

1.2. Basic properties

In this section, we will examine the basic properties of spanning trees and prove several important theorems which will serve as a foundation for our MST algorithms. We will mostly follow the theory developed by Tarjan in [Tar83].

For the whole section, we will fix a connected graph G with edge weights w and all other graphs will be spanning subgraphs of G. We will use the same notation for the subgraphs as for the corresponding sets of edges.

First of all, let us show that the weights on edges are not necessary for the definition of the MST. We can formulate an equivalent characterization using an ordering of edges instead.

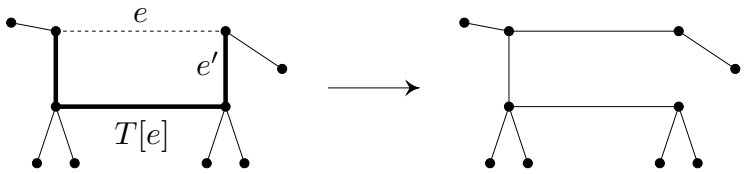
1.2.1. Definition. *(Heavy and light edges)*
Let T be a spanning tree. Then:

- For vertices x and y, let $T[x,y]$ denote the (unique) path in T joining x with y.
- For an edge $e = xy$ we will call $T[e] := T[x,y]$ the *path covered by* e and the edges of this path *edges covered by* e.
- An edge e is called *light with respect to* T (or just T*-light*) if it covers a heavier edge, i.e., if there is an edge $f \in T[e]$ such that $w(f) > w(e)$.
- An edge e is called T*-heavy* if it covers a lighter edge.

1.2.2. Remark. Edges of the tree T cover only themselves and thus they are neither heavy nor light. The same can happen if an edge outside T covers only edges of the same weight, but this will be rare because all edge weights will be usually distinct.

1.2.3. Lemma. *(Light edges)*
Let T be a spanning tree. If there exists a T-light edge, then T is not minimum.

Proof. If there is a T-light edge e, then there exists an edge $e' \in T[e]$ such that $w(e') > w(e)$. Now $T - e'$ (T with the edge e' removed) is a forest of two trees with endpoints of e located in different components, so adding e to this forest must restore connectivity and $T' := T - e' + e$ is another spanning tree with weight $w(T') = w(T) - w(e') + w(e) < w(T)$. Hence T could not have been minimum. ♠

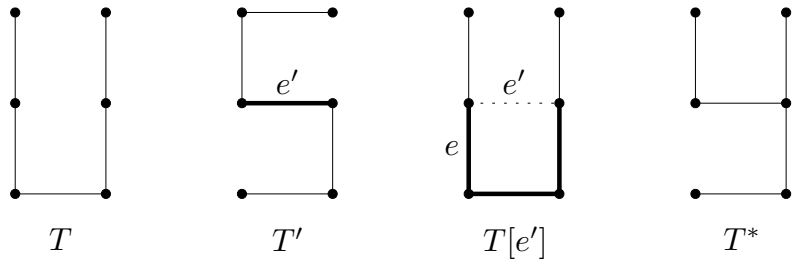


An edge exchange as in the proof of Lemma 1.2.3

The converse of this lemma is also true and to prove it, we will once again use the technique of transforming trees by *exchanges of edges.* In the proof of the lemma, we have made use of the fact that whenever we exchange an edge e of a spanning tree for another edge f covered by e, the result is again a spanning tree. In fact, it is possible to transform any spanning tree to any other spanning tree by a sequence of exchanges.

1.2.4. Lemma. *(Exchange property for trees)*
Let T and T' be spanning trees of a common graph. Then there exists a sequence of edge exchanges that transforms T to T'. More formally, there exists a sequence of spanning trees $T = T_0, T_1, \ldots, T_k = T'$ such that $T_{i+1} = T_i - e_i + e'_i$ where $e_i \in T_i$ and $e'_i \in T'$.

Proof. By induction on $d(T, T') := |T \,\Delta\, T'|$. When $d(T, T') = 0$, both trees are identical and no exchanges are needed. Otherwise, the trees are different, but as they have the same number of edges, there must exist an edge $e' \in T' \setminus T$. The cycle $T[e'] + e'$ cannot be wholly contained in T', so there also must exist an edge $e \in T[e'] \setminus T'$. Exchanging e for e' yields a spanning tree $T^* := T - e + e'$ such that $d(T^*, T') = d(T, T') - 2$. Now we can apply the induction hypothesis to T^* and T' to get the rest of the exchange sequence. ♠



One step of the proof of Lemma 1.2.4

In some cases, a much stronger statement is true:

1.2.5. Lemma. *(Monotone exchanges)*
Let T be a spanning tree such that there are no T-light edges and T' be an arbitrary spanning tree. Then there exists a sequence of edge exchanges transforming T to T' such that the weight of the tree does not decrease in any step.

Proof. We improve the argument from the previous proof, refining the induction step. When we exchange $e \in T$ for $e' \in T' \setminus T$ such that $e \in T[e']$, the weight never drops, since e' is not a T-light edge and therefore $w(e') \geq w(e)$, so $w(T^*) = w(T) - w(e) + w(e') \geq w(T)$.

To keep the induction going, we have to make sure that there are still no light edges with respect to T^*. In fact, it is enough to avoid such edges in $T' \setminus T^*$, since these are the only edges considered by the induction steps. To accomplish that, we replace the so far arbitrary choice of $e' \in T' \setminus T$ by picking the lightest such edge.

Let us consider an edge $f \in T' \setminus T^*$. We want to show that f is not T^*-light, i.e., that it is heavier than all edges on $T^*[f]$. The path $T^*[f]$ is either identical to the original path $T[f]$ (if $e \notin T[f]$) or to $T[f] \,\Delta\, C$, where C is the cycle $T[e'] + e'$. The former case is trivial, in the latter we have $w(f) \geq w(e')$ due to the choice of e' and all other edges on C are lighter than e' as e' was not T-light. ♠

This lemma immediately implies that Lemma 1.2.3 works in both directions:

1.2.6. Theorem. *(Minimality of spanning trees)*
A spanning tree T is minimum iff there is no T-light edge.

Proof. If T is minimum, then by Lemma 1.2.3 there are no T-light edges. Conversely, when T is a spanning tree without T-light edges and T_{min} is an arbitrary

minimum spanning tree, then according to the Monotone exchange lemma (1.2.5) there exists a non-decreasing sequence of exchanges transforming T to T_{min}, so $w(T) \leq w(T_{min})$ and thus T is also minimum. ♠

In general, a single graph can have many minimum spanning trees (for example a complete graph on n vertices with unit edge weights has n^{n-2} minimum spanning trees according to the Cayley's formula [Cay89]). However, as the following theorem shows, this is possible only if the weight function is not injective.

1.2.7. Theorem. *(Uniqueness of MST)*
If all edge weights are distinct, then the minimum spanning tree is unique.

Proof. Consider two minimum spanning trees T_1 and T_2. According to the previous theorem, there are no light edges with respect to neither of them, so by the Monotone exchange lemma (1.2.5) there exists a sequence of non-decreasing edge exchanges going from T_1 to T_2. As all edge weights all distinct, these edge exchanges must be in fact strictly increasing. On the other hand, we know that $w(T_1) = w(T_2)$, so the exchange sequence must be empty and indeed T_1 and T_2 must be identical. ♠

1.2.8. Notation. When G is a graph with distinct edge weights, we will use $\mathrm{mst}(G)$ to denote its unique minimum spanning tree.

Also the following trivial lemma will be often invaluable:

1.2.9. Lemma. *(Edge removal)*
Let G be a graph with distinct edge weights and $e \in G \setminus \mathrm{mst}(G)$. Then $\mathrm{mst}(G-e) = \mathrm{mst}(G)$.

Proof. The tree $T = \mathrm{mst}(G)$ is also a MST of $G - e$, because every T-light edge in $G-e$ is also T-light in G. Then we apply the uniqueness of the MST of $G-e$. ♠

1.2.10. *Comparison oracles.* To simplify the description of MST algorithms, we will assume that the weights of all edges are distinct and that instead of numeric weights we are given a comparison oracle. The oracle is a function that answers questions of type "Is $w(e) < w(f)$?" in constant time. This will conveniently shield us from problems with representation of real numbers in algorithms and in the few cases where we need a more concrete input, we will explicitly state so.

In case the weights are not distinct, we can easily break ties by comparing some unique identifiers of edges. According to our characterization of minimum spanning trees, the unique MST of the new graph will still be a MST of the original graph. Sometimes, we could be interested in finding all solutions, but as this is an uncommon problem, we will postpone it until Section 5.5. For the time being, we will always assume distinct weights.

1.2.11. Observation. If all edge weights are distinct and T is an arbitrary spanning tree, then every edge of G is either T-heavy, or T-light, or contained in T.

1.2.12. *Monotone isomorphism.* Another useful consequence of the Minimality theorem is that whenever two graphs are isomorphic and the isomorphism preserves the relative order of weights, the isomorphism applies to their MST's as well:

1.2.13. Definition. A *monotone isomorphism* between two weighted graphs $G_1 = (V_1, E_1, w_1)$ and $G_2 = (V_2, E_2, w_2)$ is a bijection $\pi : V_1 \to V_2$ such that for each $u, v \in V_1 : uv \in E_1 \Leftrightarrow \pi(u)\pi(v) \in E_2$ and for each $e, f \in E_1 : w_1(e) < w_1(f) \Leftrightarrow w_2(\pi[e]) < w_2(\pi[f])$.

1.2.14. Lemma. *(MST of isomorphic graphs)*
Let G_1 and G_2 be two weighted graphs with distinct edge weights and π a monotone isomorphism between them. Then $\mathrm{mst}(G_2) = \pi[\mathrm{mst}(G_1)]$.

Proof. The isomorphism π maps spanning trees to spanning trees bijectively and it preserves the relation of covering. Since it is monotone, it preserves the property of being a light edge (an edge $e \in E(G_1)$ is T-light $\Leftrightarrow$ the edge $\pi[e] \in E(G_2)$ is $f[T]$-light). Therefore by the Minimality Theorem (1.2.6), T is the MST of G_1 if and only if $\pi[T]$ is the MST of G_2. ♠

1.3. The Red-Blue meta-algorithm

Most MST algorithms can be described as special cases of the following procedure (again following Tarjan [Tar83]):

1.3.1. Algorithm. *(Red-Blue Meta-Algorithm)*

Input: A graph G with an edge comparison oracle (see 1.2.10)

1. At the beginning, all edges are colored black.
2. Apply rules as long as possible:
3. Either pick a cut C such that its lightest edge is not blue and color this edge blue, *(Blue rule)*
4. or pick a cycle C such that its heaviest edge is not red and color this edge red. *(Red rule)*

Output: Minimum spanning tree of G consisting of edges colored blue.

1.3.2. This procedure is not a proper algorithm, since it does not specify how to choose the rule to apply. We will however prove that no matter how the rules are applied, the procedure always stops and it gives the correct result. Also, it will turn out that each of the classical MST algorithms can be described as a specific way of choosing the rules in this procedure, which justifies the name meta-algorithm.

1.3.3. Notation. We will denote the unique minimum spanning tree of the input graph by T_{min}. We intend to prove that this is also the output of the procedure.

1.3.4. *Correctness.* Let us prove that the meta-algorithm is correct. First we show that the edges colored blue in any step of the procedure always belong to T_{min} and that the edges colored red are guaranteed to be outside T_{min}. Then we demonstrate that the procedure always stops. Some parts of the proof will turn out to be useful in the upcoming chapters, so we will state them in a slightly more general way.

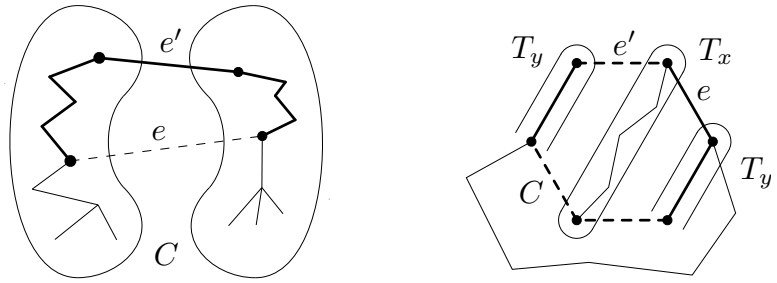
1.3.5. Lemma. *(Blue lemma, also known as the Cut rule)*
The lightest edge of every cut is contained in the MST.

Proof. By contradiction. Let e be the lightest edge of a cut C. If $e \notin T_{min}$, then there must exist an edge $e' \in T_{min}$ that is contained in C (take any pair of vertices separated by C: the path in T_{min} joining these vertices must cross C at least once). Exchanging e for e' in T_{min} yields an even lighter spanning tree since $w(e) < w(e')$. ♠

1.3.6. Lemma. *(Red lemma, also known as the Cycle rule)*
An edge e is not contained in the MST iff it is the heaviest on some cycle.

Proof. The implication from the left to the right follows directly from the Minimality theorem: if $e \notin T_{min}$, then e is T_{min}-heavy and so it is the heaviest edge on the cycle $T_{min}[e] + e$.

We will prove the other implication again by contradiction. Suppose that e is the heaviest edge of a cycle C and that $e \in T_{min}$. Removing e causes T_{min} to split to two components, let us call them T_x and T_y. Some vertices of C now lie in T_x, the others in T_y, so there must exist in edge $e' \neq e$ such that its endpoints lie in different components. Since $w(e') < w(e)$, exchanging e for e' yields a spanning tree lighter than T_{min}. ♠

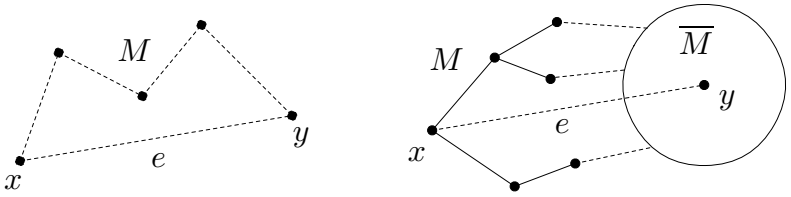


Proof of the Blue (left) and Red (right) lemma

1.3.7. Lemma. *(Black lemma)*
As long as there exists a black edge, at least one rule can be applied.

Proof. Assume that $e = xy$ is a black edge. Let us define M as the set of vertices reachable from x using only blue edges. If y lies in M, then e together with some blue path between x and y forms a cycle and e must be the heaviest edge on this cycle. This holds because all blue edges have been already proven to be in T_{min} and there can be no T_{min}-light edges. In this case, we can apply the Red rule.

On the other hand, if $y \notin M$, then the cut formed by all edges between M and $V \setminus M$ contains no blue edges, therefore we can use the Blue rule. ♠



Configurations in the proof of the Black lemma

1.3.8. Notation. We will use $\delta(M)$ to denote the cut separating M from its complement. That is, $\delta(M) = E \cap (M \times (V \setminus M))$. We will also abbreviate $\delta(\{v\})$ as $\delta(v)$.

1.3.9. Theorem. *(Red-Blue correctness)*
For any selection of rules, the Red-Blue procedure stops and the blue edges form the minimum spanning tree of the input graph.

Proof. To prove that the procedure stops, let us notice that no edge is ever recolored, so we must run out of black edges after at most m steps. Recoloring to the same color is avoided by the conditions built in the rules, recoloring to a different color

would mean that the edge would be both inside and outside T_{min} due to our Red and Blue lemmata.

When no further rules can be applied, the Black lemma guarantees that all edges are colored, so by the Blue lemma all blue edges are in T_{min} and by the Red lemma all other (red) edges are outside T_{min}. Thus the blue edges are exactly T_{min}. ♠

1.3.10. Remark. The MST problem is a special case of the problem of finding the minimum basis of a weighted matroid. Surprisingly, when we modify the Red-Blue procedure to use the standard definitions of cycles and cuts in matroids, it will always find the minimum basis. Some of the other MST algorithms also easily generalize to matroids and in some sense matroids are exactly the objects where "the greedy approach works". We will however not pursue this direction in our work, referring the reader to the Oxley's monograph [Oxl92] instead.

1.4. Classical algorithms

The three classical MST algorithms (Borůvka's, Jarník's, and Kruskal's) can be easily stated in terms of the Red-Blue meta-algorithm. For each of them, we first show the general version of the algorithm, then we prove that it gives the correct result and finally we discuss the time complexity of various implementations.

1.4.1. *Borůvka's algorithm.* The oldest MST algorithm is based on a simple idea: grow a forest in a sequence of iterations until it becomes connected. We start with a forest of isolated vertices. In each iteration, we let each tree of the forest select the lightest edge of those having exactly one endpoint in the tree (we will call such edges the *neighboring edges* of the tree). We add all such edges to the forest and proceed with the next iteration.

1.4.2. Algorithm. *(Borůvka [Bor26a], Choquet [Cho38], Sollin [Sol65], and others)*

Input: A graph G with an edge comparison oracle.

1. $T \leftarrow$ a forest consisting of vertices of G and no edges.
2. While T is not connected:
3. For each component T_i of T, choose the lightest edge e_i from the cut separating T_i from the rest of T.
4. Add all e_i's to T.

Output: Minimum spanning tree T.

1.4.3. Lemma. In each iteration of the algorithm, the number of trees in T decreases by at least a factor of two.

Proof. Each tree gets merged with at least one of its neighbors, so each of the new trees contains two or more original trees. ♠

1.4.4. Corollary. The algorithm stops in $\mathcal{O}(\log n)$ iterations.

1.4.5. Lemma. The Borůvka's algorithm outputs the MST of the input graph.

Proof. In every iteration of the algorithm, T is a blue subgraph, because every addition of some edge e_i to T is a straightforward application of the Blue rule. We stop when the blue subgraph is connected, so we do not need the Red rule to explicitly exclude edges.

It remains to show that adding the edges simultaneously does not produce a cycle. Consider the first iteration of the algorithm where T contains a cycle C. Without loss of generality we can assume that:

$$C = T_1[u_1, v_1]\, v_1u_2\, T_2[u_2, v_2]\, v_2u_3\, T_3[u_3, v_3]\, \ldots\, T_k[u_k, v_k]\, v_ku_1.$$

Each component T_i has chosen its lightest incident edge e_i as either the edge v_iu_{i+1} or $v_{i-1}u_i$ (indexing cyclically). Suppose that $e_1 = v_1u_2$ (otherwise we reverse the orientation of the cycle). Then $e_2 = v_2u_3$ and $w(e_2) < w(e_1)$ and we can continue in the same way, getting $w(e_1) > w(e_2) > \ldots > w(e_k) > w(e_1)$, which is a contradiction. (Note that distinctness of edge weights was crucial here.) ♠

1.4.6. Lemma. Each iteration can be carried out in time $\mathcal{O}(m)$.

Proof. We assign a label to each tree and we keep a mapping from vertices to the labels of the trees they belong to. We scan all edges, map their endpoints to the particular trees and for each tree we maintain the lightest incident edge so far encountered. Instead of merging the trees one by one (which would be too slow), we build an auxiliary graph whose vertices are the labels of the original trees and edges correspond to the chosen lightest inter-tree edges. We find the connected components of this graph, and these determine how are the original labels translated to the new labels. ♠

1.4.7. Theorem. The Borůvka's algorithm finds the MST in time $\mathcal{O}(m \log n)$.

Proof. Follows from the previous lemmata. ♠

1.4.8. *Jarník's algorithm.* The next algorithm, discovered independently by Jarník, Prim and Dijkstra, is similar to the Borůvka's algorithm, but instead of the whole forest it concentrates on a single tree. It starts with a single vertex and it repeatedly extends the tree by the lightest neighboring edge until the tree spans the whole graph.

1.4.9. Algorithm. *(Jarník [Jar30], Prim [Pri57], Dijkstra [Dij59])*

Input: A graph G with an edge comparison oracle.

1. $T \leftarrow$ a single-vertex tree containing an arbitrary vertex of G.
2. While there are vertices outside T:
3. Pick the lightest edge uv such that $u \in V(T)$ and $v \notin V(T)$.
4. $T \leftarrow T + uv$.

Output: Minimum spanning tree T.

1.4.10. Lemma. The Jarník's algorithm computes the MST of the input graph.

Proof. If G is connected, the algorithm always stops. In every step of the algorithm, T is always a blue tree. because Step 4 corresponds to applying the Blue rule to the cut $\delta(T)$ separating T from the rest of the given graph. We need not care about the remaining edges, since for a connected graph the algorithm always stops with the right number of blue edges. ♠

1.4.11. Implementation. The most important part of the algorithm is finding the *neighboring edges*. In a straightforward implementation, searching for the lightest neighboring edge takes $\Theta(m)$ time, so the whole algorithm runs in time $\Theta(mn)$.

We can do much better by using a binary heap to hold all neighboring edges. In each iteration, we find and delete the minimum edge from the heap and once we

expand the tree, we insert the newly discovered neighboring edges to the heap and delete the neighboring edges that became internal to the new tree. Since there are always at most m edges in the heap, each heap operation takes $\mathcal{O}(\log m) = \mathcal{O}(\log n)$ time. For every edge, we perform at most one insertion and at most one deletion, so we spend $\mathcal{O}(m \log n)$ time in total. From this, we can conclude:

1.4.12. Theorem. The Jarník's algorithm computes the MST of a given graph in time $\mathcal{O}(m \log n)$.

1.4.13. Remark. We will show several faster implementations in Section 3.2.

1.4.14. *Kruskal's algorithm.* The last of the three classical algorithms processes the edges of the graph G greedily. It starts with an empty forest and it takes the edges of G in order of their increasing weights. For every edge, it checks whether its addition to the forest produces a cycle and if it does not, the edge is added. Otherwise, the edge is dropped and not considered again.

1.4.15. Algorithm. *(Kruskal [Kru56])*

Input: A graph G with an edge comparison oracle.

1. Sort edges of G by their increasing weights.
2. $T \leftarrow$ an empty spanning subgraph.
3. For all edges e in their sorted order:
4. If $T + e$ is acyclic, add e to T.
5. Otherwise drop e.

Output: Minimum spanning tree T.

1.4.16. Lemma. The Kruskal's algorithm returns the MST of the input graph.

Proof. In every step, T is a forest of blue trees. Adding e to T in step 4 applies the Blue rule on the cut separating some pair of components of T (e is the lightest, because all other edges of the cut have not been considered yet). Dropping e in step 5 corresponds to the Red rule on the cycle found (e must be the heaviest, since all other edges of the cycle have been already processed). At the end of the algorithm, all edges are colored, so T must be the MST. ♠

1.4.17. Implementation. Except for the initial sorting, which in general requires $\Theta(m \log m)$ time, the only other non-trivial operation is the detection of cycles. What we need is a data structure for maintaining connected components, which supports queries and edge insertion. This is closely related to the well-known Disjoint Set Union problem:

1.4.18. Problem. *(Disjoint Set Union, DSU)*
Maintain an equivalence relation on a finite set under a sequence of operations *Union* and *Find*. The *Find* operation tests whether two elements are equivalent and *Union* joins two different equivalence classes into one.

1.4.19. We can maintain the connected components of our forest T as equivalence classes. When we want to add an edge uv, we first call $\mathit{Find}(u, v)$ to check if both endpoints of the edge lie in the same component. If they do not, addition of this edge connects both components into one, so we perform $\mathit{Union}(u, v)$ to merge the equivalence classes.

Tarjan has shown that there is a data structure for the DSU problem of surprising efficiency:

1.4.20. Theorem. *(Disjoint Set Union, Tarjan [Tar75])*
Starting with a trivial equivalence with single-element classes, a sequence of operations comprising of n *Union*s intermixed with $m \geq n$ *Find*s can be processed in time $\mathcal{O}(m\,\alpha(m,n))$, where $\alpha(m,n)$ is a certain inverse of the Ackermann's function (see Definition A.3.4).

Proof. See [Tar75]. ♠

This completes the following theorem:

1.4.21. Theorem. The Kruskal's algorithm finds the MST of a given graph in time $\mathcal{O}(m \log n)$. If the edges are already sorted by their weights, the time drops to $\mathcal{O}(m\,\alpha(m,n))$.

Proof. We spend $\mathcal{O}(m \log n)$ time on sorting, $\mathcal{O}(m\,\alpha(m,n))$ on processing the sequence of *Union*s and *Find*s, and $\mathcal{O}(m)$ on all other work. ♠

1.4.22. Remark. The cost of the *Union* and *Find* operations is of course dwarfed by the complexity of sorting, so a much simpler (at least in terms of its analysis) data structure would be sufficient, as long as it has $\mathcal{O}(\log n)$ amortized complexity per operation. For example, we can label vertices with identifiers of the corresponding components and always relabel the smaller of the two components.

We will study dynamic maintenance of connected components in more detail in Chapter 5.

1.5. Contractive algorithms

While the classical algorithms are based on growing suitable trees, they can be also reformulated in terms of edge contraction. Instead of keeping a forest of trees, we can keep each tree contracted to a single vertex. This replaces the relatively complex tree-edge incidencies by simple vertex-edge incidencies, potentially speeding up the calculation at the expense of having to perform the contractions.

We will show a contractive version of the Borůvka's algorithm in which these costs are carefully balanced, leading for example to a linear-time algorithm for MST in planar graphs.

There are two definitions of edge contraction that differ when an edge of a triangle is contracted. Either we unify the other two edges to a single edge or we keep them as two parallel edges, leaving us with a multigraph. We will use the multigraph version and we will show that we can easily reduce the multigraph to a simple graph later. (See A.2.3 for the exact definitions.)

We only need to be able to map edges of the contracted graph to the original edges, so we let each edge carry a unique label $\ell(e)$ that will be preserved by contractions.

1.5.1. Lemma. *(Flattening a multigraph)*
Let G be a multigraph and G' its subgraph obtaining by removing loops and replacing each bundle of parallel edges by its lightest edge. Then G' has the same MST as G.

Proof. Every spanning tree of G' is a spanning tree of G. In the other direction: Loops can be never contained in a spanning tree. If there is a spanning tree T containing a removed edge e parallel to an edge $e' \in G'$, exchanging e' for e makes T

lighter. (This is indeed the multigraph version of the Red lemma applied to a two-edge cycle, as we will see in 1.6.2.) ♠

1.5.2. Algorithm. *(Contractive version of Borůvka's algorithm)*

Input: A graph G with an edge comparison oracle.

1. $T \leftarrow \emptyset$.
2. $\ell(e) \leftarrow e$ for all edges e. *(Initialize the labels.)*
3. While $n(G) > 1$:
4. For each vertex v_k of G, let e_k be the lightest edge incident to v_k.
5. $T \leftarrow T \cup \{\ell(e_1), \ldots, \ell(e_n)\}$.
   (Remember labels of all selected edges.)
6. Contract all edges e_k, inheriting labels and weights.[2]
7. Flatten G (remove parallel edges and loops).

Output: Minimum spanning tree T.

1.5.3. Notation. For the analysis of the algorithm, we will denote the graph considered by the algorithm at the beginning of the i-th iteration by G_i (starting with $G_0 = G$) and the number of vertices and edges of this graph by n_i and m_i respectively. A single iteration of the algorithm will be called a *Borůvka step*.

1.5.4. Lemma. The i-th Borůvka step can be carried out in time $\mathcal{O}(m_i)$.

Proof. The only non-trivial parts are steps 6 and 7. Contractions can be handled similarly to the unions in the original Borůvka's algorithm (see 1.4.6): We build an auxiliary graph containing only the selected edges e_k, find connected components of this graph and renumber vertices in each component to the identifier of the component. This takes $\mathcal{O}(m_i)$ time.

Flattening is performed by first removing the loops and then bucket-sorting the edges (as ordered pairs of vertex identifiers) lexicographically, which brings parallel edges together. The bucket sort uses two passes with n_i buckets, so it takes $\mathcal{O}(n_i + m_i) = \mathcal{O}(m_i)$. ♠

1.5.5. Theorem. The Contractive Borůvka's algorithm finds the MST of the input graph in time $\mathcal{O}(\min(n^2, m \log n))$.

Proof. As in the original Borůvka's algorithm, the number of iterations is $\mathcal{O}(\log n)$. When combined with the previous lemma, it gives an $\mathcal{O}(m \log n)$ upper bound.

To get the $\mathcal{O}(n^2)$ bound, we observe that the number of trees in the non-contracting version of the algorithm drops at least by a factor of two in each iteration (Lemma 1.4.3) and the same must hold for the number of vertices in the contracting version. Therefore $n_i \leq n/2^i$. While the number of edges need not decrease geometrically, we still have $m_i \leq n_i^2$ as the graphs G_i are simple (we explicitly removed multiple edges and loops at the end of the previous iteration). Hence the total time spent in all iterations is $\mathcal{O}(\sum_i n_i^2) = \mathcal{O}(\sum_i n^2/4^i) = \mathcal{O}(n^2)$. ♠

On planar graphs, the algorithm runs much faster:

1.5.6. Theorem. *(Contractive Borůvka on planar graphs)*
When the input graph is planar, the Contractive Borůvka's algorithm runs in time $\mathcal{O}(n)$.

[2] In other words, we will ask the comparison oracle for the edge $\ell(e)$ instead of e.

Proof. Let us refine the previous proof. We already know that $n_i \leq n/2^i$. We will prove that when G is planar, the m_i's are decreasing geometrically. We know that every G_i is planar, because the class of planar graphs is closed under edge deletion and contraction. Moreover, G_i is also simple, so we can use the standard bound on the number of edges of planar simple graphs (see for example [Die05]) to get $m_i \leq 3n_i \leq 3n/2^i$. The total time complexity of the algorithm is therefore $\mathcal{O}(\sum_i m_i) = \mathcal{O}(\sum_i n/2^i) = \mathcal{O}(n)$. ♠

1.5.7. Remark. There are several other possibilities how to find the MST of a planar graph in linear time. For example, Matsui [Mat95] has described an algorithm based on simultaneously working on the graph and its topological dual. The advantage of our approach is that we do not need to construct the planar embedding explicitly. We will show another simpler linear-time algorithm in section 3.1.

1.5.8. Remark. To achieve the linear time complexity, the algorithm needs a very careful implementation, but we defer the technical details to section 2.2.

1.5.9. *General contractions.* Graph contractions are indeed a very powerful tool and they can be used in other MST algorithms as well. The following lemma shows the gist:

1.5.10. Lemma. *(Contraction of MST edges)*
Let G be a weighted graph, e an arbitrary edge of $\mathrm{mst}(G)$, G/e the multigraph produced by contracting e in G, and π the bijection between edges of $G - e$ and their counterparts in G/e. Then $\mathrm{mst}(G) = \pi^{-1}[\mathrm{mst}(G/e)] + e$.

Proof. The right-hand side of the equality is a spanning tree of G. Let us denote it by T and the MST of G/e by T'. If T were not minimum, there would exist a T-light edge f in G (by the Minimality Theorem, 1.2.6). If the path $T[f]$ covered by f does not contain e, then $\pi[T[f]]$ is a path covered by $\pi(f)$ in T'. Otherwise $\pi(T[f] - e)$ is such a path. In both cases, f is T'-light, which contradicts the minimality of T'. (We do not have a multigraph version of the theorem, but the direction we need is a straightforward edge exchange, which obviously works in multigraphs as well as in simple graphs.) ♠

1.5.11. Remark. In the Contractive Borůvka's algorithm, the role of the mapping π^{-1} is of course played by the edge labels ℓ.

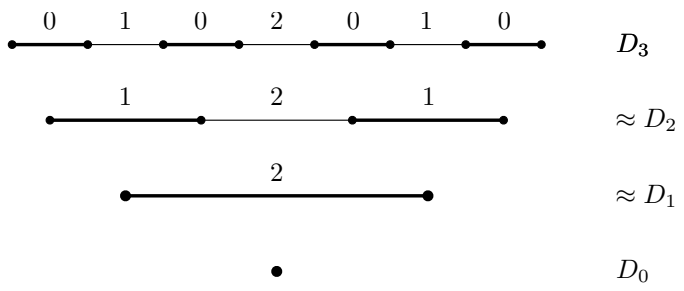
1.5.12. *A lower bound.* Finally, we will show a family of graphs for which the $\mathcal{O}(m \log n)$ bound on time complexity is tight. The graphs do not have unique weights, but they are constructed in a way that the algorithm never compares two edges with the same weight. Therefore, when two such graphs are monotonically isomorphic (see 1.2.14), the algorithm processes them in the same way.

1.5.13. Definition. A *distractor of order k,* denoted by D_k, is a path on $n = 2^k$ vertices $v_1, \ldots, v_n$, where each edge $v_i v_{i+1}$ has its weight equal to the number of trailing zeroes in the binary representation of the number i. The vertex v_1 is called a *base* of the distractor.

1.5.14. Remark. Alternatively, we can use a recursive definition: D_0 is a single vertex, D_{k+1} consists of two disjoint copies of D_k joined by an edge of weight k.

1.5.15. Lemma. A single iteration of the contractive algorithm reduces the distractor D_k to a graph isomorphic with D_{k-1}.

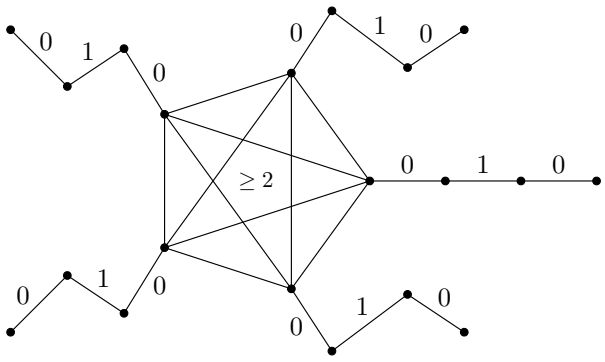
Proof. Each vertex v of D_k is incident with a single edge of weight 1. The algorithm



A distractor D_3 and its evolution (bold edges are contracted)

therefore selects all weight 1 edges and contracts them. This produces a graph that is equal to D_{k-1} with all weights increased by 1, which does not change the relative order of edges. ♠

1.5.16. Definition. A *hedgehog* $H_{a,k}$ is a graph consisting of a distractors $D_k^1, \ldots, D_k^a$ of order k together with edges of a complete graph on the bases of these distractors. The additional edges have arbitrary weights that are heavier than the edges of all the distractors.



A hedgehog $H_{5,2}$ (quills bent to fit in the picture)

1.5.17. Lemma. A single iteration of the contractive algorithm reduces $H_{a,k}$ to a graph isomorphic with $H_{a,k-1}$.

Proof. Each vertex is incident with an edge of some distractor, so the algorithm does not select any edge of the complete graph. Contraction therefore reduces each distractor to a smaller distractor (modulo an additive factor in weight) and it leaves the complete graph intact. The resulting graph is monotonely isomorphic to $H_{a,k-1}$. ♠

When we set the parameters appropriately, we get the following lower bound:

1.5.18. Theorem. *(Lower bound for Contractive Borůvka)*
For each n there exists a graph on $\Theta(n)$ vertices and $\Theta(n)$ edges such that the Contractive Borůvka's algorithm spends time $\Omega(n \log n)$ on it.

Proof. Consider the hedgehog $H_{a,k}$ for $a = \lceil \sqrt{n} \rceil$ and $k = \lceil \log_2 a \rceil$. It has $a \cdot 2^k = \Theta(n)$ vertices and $\binom{a}{2} + a \cdot 2^k = \Theta(a^2) + \Theta(a^2) = \Theta(n)$ edges as we wanted.

By the previous lemma, the algorithm proceeds through a sequence of hedgehogs $H_{a,k}, H_{a,k-1}, \ldots, H_{a,0}$ (up to monotone isomorphism), so it needs a logarithmic

number of iterations plus some more to finish on the remaining complete graph. Each iteration runs on a graph with $\Omega(n)$ edges as every $H_{a,k}$ contains a complete graph on a vertices. ♠

1.6. Lifting restrictions

In order to have a simple and neat theory, we have introduced several restrictions on the graphs in which we search for the MST. As in some rare cases we are going to meet graphs that do not fit into this simplified world, let us quickly examine what happens when the restrictions are lifted.

1.6.1. *Disconnected graphs.* The basic properties of minimum spanning trees and the algorithms presented in this chapter apply to minimum spanning forests of disconnected graphs, too. The proofs of our theorems and the steps of our algorithms are based on adjacency of vertices and existence of paths, so they are always local to a single connected component. The Borůvka's and Kruskal's algorithm need no changes, the Jarník's algorithm has to be invoked separately for each component.

We can also extend the notion of light and heavy edges with respect to a tree to forests: When an edge e connects two vertices lying in the same tree T of a forest F, it is F-heavy iff it is T-heavy (similarly for F-light). Edges connecting two different trees are always considered F-light. Again, a spanning forest F is minimum iff there are no F-light edges.

1.6.2. *Multigraphs.* All theorems and algorithms from this chapter work for multigraphs as well, only the notation sometimes gets crabbed, which we preferred to avoid. The Minimality theorem and the Blue rule stay unchanged. The Red rule is naturally extended to self-loops (which are never in the MST) and two-edge cycles (where the heavier edge can be dropped) as already suggested in the Flattening lemma (1.5.1).

1.6.3. *Multiple edges of the same weight.* In case when the edge weights are not distinct, the characterization of minimum spanning trees using light edges is still correct, but the MST is no longer unique (as already mentioned, there can be as much as n^{n-2} MST's).

In the Red-Blue procedure, we have to avoid being too zealous. The Blue lemma cannot guarantee that when a cut contains multiple edges of the minimum weight, all of them are in the MST. It will however tell that if we pick one of these edges, an arbitrary MST can be modified to another MST that contains this edge. Therefore the Blue rule will change to "Pick a cut C such that it does not contain any blue edge and color one of its lightest edges blue." The Red lemma and the Red rule can be handled in a similar manner. The modified algorithm will be then guaranteed to find one of the possible MST's.

The Kruskal's and Jarník's algorithms keep working. This is however not the case of the Borůvka's algorithm, whose proof of correctness in Lemma 1.4.5 explicitly referred to distinct weights and indeed, if they are not distinct, the algorithm will occasionally produce cycles. To avoid the cycles, the ties in edge weight comparisons have to be broken in a systematic way. The same applies to the contractive version of this algorithm.

2. Fine Details of Computation

2.1. Models and machines

Traditionally, computer scientists have been using a variety of computational models as a formalism in which their algorithms are stated. If we were studying NP-completeness, we could safely assume that all these models are equivalent, possibly up to polynomial slowdown which is negligible. In our case, the differences between good and not-so-good algorithms are on a much smaller scale. In this chapter, we will replace the usual "tape measure" by a micrometer, state our computation models carefully and develop a repertoire of basic data structures tailor-made for the fine details of the models.

We would like to keep the formalism close enough to the reality of the contemporary computers. This rules out Turing machines and similar sequentially addressed models, but even the remaining models are subtly different from each other. For example, some of them allow indexing of arrays in constant time, while on the others, arrays have to be emulated with pointer structures, requiring $\Omega(\log n)$ time to access a single element of an n-element array. It is hard to say which way is superior — while most "real" computers have instructions for constant-time indexing, it seems to be physically impossible to fulfil this promise regardless of the size of addressable memory. Indeed, at the level of logical gates inside the computer, the depth of the actual indexing circuits is logarithmic.

In recent decades, most researchers in the area of combinatorial algorithms have been considering two computational models: the Random Access Machine and the Pointer Machine. The former is closer to the programmer's view of a real computer, the latter is slightly more restricted and "asymptotically safe." We will follow this practice and study our algorithms in both models.

2.1.1. The *Random Access Machine (RAM)* is not a single coherent model, but rather a family of closely related machines which share the following properties. (See Cook and Reckhow [CR72] for one of the usual formal definitions and Hagerup [Hag98] for a thorough description of the differences between the RAM variants.)

The *memory* of the machine is represented by an array of *memory cells* addressed by non-negative integers. Each cell contains a single non-negative integer. The *program* is a finite sequence of *instructions* of two basic kinds: calculation instructions and control instructions.

Calculation instructions have two source arguments and one destination argument, each *argument* being either an immediate constant (not available as destination), a directly addressed memory cell (specified by its number) or an indirectly addressed memory cell (its address is stored in a directly addressed memory cell).

Control instructions include branches (to a specific instruction in the program), conditional branches (e.g., jump if two arguments specified as in the calculation instructions are equal) and an instruction to halt the program.

At the beginning of the computation, the memory contains the input data in specified cells and arbitrary values in all other cells. Then the program is executed

one instruction at a time. When it halts, specified memory cells are interpreted as the program's output.

2.1.2. In the description of the RAM family, we have omitted several details on purpose, because different members of the family define them differently. These are: the size of the available integers, the time complexity of a single instruction, the space complexity assigned to a single memory cell and the set of operations available in calculation instructions.

If we impose no limits on the magnitude of the numbers and we assume that arithmetic and logical operations work on them in constant time, we get a very powerful parallel computer — we can emulate an exponential number of parallel processors using arithmetics and suddenly almost everything can be computed in constant time, modulo encoding and decoding of input and output. Such models are unrealistic and there are two basic possibilities how to avoid this behavior:

1. Keep unbounded numbers, but increase costs of instructions: each instruction consumes time proportional to the number of bits of the numbers it processes, including memory addresses. Similarly, space usage is measured in bits, counting not only the values, but also the addresses of the respective memory cells.
2. Place a limit on the size of the numbers —define the *word size* W, the number of bits available in each memory cell— and keep the cost of instructions and memory cells constant. The word size must not be constant, since we can address only 2^W cells of memory. If the input of the algorithm is stored in N cells, we need $W \geq \log N$ just to be able to read the input. On the other hand, we are interested in polynomial-time algorithms only, so $\Theta(\log N)$-bit numbers should be sufficient. In practice, we pick W to be the larger of $\Theta(\log N)$ and the size of integers used in the algorithm's input and output. We will call an integer that fits in a single memory cell a *machine word.*

Both restrictions easily avoid the problems of unbounded parallelism. The first choice is theoretically cleaner and Cook et al. show nice correspondences to the standard complexity classes, but the calculations of time and space complexity tend to be somewhat tedious. What more, when compared with the RAM with restricted word size, the complexities are usually exactly $\Theta(W)$ times higher. This does not hold in general (consider a program that uses many small numbers and $\mathcal{O}(1)$ large ones), but it is true for the algorithms we are interested in. Therefore we will always assume that the operations have unit cost and we make sure that all numbers are limited by the available word size.

2.1.3. As for the choice of RAM operations, the following three instruction sets are often used:

- *Word-RAM* — allows the "C-language operators", i.e., addition, subtraction, multiplication, division, remainder, bitwise AND, OR, exclusive OR (XOR) and negation (NOT), and bitwise shifts ($<<$ and $>>$).
- AC^0*-RAM* — allows all operations from the class AC^0, i.e., those computable by constant-depth polynomial-size boolean circuits with unlimited fan-in and fan-out. This includes all operations of the Word-RAM except

for multiplication, division and remainders, and also many other operations like computing the Hamming weight (number of bits set in a given number).

- Both restrictions combined.

Thorup [Tho03] discusses the usual techniques employed by RAM algorithms and he shows that they work on both Word-RAM and AC^0-RAM, but the combination of the two restrictions is too weak. On the other hand, the intersection of AC^0 with the instruction set of modern processors is already strong enough (e.g., when we add some floating-point operations and multimedia instructions available on the Intel's Pentium 4 [Int07]).

We will therefore use the Word-RAM instruction set, mentioning differences from the AC^0-RAM where necessary.

2.1.4. Notation. When speaking of the *RAM,* we implicitly mean the version with numbers limited by a specified word size of W bits, unit cost of operations and memory cells and the instruction set of the Word-RAM. This corresponds to the usage in recent algorithmic literature, although the authors rarely mention the details.

In some cases, a non-uniform variant of the Word-RAM is considered as well (e.g., by Hagerup [HMP01]):

2.1.5. Definition. A Word-RAM is called *weakly non-uniform,* if it is equipped with $\mathcal{O}(1)$-time access to a constant number of word-sized constants, which depend only on the word size. These are called *native constants* and they are available in fixed memory cells when the program starts. (By analogy with the high-level programming languages, these constants can be thought of as computed at "compile time".)

2.1.6. The *Pointer Machine (PM)* also does not seem to have any well established definition. The various kinds of pointer machines are examined by Ben-Amram in [BA95], but unlike the RAM's they turn out to be equivalent up to constant slowdown. Our definition will be closely related to the *linking automaton* proposed by Knuth in [Knu97a], we will only adapt it to use RAM-like instructions instead of an opaque control unit.

The PM works with two different types of data: *symbols* from a finite alphabet and *pointers*. The memory of the machine consists of a fixed amount of *registers* (some of them capable of storing a single symbol, each of the others holds a single pointer) and an arbitrary amount of *cells*. The structure of all cells is the same: each cell again contains a fixed number of fields for symbols and pointers. Registers can be addressed directly, the cells only via pointers — by using a pointer stored either in a register, or in a cell pointed to by a register. Longer chains of pointers cannot be followed in constant time.

We can therefore view the whole memory as a directed graph, whose vertices correspond to the cells (the registers are stored in a single special cell). The outgoing edges of each vertex correspond to pointer fields of the cells and they are labeled with distinct labels drawn from a finite set. In addition to that, each vertex contains a fixed amount of symbols. The machine can directly access vertices within distance 2 from the register vertex.

The program is a finite sequence of instructions of the following kinds:

- *symbol instructions,* which read a pair of symbols, apply an arbitrary function to them and write the result to a symbol register or field;
- *pointer instructions* for assignment of pointers to pointer registers/fields and for creation of new memory cells (a pointer to the new cell is stored into a register immediately);
- *control instructions* — similarly to the RAM; conditional jumps can decide arbitrary unary relations on symbols and compare pointers for equality.

Time and space complexity are defined in the straightforward way: all instructions have unit cost and so do all memory cells.

Both input and output of the machine are passed in the form of a linked structure pointed to by a designated register. For example, we can pass graphs back and forth without having to encode them as strings of numbers or symbols. This is important, because with the finite alphabet of the PM, symbolic representations of graphs generally require super-linear space and therefore also time.[1]

2.1.7. Compared to the RAM, the PM lacks two important capabilities: indexing of arrays and arithmetic instructions. We can emulate both with poly-logarithmic slowdown, but it will turn out that they are rarely needed in graph algorithms. We are also going to prove that the RAM is strictly stronger, so we will prefer to formulate our algorithms for the PM and use the RAM only when necessary.

2.1.8. Theorem. Every program for the Word-RAM with word size W can be translated to a PM program computing the same with $\mathcal{O}(W^2)$ slowdown (given a suitable encoding of inputs and outputs, of course). If the RAM program does not use multiplication, division and remainder operations, $\mathcal{O}(W)$ slowdown is sufficient.

Proof sketch. Represent the memory of the RAM by a balanced binary search tree or by a radix trie of depth $\mathcal{O}(W)$. Values are encoded as linked lists of symbols pointed to by the nodes of the tree. Both direct and indirect accesses to the memory can therefore be done in $\mathcal{O}(W)$ time. Use standard algorithms for arithmetic on big numbers: $\mathcal{O}(W)$ per operation except for multiplication, division and remainders which take $\mathcal{O}(W^2)$.[2] ♠

2.1.9. Theorem. Every program for the PM running in polynomial time can be translated to a program computing the same on the Word-RAM with only $\mathcal{O}(1)$ slowdown.

Proof sketch. Encode each cell of the PM's memory to $\mathcal{O}(1)$ integers. Store the encoded cells to the memory of the RAM sequentially and use memory addresses as pointers. As the symbols are finite and there is only a polynomial number of cells allocated during execution of the program, $\mathcal{O}(\log N)$-bit integers suffice (N is the size of the program's input). ♠

[1] The usual representation of edges as pairs of vertex labels uses $\Theta(m \log n)$ bits and as a simple counting argument shows, this is asymptotically optimal for general sparse graphs. On the other hand, specific families of sparse graphs can be stored more efficiently, e.g., by a remarkable result of Turán [Tur84], planar graphs can be encoded in $\mathcal{O}(n)$ bits. Encoding of dense graphs is of course trivial as the adjacency matrix has only $\Theta(n^2)$ bits.

[2] We could use more efficient arithmetic algorithms, but the quadratic bound is good enough for our purposes.

2.1.10. There are also *randomized* versions of both machines. These are equipped with an additional instruction for generating a single random bit. The standard methods of design and analysis of randomized algorithms can be used (see for example Motwani and Raghavan [MR95]).

2.1.11. Remark. There is one more interesting machine: the *Immutable Pointer Machine* (mentioned for example in the description of LISP machines in [BA95]). It differs from the ordinary PM by the inability to modify existing memory cells. Only the contents of the registers are allowed to change. All cell modifications thus have to be performed by creating a copy of the particular cell with some fields changed. This in turn requires the pointers to the cell to be updated, possibly triggering a cascade of further cell copies. For example, when a node of a binary search tree is updated, all nodes on the path from that node to the root have to be copied.

One of the advantages of this model is that the states of the machine are persistent — it is possible to return to a previously visited state by recalling the $\mathcal{O}(1)$ values of the registers (everything else could not have changed since that time) and "fork" the computations. This corresponds to the semantics of pure functional languages, e.g., of Haskell [JS03].

Unless we are willing to accept a logarithmic penalty in execution time and space (in fact, our emulation of the Word-RAM on the PM can be easily made immutable), the design of efficient algorithms for the immutable PM requires very different techniques. Therefore, we will be interested in the imperative models only and refer the interested reader to the thorough treatment of purely functional data structures in the Okasaki's monograph [Oka99].

2.2. Bucket sorting and unification

The Contractive Borůvka's algorithm (1.5.2) needs to contract a given set of edges in the current graph and then flatten the graph, all this in time $\mathcal{O}(m)$. We have spared the technical details for this section, in which we are going to explain several rather general techniques based on bucket sorting.

As we have already suggested in the proof of Lemma 1.5.2, contractions can be performed in linear time by building an auxiliary graph and finding its connected components. We will thus take care only of the subsequent flattening.

2.2.1. *Flattening on RAM.* On the RAM, we can view the edges as ordered pairs of vertex identifiers with the smaller of the identifiers placed first. We sort these pairs lexicographically. This brings parallel edges together, so that a simple linear scan suffices to find each bunch of parallel edges and to remove all but the lightest one. Lexicographic sorting of pairs can be accomplished in linear time by a two-pass bucket sort with n buckets corresponding to the vertex identifiers.

However, there is a catch. Suppose that we use the standard representation of graphs by adjacency lists whose heads are stored in an array indexed by vertex identifiers. When we contract and flatten the graph, the number of vertices decreases, but if we inherit the original vertex identifiers, the arrays will still have the same size. We could then waste a super-linear amount of time by scanning the increasingly sparse arrays, most of the time skipping unused entries.

To avoid this problem, we have to renumber the vertices after each contraction to component identifiers from the auxiliary graph and create a new vertex array. This helps keep the size of the representation of the graph linear with respect to its current size.

2.2.2. *Flattening on PM.* The pointer representation of graphs does not suffer from sparsity since the vertices are always identified by pointers to per-vertex structures. Each such structure then contains all attributes associated with the vertex, including the head of its adjacency list. However, we have to find a way how to perform bucket sorting without indexing of arrays.

We will keep a list of the per-vertex structures and we will use it to establish the order of vertices. Each such structure will be endowed with a pointer to the head of the list of items in the corresponding bucket. Inserting an edge to a bucket can be then done in constant time and scanning the contents of all n buckets takes $\mathcal{O}(n+m)$ time.

At last, we must not forget that while it was easy to *normalize* the pairs on the RAM by putting the smaller identifier first, this fails on the PM because we can directly compare the identifiers only for equality. We can work around this again by bucket-sorting: we sort the multiset $\{(x,i) \mid x \text{ occurs in the } i\text{-th pair}\}$ on x. Then we reset all pairs and re-insert the values back in their increasing order. This also takes $\mathcal{O}(n+m)$.

2.2.3. *Tree isomorphism.* Another nice example of pointer-based radix sorting is a Pointer Machine algorithm for deciding whether two rooted trees are isomorphic. Let us assume for a moment that the outdegree of each vertex is at most a fixed constant k. We begin by sorting the subtrees of both trees by their depth. This can be accomplished by running depth-first search to calculate the depths and bucket-sorting them with n buckets afterwards.

Then we proceed from depth 0 to the maximum depth and for each depth we identify the isomorphism equivalence classes of the particular subtrees. We will assign unique *codes* (identifiers) to all such classes; at most $n+1$ of them are needed as there are $n+1$ subtrees in the tree (including the empty subtree). As the PM does not have numbers as an elementary type, we create a "*yardstick*" —a list of $n+1$ distinct items— and we use pointers to these "ticks" as identifiers. When we are done, isomorphism of the whole trees can be decided by comparing the codes assigned to their roots.

Suppose that classes of depths $0, \ldots, d-1$ are already computed and we want to identify those of depth d. We will denote their count of n_d. We take a root of every such tree and label it with an ordered k-tuple of codes of its subtrees; when it has less than k sons, we pad the tuple with empty subtrees. Tuples corresponding to isomorphic subtrees are identical up to reordering of elements. We therefore sort the codes inside each tuple and then sort the tuples, which brings the equivalent tuples together.

The first sort (inside the tuples) would be easy on the RAM, but on the PM we have to use the normalization trick mentioned above. The second sort is a straightforward k-pass bucket sort.

If we are not careful, a single sorting pass takes $\mathcal{O}(n_d+n)$ time, because while we have only n_d items to sort, we have to scan all n buckets. This can be easily

avoided if we realize that the order of the buckets does not need to be fixed — in every pass, we can use a completely different order and it still does bring the equivalent tuples together. Thus we can keep a list of buckets that are used in the current pass and look only inside these buckets. This way, we reduce the time spent in a single pass to $\mathcal{O}(n_d)$ and the whole algorithm takes just $\mathcal{O}(\sum_d n_d) = \mathcal{O}(n)$.

Our algorithm can be easily modified for trees with unrestricted degrees. We replace the fixed d-tuples by general sequences of codes. The first sort does not need any changes. In the second sort, we proceed from the first position to the last one and after each bucket-sorting pass we put aside the sequences that have just ended. They are obviously not equivalent to any other sequences. The time complexity of the second sort is linear in the sum of the lengths of the sequences, which is n_{d+1} for depth d. We can therefore decide isomorphism of the whole trees in time $\mathcal{O}(\sum_d (n_d + n_{d+1})) = \mathcal{O}(n)$.

The unification of sequences by bucket sorting will be useful in many other situations, so we will state it as a separate lemma:

2.2.4. Lemma. *(Sequence unification)*
Partitioning of a collection of sequences $S_1, \ldots, S_n$, whose elements are arbitrary pointers and symbols from a finite alphabet, to equality classes can be performed on the Pointer Machine in time $\mathcal{O}(n + \sum_i |S_i|)$.

2.2.5. Remark. The first linear-time algorithm that partitions all subtrees to isomorphism equivalence classes is probably due to Zemlayachenko [Zem73], but it lacks many details. Dinitz et al. [DIR99] have recast this algorithm in modern terminology and filled the gaps. Our algorithm is easier to formulate than those, because it replaces the need for auxiliary data structures by more elaborate bucket sorting.

2.2.6. *Topological graph computations.* Many graph algorithms are based on the idea of so called *micro/macro decomposition:* We decompose a graph to subgraphs on roughly k vertices and solve the problem separately inside these "micrographs" and in the "macrograph" obtained by contraction of the micrographs. If k is small enough, many of the micrographs are isomorphic, so we can compute the result only once for each isomorphism class and recycle it for all micrographs of that class. On the other hand, the macrograph is roughly k times smaller than the original graph, so we can use a less efficient algorithm and it will still run in linear time with respect to the size of the original graph.

This kind of decomposition is traditionally used for trees, especially in the algorithms for the Lowest Common Ancestor problem (cf. Section 3.4 and the survey paper [AGKR02]) and for online maintenance of marked ancestors (cf. Alstrup et al. [AHR98]). Let us take a glimpse at what happens when we decompose a tree with k set to $1/4 \cdot \log n$. There are at most $2^{2k} = \sqrt{n}$ non-isomorphic subtrees of size k, because each isomorphism class is uniquely determined by the sequence of $2k$ up/down steps performed by depth-first search of the tree. Suppose that we are able to decompose the input and identify the equivalence classes of microtrees in linear time, then solve the problem in time $\mathcal{O}(\text{poly}(k))$ for each microtree and finally in $\mathcal{O}(n' \log n')$ for the macrotree of size $n' = n/k$. When we put these pieces together, we get an algorithm for the whole problem which achieves time complexity $\mathcal{O}(n + \sqrt{n} \cdot \text{poly}(\log n) + n/\log n \cdot \log(n/\log n)) = \mathcal{O}(n)$.

Decompositions are usually implemented on the RAM, because subgraphs can be easily encoded in numbers, and these can be then used to index arrays containing the precomputed results. As the previous algorithm for subtree isomorphism shows, indexing is not strictly required for identifying equivalent microtrees and it can be replaced by bucket sorting on the Pointer Machine. Buchsbaum et al. [BKRW98] have extended this technique to general graphs in form of so called topological graph computations. Let us define them.

2.2.7. Definition. A *graph computation* is a function that takes a *labeled undirected graph* as its input. The labels of vertices and edges can be arbitrary symbols drawn from a finite alphabet. The output of the computation is another labeling of the same graph. This time, the vertices and edges can be labeled with not only symbols of the alphabet, but also with pointers to the vertices and edges of the input graph, and possibly also with pointers to outside objects. A graph computation is called *topological* if it produces isomorphic outputs for isomorphic inputs. The isomorphism of course has to preserve not only the structure of the graph, but also the labels in the obvious way.

2.2.8. Observation. The topological graph computations cover a great variety of graph problems, ranging from searching for matchings or Eulerian tours to finding Hamilton circuits. The MST problem itself however does not belong to this class, because we do not have any means of representing the edge weights as labels, unless there is only a fixed amount of possible values.

As in the case of tree decompositions, we would like to identify the equivalent subgraphs and process only a single instance from each equivalence class. We need to be careful with the definition of the equivalence classes, because graph isomorphism is known to be computationally hard (it is one of the few problems that are neither known to lie in P nor to be NP-complete; see Arvind and Kurur [AK02] for recent results on its complexity). We will therefore manage with a weaker form of equivalence, based on some sort of graph encodings:

2.2.9. Definition. A *canonical encoding* of a given labeled graph represented by adjacency lists is obtained by running the depth-first search on the graph and recording its traces. We start with an empty encoding. When we enter a vertex, we assign an identifier to it (again using a yardstick to represent numbers) and we append the label of this vertex to the encoding. Then we scan all back edges going from this vertex and append the identifiers of their destinations, accompanied by the edges' labels. Finally we append a special terminator to mark the boundary between the code of this vertex and its successor.

2.2.10. Observation. The canonical encoding is well defined in the sense that non-isomorphic graphs always receive different encodings. Obviously, encodings of isomorphic graphs can differ, depending on the order of vertices and also of the adjacency lists. A graph on n vertices with m edges is assigned an encoding of length at most $2n + 2m$ — for each vertex, we record its label and a single terminator; edges contribute by identifiers and labels. These encodings can be constructed in linear time and in the same time we can also create a graph corresponding to a given encoding. We will use the encodings for our unification of graphs:

2.2.11. Definition. For a collection $\mathcal{C}$ of graphs, we define $|\mathcal{C}|$ as the number of graphs in the collection and $\|\mathcal{C}\|$ as their total size, i.e., $\|\mathcal{C}\| = \sum_{G \in \mathcal{C}} n(G) + m(G)$.

2.2.12. Lemma. *(Graph unification)*
A collection $\mathcal{C}$ of labeled graphs can be partitioned into classes which share the same canonical encoding in time $\mathcal{O}(\|\mathcal{C}\|)$ on the Pointer Machine.

Proof. Construct canonical encodings of all the graphs and then apply the Sequence unification lemma (2.2.4) on them. ♠

2.2.13. When we want to perform a topological computation on a collection $\mathcal{C}$ of graphs with k vertices, we first precompute its result for a collection $\mathcal{G}$ of *generic graphs* corresponding to all possible canonical encodings on k vertices. Then we use unification to match the *actual graphs* in $\mathcal{C}$ to the generic graphs in $\mathcal{G}$. This gives us the following theorem:

2.2.14. Theorem. *(Topological computations, Buchsbaum et al. [BKRW98])*
Suppose that we have a topological graph computation $\mathcal{T}$ that can be performed in time $T(k)$ for graphs on k vertices. Then we can run $\mathcal{T}$ on a collection $\mathcal{C}$ of labeled graphs on k vertices in time $\mathcal{O}(\|\mathcal{C}\|+(k+s)^{k(k+2)}\cdot(T(k)+k^2))$, where s is a constant depending only on the number of symbols used as vertex/edge labels.

Proof. A graph on k vertices has less than $k^2/2$ edges, so the canonical encodings of all such graphs are shorter than $2k + 2k^2/2 = k(k + 2)$. Each element of the encoding is either a vertex identifier, or a symbol, or a separator, so it can attain at most $k + s$ possible values for some fixed s. We can therefore enumerate all possible encodings and convert them to a collection $\mathcal{G}$ of all generic graphs such that $|\mathcal{G}| \le (k + s)^{k(k+2)}$ and $\|\mathcal{G}\| \le |\mathcal{G}| \cdot k^2$.

We run the computation on all generic graphs in time $\mathcal{O}(|\mathcal{G}| \cdot T(k))$ and then we use the Unification lemma (2.2.12) on the union of the collections $\mathcal{C}$ and $\mathcal{G}$ to match the generic graphs with the equivalent actual graphs in $\mathcal{C}$ in time $\mathcal{O}(\|\mathcal{C}\| + \|\mathcal{G}\|)$. Finally we create a copy of the generic result for each of the actual graphs. If the computation uses pointers to the input vertices in its output, we have to redirect them to the actual input vertices, which we can do by associating the output vertices that refer to an input vertex with the corresponding places in the encoding of the input graph. This way, the whole output can be generated in time $\mathcal{O}(\|\mathcal{C}\| + \|\mathcal{G}\|)$. ♠

2.2.15. Remark. The topological computations and the Graph unification lemma will play important roles in Sections 3.4 and 4.4.

2.3. Data structures on the RAM

There is a lot of data structures designed specifically for the RAM. These structures take advantage of both indexing and arithmetics and they often surpass the known lower bounds for the same problem on the PM. In many cases, they achieve constant time per operation, at least when either the magnitude of the values or the size of the data structure is suitably bounded.

A classical result of this type is the tree of van Emde Boas [vEB77] which represent a subset of the integers $\{0, \ldots, U-1\}$. It allows insertion, deletion and order operations (minimum, maximum, successor etc.) in time $\mathcal{O}(\log\log U)$, regardless of the size of the subset. If we replace the heap used in the Jarník's algorithm (1.4.9) by this structure, we immediately get an algorithm for finding the MST in integer-weighted graphs in time $\mathcal{O}(m \log\log w_{max})$, where w_{max} is the maximum weight.

A real breakthrough has however been made by Fredman and Willard who introduced the Fusion trees [FW93]. They again perform membership and predecessor operation on a set of n integers, but with time complexity $\mathcal{O}(\log_W n)$ per operation on a Word-RAM with W-bit words. This of course assumes that each element of the set fits in a single word. As W must at least $\log n$, the operations take $\mathcal{O}(\log n / \log\log n)$ time and thus we are able to sort n integers in time $o(n \log n)$. This was a beginning of a long sequence of faster and faster sorting algorithms, culminating with the work of Thorup and Han. They have improved the time complexity of integer sorting to $\mathcal{O}(n \log\log n)$ deterministically [Han02] and expected $\mathcal{O}(n\sqrt{\log\log n})$ for randomized algorithms [HT02], both in linear space.

The Fusion trees themselves have very limited use in graph algorithms, but the principles behind them are ubiquitous in many other data structures and these will serve us well and often. We are going to build the theory of Q-heaps in Section 2.5, which will later lead to a linear-time MST algorithm for arbitrary integer weights in Section 3.2. Other such structures will help us in building linear-time RAM algorithms for computing the ranks of various combinatorial structures in Chapter 7.

Outside our area, important consequences of RAM data structures include the Thorup's $\mathcal{O}(m)$ algorithm for single-source shortest paths in undirected graphs with positive integer weights [Tho99] and his $\mathcal{O}(m \log\log n)$ algorithm for the same problem in directed graphs [Tho04]. Both algorithms have been then significantly simplified by Hagerup [Hag00].

Despite the progress in the recent years, the corner-stone of all RAM structures is still the representation of combinatorial objects by integers introduced by Fredman and Willard. It will also form a basis for the rest of this chapter.

2.4. Bits and vectors

In this rather technical section, we will show how the RAM can be used as a vector computer to operate in parallel on multiple elements, as long as these elements fit in a single machine word. At the first sight this might seem useless, because we cannot require large word sizes, but surprisingly often the elements are small enough relative to the size of the algorithm's input and thus also relative to the minimum possible word size. Also, as the following lemma shows, we can easily emulate slightly longer words:

2.4.1. Lemma. *(Multiple-precision calculations)*
Given a RAM with W-bit words, we can emulate all calculation and control instructions of a RAM with word size kW in time depending only on the k. (This is usually called *multiple-precision arithmetics.*)

Proof. We split each word of the "big" machine to W'-bit blocks, where $W' = W/2$, and store these blocks in $2k$ consecutive memory cells. Addition, subtraction, comparison and bitwise logical operations can be performed block-by-block. Shifts by a multiple of W' are trivial, otherwise we can combine each block of the result from shifted versions of two original blocks. To multiply two numbers, we can use the elementary school algorithm using the W'-bit blocks as digits in base $2^{W'}$ — the product of any two blocks fits in a single word.

Division is harder, but Newton-Raphson iteration (see [ITY95]) converges to the quotient in a constant number of iterations, each of them involving $\mathcal{O}(1)$

multiple-precision additions and multiplications. A good starting approximation can be obtained by dividing the two most-significant (non-zero) blocks of both numbers.

Another approach to division is using the improved elementary school algorithm as described by Knuth in [Knu97b]. It uses $\mathcal{O}(k^2)$ steps, but the steps involve calculation of the most significant bit set in a word. We will show below that it can be done in constant time, but we have to be careful to avoid division instructions in it. ♠

2.4.2. Notation. *(Bit strings)*
We will work with binary representations of natural numbers by strings over the alphabet $\{\mathbf{0}, \mathbf{1}\}$: we will use $\langle x \rangle$ for the number x written in binary, $\langle x \rangle_b$ for the same padded to exactly b bits by adding leading zeroes, and $x[k]$ for the value of the k-th bit of x (with a numbering of bits such that $2^k[k] = 1$). The usual conventions for operations on strings will be utilized: When s and t are strings, we write st for their concatenation and s^k for the string s repeated k times. When the meaning is clear from the context, we will use x and $\langle x \rangle$ interchangeably to avoid outbreak of symbols.

2.4.3. Definition. The *bitwise encoding* of a vector $\mathbf{x} = (x_0, \ldots, x_{d-1})$ of b-bit numbers is an integer x such that $\langle x \rangle = \langle x_{d-1} \rangle_b \mathbf{0} \langle x_{d-2} \rangle_b \mathbf{0} \ldots \mathbf{0} \langle x_0 \rangle_b$. In other words, $x = \sum_i 2^{(b+1)i} \cdot x_i$. (We have interspersed the elements with *separator bits.*)

2.4.4. Notation. *(Vectors)*
We will use boldface letters for vectors and the same letters in normal type for the encodings of these vectors. The elements of a vector $\mathbf{x}$ will be written as $x_0, \ldots, x_{d-1}$.

2.4.5. If we want to fit the whole vector in a single word, the parameters b and d must satisfy the condition $(b+1)d \leq W$. By using multiple-precision arithmetics, we can encode all vectors satisfying $bd = \mathcal{O}(W)$. We will now describe how to translate simple vector manipulations to sequences of $\mathcal{O}(1)$ RAM operations on their codes. As we are interested in asymptotic complexity only, we will prefer clarity of the algorithms over saving instructions. Among other things, we will freely use calculations on words of size $\mathcal{O}(bd)$, assuming that the Multiple-precision lemma comes to save us when necessary.

2.4.6. First of all, let us observe that we can use AND and OR with suitable constants to write zeroes or ones to an arbitrary set of bit positions at once. These operations are usually called *bit masking*. Also, any element of a vector can be extracted or replaced by a different value in $\mathcal{O}(1)$ time by masking and shifts.

2.4.7. Algorithm. *(Operations on vectors with d elements of b bits each)*

- *Replicate*(α) — Create a vector $(\alpha, \ldots, \alpha)$:

  $$\mathit{Replicate}(\alpha) = \alpha \cdot (\mathbf{0}^b \mathbf{1})^d.$$

- *Sum*(x) — Calculate the sum of the elements of $\mathbf{x}$, assuming that the result fits in b bits:

  $$\mathit{Sum}(x) = x \bmod \mathbf{1}^{b+1}.$$

  This is correct because when we calculate modulo $\mathbf{1}^{b+1}$, the number $2^{b+1} = \mathbf{10}^{b+1}$ is congruent to 1 and thus $x = \sum_i 2^{(b+1)i} \cdot x_i \equiv \sum_i 1^i \cdot x_i \equiv \sum_i x_i$. As the result should fit in b bits, the modulo makes no difference.

If we want to avoid division, we can use double-precision multiplication instead:

$$\begin{array}{cccccccccc}
 & & & & & \mathbf{0}x_{d-1} & \cdots & \mathbf{0}x_2 & \mathbf{0}x_1 & \mathbf{0}x_0 \\
 & & & & * & \mathbf{0}^b\mathbf{1} & \cdots & \mathbf{0}^b\mathbf{1} & \mathbf{0}^b\mathbf{1} & \mathbf{0}^b\mathbf{1} \\
\hline
 & & & & & x_{d-1} & \cdots & x_2 & x_1 & x_0 \\
 & & & & x_{d-1} & x_{d-2} & \cdots & x_1 & x_0 & \\
 & & & x_{d-1} & x_{d-2} & x_{d-3} & \cdots & x_0 & & \\
 & & & \vdots & \vdots & \vdots & \vdots & & & \\
x_{d-1} & \cdots & x_2 & x_1 & x_0 & & & & & \\
\hline
r_{d-1} & \cdots & r_2 & r_1 & s_d & \cdots & s_3 & s_2 & s_1 &
\end{array}$$

This way, we also get all partial sums: $s_k = \sum_{i=0}^{k-1} x_i$, $r_k = \sum_{i=k}^{d-1} x_i$.

- *Cmp*(x, y) — Compare vectors $\mathbf{x}$ and $\mathbf{y}$ element-wise, i.e., make a vector $\mathbf{z}$ such that $z_i = 1$ if $x_i < y_i$ and $z_i = 0$ otherwise.
  We replace the separator zeroes in x by ones and subtract y. These ones change back to zeroes exactly at the positions where $x_i < y_i$ and they stop carries from propagating, so the fields do not interact with each other:

$$\begin{array}{cccccccccc}
 & \mathbf{1} & x_{d-1} & \mathbf{1} & x_{d-2} & \cdots & \mathbf{1} & x_1 & \mathbf{1} & x_0 \\
- & \mathbf{0} & y_{d-1} & \mathbf{0} & y_{d-2} & \cdots & \mathbf{0} & y_1 & \mathbf{0} & y_0 \\
\hline
 & ? & \ldots & ? & \ldots & \cdots & ? & \ldots & ? & \ldots
\end{array}$$

  It only remains to shift the separator bits to the right positions, negate them and mask out all other bits.
- *Rank*(x, α) — Return the number of elements of $\mathbf{x}$ which are less than α, assuming that the result fits in b bits:

$$\mathit{Rank}(x, \alpha) = \mathit{Sum}(\mathit{Cmp}(x, \mathit{Replicate}(\alpha))).$$

- *Insert*(x, α) — Insert α into a sorted vector $\mathbf{x}$:
  We calculate the rank of α in x first, then we insert α into the particular field of $\mathbf{x}$ using masking operations and shifts.

  1. $k \leftarrow \mathit{Rank}(x, \alpha)$.
  2. $\ell \leftarrow x$ AND $\mathbf{1}^{(b+1)(n-k)}\mathbf{0}^{(b+1)k}$. *("left" part of the vector)*
  3. $r = x$ AND $\mathbf{1}^{(b+1)k}$. *("right" part)*
  4. Return $(\ell \ll (b+1))$ OR $(\alpha \ll ((b+1)k))$ OR r.

- *Unpack*(α) — Create a vector whose elements are the bits of $\langle\alpha\rangle_d$. In other words, insert blocks $\mathbf{0}^b$ between the bits of α. Assuming that $b \geq d$, we can do it as follows:

  1. $x \leftarrow \mathit{Replicate}(\alpha)$.
  2. $y \leftarrow (2^{b-1}, 2^{b-2}, \ldots, 2^0)$. *(bitwise encoding of this vector)*
  3. $z \leftarrow x$ AND y.
  4. Return $\mathit{Cmp}(z, y)$ XOR $(\mathbf{0}^b\mathbf{1})^d$.

  Let us observe that z_i is either zero or equal to y_i depending on the value of the i-th bit of the number α. Comparing it with y_i normalizes it to

either zero or one, but in the opposite way than we need, so we flip the bits by an additional XOR.

- *Unpack*$_\pi(\alpha)$ — Like *Unpack*, but change the order of the bits according to a fixed permutation π: The i-th element of the resulting vector is equal to $\alpha[\pi(i)]$.
  Implemented as above, but with a mask $y = (2^{\pi(b-1)}, \ldots, 2^{\pi(0)})$.
- *Pack*(x) — The inverse of *Unpack*: given a vector of zeroes and ones, produce a number whose bits are the elements of the vector (in other words, it crosses out the $\mathbf{0}^b$ blocks).
  We interpret the x as an encoding of a vector with elements one bit shorter and we sum these elements. For example, when $n = 4$ and $b = 4$:

$$\begin{array}{|cccc|cccc|cccc|cccc|}
\hline
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\, x_3 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\, x_2 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\, x_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\, x_0 \\
\hline
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\, x_3 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\, x_2 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\, x_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\, x_0 \\
\hline
\end{array}$$

  However, this "reformatting" does not produce a correct encoding of a vector, because the separator zeroes are missing. For this reason, the implementation of *Sum* using modulo does not work correctly (it produces $\mathbf{0}^b$ instead of $\mathbf{1}^b$). We therefore use the technique based on multiplication instead, which does not need the separators. (Alternatively, we can observe that $\mathbf{1}^b$ is the only case affected, so we can handle it separately.)

2.4.8. *Scalar operations.* We can use the aforementioned tricks to perform interesting operations on individual numbers in constant time, too. Let us assume for a while that we are operating on b-bit numbers and the word size is at least b^2. This enables us to make use of intermediate vectors with b elements of b bits each.

2.4.9. Algorithm. *(Integer operations in quadratic workspace)*

- *Weight*(α) — Compute the Hamming weight of α, i.e., the number of ones in $\langle\alpha\rangle$.
  We perform *Unpack* and then *Sum*.
- *Permute*$_\pi(\alpha)$ — Shuffle the bits of α according to a fixed permutation π.
  We perform *Unpack*$_\pi$ and *Pack* back.
- *LSB*(α) — Find the least significant bit of α, i.e., the smallest i such that $\alpha[i] = 1$.
  By a combination of subtraction with XOR, we create a number that contains ones exactly at the position of *LSB*(α) and below:

$$\begin{aligned}
\alpha &= \cdots\cdots\mathbf{10000} \\
\alpha - 1 &= \cdots\cdots\mathbf{01111} \\
\alpha \operatorname{XOR} (\alpha - 1) &= \mathbf{0}\cdots\mathbf{011111}
\end{aligned}$$

  Then we calculate the *Weight* of the result and subtract 1.
- *MSB*(α) — Find the most significant bit of α (the position of the highest bit set).
  Reverse the bits of the number α first by calling *Permute*, then apply *LSB* and subtract the result from $b - 1$.

2.4.10. *LSB and MSB.* As noted by Brodnik [Bro93] and others, the space requirements of the *LSB* operation can be lowered to linear. We split the w-bit input to

$\sqrt{w}$ blocks of $\sqrt{w}$ bits each. Then we determine which blocks are non-zero and identify the lowest such block (this is a *LSB* of a number whose bits correspond to the blocks). Finally we calculate the *LSB* of this block. In both calls to *LSB,* we have a $\sqrt{w}$-bit number in a w-bit word, so we can use the previous algorithm. The same trick of course applies to for finding the *MSB*, too.

The following algorithm shows the details:

2.4.11. Algorithm. *(LSB in linear workspace)*

Input: A w-bit number α.

1. $b \leftarrow \lceil \sqrt{w} \rceil$. *(the size of a block)*
2. $\ell \leftarrow b$. *(the number of blocks is the same)*
3. $x \leftarrow (\alpha \text{ AND } (\mathbf{01}^b)^\ell) \text{ OR } ((\alpha \text{ AND } (\mathbf{10}^b)^\ell) >> 1)$.
   (encoding of a vector $\mathbf{x}$ *such that* $x_i \neq 0$ *iff the* i*-th block is non-zero)*[3]
4. $y \leftarrow Cmp(0, x)$. *($y_i = 1$ if the i-th block is non-zero, otherwise $y_0 = 0$)*
5. $\beta \leftarrow Pack(y)$. *(each block compressed to a single bit)*
6. $p \leftarrow LSB(\beta)$. *(the index of the lowest non-zero block)*
7. $\gamma \leftarrow (\alpha >> bp) \text{ AND } \mathbf{1}^b$. *(the contents of that block)*
8. $q \leftarrow LSB(\gamma)$. *(the lowest bit set there)*

Output: $LSB(\alpha) = bp + q$.

2.4.12. *Constants.* We have used a plenty of constants that depend on the format of the vectors. Either we can write non-uniform programs (see 2.1.5) and use native constants, or we can observe that all such constants can be easily manufactured. For example, $(\mathbf{0}^b\mathbf{1})^d = \mathbf{1}^{(b+1)d}/\mathbf{1}^{b+1} = (2^{(b+1)d} - 1)/(2^{b+1} - 1) = ((1 << (b+1)d) - 1)/((2 << b) - 1)$. The only exceptions are the w and b in the LSB algorithm 2.4.11, which we are unable to produce in constant time. In practice we use the "bit tricks" as frequently called subroutines in an encompassing algorithm, so we usually can afford spending a lot of time on the precalculation of constants performed once during algorithm startup.

2.4.13. *History.* The history of combining arithmetic and logical operations to obtain fast programs for various interesting functions is blurred. Many of the bit tricks, which we have described, have been discovered independently by numerous people in the early ages of digital computers. Since then, they have become a part of the computer science folklore. Probably the earliest documented occurrence is in the 1972's memo of the MIT Artificial Intelligence Lab [BGS72]. However, until the work of Fredman and Willard nobody seemed to realize the full consequences.

[3] Why is this so complicated? It is tempting to take α itself as a code of this vector, but we must not forget the separator bits between elements, so we create them and relocate the bits we have overwritten.

2.5. Q-Heaps

We have shown how to perform non-trivial operations on a set of values in constant time, but so far only under the assumption that the number of these values is small enough and that the values themselves are also small enough (so that the whole set fits in $\mathcal{O}(1)$ machine words). Now we will show how to lift the restriction on the magnitude of the values and still keep constant time complexity. We will describe a slightly simplified version of the Q-heaps developed by Fredman and Willard in [FW90].

The Q-heap represents a set of at most k word-sized integers, where $k \leq W^{1/4}$ and W is the word size of the machine. It will support insertion, deletion, finding of minimum and maximum, and other operations described below, in constant time, provided that we are willing to spend $\mathcal{O}(2^{k^4})$ time on preprocessing.

The exponential-time preprocessing may sound alarming, but a typical application uses Q-heaps of size $k = \log^{1/4} N$, where N is the size of the algorithm's input. This guarantees that $k \leq W^{1/4}$ and $\mathcal{O}(2^{k^4}) = \mathcal{O}(N)$. Let us however remark that the whole construction is primarily of theoretical importance — the huge multiplicative constants hidden in the $\mathcal{O}$ make these heaps useless in practical algorithms. Despite this, many of the tricks we develop have proven themselves useful even in real-life data structures.

Spending so much time on preprocessing makes it possible to precompute tables of almost arbitrary functions and then assume that the functions can be evaluated in constant time:

2.5.1. Lemma. When f is a function computable in polynomial time, $\mathcal{O}(2^{k^4})$ time is enough to precompute a table of the values of f for all arguments whose size is $\mathcal{O}(k^3)$ bits.

Proof. There are $2^{\mathcal{O}(k^3)}$ possible combinations of arguments of the given size and for each of them we spend $\mathrm{poly}(k)$ time on calculating the function. It remains to observe that $2^{\mathcal{O}(k^3)} \cdot \mathrm{poly}(k) = \mathcal{O}(2^{k^4})$. ♠

2.5.2. *Tries and ranks.* We will first develop an auxiliary construction based on tries and then derive the real definition of the Q-heap from it.

2.5.3. Notation.

- W — the word size of the RAM,
- $k = \mathcal{O}(W^{1/4})$ — the limit on the size of the heap,
- $n \leq k$ — the number of elements stored in the heap,
- $X = \{x_1, \ldots, x_n\}$ — the elements themselves: distinct W-bit numbers indexed in a way that $x_1 < \ldots < x_n$,
- $g_i = MSB(x_i \operatorname{XOR} x_{i+1})$ — the position of the most significant bit in which x_i and x_{i+1} differ,
- $R_X(x)$ — the rank of x in X, that is the number of elements of X which are less than x (where x itself need not be an element of X).[4]

2.5.4. Definition. A *trie* for a set of strings S over a finite alphabet Σ is a rooted tree whose vertices are the prefixes of the strings in S and there is an edge going

[4] We will dedicate the whole Chapter 7 to the study of various ranks.

from a prefix α to a prefix β iff β can be obtained from α by appending a single symbol of the alphabet. The edge will be labeled with that particular symbol. We will also define the *letter depth* of a vertex as the length of the corresponding prefix. We mark the vertices which match a string of S.

A *compressed trie* is obtained by removing the vertices of outdegree 1 except for the root and the marked vertices. Wherever there is a directed path whose internal vertices have outdegree 1 and they carry no mark, we replace this path by a single edge labeled with the concatenation of the original edges' labels.

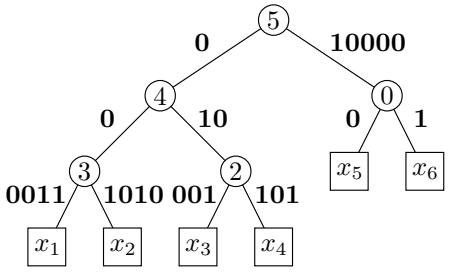
In both kinds of tries, we order the outgoing edges of every vertex by their labels lexicographically.

2.5.5. Observation. In both tries, the root of the tree is the empty word. Generally, the prefix in a vertex is equal to the concatenation of edge labels on the path leading from the root to that vertex. The letter depth of the vertex is equal to the total size of these labels. All leaves correspond to strings in S, but so can some internal vertices if there are two strings in S such that one is a prefix of the other.

Furthermore, the labels of all edges leaving a common vertex are always distinct and when we compress the trie, no two such labels share their initial symbols. This allows us to search in the trie efficiently: when looking for a string x, we follow the path from the root and whenever we visit an internal vertex of letter depth d, we test the d-th character of x, we follow the edge whose label starts with this character, and we check that the rest of the label matches.

The compressed trie is also efficient in terms of space consumption — it has $\mathcal{O}(|S|)$ vertices (this can be easily shown by induction on $|S|$) and all edge labels can be represented in space linear in the sum of the lengths of the strings in S.

2.5.6. Definition. For our set X, we define T as a compressed trie for the set of binary encodings of the numbers x_i, padded to exactly W bits, i.e., for $S = \{\langle x \rangle_W \mid x \in X\}$.



$x_1 = \mathbf{000011}$ $g_1 = 3$
$x_2 = \mathbf{001010}$ $g_2 = 4$
$x_3 = \mathbf{010001}$ $g_3 = 2$
$x_4 = \mathbf{010101}$ $g_4 = 5$
$x_5 = \mathbf{100000}$ $g_5 = 0$
$x_6 = \mathbf{100001}$

Six numbers stored in a compressed trie

2.5.7. Observation. The trie T has several interesting properties. Since all words in S have the same length, the leaves of the trie correspond to these exact words, that is to the numbers x_i. The depth-first traversal of the trie enumerates the words of S in lexicographic order and therefore also the x_i's in the order of their values. Between each pair of leaves x_i and x_{i+1} it visits an internal vertex whose letter depth is exactly $W - 1 - g_i$.

2.5.8. Let us now modify the algorithm for searching in the trie and make it compare only the first symbols of the edges. In other words, we will test only the bits g_i

which will be called *guides* (as they guide us through the tree). For $x \in X$, the modified algorithm will still return the correct leaf. For all x outside X it will no longer fail and instead it will land on some leaf x_i. At the first sight the number x_i may seem unrelated, but we will show that it can be used to determine the rank of x in X, which will later form a basis for all Q-heap operations:

2.5.9. Lemma. The rank $R_X(x)$ is uniquely determined by a combination of:

- the trie T,
- the index i of the leaf found when searching for x in T,
- the relation $(<, =, >)$ between x and x_i,
- the bit position $b = MSB(x \text{ XOR } x_i)$ of the first disagreement between x and x_i.

Proof. If $x \in X$, we detect that from $x_i = x$ and the rank is obviously $i - 1$. Let us assume that $x \notin X$ and imagine that we follow the same path as when searching for x, but this time we check the full edge labels. The position b is the first position where $\langle x \rangle$ disagrees with a label. Before this point, all edges not taken by the search were leading either to subtrees containing elements all smaller than x or all larger than x and the only values not known yet are those in the subtree below the edge that we currently consider. Now if $x[b] = 0$ (and therefore $x < x_i$), all values in that subtree have $x_j[b] = 1$ and thus they are larger than x. In the other case, $x[b] = 1$ and $x_j[b] = 0$, so they are smaller. ♠

2.5.10. *A better representation.* The preceding lemma shows that the rank can be computed in polynomial time, but unfortunately the variables on which it depends are too large for a table to be efficiently precomputed. We will carefully choose an equivalent representation of the trie which is compact enough.

2.5.11. Lemma. The compressed trie is uniquely determined by the order of the guides $g_1, \ldots, g_{n-1}$.

Proof. We already know that the letter depths of the trie vertices are exactly the numbers $W - 1 - g_i$. The root of the trie must have the smallest of these letter depths, i.e., it must correspond to the highest numbered bit. Let us call this bit g_i. This implies that the values $x_1, \ldots, x_i$ must lie in the left subtree of the root and $x_{i+1}, \ldots, x_n$ in its right subtree. Both subtrees can be then constructed recursively.[5] ♠

2.5.12. Unfortunately, the vector of the g_i's is also too long (is has $k \log W$ bits and we have no upper bound on W in terms of k), so we will compress it even further:

2.5.13. Notation.

- $B = \{g_1, \ldots, g_n\}$ — the set of bit positions of all the guides, stored as a sorted array,
- $G : \{1, \ldots, n\} \to \{1, \ldots, n\}$ — a function mapping the guides to their bit positions in B: $g_i = B[G(i)]$,
- $x[B]$ — a bit string containing the bits of x originally located at the positions given by B, i.e., the concatenation of bits $x[B[1]], x[B[2]], \ldots, x[B[n]]$.

[5] This construction is also known as the *cartesian tree* for the sequence $g_1, \ldots, g_{n-1}$ and it is useful in many other algorithms as it can be built in $\mathcal{O}(n)$ time. A nice application on the Lowest Common Ancestor and Range Minimum problems has been described by Bender et al. in [BFC00].

2.5.14. Observation. The set B has $\mathcal{O}(k \log W) = \mathcal{O}(W)$ bits, so it can be stored in a constant number of machine words in the form of a sorted vector. The function G can be also stored as a vector of $\mathcal{O}(k \log k)$ bits. We can change a single g_i in constant time using vector operations: First we delete the original value of g_i from B if it is not used anywhere else. Then we add the new value to B if it was not there yet and we write its position in B to $G(i)$. Whenever we insert or delete a value in B, the values at the higher positions shift one position up or down and we have to update the pointers in G. This can be fortunately accomplished by adding or subtracting a result of vector comparison.

In this representation, we can reformulate our lemma on ranks as follows:

2.5.15. Lemma. The rank $R_X(x)$ can be computed in constant time from:

- the function G,
- the values $x_1, \ldots, x_n$,
- the bit string $x[B]$,
- x itself.

Proof. Let us prove that all ingredients of Lemma 2.5.9 are either small enough or computable in constant time.

We know that the shape of the trie T is uniquely determined by the order of the g_i's and therefore by the function G since the array B is sorted. The shape of the trie together with the bits in $x[B]$ determine the leaf x_i found when searching for x using only the guides. This can be computed in polynomial time and it depends on $\mathcal{O}(k \log k)$ bits of input, so according to Lemma 2.5.1 we can look it up in a precomputed table.

The relation between x and x_i can be obtained directly as we know the x_i. The bit position of the first disagreement can be calculated in constant time using the Brodnik's LSB/MSB algorithm (2.4.11).

All these ingredients can be stored in $\mathcal{O}(k \log k)$ bits, so we may assume that the rank can be looked up in constant time as well. ♠

2.5.16. We would like to store the set X as a sorted array together with the corresponding trie, which will allow us to determine the position for a newly inserted element in constant time. However, the set is too large to fit in a vector and we cannot perform insertion on an ordinary array in constant time. This can be worked around by keeping the set in an unsorted array and storing a separate vector containing the permutation that sorts the array. We can then insert a new element at an arbitrary place in the array and just update the permutation to reflect the correct order.

2.5.17. *The Q-heap.* We are now ready for the real definition of the Q-heap and for the description of the basic operations on it.

2.5.18. Definition. A *Q-heap* consists of:

- k, n — the capacity of the heap and the current number of elements (word-sized integers),
- X — the set of word-sized elements stored in the heap (an array of words in an arbitrary order),
- ϱ — a permutation on $\{1, \ldots, n\}$ such that $X[\varrho(1)] < \ldots < X[\varrho(n)]$ (a vector of $\mathcal{O}(n \log k)$ bits; we will write x_i for $X[\varrho(i)]$),

- B — a set of "interesting" bit positions (a sorted vector of $\mathcal{O}(n \log W)$ bits),
- G — the function that maps the guides to the bit positions in B (a vector of $\mathcal{O}(n \log k)$ bits),
- precomputed tables of various functions.

2.5.19. Algorithm. *(Search in the Q-heap)*

Input: A Q-heap and an integer x to search for.

1. $i \leftarrow R_X(x) + 1$, using Lemma 2.5.15 to calculate the rank.
2. If $i \leq n$ return x_i, otherwise return UNDEFINED.

Output: The smallest element of the heap which is greater or equal to x.

2.5.20. Algorithm. *(Insertion to the Q-heap)*

Input: A Q-heap and an integer x to insert.

1. $i \leftarrow R_X(x) + 1$, using Lemma 2.5.15 to calculate the rank.
2. If $x = x_i$, return immediately (the value is already present).
3. Insert the new value to X:
4. $n \leftarrow n + 1$.
5. $X[n] \leftarrow x$.
6. Insert n at the i-th position in the permutation ϱ.
7. Update the g_j's:
8. Move all g_j for $j \geq i$ one position up.
   This translates to insertion in the vector representing G.
9. Recalculate g_{i-1} and g_i according to the definition.
   Update B and G as described in 2.5.14.

Output: The updated Q-heap.

2.5.21. Algorithm. *(Deletion from the Q-heap)*

Input: A Q-heap and an integer x to be deleted from it.

1. $i \leftarrow R_X(x) + 1$, using Lemma 2.5.15 to calculate the rank.
2. If $i > n$ or $x_i \neq x$, return immediately (the value is not in the heap).
3. Delete the value from X:
4. $X[\varrho(i)] \leftarrow X[n]$.
5. Find j such that $\varrho(j) = n$ and set $\varrho(j) \leftarrow \varrho(i)$.
6. $n \leftarrow n - 1$.
7. Update the g_j's like in the previous algorithm.

Output: The updated Q-heap.

2.5.22. Algorithm. *(Finding the i-th smallest element in the Q-heap)*

Input: A Q-heap and an index i.

1. If $i < 1$ or $i > n$, return UNDEFINED.
2. Return x_i.

Output: The i-th smallest element in the heap.

2.5.23. *Extraction.* The heap algorithms we have just described have been built from primitives operating in constant time, with one notable exception: the extraction $x[B]$ of all bits of x at positions specified by the set B. This cannot be done

in $\mathcal{O}(1)$ time on the Word-RAM, but we can implement it with AC^0 instructions as suggested by Andersson in [AMT99] or even with those AC^0 instructions present on real processors (see Thorup [Tho03]). On the Word-RAM, we need to make use of the fact that the set B is not changing too much — there are $\mathcal{O}(1)$ changes per Q-heap operation. As Fredman and Willard have shown, it is possible to maintain a "decoder", whose state is stored in $\mathcal{O}(1)$ machine words and which helps us to extract $x[B]$ in a constant number of operations:

2.5.24. Lemma. *(Extraction of bits)*
Under the assumptions on k, W and the preprocessing time as in the Q-heaps,[6] it is possible to maintain a data structure for a set B of bit positions, which allows $x[B]$ to be extracted in $\mathcal{O}(1)$ time for an arbitrary x. When a single element is inserted to B or deleted from B, the structure can be updated in constant time, as long as $|B| \leq k$.

Proof. See Fredman and Willard [FW90]. ♠

2.5.25. This was the last missing bit of the mechanics of the Q-heaps. We are therefore ready to conclude this section by the following theorem and its consequences:

2.5.26. Theorem. *(Q-heaps, Fredman and Willard [FW90])*
Let W and k be positive integers such that $k = \mathcal{O}(W^{1/4})$. Let Q be a Q-heap of at most k-elements of W bits each. Then the Q-heap operations 2.5.19 to 2.5.22 on Q (insertion, deletion, search for a given value and search for the i-th smallest element) run in constant time on a Word-RAM with word size W, after spending time $\mathcal{O}(2^{k^4})$ on the same RAM on precomputing of tables.

Proof. Every operation on the Q-heap can be performed in a constant number of vector operations and calculations of ranks. The ranks are computed in $\mathcal{O}(1)$ steps involving again $\mathcal{O}(1)$ vector operations, binary logarithms and bit extraction. All these can be calculated in constant time using the results of Section 2.4 and Lemma 2.5.24. ♠

2.5.27. *Combining Q-heaps.* We can also use the Q-heaps as building blocks of more complex structures like Atomic heaps and AF-heaps (see once again [FW90]). We will show a simpler, but often sufficient construction, sometimes called the *Q-heap tree.* Suppose we have a Q-heap of capacity k and a parameter $d \in \mathbb{N}^+$. We can build a balanced k-ary tree of depth d such that its leaves contain a given set and every internal vertex keeps the minimum value in the subtree rooted in it, together with a Q-heap containing the values in all its sons. This allows minimum to be extracted in constant time (it is placed in the root) and when any element is changed, it is sufficient to recalculate the values from the path from this element to the root, which takes $\mathcal{O}(d)$ Q-heap operations.

2.5.28. Corollary. *(Q-heap trees)*
For every positive integer r and $\delta > 0$ there exists a data structure capable of maintaining the minimum of a set of at most r word-sized numbers under insertions and deletions. Each operation takes $\mathcal{O}(1)$ time on a Word-RAM with word size

[6] Actually, this is the only place where we need k to be as low as $W^{1/4}$. In the AC^0 implementation, it is enough to ensure $k \log k \leq W$. On the other hand, we need not care about the exponent because it can be increased arbitrarily using the Q-heap trees described below.

$W = \Omega(r^\delta)$, after spending time $\mathcal{O}(2^{r^\delta})$ on precomputing of tables.

Proof. Choose $\delta' \leq \delta$ such that $r^{\delta'} = \mathcal{O}(W^{1/4})$. Build a Q-heap tree of depth $d = \lceil \delta/\delta' \rceil$ containing Q-heaps of size $k = r^{\delta'}$. ♠

2.5.29. Remark. When we have an algorithm with input of size N, the word size is at least $\log N$ and we can spend time $\mathcal{O}(N)$ on preprocessing, so we can choose $r = \log N$ and $\delta = 1$ in the above corollary and get a heap of size $\log N$ working in constant time per operation.

3. Advanced MST Algorithms

3.1. Minor-closed graph classes

The contractive algorithm given in Section 1.5 has been found to perform well on planar graphs, but in the general case its time complexity was not linear. Can we find any broader class of graphs where this algorithm is still efficient? The right context turns out to be the minor-closed graph classes, which are closed under contractions and have bounded density.

3.1.1. Definition. A graph H is a *minor* of a graph G (written as $H \preccurlyeq G$) iff it can be obtained from a subgraph of G by a sequence of simple graph contractions (see A.2.4).

3.1.2. Definition. A class $\mathcal{C}$ of graphs is *minor-closed*, when for every $G \in \mathcal{C}$ and every minor H of G, the graph H lies in $\mathcal{C}$ as well. A class $\mathcal{C}$ is called *non-trivial* if at least one graph lies in $\mathcal{C}$ and at least one lies outside $\mathcal{C}$.

3.1.3. Example. Non-trivial minor-closed classes include:

- planar graphs,
- graphs embeddable in any fixed surface (i.e., graphs of bounded genus),
- graphs embeddable in $\mathbb{R}^3$ without knots or without interlocking cycles,
- graphs of bounded tree-width or path-width.

3.1.4. Many of the nice structural properties of planar graphs extend to minor-closed classes, too (see Lovász [Lov05] for a nice survey of this theory and Diestel [Die05] for some of the deeper results). The most important property is probably the characterization of such classes in terms of their forbidden minors.

3.1.5. Definition. For a class $\mathcal{H}$ of graphs we define $\mathrm{Forb}(\mathcal{H})$ as the class of graphs that do not contain any of the graphs in $\mathcal{H}$ as a minor. We will call $\mathcal{H}$ the set of *forbidden (or excluded) minors* for this class. We will often abbreviate $\mathrm{Forb}(\{M_1, \ldots, M_n\})$ to $\mathrm{Forb}(M_1, \ldots, M_n)$.

3.1.6. Observation. For every $\mathcal{H} \neq \emptyset$, the class $\mathrm{Forb}(\mathcal{H})$ is non-trivial and closed on minors. This works in the opposite direction as well: for every minor-closed class $\mathcal{C}$ there is a class $\mathcal{H}$ such that $\mathcal{C} = \mathrm{Forb}(\mathcal{H})$. One such $\mathcal{H}$ is the complement of $\mathcal{C}$, but smaller ones can be found, too. For example, the planar graphs can be equivalently described as the class $\mathrm{Forb}(K_5, K_{3,3})$ — this follows from the Kuratowski's theorem (the theorem speaks of forbidden subdivisions, but while in general this is not the same as forbidden minors, it is for K_5 and $K_{3,3}$). The celebrated theorem by Robertson and Seymour guarantees that we can always find a finite set of forbidden minors:

3.1.7. Theorem. *(Excluded minors, Robertson & Seymour [RS04])*
For every non-trivial minor-closed graph class $\mathcal{C}$ there exists a finite set $\mathcal{H}$ of graphs such that $\mathcal{C} = \mathrm{Forb}(\mathcal{H})$.

Proof. This theorem has been proven in a long series of papers on graph minors culminating with [RS04]. See this paper and follow the references to the previous articles in the series. ♠

3.1.8. For analysis of the contractive algorithm, we will make use of another important property — the bounded density of minor-closed classes. The connection between minors and density dates back to Mader in the 1960's and it can be proven without use of the Robertson-Seymour theory.

3.1.9. Definition. Let G be a graph and $\mathcal{C}$ be a class of graphs. We define the *edge density* $\varrho(G)$ of G as the average number of edges per vertex, i.e., $m(G)/n(G)$. The edge density $\varrho(\mathcal{C})$ of the class is then defined as the infimum of $\varrho(G)$ over all $G \in \mathcal{C}$.

3.1.10. Theorem. *(Mader [Mad67])*
For every $k \in \mathbb{N}$ there exists $h(k) \in \mathbb{R}$ such that every graph of average degree at least $h(k)$ contains a subdivision of K_k as a subgraph.

Proof sketch. (See Lemma 3.5.1 in [Die05] for a complete proof in English.)

Let us fix k and prove by induction on m that every graph of average degree at least 2^m contains a subdivision of some graph with k vertices and m edges (for $k \leq m \leq \binom{k}{2}$). When we reach $m = \binom{k}{2}$, the theorem follows as the only graph with k vertices and $\binom{k}{2}$ edges is K_k.

The base case $m = k$: Let us observe that when the average degree is a, removing any vertex of degree less than $a/2$ does not decrease the average degree. A graph with $a \geq 2^k$ therefore has a subgraph with minimum degree $\delta \geq a/2 = 2^{k-1}$. Such subgraph contains a cycle on more than δ vertices, in other words a subdivision of the cycle C_k.

Induction step: Let G be a graph with average degree at least 2^m and assume that the theorem already holds for $m-1$. Without loss of generality, G is connected. Consider a maximal set $U \subseteq V$ such that the subgraph $G[U]$ induced by U is connected and the graph $G\,.\,U$ (G with U contracted to a single vertex) has average degree at least 2^m (such U exists, because $G = G\,.\,U$ whenever $|U| = 1$). Now consider the subgraph H induced in G by the neighbors of U. Every $v \in V(H)$ must have $\deg_H(v) \geq 2^{m-1}$, as otherwise we can add this vertex to U, contradicting its maximality. By the induction hypothesis, H contains a subdivision of some graph R with k vertices and $m - 1$ edges. Any two non-adjacent vertices of R can be connected in the subdivision by a path lying entirely in $G[U]$, which reveals a subdivision of a graph with m edges. ♠

3.1.11. Theorem. *(Density of minor-closed classes, Mader [Mad67])*
Every non-trivial minor-closed class of graphs has finite edge density.

Proof. Let $\mathcal{C}$ be any such class, X its excluded minor with the smallest number of vertices x. As $X \preccurlyeq K_x$, the class $\mathcal{C}$ is entirely contained in $\mathcal{C}' = \mathrm{Forb}(K_x)$, so $\varrho(\mathcal{C}) \leq \varrho(\mathcal{C}')$ and therefore it suffices to prove the theorem for classes excluding a single complete graph K_x.

We will show that $\varrho(\mathcal{C}) \leq 2h(x)$, where h is the function from the previous theorem. If any $G \in \mathcal{C}$ had more than $2h(x) \cdot n(G)$ edges, its average degree would be at least $h(x)$, so by the previous theorem G would contain a subdivision of K_x and hence K_x as a minor. ♠

Let us return to the analysis of our algorithm.

3.1.12. Theorem. *(MST on minor-closed classes, Mareš [Mar04])*
For any fixed non-trivial minor-closed class $\mathcal{C}$ of graphs, the Contractive Borůvka's algorithm (1.5.2) finds the MST of any graph of this class in time $\mathcal{O}(n)$. (The

constant hidden in the $\mathcal{O}$ depends on the class.)

Proof. Following the proof for planar graphs (1.5.6), we denote the graph considered by the algorithm at the beginning of the i-th Borůvka step by G_i and its number of vertices and edges by n_i and m_i respectively. Again the i-th phase runs in time $\mathcal{O}(m_i)$ and we have $n_i \leq n/2^i$, so it remains to show a linear bound for the m_i's.

Since each G_i is produced from G_{i-1} by a sequence of edge contractions, all G_i's are minors of the input graph.[1] So they also belong to $\mathcal{C}$ and by the Density theorem $m_i \leq \varrho(\mathcal{C}) \cdot n_i$. The time complexity is therefore $\sum_i \mathcal{O}(m_i) = \sum_i \mathcal{O}(n_i) = \mathcal{O}(\sum_i n/2^i) = \mathcal{O}(n)$. ♠

3.1.13. *Local contractions.* The contractive algorithm uses "batch processing" to perform many contractions in a single step. It is also possible to perform them one edge at a time, batching only the flattenings. A contraction of an edge uv can be done in time $\mathcal{O}(\deg(u))$ by removing all edges incident with u and inserting them back with u replaced by v. Therefore we need to find a lot of vertices with small degrees. The following lemma shows that this is always the case in minor-closed classes.

3.1.14. Lemma. *(Low-degree vertices)*
Let $\mathcal{C}$ be a graph class with density ϱ and $G \in \mathcal{C}$ a graph with n vertices. Then at least $n/2$ vertices of G have degree at most 4ϱ.

Proof. Assume the contrary: Let there be at least $n/2$ vertices with degree greater than 4ϱ. Then $\sum_v \deg(v) > n/2 \cdot 4\varrho = 2\varrho n$, which is in contradiction with the number of edges being at most ϱn. ♠

3.1.15. Remark. The proof can be also viewed probabilistically: let X be the degree of a vertex of G chosen uniformly at random. Then $\mathbb{E}X \leq 2\varrho$, hence by the Markov's inequality $\Pr[X > 4\varrho] < 1/2$, so for at least $n/2$ vertices v we have $\deg(v) \leq 4\varrho$.

3.1.16. Algorithm. *(Local Borůvka's Algorithm, Mareš [Mar04])*

Input: A graph G with an edge comparison oracle and a parameter $t \in \mathbb{N}$.

1. $T \leftarrow \emptyset$.
2. $\ell(e) \leftarrow e$ for all edges e.
3. While $n(G) > 1$:
4. While there exists a vertex v such that $\deg(v) \leq t$:
5. Select the lightest edge e incident with v.
6. Contract e.
7. $T \leftarrow T + \ell(e)$.
8. Flatten G, removing parallel edges and loops.

Output: Minimum spanning tree T.

3.1.17. Theorem. When $\mathcal{C}$ is a minor-closed class of graphs with density ϱ, the Local Borůvka's Algorithm with the parameter t set to 4ϱ finds the MST of any graph from this class in time $\mathcal{O}(n)$. (The constant in the $\mathcal{O}$ depends on the class.)

Proof. Let us denote by G_i, n_i and m_i the graph considered by the algorithm at the beginning of the i-th iteration of the outer loop, and the number of its vertices and

[1] Technically, these are multigraph contractions, but followed by flattening, so they are equivalent to contractions on simple graphs.

edges respectively. As in the proof of the previous algorithm (3.1.12), we observe that all the G_i's are minors of the graph G given as the input.

For the choice $t = 4\varrho$, the Lemma on low-degree vertices (3.1.14) guarantees that at the beginning of the i-th iteration, at least $n_i/2$ vertices have degree at most t. Each selected edge removes one such vertex and possibly increases the degree of another one, so at least $n_i/4$ edges get selected. Hence $n_i \leq 3/4 \cdot n_{i-1}$ and $n_i \leq n \cdot (3/4)^i$, so the algorithm terminates after $\mathcal{O}(\log n)$ iterations.

Each selected edge belongs to $\mathrm{mst}(G)$, because it is the lightest edge of the trivial cut $\delta(v)$ (see the Blue rule, Lemma 1.3.1). The steps 6 and 7 therefore correspond to the operation described by the Contraction Lemma (1.5.10) and when the algorithm stops, T is indeed the minimum spanning tree.

It remains to analyse the time complexity of the algorithm. Since $G_i \in \mathcal{C}$, we know that $m_i \leq \varrho n_i \leq \varrho n/2^i$. We will show that the i-th iteration is carried out in time $\mathcal{O}(m_i)$. Steps 5 and 6 run in time $\mathcal{O}(\deg(v)) = \mathcal{O}(t)$ for each v, so summed over all v's they take $\mathcal{O}(tn_i)$, which is $\mathcal{O}(n_i)$ for a fixed class $\mathcal{C}$. Flattening takes $\mathcal{O}(m_i)$ as already noted in the analysis of the Contracting Borůvka's Algorithm (see 1.5.4).

The whole algorithm therefore runs in time $\mathcal{O}(\sum_i m_i) = \mathcal{O}(\sum_i n/2^i) = \mathcal{O}(n)$. ♠

3.1.18. *Back to planar graphs.* For planar graphs, we can obtain a sharper version of the low-degree lemma showing that the algorithm works with $t = 8$ as well (we had $t = 12$ from $\varrho = 3$). While this does not change the asymptotic time complexity of the algorithm, the constant-factor speedup can still delight the hearts of its practical users.

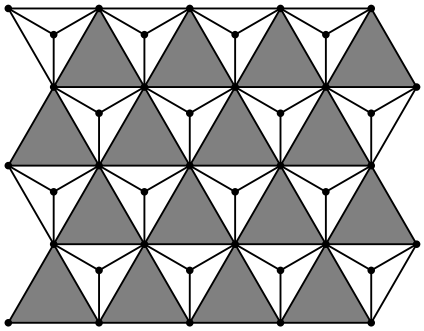
3.1.19. Lemma. *(Low-degree vertices in planar graphs)*
Let G be a planar graph with n vertices. Then at least $n/2$ vertices of v have degree at most 8.

Proof. It suffices to show that the lemma holds for triangulations (if there are any edges missing, the situation can only get better) with at least 4 vertices. Since G is planar, we have $\sum_v \deg(v) < 6n$. The numbers $d(v) := \deg(v) - 3$ are non-negative and $\sum_v d(v) < 3n$, so by the same argument as in the proof of the general lemma, for at least $n/2$ vertices v it holds that $d(v) < 6$, and thus $\deg(v) \leq 8$. ♠

3.1.20. Remark. The constant 8 in the previous lemma is the best we can have. Consider a $k \times k$ triangular grid. It has $n = k^2$ vertices, $\mathcal{O}(k)$ of them lie on the outer face and they have degree at most 6, the remaining $n - \mathcal{O}(k)$ interior vertices have degree exactly 6. Therefore the number of faces f is $6/3 \cdot n = 2n$, ignoring terms of order $\mathcal{O}(k)$. All interior triangles can be properly colored with two colors, black and white. Now add a new vertex inside each white face and connect it to all three vertices on the boundary of that face (see the picture). This adds $f/2 \approx n$ vertices of degree 3 and it increases the degrees of the original $\approx n$ interior vertices to 9, therefore about a half of the vertices of the new planar graph has degree 9.

3.1.21. Remark. The observation in Theorem 3.1.12 was also independently made by Gustedt [Gus98], who studied a parallel version of the Contractive Borůvka's algorithm applied to minor-closed classes.

3.1.22. Remark. The bound on the average degree needed to enforce a K_k minor, which we get from Theorem 3.1.10, is very coarse. Kostochka [Kos84] and inde-



The construction from Remark 3.1.20

pendently Thomason [Tho84] have proven that an average degree $\Omega(k\sqrt{\log k})$ is sufficient and that this is the best what we can get.

3.1.23. Remark. Minor-closed classes share many other interesting properties, for example bounded chromatic numbers of various kinds, as shown by Theorem 6.1 of [NdM03]. We can expect that many algorithmic problems will turn out to be easy for them.

3.2. Iterated algorithms

We have seen that the Jarník's Algorithm 1.4.9 runs in $\Theta(m \log n)$ time. Fredman and Tarjan [FT87] have shown a faster implementation using their Fibonacci heaps. In this section, we will convey their results and we will show several interesting consequences.

The previous implementation of the algorithm used a binary heap to store all edges separating the current tree T from the rest of the graph, i.e., edges of the cut $\delta(T)$. Instead of that, we will remember the vertices adjacent to T and for each such vertex v we will maintain the lightest edge uv such that u lies in T. We will call these edges *active edges* and keep them in a Fibonacci heap, ordered by weight.

When we want to extend T by the lightest edge of $\delta(T)$, it is sufficient to find the lightest active edge uv and add this edge to T together with the new vertex v. Then we have to update the active edges as follows. The edge uv has just ceased to be active. We scan all neighbors w of the vertex v. When w is already in T, no action is needed. If w is outside T and it was not adjacent to T (there is no active edge remembered for it so far), we set the edge vw as active. Otherwise we check the existing active edge for w and replace it by vw if the new edge is lighter.

The following algorithm shows how these operations translate to insertions, decreases and deletions in the heap.

3.2.1. Algorithm. *(Active Edge Jarník; Fredman and Tarjan [FT87])*

Input: A graph G with an edge comparison oracle.

1. $v_0 \leftarrow$ an arbitrary vertex of G.
2. $T \leftarrow$ a tree containing just the vertex v_0.
3. $H \leftarrow$ a Fibonacci heap of active edges stored as pairs (u, v) where $u \in T, v \notin T$, ordered by the weights $w(uv)$, and initially empty.
4. $A \leftarrow$ a mapping of vertices outside T to their active edges in the heap; initially all elements undefined.

5. *Insert* all edges incident with v_0 to H and update A accordingly.
6. While H is not empty:
7. $\quad (u, v) \leftarrow \mathit{DeleteMin}(H)$.
8. $\quad T \leftarrow T + uv$.
9. $\quad$ For all edges vw such that $w \notin T$:
10. $\quad\quad$ If there exists an active edge $A(w)$:
11. $\quad\quad\quad$ If vw is lighter than $A(w)$, *Decrease* $A(w)$ to (v, w) in H.
12. $\quad\quad$ If there is no such edge, then *Insert* (v, w) to H and set $A(w)$.

Output: Minimum spanning tree T.

3.2.2. *Analysis.* To analyse the time complexity of this algorithm, we will use the standard theorem on complexity of the Fibonacci heap:

3.2.3. Theorem. *(Fibonacci heaps, Fredman and Tarjan [FT87])*
The Fibonacci heap performs the following operations with the indicated amortized time complexities:

- *Insert* (insertion of a new element) in $\mathcal{O}(1)$,
- *Decrease* (decreasing the value of an existing element) in $\mathcal{O}(1)$,
- *Merge* (merging of two heaps into one) in $\mathcal{O}(1)$,
- *DeleteMin* (deletion of the minimal element) in $\mathcal{O}(\log n)$,
- *Delete* (deletion of an arbitrary element) in $\mathcal{O}(\log n)$,

where n is the number of elements present in the heap at the time of the operation.

Proof. See Fredman and Tarjan [FT87] for both the description of the Fibonacci heap and the proof of this theorem. ♠

3.2.4. Theorem. Algorithm 3.2.1 with the Fibonacci heap finds the MST of the input graph in time $\mathcal{O}(m + n \log n)$.

Proof. The algorithm always stops, because every edge enters the heap H at most once. As it selects exactly the same edges as the original Jarník's algorithm, it gives the correct answer.

The time complexity is $\mathcal{O}(m)$ plus the cost of the heap operations. The algorithm performs at most one *Insert* or *Decrease* per edge and exactly one *DeleteMin* per vertex. There are at most n elements in the heap at any given time, thus by the previous theorem the operations take $\mathcal{O}(m + n \log n)$ time in total. ♠

3.2.5. Corollary. For graphs with edge density $\Omega(\log n)$, this algorithm runs in linear time.

3.2.6. Remark. *(Other heaps)*
We can consider using other kinds of heaps that have the property that inserts and decreases are faster than deletes. Of course, the Fibonacci heaps are asymptotically optimal (by the standard $\Omega(n \log n)$ lower bound on sorting by comparisons, see for example [LRCS01]), so the other data structures can improve only multiplicative constants or offer an easier implementation.

A nice example is the *d-regular heap* — a variant of the usual binary heap in the form of a complete d-regular tree. *Insert*, *Decrease* and other operations involving bubbling the values up spend $\mathcal{O}(1)$ time at a single level, so they run in $\mathcal{O}(\log_d n)$ time. *Delete* and *DeleteMin* require bubbling down, which incurs

comparison with all d sons at every level, so they spend $\mathcal{O}(d \log_d n)$. With this structure, the time complexity of the whole algorithm is $\mathcal{O}(nd \log_d n + m \log_d n)$, which suggests setting $d = m/n$, yielding $\mathcal{O}(m \log_{m/n} n)$. This is still linear for graphs with density at least $n^{1+\varepsilon}$.

Another possibility is to use the 2-3-heaps [TC99] or Trinomial heaps [Tak00]. Both have the same asymptotic complexity as Fibonacci heaps (the latter even in the worst case, but it does not matter here) and their authors claim faster implementation. For integer weights, we can use Thorup's priority queues described in [Tho04] which have constant-time *Insert* and *Decrease* and $\mathcal{O}(\log \log n)$ time *DeleteMin*. (We will however omit the details since we will show a faster integer algorithm soon.)

3.2.7. *Combining MST algorithms.* As we already noted, the improved Jarník's algorithm runs in linear time for sufficiently dense graphs. In some cases, it is useful to combine it with another MST algorithm, which identifies a part of the MST edges and contracts them to increase the density of the graph. For example, we can perform several Borůvka steps and then find the rest of the MST by the Active Edge Jarník's algorithm.

3.2.8. Algorithm. *(Mixed Borůvka-Jarník)*

Input: A graph G with an edge comparison oracle.

1. Run $\log \log n$ Borůvka steps (1.5.2), getting a MST T_1.
2. Run the Active Edge Jarník's algorithm (3.2.1) on the resulting graph, getting a MST T_2.
3. Combine T_1 and T_2 to T as in the Contraction lemma (1.5.10).

Output: Minimum spanning tree T.

3.2.9. Theorem. The Mixed Borůvka-Jarník algorithm finds the MST of the input graph in time $\mathcal{O}(m \log \log n)$.

Proof. Correctness follows from the Contraction lemma and from the proofs of correctness of the respective algorithms. As for time complexity: The first step takes $\mathcal{O}(m \log \log n)$ time (by Lemma 1.5.4) and it gradually contracts G to a graph G' of size $m' \leq m$ and $n' \leq n/\log n$. The second step then runs in time $\mathcal{O}(m' + n' \log n') = \mathcal{O}(m)$ and both trees can be combined in linear time, too. ♠

3.2.10. *Iterating Jarník's algorithm.* Actually, there is a much better choice of the algorithms to combine: use the Active Edge Jarník's algorithm multiple times, each time stopping it after a while. A good choice of the stopping condition is to place a limit on the size of the heap. We start with an arbitrary vertex, grow the tree as usually and once the heap gets too large, we conserve the current tree and start with a different vertex and an empty heap. When this process runs out of vertices, it has identified a sub-forest of the MST, so we can contract the edges of this forest and iterate.

3.2.11. Algorithm. *(Iterated Jarník; Fredman and Tarjan [FT87])*

Input: A graph G with an edge comparison oracle.

1. $T \leftarrow \emptyset$. *(edges of the MST)*
2. $\ell(e) \leftarrow e$ for all edges e. *(edge labels as usually)*
3. $m_0 \leftarrow m$.

4. While $n > 1$: *(We will call iterations of this loop phases.)*
5. $F \leftarrow \emptyset$. *(forest built in the current phase)*
6. $t \leftarrow 2^{\lceil 2m_0/n \rceil}$. *(the limit on heap size)*
7. While there is a vertex $v_0 \notin F$:
8. Run the Active Edge Jarník's algorithm (3.2.1) from v_0, stop when:
9. all vertices have been processed, or
10. a vertex of F has been added to the tree, or
11. the heap has grown to more than t elements.
12. Denote the resulting tree R.
13. $F \leftarrow F \cup R$.
14. $T \leftarrow T \cup \ell[F]$. *(Remember MST edges found in this phase.)*
15. Contract all edges of F and flatten G.

Output: Minimum spanning tree T.

3.2.12. Notation. For analysis of the algorithm, let us denote the graph entering the i-th phase by G_i and likewise with the other parameters. Let the trees from which F_i has been constructed be called $R_i^1, \ldots, R_i^{z_i}$. The non-indexed G, m and n will correspond to the graph given as input.

3.2.13. However the choice of the parameter t can seem mysterious, the following lemma makes the reason clear:

3.2.14. Lemma. Each phase of the Iterated Jarník's algorithm runs in time $\mathcal{O}(m)$.

Proof. During the i-th phase, the heap always contains at most t_i elements, so it takes time $\mathcal{O}(\log t_i) = \mathcal{O}(m/n_i)$ to delete an element from the heap. The trees R_i^j are edge-disjoint, so there are at most n_i *DeleteMin*'s over the course of the phase. Each edge is considered at most twice (once per its endpoint), so the number of the other heap operations is $\mathcal{O}(m_i)$. Together, it equals $\mathcal{O}(m_i + n_i \log t_i) = \mathcal{O}(m_i + m) = \mathcal{O}(m)$. ♠

3.2.15. Lemma. Unless the i-th phase is final, the forest F_i consists of at most $2m_i/t_i$ trees.

Proof. As every edge of G_i is incident with at most two trees of F_i, it is sufficient to establish that there are at least t_i edges incident with every such tree, including edges connecting two vertices of the same tree.

The forest F_i evolves by additions of the trees R_i^j. Let us consider the possibilities how the algorithm could have stopped growing the tree R_i^j:

- the heap had more than t_i elements (step 10): since the each elements stored in the heap corresponds to a unique edge incident with R_i^j, we have enough such edges;
- the algorithm just added a vertex of F_i to R_i^j (step 9): in this case, an existing tree of F_i is extended, so the number of edges incident with it cannot decrease;[2]
- all vertices have been processed (step 8): this can happen only in the final phase. ♠

[2] This is the place where we needed to count the interior edges as well.

3.2.16. Theorem. The Iterated Jarník's algorithm finds the MST of the input graph in time $\mathcal{O}(m\,\beta(m,n))$, where $\beta(m,n) := \min\{i \mid \log^{(i)} n \leq m/n\}$.

Proof. Phases are finite and in every phase at least one edge is contracted, so the outer loop is eventually terminated. The resulting subgraph T is equal to $\mathrm{mst}(G)$, because each F_i is a subgraph of $\mathrm{mst}(G_i)$ and the F_i's are glued together according to the Contraction lemma (1.5.10).

Let us bound the sizes of the graphs processed in the individual phases. As the vertices of G_{i+1} correspond to the components of F_i, by the previous lemma $n_{i+1} \leq 2m_i/t_i$. Then $t_{i+1} = 2^{\lceil 2m/n_{i+1} \rceil} \geq 2^{2m/n_{i+1}} \geq 2^{2m/(2m_i/t_i)} = 2^{(m/m_i)\cdot t_i} \geq 2^{t_i}$, therefore:

$$t_i \geq 2^{2^{2^{\cdot^{\cdot^{\cdot^{m/n}}}}}} \Bigg\} \text{ a tower of } i \text{ exponentials.}$$

As soon as $t_i \geq n$, the i-th phase is final, because at that time there is enough space in the heap to process the whole graph without stopping. So there are at most $\beta(m,n)$ phases and we already know that each phase runs in linear time (Lemma 3.2.14). ♠

3.2.17. Corollary. The Iterated Jarník's algorithm runs in time $\mathcal{O}(m \log^* n)$.

Proof. $\beta(m,n) \leq \beta(1,n) \leq \log^* n$. ♠

3.2.18. Corollary. When we use the Iterated Jarník's algorithm on graphs with edge density at least $\log^{(k)} n$ for some $k \in \mathbb{N}^+$, it runs in time $\mathcal{O}(km)$.

Proof. If $m/n \geq \log^{(k)} n$, then $\beta(m,n) \leq k$. ♠

3.2.19. *Integer weights.* The algorithm spends most of the time in phases which have small heaps. Once the heap grows to $\Omega(\log^{(k)} n)$ for any fixed k, the graph gets dense enough to guarantee that at most k phases remain. This means that if we are able to construct a heap of size $\Omega(\log^{(k)} n)$ with constant time per operation, we can get a linear-time algorithm for MST. This is the case when the weights are integers:

3.2.20. Theorem. *(MST for integer weights, Fredman and Willard [FW90])*
MST of a graph with integer edge weights can be found in time $\mathcal{O}(m)$ on the Word-RAM.

Proof. We will combine the Iterated Jarník's algorithm with the Q-heaps from Section 2.5. We modify the first pass of the algorithm to choose $t = \log n$ and use the Q-heap tree instead of the Fibonacci heap. From Theorem 2.5.26 and Remark 2.5.29 we know that the operations on the Q-heap tree run in constant time, so the modified first phase takes time $\mathcal{O}(m)$. Following the analysis of the original algorithm in the proof of Theorem 3.2.16 we obtain $t_2 \geq 2^{t_1} = 2^{\log n} = n$, so the algorithm stops after the second phase.[3] ♠

3.2.21. *Further improvements.* Gabow et al. [GGST86] have shown how to speed up the Iterated Jarník's algorithm to $\mathcal{O}(m \log \beta(m,n))$. They split the adjacency lists of the vertices to small buckets, keep each bucket sorted and consider only the lightest edge in each bucket until it is removed. The mechanics of the algorithm is complex and there is a lot of technical details which need careful handling, so

[3] Alternatively, we can use the Q-heaps directly with $k = \log^{1/4} n$ and then the algorithm stops after the third phase.

we omit the description of this algorithm. A better algorithm will be shown in Chapter 4.

3.3. Verification of minimality

Now we will turn our attention to a slightly different problem: given a spanning tree, how to verify that it is minimum? We will show that this can be achieved in linear time and it will serve as a basis for a randomized linear-time MST algorithm in Section 3.5.

MST verification has been studied by Komlós [Kom85], who has proven that $\mathcal{O}(m)$ edge comparisons are sufficient, but his algorithm needed super-linear time to find the edges to compare. Dixon, Rauch and Tarjan [DRT92] have later shown that the overhead can be reduced to linear time on the RAM using preprocessing and table lookup on small subtrees. Later, King has given a simpler algorithm in [Kin97].

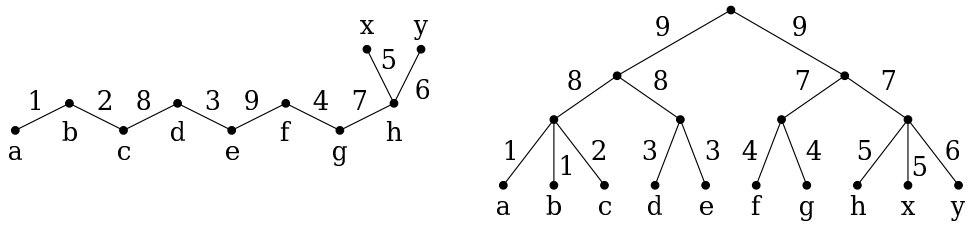
In this section, we will follow Komlós's steps and study the comparisons needed, saving the actual efficient implementation for later.

3.3.1. To verify that a spanning tree T is minimum, it is sufficient to check that all edges outside T are T-heavy (by the Minimality Theorem, 1.2.6). In fact, we will be able to find all T-light edges efficiently. For each edge $uv \in E \setminus T$, we will find the heaviest edge of the tree path $T[u,v]$ and compare its weight to $w(uv)$. It is therefore sufficient to solve the following problem:

3.3.2. Problem. Given a weighted tree T and a set of *query paths* $Q \subseteq \{T[u,v] \mid u,v \in V(T)\}$ specified by their endpoints, find the heaviest edge *(peak)* of every path in Q.

3.3.3. *Borůvka trees.* Finding the peaks can be burdensome if the tree T is degenerated, so we will first reduce it to the same problem on a balanced tree. We run the Borůvka's algorithm on T, which certainly produces T itself, and we record the order, in which the subtrees have been merged, in another tree $B(T)$. The peak queries on T can be then easily translated to peak queries on $B(T)$.

3.3.4. Definition. For a weighted tree T we define its *Borůvka tree* $B(T)$ as a rooted tree which records the execution of the Borůvka's algorithm run on T. The leaves of $B(T)$ are all the vertices of T, an internal vertex v at level i from the bottom corresponds to a component tree $C(v)$ formed in the i-th iteration of the algorithm. When a tree $C(v)$ selects an adjacent edge e and gets merged with some other trees to form a component $C(u)$, we add an edge uv to $B(T)$ and set its weight to $w(e)$.



An octipede and its Borůvka tree

3.3.5. Observation. As the algorithm finishes with a single component in the last phase, the Borůvka tree is really a tree. All its leaves are on the same level and each internal vertex has at least two sons. Such trees will be called *complete branching trees.*

3.3.6. Lemma. For every tree T and every pair of its vertices $x, y \in V(T)$, the peak of the path $T[x,y]$ has the same weight as the peak of the path $B(T)[x,y]$.

Proof. Let us denote the path $T[x,y]$ by P and its heaviest edge by $h = ab$. Similarly, let us use P' for $B(T)[x,y]$ and h' for the heaviest edge of P'.

We will first prove that h has its counterpart of the same weight in P', so $w(h') \geq w(h)$. Consider the lowest vertex u of $B(T)$ such that the component $C(u)$ contains both a and b, and consider the sons v_a and v_b of u for which $a \in C(v_a)$ and $b \in C(v_b)$. As the edge h must have been selected by at least one of these components, we assume without loss of generality that it was $C(v_a)$, and hence we have $w(uv_a) = w(h)$. We will show that the edge uv_a lies in P', because exactly one of the vertices x, y lies in $C(v_a)$. Both cannot lie there, since it would imply that $C(v_a)$, being connected, contains the whole path P, including h. On the other hand, if $C(v_a)$ contained neither x nor y, it would have to be incident with another edge of P different from h, so this lighter edge would be selected instead of h.

In the other direction: for any edge $uv \in P'$, the tree $C(v)$ is incident with at least one edge of P, so the selected edge must be lighter or equal to this edge and hence also to h. ♠

3.3.7. We will simplify the problem even further: For an arbitrary tree T, we split each query path $T[x,y]$ to two half-paths $T[x,a]$ and $T[a,y]$ where a is the *lowest common ancestor* of x and y in T. It is therefore sufficient to consider only paths that connect a vertex with one of its ancestors.

When we combine the two transforms, we get:

3.3.8. Lemma. *(Balancing of trees)*
For each tree T on n vertices and a set Q of q query paths on T, it is possible to find a complete branching tree T', together with a set Q' of paths on T', such that the weights of the heaviest edges of the paths in Q can be deduced from the same of the paths in Q'. The tree T' has at most $2n$ vertices and $\mathcal{O}(\log n)$ levels. The set Q' contains at most $2q$ paths and each of them connects a vertex of T' with one of its ancestors. The construction of T' involves $\mathcal{O}(n)$ comparisons and the transformation of the answers takes $\mathcal{O}(q)$ comparisons.

Proof. The tree T' will be the Borůvka tree for T, obtained by running the contractive version of the Borůvka's algorithm (Algorithm 1.5.2) on T. The algorithm runs in linear time, for example because trees are planar (Theorem 1.5.6). We therefore spend $\mathcal{O}(n)$ comparisons in it.

As T' has n leaves and it is a complete branching tree, it has at most n internal vertices, so $n(T') \leq 2n$ as promised. Since the number of iterations of the Borůvka's algorithm is $\mathcal{O}(\log n)$, the depth of the Borůvka tree must be logarithmic as well.

For each query path $T[x,y]$ we find the lowest common ancestor of x and y and split the path by the two half-paths. This produces a set Q' of at most $2q$ half-paths. The peak of every original query path is then the heavier of the peaks of its halves. ♠

3.3.9. *Bounding comparisons.* We will now describe a simple variant of the depth-first search which finds the peaks of all query paths of the balanced problem. As we promised, we will take care of the number of comparisons only, as long as all other operations are well-defined and they can be performed in polynomial time.

3.3.10. Definition. For every edge $e = uv$, we consider the set Q_e of all query paths containing e. The vertex of a path, that is closer to the root, will be called the *top* of the path, the other vertex its *bottom.* We define arrays T_e and P_e as follows: T_e contains the tops of the paths in Q_e in order of their increasing depth (we will call them *active tops* and each of them will be stored exactly once). For each active top $t = T_e[i]$, we define $P_e[i]$ as the peak of the path $T[v, t]$.

3.3.11. Observation. As for every i the path $T[v, T_e[i+1]]$ is contained within $T[v, T_e[i]]$, the edges of P_e must have non-increasing weights, that is $w(P_e[i+1]) \leq w(P_e[i])$. This leads to the following algorithm:

3.3.12. Algorithm. *FindPeaks*(u, p, T_p, P_p) — process all queries located in the subtree rooted at u entered from its parent via an edge p.

1. Process all query paths whose bottom is u and record their peaks. This is accomplished by finding the index i of each path's top in T_p and reading the desired edge from $P_p[i]$.
2. For every son v of u, process the edge $e = uv$:
3. Construct the array of tops T_e for the edge e: Start with T_p, remove the tops of the paths that do not contain e and add the vertex u itself if there is a query path which has u as its top and whose bottom lies somewhere in the subtree rooted at v.
4. Prepare the array of the peaks P_e: Start with P_p, remove the entries corresponding to the tops that are no longer active. If u became an active top, append e to the array.
5. Finish P_e: Since the paths leading to all active tops have been extended by the edge e, compare $w(e)$ with weights of the edges recorded in P_e and replace those edges which are lighter by e. Since P_p was sorted, we can use binary search to locate the boundary between the lighter and heavier edges in P_e.
6. Recurse on v: call *FindPeaks*(v, e, T_e, P_e).

As we need a parent edge to start the recursion, we add an imaginary parent edge p_0 of the root vertex r, for which no queries are defined. We can therefore start with *FindPeaks*$(r, p_0, \emptyset, \emptyset)$.

Let us account for the comparisons:

3.3.13. Lemma. When the procedure *FindPeaks* is called on the balanced problem, it performs $\mathcal{O}(n+q)$ comparisons, where n is the size of the tree and q is the number of query paths.

Proof. We will calculate the number of comparisons c_i performed when processing the edges going from the $(i+1)$-th to the i-th level of the tree. The levels are numbered from the bottom, so leaves are at level 0 and the root is at level $\ell \leq \lceil \log_2 n \rceil$. There are $n_i \leq n/2^i$ vertices at the i-th level, so we consider exactly n_i

edges. To avoid taking a logarithm[4] of zero, we define $|T_e| = 1$ for $T_e = \emptyset$.

$$\begin{aligned}
c_i &\le \sum_e (1 + \log |T_e|) && \text{(Total cost of the binary searches.)} \\
&\le n_i + \sum_e \log |T_e| && \text{(We sum over } n_i \text{ edges.)} \\
&\le n_i + n_i \cdot \frac{\sum_e \log |T_e|}{n_i} && \text{(Consider the average of logarithms.)} \\
&\le n_i + n_i \cdot \log \frac{\sum_e |T_e|}{n_i} && \text{(Logarithm is concave.)} \\
&\le n_i + n_i \cdot \log \frac{q+n}{n_i} && \text{(Bound the number of tops by queries.)} \\
&= n_i \cdot \left(1 + \log\left(\frac{q+n}{n} \cdot \frac{n}{n_i}\right)\right) \\
&= n_i + n_i \log \frac{q+n}{n} + n_i \log \frac{n}{n_i}.
\end{aligned}$$

Summing over all levels, we estimate the total number of comparisons as:

$$c = \sum_i c_i = \left(\sum_i n_i\right) + \left(\sum_i n_i \log \frac{q+n}{n}\right) + \left(\sum_i n_i \log \frac{n}{n_i}\right).$$

The first part is equal to n, the second one to $n \log((q+n)/n) \le q + n$. For the third one, we would like to plug in the bound $n_i \le n/2^i$, but we unfortunately have one n_i in the denominator. We save the situation by observing that the function $f(x) = x \log(n/x)$ is decreasing[5] for $x > n/e$, so for $i \ge 2$ it holds that:

$$n_i \log \frac{n}{n_i} \le \frac{n}{2^i} \cdot \log \frac{n}{n/2^i} = \frac{n}{2^i} \cdot i.$$

We can therefore rewrite the third part as:

$$\begin{aligned}
\sum_i n_i \log \frac{n}{n_i} &\le n_0 \log \frac{n}{n_0} + n_1 \log \frac{n}{n_1} + n \cdot \sum_{i \ge 2} \frac{i}{2^i} \le \\
&\le n \log 1 + n_1 \cdot \frac{n}{n_1} + n \cdot \mathcal{O}(1) = \mathcal{O}(n).
\end{aligned}$$

Putting all three parts together, we conclude that:

$$c \le n + (q+n) + \mathcal{O}(n) = \mathcal{O}(n+q). \qquad \spadesuit$$

3.3.14. When we combine this lemma with the above reduction from general trees to balanced trees, we get the following theorem:

[4] All logarithms are binary.

[5] We can easily check the derivative: $f(x) = (x \ln n - x \ln x)/\ln 2$, so $f'(x) \cdot \ln 2 = \ln n - \ln x - 1$. We want $f'(x) < 0$ and therefore $\ln x > \ln n - 1$, i.e., $x > n/e$.

3.3.15. Theorem. *(Verification of the MST, Komlós [Kom85])*
For every weighted graph G and its spanning tree T, it is sufficient to perform $\mathcal{O}(m)$ comparisons of edge weights to determine whether T is minimum and to find all T-light edges in G.

Proof. We first transform the problem to finding all peaks of a set of query paths in T (these are exactly the paths covered by the edges of $G \backslash T$). We use the reduction from Lemma 3.3.8 to get an equivalent problem with a full branching tree and a set of parent-descendant paths. The reduction costs $\mathcal{O}(m+n)$ comparisons. Then we run the *FindPeaks* procedure (Algorithm 3.3.12) to find the tops of all query paths. According to Lemma 3.3.13, this spends another $\mathcal{O}(m+n)$ comparisons. Since we (as always) assume that G is connected, $\mathcal{O}(m+n) = \mathcal{O}(m)$. ♠

3.3.16. *Other applications.* The problem of computing path maxima or minima in a weighted tree has several other interesting applications. One of them is computing minimum cuts separating given pairs of vertices in a given weighted undirected graph G. We construct a Gomory-Hu tree T for the graph as described in [GH61] (see also [HKPB07] for a more efficient algorithm running in time $\widetilde{\mathcal{O}}(mn)$ for unit-cost graphs). The crucial property of this tree is that for every two vertices u, v of the graph G, the minimum-cost edge on $T[u,v]$ has the same cost as the minimum cut separating u and v in G. Since the construction of T generally takes $\Omega(n^2)$ time, we could of course invest this time in precomputing the minima for all pairs of vertices. This would however require quadratic space, so we can better use the method of this section which fits in $\mathcal{O}(n+q)$ space for q queries.

3.3.17. *Dynamic verification.* A dynamic version of the problem is also often considered. It calls for a data structure representing a weighted forest with operations for modifying the structure of the forest and querying minima or maxima on paths. Sleator and Tarjan have shown in [ST83] how to do this in $\mathcal{O}(\log n)$ time amortized per operation, which leads to an implementation of the Dinic's maximum flow algorithm [Din70] in time $\mathcal{O}(mn \log n)$.

3.4. Verification in linear time

We have proven that $\mathcal{O}(m)$ edge weight comparisons suffice to verify minimality of a given spanning tree. Now we will show an algorithm for the RAM which finds the required comparisons in linear time. We will follow the idea of King from [Kin97], but as we have the power of the RAM data structures from Section 2.4 at our command, the low-level details will be easier, especially the construction of vertex and edge labels.

3.4.1. First of all, let us make sure that the reduction to fully branching trees in the Balancing lemma (3.3.8) can be made run in linear time. As already noticed in the proof, the Borůvka's algorithm runs in linear time. Constructing the Borůvka tree in the process adds at most a constant overhead to every step of the algorithm.

Finding the common ancestors is not trivial, but Harel and Tarjan have shown in [HT84] that linear time is sufficient on the RAM. Several more accessible algorithms have been developed since then (see the Alstrup's survey paper [AGKR02] and a particularly elegant algorithm described by Bender and Falach-Colton in [BFC00]). Any of them implies the following theorem:

3.4.2. Theorem. *(Lowest common ancestors)*
On the RAM, it is possible to preprocess a tree T in time $\mathcal{O}(n)$ and then answer lowest common ancestor queries presented online in constant time.

3.4.3. Corollary. The reductions in Lemma 3.3.8 can be performed in time $\mathcal{O}(m)$.

3.4.4. Having the balanced problem at hand, it remains to implement the procedure *FindPeaks* of Algorithm 3.3.12 efficiently. We need a compact representation of the arrays T_e and P_e, which will allow to reduce the overhead of the algorithm to time linear in the number of comparisons performed. To achieve this goal, we will encode the arrays in RAM vectors (see Section 2.4 for the vector operations).

3.4.5. Definition.

Vertex identifiers: Since all vertices processed by the procedure lie on the path from the root to the current vertex u, we modify the algorithm to keep a stack of these vertices in an array. We will often refer to each vertex by its index in this array, i.e., by its depth. We will call these identifiers *vertex labels* and we note that each label requires only $\ell = \lceil \log \lceil \log n \rceil \rceil$ bits. As every tree edge is uniquely identified by its bottom vertex, we can use the same encoding for *edge labels.*

Slots: As we are going to need several operations which are not computable in constant time on the RAM, we precompute tables for these operations like we did in the Q-heaps (cf. Lemma 2.5.1). A table for word-sized arguments would take too much time to precompute, so we will generally store our data structures in *slots* of $s = \lceil 1/3 \cdot \log n \rceil$ bits each. We will soon show that it is possible to precompute a table of any reasonable function whose arguments fit in two slots.

Top masks: The array T_e will be represented by a bit mask M_e called the *top mask.* For each of the possible tops t (i.e., the ancestors of the current vertex), we store a single bit telling whether $t \in T_e$. Each top mask fits in $\lceil \log n \rceil$ bits and therefore in a single machine word. If needed, it can be split to three slots. Unions and intersections of sets of tops then translate to AND/OR on the top masks.

Small and big lists: The heaviest edge found so far for each top is stored by the algorithm in the array P_e. Instead of keeping the real array, we store the labels of these edges in a list encoded in a bit string. Depending on the size of the list, we use one of two possible encodings: *Small lists* are stored in a vector that fits in a single slot, with the unused fields filled by a special constant, so that we can easily infer the length of the list.

If the data do not fit in a small list, we use a *big list* instead. It is stored in $\mathcal{O}(\log \log n)$ words, each of them containing a slot-sized vector. Unlike the small lists, we use the big lists as arrays. If a top t of depth d is active, we keep the corresponding entry of P_e in the d-th field of the big list. Otherwise, we keep that entry unused.

We want to perform all operations on small lists in constant time, but we can afford spending time $\mathcal{O}(\log \log n)$ on every big list. This is true because whenever we use a big list, $|T_e| = \Omega(\log n / \log \log n)$, hence we need $\log |T_e| = \Omega(\log \log n)$ comparisons anyway.

Pointers: When we need to construct a small list containing a sub-list of a big list, we do not have enough time to see the whole big list. To handle this, we introduce *pointers* as another kind of edge identifiers. A pointer is an index to the

nearest big list on the path from the small list containing the pointer to the root. As each big list has at most $\lceil \log n \rceil$ fields, the pointer fits in ℓ bits, but we need one extra bit to distinguish between regular labels and pointers.

3.4.6. Lemma. *(Precomputation of tables)*
When f is a function of up to two arguments computable in polynomial time, we can precompute a table of the values of f for all values of arguments that fit in a single slot. The precomputation takes $\mathcal{O}(n)$ time.

Proof. Similar to the proof of Lemma 2.5.1. There are $\mathcal{O}(2^{2s}) = \mathcal{O}(n^{2/3})$ possible values of arguments, so the precomputation takes time $\mathcal{O}(n^{2/3} \cdot \mathrm{poly}(s)) = \mathcal{O}(n^{2/3} \cdot \mathrm{poly}(\log n)) = \mathcal{O}(n)$. ♠

3.4.7. Example. As we can afford spending $\mathcal{O}(n)$ time on preprocessing, we can assume that we can compute the following functions in constant time:

- *Weight*(x) — the Hamming weight of a slot-sized number x (we already considered this operation in Algorithm 2.4.9, but we needed quadratic word size for it). We can easily extend this function to $\log n$-bit numbers by splitting the number in three slots and adding their weights.
- *FindKth*(x, k) — the k-th set bit from the top of the slot-sized number x. Again, this can be extended to multi-slot numbers by calculating the *Weight* of each slot first and then finding the slot containing the k-th **1**.
- *Bits*(m) — for a slot-sized bit mask m, it returns a small list of the positions of the bits set in $\langle m \rangle$.
- *Select*(x, m) — constructs a slot containing the substring of $\langle x \rangle$ selected by the bits set in $\langle m \rangle$.
- *SubList*(x, m) — when x is a small list and m a bit mask, it returns a small list containing the elements of x selected by the bits set in m.

3.4.8. We will now show how to perform all parts of the procedure *FindPeaks* in the required time. We will denote the size of the tree by n and the number of query paths by q.

3.4.9. Lemma. Depths of all vertices and all top masks can be computed in time $\mathcal{O}(n + q)$.

Proof. Run depth-first search on the tree, assign the depth of a vertex when entering it and construct its top mask when leaving it. The top mask can be obtained by OR-ing the masks of its sons, excluding the level of the sons and including the tops of all query paths that have their bottoms at the current vertex (the depths of the tops are already assigned). ♠

3.4.10. Lemma. The arrays T_e and P_e can be indexed in constant time.

Proof. Indexing T_e is exactly the operation *FindKth* applied on the corresponding top mask M_e.

If P_e is stored in a big list, we calculate the index of the particular slot and the position of the field inside the slot. This field can be then extracted using bit masking and shifts.

If it is a small list, we extract the field directly, but we have to dereference it in case it is a pointer. We modify the recursion in *FindPeaks* to pass the depth of the lowest edge endowed with a big list and when we encounter a pointer, we index this big list. ♠

3.4.11. Lemma. For an arbitrary active top t, the corresponding entry of P_e can be extracted in constant time.

Proof. We look up the precomputed depth d of t first. If P_e is stored in a big list, we extract the d-th entry of the list. If the list is small, we find the position of the particular field by counting bits of the top mask M_e at position d and higher (this is *Weight* of M_e with the lower bits masked out). ♠

3.4.12. Lemma. *FindPeaks* processes an edge e in time $\mathcal{O}(\log |T_e| + q_e)$, where q_e is the number of query paths having e as its bottom edge.

Proof. The edge is examined in steps 1, 3, 4 and 5 of the algorithm. We will show how to perform each of these steps in constant time if P_e is a small list or $\mathcal{O}(\log \log n)$ if it is big.

Step 1 looks up q_e tops in P_e and we already know from Lemma 3.4.11 how to do that in constant time per top.

Step 3 is trivial as we have already computed the top masks and we can reconstruct the entries of T_e in constant time according to Lemma 3.4.10.

Step 5 involves binary search on P_e in $\mathcal{O}(\log |T_e|)$ comparisons, each of them indexes P_e, which is $\mathcal{O}(1)$ again by Lemma 3.4.10. Rewriting the lighter edges is $\mathcal{O}(1)$ for small lists by replication and bit masking, for a big list we do the same for each of its slots.

Step 4 is the only non-trivial one. We already know which tops to select (we have the top masks M_e and M_p precomputed), but we have to carefully extract the sublist. We need to handle these four cases:

- *Small from small:* We use $Select(T_e, T_p)$ to find the fields of P_p that shall be deleted by a subsequent call to *SubList*. Pointers can be retained as they still refer to the same ancestor list.
- *Big from big:* We can copy the whole P_p, since the layout of the big lists is fixed and the items, which we do not want, simply end up as unused fields in P_e.
- *Small from big:* We use the operation *Bits* to construct a list of pointers (we use bit masking to add the "this is a pointer" flags).
- *Big from small:* First we have to dereference the pointers in the small list S. For each slot B_i of the ancestor big list, we construct a subvector of S containing only the pointers referring to that slot, adjusted to be relative to the beginning of the slot (we use *Compare* and *Replicate* from Algorithm 2.4.7 and bit masking). Then we use a precomputed table to replace the pointers by the fields of B_i they point to. We OR together the partial results and we again have a small list.

  Finally, we have to spread the fields of this small list to the whole big list. This is similar: for each slot of the big list, we find the part of the small list keeping the fields we want (we call *Weight* on the sub-words of M_e before and after the intended interval of depths) and we use a tabulated function to shift the fields to the right locations in the slot (controlled by the sub-word of M_e in the intended interval). ♠

We now have all the necessary ingredients to prove the following theorem and thus conclude this section:

3.4.13. Theorem. *(Verification of MST on the RAM)*
There is a RAM algorithm which for every weighted graph G and its spanning tree T determines whether T is minimum and finds all T-light edges in G in time $\mathcal{O}(m)$.

Proof. Implement the Komlós's algorithm from Theorem 3.3.15 with the data structures developed in this section. According to Lemma 3.4.12, the algorithm runs in time $\sum_e \mathcal{O}(\log |T_e| + q_e) = \mathcal{O}(\sum_e \log |T_e|) + \mathcal{O}(\sum_e q_e)$. The second sum is $\mathcal{O}(m)$ as there are $\mathcal{O}(1)$ query paths per edge, the first sum is $\mathcal{O}(\#\text{comparisons})$, which is $\mathcal{O}(m)$ by Theorem 3.3.15. ♠

In Section 5.5, we will need a more specialized statement:

3.4.14. Corollary. There is a RAM algorithm which for every weighted tree T and a set P of paths in T calculates the peaks of these paths in time $\mathcal{O}(n(T) + |P|)$.

3.4.15. *Verification on the Pointer Machine.* Buchsbaum et al. [BKRW98] have recently shown that linear-time verification can be achieved even on the Pointer Machine. They first solve the problem of finding the lowest common ancestors for a set of pairs of vertices by batch processing: They combine an algorithm of time complexity $\mathcal{O}(m\,\alpha(m,n))$ based on the Disjoint Set Union data structure with the framework of topological graph computations described in Section 2.2. Then they use a similar technique for finding the peaks themselves.

3.4.16. *Online verification.* The online version of this problem has turned out to be more difficult. It calls for an algorithm that preprocesses the tree and then answers queries for peaks of paths presented online. Pettie [Pet06] has proven an interesting lower bound based on the inverses of the Ackermann's function. If we want to answer queries within t comparisons, we have to invest $\Omega(n \log \lambda_t(n))$ time into preprocessing.[6] This implies that with preprocessing in linear time, the queries require $\Omega(\alpha(n))$ time.

3.5. A randomized algorithm

When we analysed the Contractive Borůvka's algorithm in Section 1.5, we observed that while the number of vertices per iteration decreases exponentially, the number of edges generally does not, so we spend $\Theta(m)$ time on every phase. Karger, Klein and Tarjan [KKT95] have overcome this problem by combining the Borůvka's algorithm with filtering based on random sampling. This leads to a randomized algorithm which runs in linear expected time.

The principle of the filtering is simple: Let us consider any spanning tree T of the input graph G. Each edge of G that is T-heavy is the heaviest edge of some cycle, so by the Red lemma (1.3.6) it cannot participate in the MST of G. We can therefore discard all T-heavy edges and continue with finding the MST on the reduced graph. Of course, not all choices of T are equally good, but it will soon turn out that when we take T as the MST of a randomly selected subgraph, only a small expected number of edges remains.

Selecting a subgraph at random will unavoidably produce disconnected subgraphs at occasion, so we will drop the implicit assumption that all graphs are

[6] $\lambda_t(n)$ is the t-th row inverse of the Ackermann's function, $\alpha(n)$ is its diagonal inverse. See A.3.4 for the exact definitions.

connected for this section and we will always search for the minimum spanning forest. As we already noted (1.6.1), with a little bit of care our algorithms and theorems keep working.

Since we need the MST verification algorithm for finding the T-heavy edges, we will assume that we are working on the RAM.

3.5.1. Lemma. *(Random sampling, Karger [Kar93])*
Let H be a subgraph of G obtained by including each edge independently with probability p. Let further F be the minimum spanning forest of H. Then the expected number of F-nonheavy[7] edges in G is at most n/p.

Proof. Let us observe that we can obtain the forest F by running the Kruskal's algorithm (1.4.21) combined with the random process producing H from G. We sort all edges of G by their weights and we start with an empty forest F. For each edge, we first flip a biased coin (that gives heads with probability p) and if it comes up tails, we discard the edge. Otherwise we perform a single step of the Kruskal's algorithm: We check whether $F + e$ contains a cycle. If it does, we discard e, otherwise we add e to F. At the end, we have produced the subgraph H and its MSF F.

When we exchange the check for cycles with flipping the coin, we get an equivalent algorithm which will turn out to be more convenient to analyse:

1. If $F + e$ contains a cycle, we immediately discard e (we can flip the coin, but we need not to, because the edge will be discarded regardless of the outcome). We note that e is F-heavy with respect to both the current state of F and the final MSF.
2. If $F + e$ is acyclic, we flip the coin:
3. If it comes up heads, we add e to F. In this case, e is neither F-light nor F-heavy.
4. If it comes up tails, we discard e. Such edges are F-light.

The number of F-nonheavy edges is therefore equal to the total number of the coin flips in step 2 of this algorithm. We also know that the algorithm stops before it adds n edges to F. Therefore it flips at most as many coins as a simple random process that repeatedly flips until it gets n heads. As waiting for every occurrence of heads takes expected time $1/p$, waiting for n heads must take n/p. This is the bound we wanted to achieve. ♠

3.5.2. We will formulate the algorithm as a doubly-recursive procedure. It alternatively performs steps of the Borůvka's algorithm and filtering based on the above lemma. The first recursive call computes the MSF of the sampled subgraph, the second one finds the MSF of the original graph, but without the heavy edges.

As in all contractive algorithms, we use edge labels to keep track of the original locations of the edges in the input graph. For the sake of simplicity, we do not mention it in the algorithm explicitly.

3.5.3. Algorithm. *(MSF by random sampling — the KKT algorithm)*

Input: A graph G with an edge comparison oracle.

[7] That is, F-light edges and also edges of F itself.

1. Remove isolated vertices from G. If no vertices remain, stop and return an empty forest.
2. Perform two Borůvka steps (iterations of Algorithm 1.5.2) on G and remember the set B of the edges having been contracted.
3. Select a subgraph $H \subseteq G$ by including each edge independently with probability 1/2.
4. $F \leftarrow \mathrm{msf}(H)$ calculated recursively.
5. Construct $G' \subseteq G$ by removing all F-heavy edges of G.
6. $R \leftarrow \mathrm{msf}(G')$ calculated recursively.
7. Return $R \cup B$.

Output: The minimum spanning forest of G.

3.5.4. Notation. Let us analyse the time complexity of this algorithm by studying properties of its *recursion tree.* This tree describes the subproblems processed by the recursive calls. For any vertex v of the tree, we denote the number of vertices and edges of the corresponding subproblem G_v by n_v and m_v respectively. If $m_v > 0$, the recursion continues: the left son of v corresponds to the call on the sampled subgraph H_v, the right son to the reduced graph G'_v. (Similarly, we use letters subscripted with v for the state of the other variables of the algorithm.) The root of the recursion tree is obviously the original graph G, the leaves are trivial graphs with no edges.

3.5.5. Observation. The Borůvka steps together with the removal of isolated vertices guarantee that the number of vertices drops at least by a factor of four in every recursive call. The size of a subproblem G_v at level i is therefore at most $n/4^i$ and the depth of the tree is at most $\lceil \log_4 n \rceil$. As there are no more than 2^i subproblems at level i, the sum of all n_v's on that level is at most $n/2^i$, which is at most $2n$ when summed over the whole tree.

We are going to show that the worst case of the KKT algorithm is not worse than of the plain contractive algorithm, while the average case is linear.

3.5.6. Lemma. For every subproblem G_v, the KKT algorithm spends $\mathcal{O}(m_v + n_v)$ time plus the cost of the recursive calls.

Proof. We know from Lemma 1.5.4 that each Borůvka step takes time $\mathcal{O}(m_v + n_v)$.[8] The selection of the edges of H_v is straightforward. Finding the F_v-heavy edges is not, but we have already shown in Theorem 3.4.13 that linear time is sufficient on the RAM. ♠

3.5.7. Theorem. *(Worst-case complexity of the KKT algorithm)*
The KKT algorithm runs in time $\mathcal{O}(\min(n^2, m \log n))$ in the worst case on the RAM.

Proof. The argument for the $\mathcal{O}(n^2)$ bound is similar to the analysis of the plain contractive algorithm. As every subproblem G_v is a simple graph, the number of its edges m_v is less than n_v^2. By the previous lemma, we spend time $\mathcal{O}(n_v^2)$ on it. Summing over all subproblems yields $\sum_v \mathcal{O}(n_v^2) = \mathcal{O}((\sum_v n_v)^2) = \mathcal{O}(n^2)$.

In order to prove the $\mathcal{O}(m \log n)$ bound, it is sufficient to show that the total time spent on every level of the recursion tree is $\mathcal{O}(m)$. Suppose that v is a vertex of the recursion tree with its left son ℓ and right son r. Some edges of G_v are removed

[8] We need to add n_v, because the graph could be disconnected.

in the Borůvka steps, let us denote their number by b_v. The remaining edges fall either to $G_\ell = H_v$, or to $G_r = G'_v$, or possibly to both.

We can observe that the intersection $G_\ell \cap G_r$ cannot be large: The edges of H_v that are not in the forest F_v are F_v-heavy, so they do not end up in G_r. Therefore the intersection can contain only the edges of F_v. As there are at most $n_v/4$ such edges, we have $m_\ell + m_r + b_v \leq m_v + n_v/4$.

On the other hand, the first Borůvka step selects at least $n_v/2$ edges, so $b_v \geq n_v/2$. The duplication of edges between G_ℓ and G_r is therefore compensated by the loss of edges by contraction and $m_\ell + m_r \leq m_v$. So the total number of edges per level does not decrease and it remains to apply the previous lemma. ♠

3.5.8. Theorem. *(Expected complexity of the KKT algorithm)*
The expected time complexity of the KKT algorithm on the RAM is $\mathcal{O}(m)$.

Proof. The structure of the recursion tree depends on the random choices taken, but as its worst-case depth is at most $\lceil \log_4 n \rceil$, the tree is always a subtree of the complete binary tree of that depth. We will therefore prove the theorem by examining the complete tree, possibly with empty subproblems in some vertices.

The left edges of the tree (edges connecting a parent with its left son) form a set of *left paths.* Let us consider the expected time spent on a single left path. When walking the path downwards from its top vertex r, the expected size of the subproblems decreases exponentially: for a son ℓ of a vertex v, we have $n_\ell \leq n_v/4$ and $\mathbb{E}m_\ell = \mathbb{E}m_v/2$. The expected total time spend on the path is therefore $\mathcal{O}(n_r + m_r)$ and it remains to sum this over all left paths.

With the exception of the path going from the root of the tree, the top r of a left path is always a right son of a unique parent vertex v. Since the subproblem G_r has been obtained from its parent subproblem G_v by filtering out all heavy edges, we can use the Sampling lemma (3.5.1) to show that $\mathbb{E}m_r \leq 2n_v$. The sum of the expected sizes of all top subproblems is then $\sum_r n_r + m_r \leq \sum_v 3n_v = \mathcal{O}(n)$. After adding the exceptional path from the root, we get $\mathcal{O}(m + n) = \mathcal{O}(m)$. ♠

3.5.9. *High probability.* There is also a high-probability version of the above theorem. According to Karger, Klein and Tarjan [KKT95], the time complexity of the algorithm is $\mathcal{O}(m)$ with probability $1 - \exp(-\Omega(m))$. The proof again follows the recursion tree and it involves applying the Chernoff bound [Che52] to bound the tail probabilities.

3.5.10. *Different sampling.* We could also use a slightly different formulation of the Sampling lemma suggested by Chan [Cha98]. He changes the selection of the subgraph H to choosing an mp-edge subset of $E(G)$ uniformly at random. The proof is then a straightforward application of the backward analysis method. We however preferred the Karger's original version, because generating a random subset of a given size requires an unbounded number of random bits in the worst case.

3.5.11. *On the Pointer Machine.* The only place where we needed the power of the RAM is finding the heavy edges, so we can employ the pointer-machine verification algorithm mentioned in 3.4.15 to bring the results of this section to the PM.

4. Approaching Optimality

4.1. Soft heaps

A vast majority of MST algorithms that we have encountered so far is based on the Tarjan's Blue rule (Lemma 1.3.5). The rule serves to identify edges that belong to the MST, while all other edges are left in the process. This unfortunately means that the later stages of computation spend most of their time on these edges that never enter the MSF. A notable exception is the randomized algorithm of Karger, Klein and Tarjan. It adds an important ingredient: it uses the Red rule (Lemma 1.3.6) to filter out edges that are guaranteed to stay outside the MST, so that the graphs with which the algorithm works get smaller with time.

Recently, Chazelle [Cha00a] and Pettie [Pet99] have presented new deterministic algorithms for the MST which are also based on the combination of both rules. They have reached worst-case time complexity $\mathcal{O}(m\,\alpha(m,n))$ on the Pointer Machine. We will devote this chapter to their results and especially to another algorithm by Pettie and Ramachandran [PR02b] which is provably optimal.

At the very heart of all these algorithms lies the *soft heap* discovered by Chazelle [Cha00b]. It is a meldable priority queue, roughly similar to the Vuillemin's binomial heaps [Vui78] or Fredman's and Tarjan's Fibonacci heaps [FT87]. The soft heaps run faster at the expense of *corrupting* a fraction of the inserted elements by raising their values (the values are however never lowered). This allows for a trade-off between accuracy and speed, controlled by a parameter ε. The heap operations take $\mathcal{O}(\log(1/\varepsilon))$ amortized time and at every moment at most εn elements of the n elements inserted can be corrupted.

4.1.1. Definition. *(Soft heap interface)*
The *soft heap* contains a set of distinct items from a totally ordered universe and it supports the following operations:

- *Create*(ε) — Create an empty soft heap with the given accuracy parameter ε.
- *Insert*(H, x) — Insert a new item x into the heap H.
- *Meld*(P, Q) — Merge two heaps into one, more precisely move all items of a heap Q to the heap P, destroying Q in the process. Both heaps must have the same ε.
- *DeleteMin*(H) — Delete the minimum item of the heap H and return its value (optionally signalling that the value has been corrupted).
- *Explode*(H) — Destroy the heap and return a list of all items contained in it (again optionally marking those corrupted).

For every item, we will distinguish between its *original* value at the time of insertion and its *current* value in the heap, which is possibly corrupted.

4.1.2. Example. *(Linear-time selection)*
We can use soft heaps to select the median (or generally the k-th smallest element) of a sequence. We insert all n elements to a soft heap with error rate $\varepsilon = 1/3$. Then we delete the minimum about $n/3$ times and we remember the current value x of the last element deleted. This x is greater or equal than the current values of the

previously deleted elements and thus also than their original values. On the other hand, the current values of the $2n/3$ elements remaining in the heap are greater than x and at most $n/3$ of them are corrupted. Therefore at least $n/3$ original elements are greater than x and at least $n/3$ ones are smaller.

This allows us to use the x as a pivot in the traditional divide-and-conquer algorithm for selection (cf. Hoare's Quickselect algorithm [Hoa61]): We split the input elements to three parts: A contains those less than x, B those equal to x and C those greater than x. If $k \leq |A|$, the desired element lies in A, so we can continue by finding the k-th smallest element of A. If $|A| < k \leq |A| + |B|$, the desired element is x itself. Otherwise, we search for the $(k - |A| - |B|)$-th smallest element of C. Either way, we have removed at least $n/3$ items, so the total time complexity is $\mathcal{O}(n + (2/3) \cdot n + (2/3)^2 \cdot n + \ldots) = \mathcal{O}(n)$.

We have obtained a nice alternative to the standard linear-time selection algorithm of Blum [BFP+73]. The same trick can be used to select a good pivot in Quicksort [Hoa62], leading to time complexity $\mathcal{O}(n \log n)$ in the worst case.

4.1.3. Definition. *(Soft queues)*
The soft heap is built from *soft queues* (we will usually omit the adjective soft in the rest of this section). Each queue has a shape of a binary tree.[1] Each vertex v of the tree remembers a doubly-linked list of items. The item list in every left son will be used only temporarily and it will be kept empty between operations. Only right sons and the root have their lists permanently occupied. The left sons will be called *white*, the right ones *black.* (See the picture.)

The first value in every list is called the *controlling key* of the vertex, denoted by $ckey(v)$. If the list is empty, we keep the most recently used value or we set $ckey(v) = +\infty$. The *ckey*s obey the standard *heap order* — a *ckey* of a parent is always smaller than the *ckey*s of its sons.

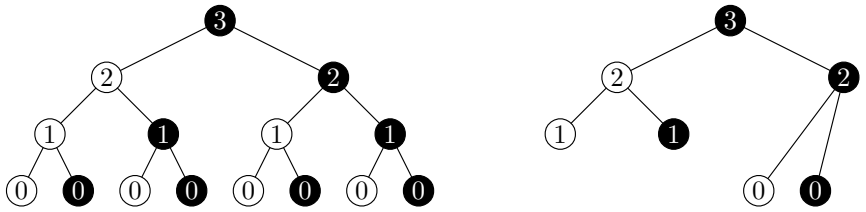
Each vertex is also assigned its *rank,* which is a non-negative integer. The ranks of leaves are always zero, the rank of every internal vertex can be arbitrary, but it must be strictly greater than the ranks of its sons. We define the rank of the whole queue to be equal to the rank of its root vertex and similarly for its *ckey*.

A queue is called *complete* if every two vertices joined by an edge have rank difference exactly one. In other words, it is a complete binary tree and the ranks correspond to vertex heights.

4.1.4. Observation. The complete queue of rank k contains exactly $2^{k+1} - 1$ vertices, 2^k of which are black (by induction). Any other queue can be trivially embedded to the complete queue of the same rank, which we will call the *master tree* of the queue. This embedding preserves vertex ranks, colors and the ancestor relation.

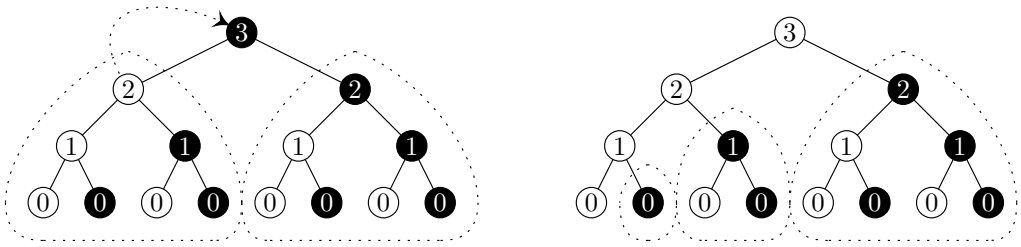
The queues have a nice recursive structure. We can construct a queue of rank k by *joining* two queues of rank $k - 1$ under a new root. The root will inherit the item list of one of the original roots and also its *ckey*. To preserve the heap order, we will choose the son whose *ckey* is smaller.

[1] Actually, Chazelle defines the queues as binomial trees, but he transforms them in ways that are somewhat counter-intuitive, albeit well-defined. We prefer describing the queues as binary trees with a special distribution of values. In fact, the original C code in the Chazelle's paper [Cha00b] uses this representation internally.



A complete and a partial soft queue tree
(black vertices contain items, numbers indicate ranks)

Sometimes, we will also need to split a queue to smaller queues. We will call this operation *dismantling* the queue and it will happen only in cases when the item list in the root is empty. It suffices to remove the leftmost (all white) path going from the root. From a queue of rank k, we get queues of ranks $0, 1, \ldots, k-1$, some of which may be missing if the original queue was not complete.



Joining and dismantling of soft queues

We will now define the real soft heap and the operations on it.

4.1.5. Definition. A *soft heap* consists of:

- a doubly linked list of soft queues of distinct ranks (in increasing order of ranks), we will call the first queue the *head* of the list, the last queue will be its *tail*;
- *suffix minima:* each queue contains a pointer to the queue with minimum *ckey* of those following it in the list;
- a global parameter r: an even integer to be set depending on ε.

We will define the *rank* of a heap as the highest of the ranks of its queues (that is, the rank of the heap's tail).

The heap always keeps the *Rank invariant:* When a root of any tree has rank k, its leftmost path contains at least $k/2$ vertices.

4.1.6. *Operations on soft heaps.*

Melding of two soft heaps involves merging of their lists of queues. We disassemble the heap of the smaller rank and we insert its queues to the other heap. If two queues of the same rank k appear in both lists, we *join* them to a single queue of rank $k+1$ as already described and we propagate the new queue as a *carry* to the next iteration. (This is similar to addition of binary numbers.) Finally, we have to update the suffix minima by walking the new list backwards from the last inserted item.

Creation of a new soft heap is trivial, *insertion* is handled by creating a single-element heap and melding it to the destination heap.

4.1.7. Algorithm. *(Creating a new soft heap)*

Input: The parameter ε (the accuracy of the heap).

1. Allocate memory for a new heap structure H.
2. Initialize the list of queues in H to an empty list.
3. Set the parameter r to $2\lceil\log(1/\varepsilon)\rceil + 2$ (to be justified later).

Output: A newly minted soft heap H.

4.1.8. Algorithm. *(Melding of two soft heaps)*

Input: Two soft heaps P and Q.

1. If $\mathit{rank}(P) < \mathit{rank}(Q)$, exchange the queue lists of P and Q.
2. $p \leftarrow \mathit{head}(P)$.
   (Whenever we run into an end of a list in this procedure, we assume that there is an empty queue of infinite rank there.)
3. While Q still has some queues:
4. $\quad q \leftarrow \mathit{head}(Q)$.
5. $\quad$ If $\mathit{rank}(p) < \mathit{rank}(q)$, then $p \leftarrow$ the successor of p,
6. $\quad$ else if $\mathit{rank}(p) > \mathit{rank}(q)$, remove q from Q and insert it to P before p,
7. $\quad$ otherwise (the ranks are equal, we need to propagate the carry):
8. $\quad\quad \mathit{carry} \leftarrow p$.
9. $\quad\quad$ Remove p from P and set $p \leftarrow$ the original successor of p.
10. $\quad\quad$ While $\mathit{rank}(q) = \mathit{rank}(\mathit{carry})$:
11. $\quad\quad\quad$ Remove q from Q.
12. $\quad\quad\quad \mathit{carry} \leftarrow \mathit{join}(q, \mathit{carry})$.
13. $\quad\quad\quad q \leftarrow \mathit{head}(Q)$.
14. $\quad\quad$ Insert *carry* before q.
15. Update the suffix minima: Walk with p backwards to the head of P:
16. $\quad p' \leftarrow \mathit{suffix\_min}$ of the successor of p.
17. $\quad$ If $\mathit{ckey}(p) < \mathit{ckey}(p')$, set $\mathit{suffix\_min}(p) \leftarrow p$.
18. $\quad$ Otherwise set $\mathit{suffix\_min}(p) \leftarrow p'$.
19. Destroy the heap Q.

Output: The merged heap P.

4.1.9. Algorithm. *(Insertion of an element to a soft heap)*

Input: A heap H and a new element x.

1. Create a new heap H' of the same parameters as H. Let H' contain a sole queue of rank 0, whose only vertex has the element x in its item list.
2. $\mathit{Meld}(H, H')$.

Output: An updated heap H.

4.1.10. *Corruption.* So far, the mechanics of the soft heaps were almost identical to the binomial heaps and the reader could rightfully yawn. The things are going to get interesting now as we approach the parts where corruption of items takes place.

If all item lists contain at most one item equal to the *ckey* of the particular vertex, no information is lost and the heap order guarantees that the minimum item of every queue stays in its root. We can however allow longer lists and let the items stored in a single list travel together between the vertices of the tree, still represented by a common *ckey*. This data-structural analogue of car pooling will allow the items to travel at a faster rate, but as only a single item can be equal to the *ckey*, all other items will be inevitably corrupted.

We of course have to be careful about the size of the lists, because we must avoid corrupting too many items. We will control the growth according to the vertex ranks. Vertices with rank at most r will always contain just a single item. Above this level, the higher is the rank, the longer list will be allowed.

4.1.11. *Deletion of minimum* will be based on this principle. The minimum is easy to locate — we follow the *suffix_min* of the head of the heap to the queue with the minimum *ckey*. There we look inside the item list of the root of the queue. We remove the *last* item from the list (we do not want the *ckey* to change) and we return it as the minimum. (It is not necessarily the real minimum of all items, but always the minimum of their current, possibly corrupted values.)

If the list becomes empty, we *refill* it with items from the lower levels of the same queue. This process can be best described recursively: We ask the left son to refill itself (remember that the left son is always white, so there are currently no items there). If the new *ckey* of the left son is smaller than of the right son, we immediately move the left son's list to its parent. Otherwise, we exchange the sons and move the list from the new left son to the parent. This way we obey the heap order and at the same time we keep the white left son free of items.

Occasionally, we repeat this process once again and we concatenate the resulting lists (we append the latter list to the former, using the smaller of the two *ckey*s). This makes the lists grow longer and we want to do that roughly on every other level of the tree. The exact condition will be that either the rank of the current vertex is odd, or the difference in ranks between this vertex and its right son is at least two.

If refilling of the left son fails because there are no more items in that subtree (we report this by setting the *ckey* to $+\infty$), the current vertex is no longer needed — the items would just pass through it unmodified. We therefore want to remove it. Instead of deleting it directly, we rather make it point to its former grandsons and we remove the (now orphaned) original son. This helps us to ensure that both sons always keep the same rank, which will be useful for the analysis.

When the refilling is over, we update the suffix minima by walking from the current queue to the head of the heap exactly as we did in the *Meld* procedure.

Our only remaining worry is that the Rank invariant can be broken after the refilling. When the leftmost path of the tree becomes too short, we just *dismantle* the tree as already described and we meld the new trees back to the heap. This is easier to handle when the item list at the root vertex is empty. We will therefore move this check before the refilling of the root list. It will turn out that we have enough time to always walk the leftmost path completely, so no explicit counters are needed.

Let us translate these ideas to real (pseudo)code:

4.1.12. Algorithm. *(Deleting the minimum item from a soft heap)*

Input: A soft heap H.

1. Use *suffix_min* of the head queue of H to locate the queue q with the smallest *ckey*.
2. Remove the final element x of the item list in the root of q.
3. If the item list became empty:
4. Count the vertices on the leftmost path of q.
5. If there are less than $rank(q)$ of them:
6. Remove q from the list of queues.
7. Recalculate the suffix minima as in the *Meld* procedure.
8. Dismantle q and create a heap H' holding the resulting trees.
9. Meld them back: $Meld(H, H')$.
10. Otherwise:
11. Call *Refill* on the root of q.
12. If $ckey(q) = +\infty$ (no items left), remove the tree q from H.
13. Recalculate the suffix minima.

Output: The deleted minimum item x (possibly corrupted).

4.1.13. Algorithm. *(Refilling the item list of a vertex)*

Input: A soft queue and its vertex v with an empty item list.

1. Handle trivial cases: If v has no children or both have $ckey = +\infty$, set $ckey(v)$ to $+\infty$ and return.
2. Let *left* and *right* denote the respective sons of v.
3. Recurse: call $Refill(left)$.
4. If $ckey(left) > ckey(right)$, swap the sons.
5. Move the item list from *left* to v (implying $ckey(v) = ckey(left)$).
6. If $rank(v) > r$ and either $rank(v)$ is odd or $rank(v) > rank(right)+1$, recurse once more:
7. Repeat steps 3–4.
8. Append the item list from *left* to the item list at v.
9. Clean up. If $ckey(right) = +\infty$:
10. If $ckey(left) = +\infty$, unlink and discard both sons.
11. Otherwise relink the sons of *left* to v and discard *left*.

Output: A modified soft queue.

4.1.14. *Explode* is trivial once we know how to recognize the corrupted items. It simply examines all queues in the heap, walks the trees and the item lists of all vertices. It records all items seen, the corrupted ones are those that different from their *ckey*.

4.1.15. *Analysis of accuracy.* The description of the operations is now complete, so let us analyse their behavior and verify that we have delivered what we promised — first the accuracy of the structure, then the time complexity of operations. In the whole analysis, we will denote the total number of elements inserted during the

history of the structure by n. We will also frequently take advantage of knowing that the threshold r is even.

We start by bounding the sizes of the item lists.

4.1.16. Lemma. For every vertex v of a soft queue, the size $\ell(v)$ of its item list satisfies:

$$\ell(v) \leq \max(1, 2^{\lceil rank(v)/2 \rceil - r/2}).$$

Proof. Initially, all item lists contain at most one item, so the inequality trivially holds. Let us continue by induction. Melds can affect the inequality only in the favorable direction (they occasionally move an item list to a vertex of a higher rank) and so do deletes (they only remove items from lists). The only potentially dangerous place is the *Refill* procedure.

Refilling sometimes just moves items upwards, which is safe, and sometimes it joins two lists into one, which generally is not. When $rank(v) \leq r$, no joining takes place and $\ell(v)$ is still 1. Otherwise we join when either $rank(v)$ is odd or $rank(w) < rank(v) - 1$ for any son w of v (remember that both sons have the same rank). In both cases, $\lceil rank(w)/2 \rceil \leq \lceil rank(v)/2 \rceil - 1$. By the induction hypothesis, the size of each of the two lists being joined is at most $2^{\max(1, \lceil rank(v)/2 \rceil - 1 - r/2)}$, so the new list has at most $2^{\lceil rank(v)/2 \rceil - r/2}$ items. (The maximum has disappeared since $rank(v) > r$ and therefore the desired bound is at least 2.) ♠

We will now sum the sizes of the lists over all vertices containing corrupted items.

4.1.17. Lemma. At any given time, the heap contains at most $n/2^{r-2}$ corrupted items.

Proof. We first prove an auxiliary claim: The master trees of all queues contain at most n black vertices. This follows by induction: If no deletions have taken place, there are exactly n black vertices, because insertion adds one black vertex and melding preserves their number. A deletion affects the master trees only when dismantling takes place and then it only removes a black vertex.

An obvious upper bound on the number of corrupted items is the total size of item lists in all vertices of rank greater than r. We already know from the previous lemma that the list sizes are limited by a function of the ranks. A complete tree is obviously the worst case, so we will prove that this lemma holds for the master tree of every queue in the heap. The actual trees can be much sparser, but the above claim guarantees that the total size of the master trees is bounded by the number of insertions properly.

So let us consider a complete tree of rank k. It has exactly 2^{k-i} vertices of rank i and each such vertex contains a list of at most $2^{\lceil i/2 \rceil - r/2}$ items by the previous lemma. Summing over all ranks greater than r, we get that the total number of corrupted items in this tree is at most:

$$\sum_{i=r+1}^{k} 2^{k-i} \cdot 2^{\lceil i/2 \rceil - r/2} = 2^{k-r/2} \cdot \sum_{i=r+1}^{k} 2^{\lceil i/2 \rceil - i} \leq 2^{k-r/2+1/2} \cdot \sum_{i=r+1}^{k} 2^{-i/2} \leq 2^{k-r} \cdot \sum_{i=0}^{\infty} 2^{-i/2}.$$

The sum of a geometric series with quotient $2^{-1/2}$ is less than four, so the last expression is less than 2^{k-r+2}. Since the tree contains $n_k = 2^k$ black vertices, this makes less than $n_k/2^{r-2}$ corrupted items as we asserted. ♠

4.1.18. *Analysis of time complexity.* Now we will examine the amortized time complexity of the individual operations. We will show that if we charge $\mathcal{O}(r)$ time against every element inserted, it is enough to cover the cost of all other operations.

All heap operations use only pointer operations, so it will be easy to derive the time bound in the Pointer Machine model. The notable exception is however that the procedures often refer to the ranks, which are integers on the order of $\log n$, so they cannot fit in a single memory cell. For the time being, we will assume that the ranks can be manipulated in constant time, postponing the proof for later.

We take a look at the melds first.

4.1.19. Lemma. The amortized cost of a meld is $\mathcal{O}(1)$, except for melds induced by dismantling which take $\mathcal{O}(\mathit{rank}(q))$, where q is the queue to be dismantled.

Proof. The real cost of a meld of heaps P and Q is linear in the smaller of their ranks, plus the time spent on carry propagation. The latter is easy to dispose of: Every time there is a carry, the total number of trees in all heaps decreases by one. So it suffices to charge $\mathcal{O}(1)$ against creation of a tree. An insert creates one tree, dismantling creates at most $\mathit{rank}(q)$ trees, and all other operations alter only the internal structure of trees.

As for the $\mathcal{O}(\min(\mathit{rank}(P), \mathit{rank}(Q)))$ part, let us assume for a while that no dismantling ever takes place and consider the *meld forest*. It is a forest whose leaves correspond to the n single-element heaps constructed by *Insert* and each internal vertex represents a heap arisen from melding its sons. The left son will be the one with the greater or equal rank. We therefore want to bound the sum of ranks of all right sons.

For every right son, we will distribute the charge for its rank k among all leaves in its subtree. There are at least 2^k such leaves. No leaf ever receives the same rank twice, because the ranks of right sons on every path from the root of the tree to a leaf are strictly decreasing. (This holds because melding of two heaps always produces a heap of a rank strictly greater than the smaller of the input ranks.) Hence at most $n/2^k$ right sons have rank k and the total time charged against the leaves is bounded by:

$$\sum_{k=0}^{\text{max.rank}} k \cdot \frac{n}{2^k} \leq n \cdot \sum_{k=0}^{\infty} \frac{k}{2^k} = 2n.$$

Let us return dismantling to the game. When a queue is dismantled, melding the parts back to the heap takes $\mathcal{O}(\mathit{rank}(q))$ time. We can therefore let the dismantling pay for it and omit such induced melds from the meld forest. As the rank of a heap is never increased by induced melds, the above calculation is still a proper upper bound on the cost of the regular melds. ♠

Before we estimate the time spent on deletions, we analyse the refills.

4.1.20. Lemma. Every invocation of *Refill* takes time $\mathcal{O}(1)$ amortized.

Proof. When *Refill* is called from the *DeleteMin* operation, it recurses on a subtree of the queue. This subtree can be split to the "lossless" lower part (rank r and below) and the upper part where list concatenation and thus also corruption takes place. Whenever we visit the lower part during the recursion, we spend at worst

$\mathcal{O}(r)$ time there. We will prove that the total time spent in the upper parts during the whole life of the data structure is $\mathcal{O}(n)$. Since each upper vertex can perform at most two calls to the lower part, the total time spent in the lower parts is $\mathcal{O}(rn)$. All this can be prepaid by the inserts.

Let us focus on the upper part. There are three possibilities of what can happen when we visit a vertex:

- We delete it: Every vertex deleted has to have been created at some time in the past. New vertices are created only during inserts and melds (when joining two trees) and we have already shown that these operations have constant amortized complexity. Then the same must hold for deletions.
- We recurse twice and concatenate the lists: The lists are disassembled only when they reach the root of the tree, otherwise they are only concatenated. We can easily model the situation by a binary tree forest similar to the meld forest. There are n leaves and every internal vertex has outdegree two, so the total number of concatenations is at most n. Each of them can be performed in constant time as the list is doubly linked.
- We recurse only once: This occurs only if the rank is even and the gap between the rank of this vertex and its sons is equal to 1. It therefore cannot happen twice in a row, thus it is clearly dominated by the cost of the other possibilities.

The total cost of all steps in the upper part is therefore $\mathcal{O}(n)$. ♠

We now proceed with examining the *DeleteMin* operation.

4.1.21. Lemma. Every *DeleteMin* takes $\mathcal{O}(1)$ time amortized.

Proof. Aside from refilling, which is $\mathcal{O}(1)$ by the previous lemma, the *DeleteMin* takes care of the Rank invariant. This happens by checking the length of the leftmost path and dismantling the tree if the length is too far from the tree's rank k. When the invariant is satisfied, the leftmost path is visited by the subsequent call to *Refill*, so we can account the check on the refilling.

When we are dismantling, we have to pay $\mathcal{O}(k)$ for the operation itself and another $\mathcal{O}(k)$ for melding the trees back to the heap. Since we have produced at most $k/2$ subtrees of distinct ranks, some subtree of rank $k/2$ or more must be missing. Its master tree contained at least $2^{k/2}$ vertices which are now permanently gone from the data structure, so we can charge the cost against them. A single vertex can participate in the master trees of several dismantlings, but their ranks are always strictly increasing. By the same argument as in the proof of Lemma 4.1.19 (complexity of *Meld*), each vertex pays $\mathcal{O}(1)$.

We must not forget that *DeleteMin* also has to recalculate the suffix minima. In the worst case, it requires touching k trees. Because of the Rank invariant, this is linear in the size of the leftmost path and therefore it can be also paid for by *Refill*. (Incidentally, this was the only place where we needed the invariant.) ♠

*Explode*s are easy not only to implement, but also to analyse:

4.1.22. Lemma. Every *Explode* takes $\mathcal{O}(1)$ time amortized.

Proof. As all queues, vertices and items examined by *Explode* are forever gone from the heap, we can charge the constant time spent on each of them against the operations that have created them. ♠

It remains to take care of the calculation with ranks:

4.1.23. Lemma. Every manipulation with ranks performed by the soft heap operations can be implemented on the Pointer Machine in constant amortized time.

Proof. We will recycle the idea of "yardsticks" from Section 2.2. We create a yardstick — a doubly linked list whose elements represent the possible values of a rank. Every vertex of a queue will store its rank as a pointer to the corresponding "tick" of the yardstick. We will extend the list whenever necessary.

Comparison of two ranks for equality is then trivial, as is incrementing or decrementing the rank by 1. Testing whether a rank is odd can be handled by storing an odd/even flag in every tick. This covers all uses of ranks except for the comparisons for inequality when melding. In step 1 of *Meld*, we just mark the ticks of the two ranks and walk the yardstick from the beginning until we come across a mark. Thus we compare the ranks in time proportional to the smaller of them, which is the real cost of the meld anyway. The comparisons in steps 5 and 6 are trickier, but since the ranks of the elements put to P are strictly increasing, we can start walking the list at the rank of the previous element in P. The cost is then the difference between the current and the previous rank and the sum of these differences telescopes, again to the real cost of the meld. ♠

We can put the bits together now and laurel our effort with the following theorem:

4.1.24. Theorem. *(Performance of soft heaps, Chazelle [Cha00b])*
A soft heap with error rate ε ($0 < \varepsilon \leq 1/2$) processes a sequence of operations starting with an empty heap and containing n *Insert*s in time $\mathcal{O}(n \log(1/\varepsilon))$ on the Pointer Machine. At every moment, the heap contains at most εn corrupted items.

Proof. We set the parameter r to $2+2\lceil\log(1/\varepsilon)\rceil$. The rest follows from the analysis above. By Lemma 4.1.17, there are always at most $n/2^{r-2} \leq \varepsilon n$ corrupted items in the heap. By Lemma 4.1.19–4.1.23, the time spent on all operations in the sequence can be paid for by charging $\mathcal{O}(r)$ time against each *Insert*. This yields the time bound. ♠

4.1.25. Remark. When we set $\varepsilon = 1/2n$, the soft heap is not allowed to corrupt any items, so it can be used like any traditional heap. By the standard lower bound on sorting it therefore requires $\Omega(\log n)$ time per operation, so the time complexity is optimal for this choice of ε. Chazelle [Cha00b] proves that it is optimal for every choice of ε.

The space consumed by the heap need not be linear in the *current* number of items, but if a case where this matters ever occurred, we could fix it easily by rebuilding the whole data structure completely after $n/2$ deletes. This increases the number of potentially corrupted items, but at worst twice, so it suffices to decrease ε twice.

4.2. Robust contractions

Having the soft heaps at hand, we would like to use them in a conventional MST algorithm in place of a normal heap. The most efficient specimen of a heap-based algorithm we have seen so far is the Iterated Jarník's algorithm (3.2.11). It is based on a simple, yet powerful idea: Run the Jarník's algorithm with limited heap size, so that it stops when the neighborhood of the tree becomes too large. Grow multiple such trees, always starting in a vertex not visited yet. All these trees are contained in the MST, so by the Contraction lemma (1.5.10) we can contract each of them to a single vertex and iterate the algorithm on the resulting graph.

We can try implanting the soft heap in this algorithm, preferably in the earlier version without active edges (1.4.9) as the soft heap lacks the *Decrease* operation. This brave, but somewhat simple-minded attempt is however doomed to fail. The reason is of course the corruption of items inside the heap, which leads to increase of weights of some subset of edges. In presence of corrupted edges, most of the theory we have so carefully built breaks down. For example, the Blue lemma (1.3.5) now holds only when we consider a cut with no corrupted edges, with a possible exception of the lightest edge of the cut. Similarly, the Red lemma (1.3.6) is true only if the heaviest edge on the cycle is not corrupted.

There is fortunately some light in this darkness. While the basic structural properties of MST's no longer hold, there is a weaker form of the Contraction lemma that takes the corrupted edges into account. Before we prove this lemma, we expand our awareness of subgraphs which can be contracted.

4.2.1. Definition. A subgraph $C \subseteq G$ is *contractible* iff for every pair of edges $e, f \in \delta(C)$[2] there exists a path in C connecting the endpoints of the edges e, f such that all edges on this path are lighter than either e or f.

4.2.2. Example. Let us see that when we stop the Jarník's algorithm at some moment and we take a subgraph C induced by the constructed tree, this subgraph is contractible. Indeed, when we consider any two distinct edges uv, xy having exactly one endpoint in C (w.l.o.g. $u, x \in C$ and $v, y \notin C$), they enter the algorithm's heap at some time. Without loss of generality uv enters it earlier. Before the algorithm reaches the vertex x, it adds the whole path ux to the tree. As the edge uv never leaves the heap, all edges on the path ux must be lighter than this edge.

We can now easily reformulate the Contraction lemma (1.5.10) in the language of contractible subgraphs. We again assume that we are working with multigraphs and that they need not be connected. Furthermore, we slightly abuse the notation in the way that we omit the explicit bijection between $G - C$ and G/C, so we can write $G = C \cup (G/C)$.

4.2.3. Lemma. *(Generalized contraction)*
When $C \subseteq G$ is a contractible subgraph, then $\mathrm{msf}(G) = \mathrm{msf}(C) \cup \mathrm{msf}(G/C)$.

Proof. As both sides of the equality are forests spanning the same graph, it suffices to show that $\mathrm{msf}(G) \subseteq \mathrm{msf}(C) \cup \mathrm{msf}(G/C)$. Let us show that edges of G that do not belong to the right-hand side do not belong to the left-hand side either. We know that the edges that do not participate in the MSF of some graph are exactly

[2] That is, of G's edges with exactly one endpoint in C.

those which are the heaviest on some cycle (this is the Cycle rule from Lemma 1.3.6).

Whenever an edge g lies in C, but not in $\mathrm{msf}(C)$, then g is the heaviest edge on some cycle in C. As this cycle is also contained in G, the edge g does not participate in $\mathrm{msf}(G)$ either.

Similarly for $g \in (G/C) \setminus \mathrm{msf}(G/C)$: when the cycle does not contain the vertex c to which we have contracted the subgraph C, this cycle is present in G, too. Otherwise we consider the edges e, f incident with c on this cycle. Since C is contractible, there must exist a path P in C connecting the endpoints of e and f in G, such that all edges of P are lighter than either e or f and hence also than g. Expanding c in the cycle to the path P then produces a cycle in G whose heaviest edge is g. ♠

We are now ready to bring corruption back to the game and state a "robust" version of this lemma. A notation for corrupted graphs will be handy:

4.2.4. Notation. When G is a weighted graph and R a subset of its edges, we will use $G \Uparrow R$ to denote an arbitrary graph obtained from G by increasing the weights of some of the edges in R. As usually, we will assume that all edges of this graph have pairwise distinct weights. While this is technically not true for the corruption caused by soft heaps, we can easily make it so.

Whenever C is a subgraph of G, we will use R^C to refer to the edges of R with exactly one endpoint in C (i.e., $R^C = R \cap \delta(C)$).

4.2.5. Lemma. *(Robust contraction, Chazelle [Cha97])*
Let G be a weighted graph and C its subgraph contractible in $G \Uparrow R$ for some set R of edges. Then $\mathrm{msf}(G) \subseteq \mathrm{msf}(C) \cup \mathrm{msf}((G/C) \setminus R^C) \cup R^C$.

Proof. We will modify the proof of the previous lemma. We will again consider all possible types of edges that do not belong to the right-hand side and we will show that they are the heaviest edges of certain cycles. Every edge g of G lies either in C, or in $H = (G/C) \setminus R^C$, or possibly in R^C.

If $g \in C \setminus \mathrm{msf}(C)$, then the same argument as before applies.

If $g \in H \setminus \mathrm{msf}(H)$, we consider the cycle in H on which g is the heaviest. When c (the vertex to which we have contracted C) is outside this cycle, we are done. Otherwise we observe that the edges e, f adjacent to c on this cycle cannot be corrupted (they would be in R^C otherwise, which is impossible). By contractibility of C there exists a path P in $C \Uparrow (R \cap C)$ such that all edges of P are lighter than e or f and hence also than g. The weights of the edges of P in the original graph G cannot be higher than in $G \Uparrow R$, so the path P is lighter than g even in G and we can again perform the trick with expanding the vertex c to P in the cycle C. Hence $g \notin \mathrm{mst}(G)$. ♠

4.2.6. We still intend to mimic the Iterative Jarník's algorithm. We will partition the given graph to a collection $\mathcal{C}$ of non-overlapping contractible subgraphs called *clusters* and we put aside all edges that got corrupted in the process. We recursively compute the MSF of those subgraphs and of the contracted graph. Then we take the union of these MSF's and add the corrupted edges. According to the previous lemma, this does not produce the MSF of G, but a sparser graph containing it, on which we can continue.

We can formulate the exact partitioning algorithm and its properties as follows:

4.2.7. Algorithm. *(Partition a graph to a collection of contractible clusters)*

Input: A graph G with an edge comparison oracle, a parameter t controlling the size of the clusters, and an accuracy parameter ε.

1. Mark all vertices as "live".
2. $\mathcal{C} \leftarrow \emptyset$, $R^{\mathcal{C}} \leftarrow \emptyset$. *(Start with an empty collection and no corrupted edges.)*
3. While there is a live vertex v_0:
4. $\quad T = \{v_0\}$. *(the tree that we currently grow)*
5. $\quad K = \emptyset$. *(edges known to be corrupted in the current iteration)*
6. $\quad$ *Create* a soft heap with accuracy ε and *Insert* the edges adjacent to v_0 into it.
7. $\quad$ While the heap is not empty and $|T| \leq t$:
8. $\quad\quad$ *DeleteMin* an edge uv from the heap, assume $u \in T$.
9. $\quad\quad$ If uv was corrupted, add it to K.
10. $\quad\quad$ If $v \in T$, drop the edge and repeat the previous two steps.
11. $\quad\quad$ $T \leftarrow T \cup \{v\}$.
12. $\quad\quad$ If v is dead, break out of the current loop.
13. $\quad\quad$ Insert all edges incident with v to the heap.
14. $\quad$ $\mathcal{C} \leftarrow \mathcal{C} \cup \{G[T]\}$. *(Record the cluster induced by the tree.)*
15. $\quad$ *Explode* the heap and add all remaining corrupted edges to K.
16. $\quad$ $R^{\mathcal{C}} \leftarrow R^{\mathcal{C}} \cup K^T$. *(Record the "interesting" corrupted edges.)*
17. $\quad$ $G \leftarrow G \setminus K^T$. *(Remove the corrupted edges from G.)*
18. $\quad$ Mark all vertices of T as "dead".

Output: The collection $\mathcal{C}$ of contractible clusters and the set $R^{\mathcal{C}}$ of corrupted edges in the neighborhood of these clusters.

4.2.8. Theorem. *(Partitioning to contractible clusters, Chazelle [Cha97])* Given a weighted graph G and parameters ε ($0 < \varepsilon \leq 1/2$) and t, the Partition algorithm (4.2.7) constructs a collection $\mathcal{C} = \{C_1, \ldots, C_k\}$ of clusters and a set $R^{\mathcal{C}}$ of edges such that:

1. All the clusters and the set $R^{\mathcal{C}}$ are mutually edge-disjoint.
2. Each cluster contains at most t vertices.
3. Each vertex of G is contained in at least one cluster.
4. The connected components of the union of all clusters have at least t vertices each, except perhaps for those which are equal to a connected component of $G \setminus R^{\mathcal{C}}$.
5. $|R^{\mathcal{C}}| \leq 2\varepsilon m$.
6. $\operatorname{msf}(G) \subseteq \bigcup_i \operatorname{msf}(C_i) \cup \operatorname{msf}\Big((G/\bigcup_i C_i) \setminus R^{\mathcal{C}}\Big) \cup R^{\mathcal{C}}$.
7. The algorithm runs in time $\mathcal{O}(n + m \log(1/\varepsilon))$.

Proof. Claim 1: The Partition algorithm grows a series of trees which induce the clusters C_i in G. A tree is built from edges adjacent to live vertices and once it is finished, all vertices of the tree die, so no edge is ever reused in another tree. The edges that enter $R^{\mathcal{C}}$ are immediately deleted from the graph, so they never participate in any tree.

Claim 2: The algorithm stops when all vertices are dead, so each vertex must have entered some tree.

Claim 3: The trees have at most t vertices each, which limits the size of the C_i's as well.

Claim 4: We can show that each connected component has t or more vertices as we already did in the proof of Lemma 3.2.15: How can a new tree stop growing? Either it acquires t vertices, or it joins one of the existing trees (this only increases the size of the component), or the heap becomes empty (which means that the tree spans a full component of the current graph and hence also of the final $G \setminus R^{\mathcal{C}}$).

Claim 5: The set $R^{\mathcal{C}}$ contains a subset of edges corrupted by the soft heaps over the course of the algorithm. As every edge is inserted to a heap at most once per its endpoint, the heaps can corrupt at worst $2\varepsilon m$ edges altogether.

We will prove the remaining two claims as separate lemmata. ♠

4.2.9. Lemma. *(Correctness of Partition, Claim 6 of Theorem 4.2.8)*

$$\operatorname{msf}(G) \subseteq \bigcup_i \operatorname{msf}(C_i) \cup \operatorname{msf}\left(\left(G/\bigcup_i C_i\right) \setminus R^{\mathcal{C}}\right) \cup R^{\mathcal{C}}.$$

Proof. By iterating the Robust contraction lemma (4.2.5). Let K_i be the set of edges corrupted when constructing the cluster C_i in the i-th phase of the algorithm, and similarly for the state of the other variables at that time. We will use induction on i to prove that the lemma holds at the end of every phase.

At the beginning, the statement is obviously true, even as an equality. In the i-th phase we construct the cluster C_i by running the partial Jarník's algorithm on the graph $G_i = G \setminus \bigcup_{j<i} K_j^{C_j}$. If it were not for corruption, the cluster C_i would be contractible, as we already know from Example 4.2.2. When the edges in K_i get corrupted, the cluster is contractible in some corrupted graph $G_i \Uparrow K_i$. (We have to be careful as the edges are becoming corrupted gradually, but it is easy to check that it can only improve the situation.) Since C_i shares no edges with C_j for any $j < i$, we know that C_i also a contractible subgraph of $(G_i/\bigcup_{j<i} C_j) \Uparrow K_i$. Now we can use the Robust contraction lemma to show that:

$$\operatorname{msf}\left(\left(G/\bigcup_{j<i} C_j\right) \setminus \bigcup_{j<i} K_j^{C_j}\right) \subseteq \operatorname{msf}(C_i) \cup \operatorname{msf}\left(\left(G/\bigcup_{j\le i} C_j\right) \setminus \bigcup_{j\le i} K_j^{C_j}\right) \cup K_i^{C_i}.$$

This completes the induction step, because $K_j^{C_j} = K_j^{\mathcal{C}_j}$ for all j. ♠

4.2.10. Lemma. *(Efficiency of Partition, Claim 7 of Theorem 4.2.8)*
The Partition algorithm runs in time $\mathcal{O}(n + m \log(1/\varepsilon))$.

Proof. The inner loop (steps 7–13) processes the edges of the current cluster C_i and also the edges in its neighborhood $\delta(C_i)$. Steps 6 and 13 insert every such edge to the heap at most once, so steps 8–12 visit each edge also at most once. For every edge, we spend $\mathcal{O}(\log(1/\varepsilon))$ time amortized on inserting it and $\mathcal{O}(1)$ on the other operations (by Theorem 4.1.24 on performance of the soft heaps).

Each edge of G can participate in some $C_i \cup \delta(C_i)$ only twice before both its endpoints die. The inner loop therefore processes $\mathcal{O}(m)$ edges total, so it takes

$\mathcal{O}(m \log(1/\varepsilon))$ time. The remaining parts of the algorithm spend $\mathcal{O}(m)$ time on operations with clusters and corrupted edges and additionally $\mathcal{O}(n)$ on identifying the live vertices. ♠

4.3. Decision trees

The Pettie's and Ramachandran's algorithm combines the idea of robust partitioning with optimal decision trees constructed by brute force for very small subgraphs. In this section, we will explain the basics of the decision trees and prove several lemmata which will turn out to be useful for the analysis of time complexity of the final algorithm.

Let us consider all computations of some comparison-based MST algorithm when we run it on a fixed graph G with all possible permutations of edge weights. The computations can be described by a binary tree. The root of the tree corresponds to the first comparison performed by the algorithm and depending to its result, the computation continues either in the left subtree or in the right subtree. There it encounters another comparison and so on, until it arrives to a leaf of the tree where the spanning tree found by the algorithm is recorded.

Formally, the decision trees are defined as follows:

4.3.1. Definition. *(Decision trees and their complexity)*
A *MSF decision tree* for a graph G is a binary tree. Its internal vertices are labeled with pairs of G's edges to be compared, each of the two outgoing tree edges corresponds to one possible result of the comparison.[3] Leaves of the tree are labeled with spanning trees of the graph G.

A *computation* of the decision tree on a specific permutation of edge weights in G is the path from the root to a leaf such that the outcome of every comparison agrees with the edge weights. The result of the computation is the spanning tree assigned to its final leaf. A decision tree is *correct* iff for every permutation the corresponding computation results in the real MSF of G with the particular weights.

The *time complexity* of a decision tree is defined as its depth. It therefore bounds the number of comparisons spent on every path. (It need not be equal since some paths need not correspond to an actual computation — the sequence of outcomes on the path could be unsatisfiable.)

A decision tree is called *optimal* if it is correct and its depth is minimum possible among the correct decision trees for the given graph. We will denote an arbitrary optimal decision tree for G by $\mathcal{D}(G)$ and its complexity by $D(G)$.

The *decision tree complexity* $D(m, n)$ of the MSF problem is the maximum of $D(G)$ over all graphs G with n vertices and m edges.

4.3.2. Observation. Decision trees are the most general deterministic comparison-based computation model possible. The only operations that count in its time complexity are comparisons. All other computation is free, including solving NP-complete problems or having access to an unlimited source of non-uniform constants. The decision tree complexity is therefore an obvious lower bound on the time complexity of the problem in all other comparison-based models.

[3] There are two possible outcomes since there is no reason to compare an edge with itself and we, as usually, expect that the edge weights are distinct.

The downside is that we do not know any explicit construction of the optimal decision trees, or at least a non-constructive proof of their complexity. On the other hand, the complexity of any existing comparison-based algorithm can be used as an upper bound on the decision tree complexity. For example:

4.3.3. Lemma. $D(m,n) \leq 4/3 \cdot n^2$.

Proof. Let us count the comparisons performed by the Contractive Borůvka's algorithm (1.5.2), tightening up the constants in its previous analysis in Theorem 1.5.5. In the first iteration, each edge participates in two comparisons (one per endpoint), so the algorithm performs at most $2m \leq 2\binom{n}{2} \leq n^2$ comparisons. Then the number of vertices drops at least by a factor of two, so the subsequent iterations spend at most $(n/2)^2, (n/4)^2, \ldots$ comparisons, which sums to less than $n^2 \cdot \sum_{i=0}^{\infty}(1/4)^i = 4/3 \cdot n^2$. Between the Borůvka steps, we flatten the multigraph to a simple graph, which also needs some comparisons, but for every such comparison we remove one of the participating edges, which saves at least one comparison in the subsequent steps. ♠

4.3.4. Of course we can get sharper bounds from the better algorithms, but we will first show how to find the optimal trees using brute force. The complexity of the search will be of course enormous, but as we already promised, we will need the optimal trees only for very small subgraphs.

4.3.5. Lemma. *(Construction of optimal decision trees)*
An optimal MST decision tree for a graph G on n vertices can be constructed on the Pointer Machine in time $\mathcal{O}(2^{2^{4n^2}})$.

Proof. We will try all possible decision trees of depth at most $2n^2$ (we know from the previous lemma that the desired optimal tree is shallower). We can obtain any such tree by taking the complete binary tree of exactly this depth and labeling its $2 \cdot 2^{2n^2} - 1$ vertices with comparisons and spanning trees. Those labeled with comparisons become internal vertices of the decision tree, the others become leaves and the parts of the tree below them are removed. There are less than n^4 possible comparisons and less than 2^{n^2} spanning trees of G, so the number of candidate decision trees is bounded by $(n^4 + 2^{n^2})^{2^{2n^2+1}} \leq 2^{(n^2+1)\cdot 2^{2n^2+1}} \leq 2^{2^{2n^2+2}} \leq 2^{2^{3n^2}}$.

We will enumerate the trees in an arbitrary order, test each of them for correctness and find the shallowest tree among those correct. Testing can be accomplished by running through all possible permutations of edges, each time calculating the MSF using any of the known algorithms and comparing it with the result given by the decision tree. The number of permutations does not exceed $(n^2)! \leq (n^2)^{n^2} \leq n^{2n^2} \leq 2^{n^3}$ and each one can be checked in time $\mathcal{O}(\text{poly}(n))$.

On the Pointer Machine, trees and permutations can be certainly enumerated in time $\mathcal{O}(\text{poly}(n))$ per object. The time complexity of the whole algorithm is therefore $\mathcal{O}(2^{2^{3n^2}} \cdot 2^{n^3} \cdot \text{poly}(n)) = \mathcal{O}(2^{2^{4n^2}})$. ♠

4.3.6. *Basic properties of decision trees.* The following properties will be useful for analysis of algorithms based on precomputed decision trees. We will omit some technical details, referring the reader to section 5.1 of the Pettie's article [PR02b].

4.3.7. Lemma. The decision tree complexity $D(m,n)$ of the MSF satisfies:

1. $D(m,n) \geq m/2$ for $m, n > 2$.
2. $D(m',n') \geq D(m,n)$ whenever $m' \geq m$ and $n' \geq n$.

Proof. For every $m, n > 2$ there is a graph on n vertices and m edges such that every edge lies on a cycle. Every correct MSF decision tree for this graph has to compare each edge at least once. Otherwise the decision tree cannot distinguish between the case when an edge has the lowest of all weights (and thus it is forced to belong to the MSF) and when it has the highest weight (so it is forced out of the MSF).

Decision trees for graphs on n' vertices can be used for graphs with n vertices as well — it suffices to add isolated vertices, which does not change the MSF. Similarly, we can increase m to m' by adding edges parallel to an existing edge and making them heavier than the rest of the graph, so that they can never belong to the MSF. ♠

4.3.8. Definition. Subgraphs $C_1, \ldots, C_k$ of a graph G are called the *compartments* of G iff they are edge-disjoint, their union is the whole graph G and $\mathrm{msf}(G) = \bigcup_i \mathrm{msf}(C_i)$ for every permutation of edge weights.

4.3.9. Lemma. The clusters $C_1, \ldots, C_k$ generated by the Partition procedure of the previous section (Algorithm 4.2.7) are compartments of the graph $H = \bigcup_i C_i$.

Proof. The first and second condition of the definition of compartments follow from the Partitioning theorem (4.2.8), so it remains to show that $\mathrm{msf}(H)$ is the union of the MSF's of the individual compartments. By the Cycle rule (Lemma 1.3.6), an edge $h \in H$ is not contained in $\mathrm{msf}(H)$ if and only if it is the heaviest edge on some cycle. It is therefore sufficient to prove that every cycle in H is contained within a single C_i.

Let us consider a cycle $K \subseteq H$ and a cluster C_i such that it contains an edge e of K and all clusters constructed later by the procedure do not contain any. If K is not fully contained in C_i, we can extend the edge e to a maximal path contained in both K and C_i. Since C_i shares at most one vertex with the earlier clusters, there can be at most one edge from K adjacent to the maximal path, which is impossible. ♠

4.3.10. Lemma. Let $C_1, \ldots, C_k$ be compartments of a graph G. Then there exists an optimal MSF decision tree for G that does not compare edges of distinct compartments.

Proof sketch. Consider a subset $\mathcal{P}$ of edge weight permutations w that satisfy $w(e) < w(f)$ whenever $e \in C_i, f \in C_j, i < j$. For such permutations, no decision tree can gain any information on relations between edge weights in a single compartment by inter-compartment comparisons — the results of all such comparisons are determined in advance.

Let us take an arbitrary correct decision tree for G and restrict it to vertices reachable by computations on $\mathcal{P}$. Whenever a vertex contained an inter-compartment comparison, it has lost one of its sons, so we can remove it by contracting its only outgoing edge. We observe that we get a decision tree satisfying the desired condition and that this tree is correct.

As for the correctness, the MSF of a single C_i is uniquely determined by comparisons of its weights and the set $\mathcal{P}$ contains all combinations of orderings of weights inside individual compartments. Therefore every spanning tree of every C_i and thus also of H is properly recognized. ♠

4.3.11. Lemma. Let $C_1, \ldots, C_k$ be compartments of a graph G. Then $D(G) = \sum_i D(C_i)$.

Proof sketch. A collection of decision trees for the individual compartments can be "glued together" to a decision tree for G. We take the decision tree for C_1, replace every its leaf by a copy of the tree for C_2 and so on. Every leaf ℓ of the compound tree will be labeled with the union of labels of the original leaves encountered on the path from the root to ℓ. This proves that $D(G) \leq \sum_i D(C_i)$.

The other inequality requires more effort. We use the previous lemma to transform the optimal decision tree for G to another of the same depth, but without inter-compartment comparisons. Then we prove by induction on k and then on the depth of the tree that this tree can be re-arranged, so that every computation first compares edges from C_1, then from C_2 and so on. This means that the tree can be decomposed to decision trees for the C_i's. Also, without loss of efficiency all trees for a single C_i can be made isomorphic to $\mathcal{D}(C_i)$. ♠

4.3.12. Corollary. If $C_1, \ldots, C_k$ are the clusters generated by the Partition procedure (Algorithm 4.2.7), then $D(\bigcup_i C_i) = \sum_i D(C_i)$.

Proof. Lemma 4.3.9 tells us that $C_1, \ldots, C_k$ are compartments of the graph $\bigcup C_i$, so we can apply Lemma 4.3.11 on them. ♠

4.3.13. Corollary. $2D(m, n) \leq D(2m, 2n)$ for every m, n.

Proof. For an arbitrary graph G with m edges and n vertices, we create a graph G_2 consisting of two copies of G sharing a single vertex. The copies of G are obviously compartments of G_2, so by Lemma 4.3.11 it holds that $D(G_2) = 2D(G)$. Taking a maximum over all choices of G yields $D(2m, 2n) \geq \max_G D(G_2) = 2D(m, n)$. ♠

4.4. An optimal algorithm

Once we have developed the soft heaps, partitioning and MST decision trees, it is now simple to state the Pettie's and Ramachandran's MST algorithm and prove that it is asymptotically optimal among all MST algorithms in comparison-based models. Several standard MST algorithms from the previous chapters will also play their roles.

We will describe the algorithm as a recursive procedure. When the procedure is called on a graph G, it sets the parameter t to roughly $\log^{(3)} n$ and it calls the *Partition* procedure to split the graph into a collection of clusters of size t and a set of corrupted edges. Then it uses precomputed decision trees to find the MSF of the clusters. The graph obtained by contracting the clusters is on the other hand dense enough, so that the Iterated Jarník's algorithm runs on it in linear time. Afterwards we combine the MSF's of the clusters and of the contracted graphs, we mix in the corrupted edges and run two iterations of the Contractive Borůvka's algorithm. This guarantees reduction in the number of both vertices and edges by a constant factor, so we can efficiently recurse on the resulting graph.

4.4.1. Algorithm. *(Optimal MST algorithm, Pettie and Ramachandran [PR02b])*

Input: A connected graph G with an edge comparison oracle.

1. If G has no edges, return an empty tree.
2. $t \leftarrow \lfloor \log^{(3)} n \rfloor$. *(the size of clusters)*

3. Call *Partition* (4.2.7) on G and t with $\varepsilon = 1/8$. It returns a collection $\mathcal{C} = \{C_1, \ldots, C_k\}$ of clusters and a set $R^{\mathcal{C}}$ of corrupted edges.
4. $F_i \leftarrow \mathrm{mst}(C_i)$ for all i, obtained using optimal decision trees.
5. $G_A \leftarrow (G/\bigcup_i C_i) \setminus R^{\mathcal{C}}$. *(the contracted graph)*
6. $F_A \leftarrow \mathrm{msf}(G_A)$ calculated by the Iterated Jarník's algorithm (3.2.11).
7. $G_B \leftarrow \bigcup_i F_i \cup F_A \cup R^{\mathcal{C}}$. *(combine subtrees with corrupted edges)*
8. Run two Borůvka steps (iterations of the Contractive Borůvka's algorithm, 1.5.2) on G_B, getting a contracted graph G_C and a set F_B of MST edges.
9. $F_C \leftarrow \mathrm{mst}(G_C)$ obtained by a recursive call to this algorithm.
10. Return $F_B \cup F_C$.

Output: The minimum spanning tree of G.

Correctness of this algorithm immediately follows from the Partitioning theorem (4.2.8) and from the proofs of the respective algorithms used as subroutines. Let us take a look at the time complexity. We will be careful to use only the operations offered by the Pointer Machine.

4.4.2. Lemma. The time complexity $T(m, n)$ of the Optimal algorithm satisfies the following recurrence:

$$T(m, n) \leq \sum_i c_1 D(C_i) + T(m/2, n/4) + c_2 m,$$

where c_1 and c_2 are some positive constants and D is the decision tree complexity from the previous section.

Proof. The first two steps of the algorithm are trivial as we have linear time at our disposal.

By the Partitioning theorem (4.2.8), the call to *Partition* with ε set to a constant takes $\mathcal{O}(m)$ time and it produces a collection of clusters of size at most t and at most $m/4$ corrupted edges. It also guarantees that the connected components of the union of the C_i's have at least t vertices (unless there is just a single component).

To apply the decision trees, we will use the framework of topological computations developed in Section 2.2. We pad all clusters in $\mathcal{C}$ with isolated vertices, so that they have exactly t vertices. We use a computation that labels the graph with a pointer to its optimal decision tree. Then we apply Theorem 2.2.14 combined with the brute-force construction of optimal decision trees from Lemma 4.3.5. Together they guarantee that we can assign the decision trees to the clusters in time:

$$\mathcal{O}\left(\|\mathcal{C}\| + t^{t(t+2)} \cdot \left(2^{2^{4t^2}} + t^2\right)\right) = \mathcal{O}\left(m + 2^{2^{2^t}}\right) = \mathcal{O}(m).$$

Execution of the decision tree on each cluster C_i then takes $\mathcal{O}(D(C_i))$ steps.

The contracted graph G_A has at most $n/t = \mathcal{O}(n/\log^{(3)} n)$ vertices and asymptotically the same number of edges as G, so according to Corollary 3.2.18, the Iterated Jarník's algorithm runs on it in linear time.

The combined graph G_B has n vertices, but less than n edges from the individual spanning trees and at most $m/4$ additional edges which were corrupted. The

Borůvka steps on G_B take $\mathcal{O}(m)$ time by Lemma 1.4.6 and they produce a graph G_C with at most $n/4$ vertices and at most $n/4 + m/4 \leq m/2$ edges. (The n tree edges in G_B are guaranteed to be reduced by the Borůvka's algorithm.) It is easy to verify that this graph is still connected, so we can recurse on it.

The remaining steps of the algorithm can be easily performed in linear time either directly or in case of the contractions by the bucket-sorting techniques of Section 2.2. ♠

4.4.3. *Optimality.* The properties of decision tree complexity, which we have proven in the previous section, will help us show that the time complexity recurrence is satisfied by a constant multiple of the decision tree complexity $D(m, n)$ itself. This way, we will prove the following theorem:

4.4.4. Theorem. *(Optimality of the Optimal algorithm)*
The time complexity of the Optimal MST algorithm 4.4.1 is $\Theta(D(m, n))$.

Proof. We will prove by induction that $T(m, n) \leq cD(m, n)$ for some $c > 0$. The base case is trivial, for the induction step we will expand on the previous lemma:

$$\begin{aligned}
T(m,n) &\leq \sum_i c_1 D(C_i) + T(m/2, n/4) + c_2 m && \text{(Lemma 4.4.2)}\\
&\leq c_1 D(\textstyle\bigcup_i C_i) + T(m/2, n/4) + c_2 m && \text{(Corollary 4.3.12)}\\
&\leq c_1 D(m,n) + T(m/2, n/4) + c_2 m && \text{(definition of } D(m,n))\\
&\leq c_1 D(m,n) + cD(m/2, n/4) + c_2 m && \text{(induction hypothesis)}\\
&\leq c_1 D(m,n) + c/2 \cdot D(m, n/2) + c_2 m && \text{(Corollary 4.3.13)}\\
&\leq c_1 D(m,n) + c/2 \cdot D(m,n) + 2c_2 D(m,n) && \text{(Lemma 4.3.7)}\\
&\leq (c_1 + c/2 + 2c_2) \cdot D(m,n)\\
&\leq cD(m,n). && \text{(by setting } c = 2c_1 + 4c_2)
\end{aligned}$$

The other inequality is obvious as $D(m, n)$ is an asymptotic lower bound on the time complexity of every comparison-based algorithm. ♠

4.4.5. *Complexity of MST.* As we have already noted, the exact decision tree complexity $D(m, n)$ of the MST problem is still open and so therefore is the time complexity of the optimal algorithm. However, every time we come up with another comparison-based algorithm, we can use its complexity (or more specifically the number of comparisons it performs, which can be even lower) as an upper bound on the optimal algorithm.

The best explicit comparison-based algorithm known to date achieves complexity $\mathcal{O}(m\,\alpha(m, n))$.[4] It has been discovered by Chazelle [Cha00a] as an improvement of his previous $\mathcal{O}(m\,\alpha(m, n) \cdot \log \alpha(m, n))$ algorithm [Cha97]. It is also based on the ideas of this chapter — in particular the soft heaps and robust contractions. The algorithm is very complex and it involves a lot of elaborate technical details, so we only refer to the original paper here. Another algorithm of the same complexity, using similar ideas, has been discovered independently by Pettie [Pet99], who, having the optimal algorithm at hand, does not take care about the low-level details

[4] $\alpha(m, n)$ is a certain inverse of the Ackermann's function, $\lambda_k(n)$ is the row inverse of the same function. See A.3.4 for the exact definitions.

and he only bounds the number of comparisons. Using any of these results, we can prove an Ackermannian upper bound on the optimal algorithm:

4.4.6. Theorem. *(Upper bound on complexity of the Optimal algorithm)*
The time complexity of the Optimal MST algorithm is $\mathcal{O}(m\alpha(m,n))$.

Proof. We bound $D(m,n)$ by the number of comparisons performed by the algorithm of Chazelle [Cha00a] or Pettie [Pet99]. ♠

Similarly to the Iterated Jarník's algorithm, this bound is actually linear for classes of graphs that do not have density extremely close to constant:

4.4.7. Corollary. The Optimal MST algorithm runs in linear time whenever $m \geq n \cdot \lambda_k(n)$ for any fixed k.

Proof. Combine the previous theorem with Lemma A.3.8. ♠

4.4.8. Remark. It is also known from [PR02b] that when we run the Optimal algorithm on a random graph drawn from either $G_{n,p}$ (random graphs on n vertices, each edge is included with probability p independently on the other edges) or $G_{n,m}$ (we draw the graph uniformly at random from the set of all graphs with n vertices and m edges), it runs in linear time with high probability, regardless of the edge weights.

4.4.9. *Models of computation.* Another important consequence of the optimal algorithm is that when we aim for a linear-time MST algorithm (or for proving that it does not exist), we do not need to care about computational models at all. The elaborate RAM data structures of Chapter 2, which have helped us so much in the case of integer weights, cannot help if we are allowed to access the edge weights by performing comparisons only. We can even make use of non-uniform objects given by some sort of oracle. Indeed, whatever trick we employ to achieve linear time complexity, we can mimic it in the world of decision trees and thus we can use it to show that the algorithm we already knew is also linear.

This however applies to deterministic algorithms only — we have shown that access to a source of random bits allows us to compute the MST in expected linear time (the KKT algorithm, 3.5.3). There were attempts to derandomize the KKT algorithm, but so far the best result in this direction is the randomized algorithm also by Pettie [PR02a] which achieves expected linear time complexity with only $\mathcal{O}(\log^* n)$ random bits.

5. Dynamic Spanning Trees

5.1. Dynamic graph algorithms

In many applications, we often need to solve a certain graph problem for a sequence of graphs that differ only a little, so recomputing the solution for every graph from scratch would be a waste of time. In such cases, we usually turn our attention to *dynamic graph algorithms.* A dynamic algorithm is in fact a data structure that remembers a graph. It offers operations that modify the structure of the graph and also operations that query the result of the problem for the current state of the graph. A typical example of a problem of this kind is dynamic maintenance of connected components:

5.1.1. Problem. *(Dynamic connectivity)*
Maintain an undirected graph under a sequence of the following operations:

- *Init*(n) — Create a graph with n isolated vertices $\{1, \ldots, n\}$.[1]
- *Insert*(G, u, v) — Insert an edge uv to G and return its unique identifier. This assumes that the edge did not exist yet.
- *Delete*(G, e) — Delete an edge specified by its identifier from G.
- *Connected*(G, u, v) — Test if vertices u and v are in the same connected component of G.

5.1.2. We have already encountered a special case of dynamic connectivity when implementing the Kruskal's algorithm in Section 1.4. At that time, we did not need to delete any edges from the graph, which makes the problem substantially easier. This special case is customarily called an *incremental* or *semidynamic* graph algorithm. We mentioned the Disjoint Set Union data structure of Tarjan (Theorem 1.4.20) which can be used for that: Connected components are represented by equivalence classes. Queries on connectedness translate to *Find*, edge insertions to *Find* followed by *Union* if the new edge joins two different components. This way, a sequence of m operations starting with an empty graph on n vertices is processed in time $\mathcal{O}(n + m\,\alpha(m, n))$ and this holds even for the Pointer Machine. Fredman and Saks [FS89] have proven a matching lower bound in the cell-probe model which is stronger than RAM with $\mathcal{O}(\log n)$-bit words.

5.1.3. *Dynamic MSF.* In this chapter, we will focus on the dynamic version of the minimum spanning forest. This problem seems to be intimately related to the dynamic connectivity. Indeed, all known algorithms for dynamic connectivity maintain some sort of a spanning forest. For example, in the incremental algorithm we have just mentioned, this forest is formed by the edges that have triggered the *Union*s. This suggests that a dynamic MSF algorithm could be obtained by modifying the mechanics of the data structure to keep the forest minimum. This will really turn out to be true, although we cannot be sure that it will lead to the most efficient solution possible — as of now, the known lower bounds are very far.

[1] The structure could support dynamic addition and removal of vertices, too, but this is easy to add and infrequently used, so we will rather keep the set of vertices fixed for clarity.

Incremental MST will be easy to achieve even in the few pages of this section, but making it fully dynamic will require more effort, so we will review some of the required building blocks before going into that.

We however have to answer one important question first: What should be the output of our MSF data structure? Adding an operation that returns the MSF of the current graph would be of course possible, but somewhat impractical as this operation would have to spend $\Omega(n)$ time on the mere writing of its output. A better way seems to be making the *Insert* and *Delete* operations report the list of modifications of the MSF implied by the change in the graph.

Let us see what happens when we *Insert* an edge e to a graph G with its minimum spanning forest F, obtaining a new graph G' with its MSF F'. If e connects two components of G (and therefore also of F), we have to add e to F. Otherwise, one of the following cases happens: Either e is F-heavy and thus the forest F is also the MSF of the new graph. Or it is F-light and we have to modify F by exchanging the heaviest edge f of the path $F[e]$ with e.

Correctness of the former case follows immediately from the Minimality Theorem (1.2.6), because any F'-light would be also F-light, which is impossible as F was minimum. In the latter case, the edge f is not contained in F' because it is the heaviest on the cycle $F[e] + e$ (by the Red lemma, 1.3.6). We can now use the Blue lemma (1.3.5) to prove that it should be replaced with e. Consider the tree T of F that contains both endpoints of the edge e. When we remove f from F, this tree falls apart to two components T_1 and T_2. The edge f was the lightest in the cut $\delta_G(T_1)$ and e is lighter than f, so e is the lightest in $\delta_{G'}(T_1)$ and hence $e \in F'$.

A *Delete* of an edge that is not contained in F does not change F. When we delete an MSF edge, we have to reconnect F by choosing the lightest edge of the cut separating the new components (again the Blue lemma in action). If there is no such replacement edge, we have deleted a bridge, so the MSF has to remain disconnected.

The idea of reporting differences in the MSF indeed works very well. We can summarize what we have shown by the following lemma and use it to define the dynamic MSF.

5.1.4. Lemma. An *Insert* or *Delete* of an edge in G causes at most one edge addition, edge removal or edge exchange in $\mathrm{msf}(G)$.

5.1.5. Problem. *(Dynamic minimum spanning forest)*
Maintain an undirected graph with distinct weights on edges (drawn from a totally ordered set) and its minimum spanning forest under a sequence of the following operations:

- *Init*(n) — Create a graph with n isolated vertices $\{1, \ldots, n\}$.
- *Insert*(G, u, v, w) — Insert an edge uv of weight w to G. Return its unique identifier and the list of additions and deletions of edges in $\mathrm{msf}(G)$.
- *Delete*(G, e) — Delete an edge specified by its identifier from G. Return the list of additions and deletions of edges in $\mathrm{msf}(G)$.

5.1.6. *Incremental MSF.* To obtain an incremental MSF algorithm, we need to keep the forest in a data structure that supports search for the heaviest edge on the path

connecting a given pair of vertices. This can be handled efficiently by the Link-Cut trees of Sleator and Tarjan:

5.1.7. Theorem. *(Link-Cut Trees, Sleator and Tarjan [ST83])*
There is a data structure that represents a forest of rooted trees on n vertices. Each edge of the forest has a weight drawn from a totally ordered set. The structure supports the following operations in time $\mathcal{O}(\log n)$ amortized:[2]

- *Parent*(v) — Return the parent of v in its tree or *null* if v is a root.
- *Root*(v) — Return the root of the tree containing v.
- *Weight*(v) — Return the weight of the edge $(\mathit{Parent}(v), v)$.
- *PathMax*(v) — Return the vertex u closest to *Root*(v) such that the edge $(\mathit{Parent}(u), u)$ is the heaviest of those on the path from the root to v. If more edges have the maximum weight, break the tie arbitrarily. If there is no such edge (v is the root itself), return *null*.
- *Link*(u, v, w) — Connect the trees containing u and v by an edge (u, v) of weight w. Assumes that v is a tree root and u lies in a different tree.
- *Cut*(v) — Split the tree containing the non-root vertex v to two trees by removing the edge $(\mathit{Parent}(v), v)$. Returns the weight of this edge.
- *Evert*(v) — Modify the orientations of edges to make v the root of its tree.

Proof. See [ST83]. ♠

Once we have this structure, we can turn our ideas on updating of the MSF to an incremental algorithm:

5.1.8. Algorithm. *(Insert in an incremental MSF)*

Input: A graph G with its MSF F represented as a Link-Cut forest, an edge uv with weight w to be inserted.

1. *Evert*(u). *(u is now the root of its tree.)*
2. If *Root*$(v) \neq u$: *(u and v lie in different trees.)*
3. *Link*(v, u, w). *(Connect the trees.)*
4. Return "uv added".
5. Otherwise: *(both are in the same tree)*
6. $y \leftarrow$ *PathMax*(v).
7. $x \leftarrow$ *Parent*(y). *(Edge xy is the heaviest on $F[uv]$.)*
8. If *Weight*$(y) > w$: *(We have to exchange xy with uv.)*
9. *Cut*(y), *Evert*(v), *Link*(u, v, w).
10. Return "uv added, xy removed".
11. Otherwise return "no changes".

Output: The list of changes in F.

5.1.9. Theorem. *(Incremental MSF)*
When only edge insertions are allowed, the dynamic MSF can be maintained in time $\mathcal{O}(\log n)$ amortized per operation.

Proof. Every *Insert* performs $\mathcal{O}(1)$ operations on the Link-Cut forest, which take $\mathcal{O}(\log n)$ each by Theorem 5.1.7. ♠

[2] The Link-Cut trees can offer a plethora of other operations, but we do not mention them as they are not needed for our problem.

5.1.10. Remark. We can easily extend the semidynamic MSF algorithm to allow an operation commonly called *Backtrack* — removal of the most recently inserted edge. It is sufficient to keep the history of all MSF changes in a stack and reverse the most recent change upon backtrack.

What are the obstacles to making the structure fully dynamic? Deletion of edges that do not belong to the MSF is trivial (we do not need to change anything) and so is deletion of bridges (we just remove the bridge from the Link-Cut tree, knowing that there is no edge to replace it). The hard part is the search for replacement edges after an edge belonging to the MSF is deleted.

This very problem also has to be solved by algorithms for fully dynamic connectivity, we will take a look at them first.

5.2. Eulerian Tour trees

An important stop on the road to fully dynamic algorithms has the name *Eulerian Tour trees* or simply *ET-trees*. It is a representation of forests introduced by Henzinger and King [HK99] in their randomized dynamic algorithms. It is similar to the Link-Cut trees, but it is much simpler and instead of path operations it offers efficient operations on subtrees. It is also possible to attach auxiliary data to vertices and edges of the original tree.

5.2.1. Definition. Let T be a rooted tree. We will call a sequence of vertices of T its *Eulerian Tour sequence (ET-sequence)* if it lists the vertices visited by the depth-first traversal of T. More precisely, it can be generated by the following procedure $ET(v)$ when it is invoked on the root of the tree:

1. Record v in the sequence.
2. For each son w of v:
3. Call $ET(w)$.
4. Record w.

A single tree can have multiple ET-sequences, corresponding to different orders in which the sons can be enumerated in step 2.

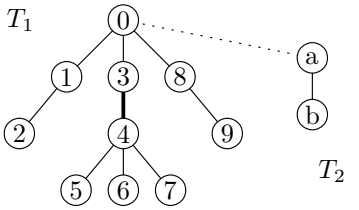
In every ET-sequence, one of the occurrences of each vertex is defined as its *active occurrence* and it will be used to store auxiliary data associated with that vertex.

5.2.2. Observation. An ET-sequence contains a vertex of degree d exactly d times except for the root which occurs $d+1$ times. The whole sequence therefore contains $2n-1$ elements. It indeed describes the order of vertices on an Eulerian tour in the tree with all edges doubled. Let us observe what happens to an ET-sequence when we modify the tree. (See the picture.)

When we *delete* an edge uv from the tree T (let u be the parent of v), the sequence $AuvBvuC$ (with no u nor v in B) splits to two sequences AuC and vBv. If there was only a single occurrence of v, then v was a leaf and thus the sequence transforms from $AuvuC$ to AuC and v alone.

Changing the root of the tree T from v to w changes its ET-sequence from $vAwBwCv$ to $wBwCvAw$. If w was a leaf, the sequence changes from $vAwCv$ to $wCvAw$. If vw was the only edge of T, the sequence vw becomes wv. Note that this works regardless of the possible presence of w inside B.

Joining the roots of two trees by a new edge makes their ET-sequences vAv and wBw combine to $vAvwBwv$. Again, we have to handle the cases when v or w has degree 1 separately: v and wBw combine to $vwBwv$, and v with w makes vwv.



T_1 : 0121034546474308980, T_2 : aba.
$T_1 - 34$: 01210308980, 4546474.
T_1 rooted at 3 : 3454647430898012103.
$T_1 + 0a + T_2$: 012103454647430898 0aba0.

Trees and their ET-sequences

If any of the occurrences that we have removed from the sequence was active, there is always a new occurrence of the same vertex that can stand in its place and inherit the auxiliary data.

5.2.3. The ET-trees will store the ET-sequences as (a, b)-trees with the parameter a set upon initialization of the structure and with $b = 2a$. We know from the standard theorems of (a, b)-trees (see for example [LRCS01]) that the depth of a tree with n leaves is always $\mathcal{O}(\log_a n)$ and that all basic operations including insertion, deletion, search, splitting and joining the trees run in time $\mathcal{O}(b \log_a n)$ in the worst case.

We will use the ET-trees to maintain a spanning forest of the dynamic graph. The auxiliary data of each vertex will hold a list of edges incident with the given vertex, that do not lie in the forest. Such edges are usually called the *non-tree edges.*

5.2.4. Definition. *Eulerian Tour trees (ET-trees)* are a data structure that represents a forest of trees and a set of non-tree edges associated with the vertices of the forest. To avoid confusion, we will distinguish between *original* vertices and edges (of the given trees) and the vertices and edges of the data structure. The structure consists of:

- A collection of (a, b)-trees of some fixed parameters a and b. Each such tree corresponds to one of the original trees T. Its leaves (in the usual tree order) correspond to the elements of an ET-sequence for T. Each two consecutive leaves u and v are separated by a unique key stored in an internal vertex of the (a, b)-tree. This key is used to represent the original edge uv. Each original edge is therefore kept in both its orientations.
- Mappings *act*, *edge* and *twin*:
  - $act(v)$ maps each original vertex to the leaf containing its active occurrence;
  - $edge(e)$ of an original edge e is one of the internal keys representing it;
  - $twin(k)$ pairs an internal key k with the other internal key of the same original edge.
- A list of non-tree edges placed in each leaf. The lists are allowed to be non-empty only in the leaves that represent active occurrences of original vertices.

- Boolean *markers* in the internal vertices that signal presence of a non-tree edge anywhere in the subtree rooted at the internal vertex.
- Counters *leaves*(v) that contain the number of leaves in the subtree rooted at v.

5.2.5. Definition. The ET-trees support the following operations on the original trees:

- *Create* — Create a single-vertex tree.
- *Link*(u, v) — Join two different trees by an edge uv and return a unique identifier of this edge.
- *Cut*(e) — Split a tree by removing the edge e given by its identifier.
- *Connected*(u, v) — Test if the vertices u and v lie in the same tree.
- *Size*(v) — Return the number of vertices in the tree containing the vertex v.
- *InsertNontree*(v, e) — Add a non-tree edge e to the list at v and return a unique identifier of this edge.
- *DeleteNontree*(e) — Delete a non-tree edge e given by its identifier.
- *ScanNontree*(v) — Return a list of non-tree edges associated with the vertices of the v's tree.

5.2.6. Implementation. We will implement the operations on the ET-trees by translating the intended changes of the ET-sequences to operations on the (a, b)-trees. The role of identifiers of the original vertices and edges will be of course played by pointers to the respective leaves and internal keys of the (a, b)-trees.

Cut of an edge splits the (a, b)-tree at both internal keys representing the given edge and joins them back in the different order.

Link of two trees can be accomplished by making both vertices the roots of their trees first and joining the roots by an edge afterwards. Re-rooting involves splits and joins of (a, b)-trees. As we can split at any occurrence of the new root vertex, we will use the active occurrence which we remember. Linking of the roots is translated to joining of the (a, b)-trees.

Connected follows parent pointers from both u and v to the roots of their trees. Then it checks if the roots are equal.

Size finds the root r and returns *leaves*(r).

InsertNontree finds the leaf $act(v)$ containing the list of v's non-tree edges and inserts the new edge there. The returned identifier will consist from the pointer to the edge and the vertex in whose list it is stored. Then we have to recalculate the markers on the path from $act(v)$ to the root. *DeleteNontree* is analogous.

Whenever any other operation changes a vertex of the tree, it will also update its marker and counter and, if necessary, the markers and counters on the path to the root.

ScanNontree traverses the tree recursively from the root, but it does not enter the subtrees whose roots are not marked.

Analysis of time complexity of the operations is now straightforward:

5.2.7. Theorem. *(Eulerian Tour trees, Henzinger and Rauch [HK99])*
The ET-trees perform the operations *Link* and *Cut* in time $\mathcal{O}(a \log_a n)$, *Create* in $\mathcal{O}(1)$, *Connected*, *Size*, *InsertNontree*, and *DeleteNontree* in $\mathcal{O}(\log_a n)$, and

ScanNontree in $\mathcal{O}(a \log_a n)$ per edge reported. Here n is the number of vertices in the original forest and $a \geq 2$ is an arbitrary constant.

Proof. We set $b = 2a$. Our implementation performs $\mathcal{O}(1)$ operations on the (a, b)-trees per operation on the ET-tree, plus $\mathcal{O}(1)$ other work. We apply the standard theorems on the complexity of (a, b)-trees [LRCS01]. ♠

5.2.8. Example. *(Connectivity acceleration)*
In most cases, the ET-trees are used with a constant, but sometimes choosing a as a function of n can also have its beauty. Suppose that there is a data structure which maintains an arbitrary spanning forest of a dynamic graph. Suppose also that the structure works in time $\mathcal{O}(\log^k n)$ per operation and that it reports $\mathcal{O}(1)$ changes in the spanning forest for every change in the graph. If we keep the spanning forest in ET-trees with $a = \log n$, the updates of the data structure cost an additional $\mathcal{O}(\log^2 n / \log \log n)$, but connectivity queries accelerate to $\mathcal{O}(\log n / \log \log n)$.

5.2.9. *ET-trees with weights.* In some cases, we will also need a representation of weighted graphs and enumerate the non-tree edges in order of their increasing weights (in fact, it will be sufficient to find the lightest one, remove it and iterate). This can be handled by a minute modification of the ET-trees.

The tree edges will remember their weight in the corresponding internal keys of the ET-tree. We replace each list of non-tree edges by an (a, b)-tree keeping the edges sorted by weight. We also store the minimum element of that tree separately, so that it can be accessed in constant time. The boolean *marker* will then become the minimum weight of a non-tree edge attached to the particular subtree, which can be recalculated as easy as the markers can. Searching for the lightest non-tree edge then just follows the modified markers.

The time complexities of all operations therefore remain the same, with a possible exception of the operations on non-tree edges, to which we have added the burden of updating the new (a, b)-trees. This however consists of $\mathcal{O}(1)$ updates per a single call to *InsertNontree* or *DeleteNontree*, which takes $\mathcal{O}(a \log_a n)$ time only. We can therefore conclude:

5.2.10. Corollary. *(Weighted ET-trees)*
In weighted ET-trees, the operations *InsertNontree* and *DeleteNontree* have time complexity $\mathcal{O}(a \log_a n)$. All other operations take the same time as indicated by Theorem 5.2.7.

5.3. Dynamic connectivity

The fully dynamic connectivity problem has a long and rich history. In the 1980's, Frederickson [Fre85] has used his topological trees to construct a dynamic connectivity algorithm of complexity $\mathcal{O}(\sqrt{m})$ per update and $\mathcal{O}(1)$ per query. Eppstein et al. [EGIN97] have introduced a sparsification technique which can bring the updates down to $\mathcal{O}(\sqrt{n})$. Later, several different algorithms with complexity on the order of n^ε were presented by Henzinger and King [HK97a] and also by Mareš [Mar00]. A polylogarithmic time bound was first reached by the randomized algorithm of Henzinger and King [HK99]. The best result known as of now is the $\mathcal{O}(\log^2 n)$ time deterministic algorithm by Holm, de Lichtenberg and Thorup [HdLT01], which will we describe in this section.

The algorithm will maintain a spanning forest F of the current graph G, represented by an ET-tree which will be used to answer connectivity queries. The edges of $G \setminus F$ will be stored as non-tree edges in the ET-tree. Hence, an insertion of an edge to G either adds it to F or inserts it as non-tree. Deletions of non-tree edges are also easy, but when a tree edge is deleted, we have to search for its replacement among the non-tree edges.

To govern the search in an efficient way, we will associate each edge e with a level $\ell(e) \leq L = \lfloor \log_2 n \rfloor$. For each level i, we will use F_i to denote the subforest of F containing edges of level at least i. Therefore $F = F_0 \supseteq F_1 \supseteq \ldots \supseteq F_L$. We will maintain the following *invariants:*

- **I1** F is the maximum spanning forest of G with respect to the levels. (In other words, if uv is a non-tree edge, then u and v are connected in $F_{\ell(uv)}$.)
- **I2** For each i, the components of F_i have at most $\lfloor n/2^i \rfloor$ vertices each. (This implies that it does not make sense to define F_i for $i > L$, because it would be empty anyway.)

At the beginning, the graph contains no edges, so both invariants are trivially satisfied. Newly inserted edges enter level 0, which cannot break I1 nor I2.

When we delete a tree edge at level ℓ, we split a tree T of F_ℓ to two trees T_1 and T_2. Without loss of generality, let us assume that T_1 is the smaller one. We will try to find the replacement edge of the highest possible level that connects the spanning tree back. From I1, we know that such an edge cannot belong to a level greater than ℓ, so we start looking for it at level ℓ. According to I2, the tree T had at most $\lfloor n/2^\ell \rfloor$ vertices, so T_1 has at most $\lfloor n/2^{\ell+1} \rfloor$ of them. Thus we can move all level ℓ edges of T_1 to level $\ell + 1$ without violating either invariant.

We now start enumerating the non-tree edges incident with T_1. Each such edge is either local to T_1 or it joins T_1 with T_2. We will therefore check each edge whether its other endpoint lies in T_2 and if it does, we have found the replacement edge, so we insert it to F_ℓ and stop. Otherwise we move the edge one level up. (This will be the grist for the mill of our amortization argument: We can charge most of the work on level increases and we know that the level of each edge can reach at most L.)

If the non-tree edges at level ℓ are exhausted, we try the same in the next lower level and so on. If there is no replacement edge at level 0, the tree T remains disconnected.

5.3.1. Implementation. For each level ℓ, we will use a separate ET-tree $\mathcal{E}_\ell$ with a set to 2, which will represent the forest F_ℓ and the non-tree edges at that particular level. Besides operations on the non-tree edges, we also need to find the tree edges of level ℓ when we want to bring them one level up. This can be accomplished either by modifying the ET-trees to attach two lists of edges attached to vertices instead of one, or by using a second ET-tree.

5.3.2. Algorithm. *(Insertion of an edge)*

Input: An edge uv to insert.

1. $\ell(uv) \leftarrow 0$.

2. Ask the ET-tree $\mathcal{E}_0$ if u and v are in the same component. If they are:
3. Add uv to the list of non-tree edges in $\mathcal{E}_0$ at both u and v.
4. Otherwise:
5. Add uv to F_0.

5.3.3. Algorithm. *(Deletion of an edge)*

Input: An edge uv to delete.

1. $\ell \leftarrow \ell(uv)$.
2. If uv is a non-tree edge:
3. Remove uv from the lists of non-tree edges at both u and v in $\mathcal{E}_\ell$.
4. Otherwise:
5. Remove uv from F_ℓ and hence also from $F_0, \ldots, F_{\ell-1}$.
6. Call *Replace*(uv, ℓ) to get the replacement edge f.
7. Insert f to $F_0, \ldots, F_{\ell(f)}$.

5.3.4. Algorithm. *(Replace(uv, i) – Search for replacement for edge uv at level i)*

Input: An edge uv to replace and a level i such that there is no replacement at levels greater than i.

1. Let T_1 and T_2 be the trees in F_i containing u and v respectively.
2. If $n(T_1) > n(T_2)$, swap T_1 with T_2.
3. Find all level i edges in T_1 using $\mathcal{E}_i$ and move them to level $i+1$.
4. Enumerate non-tree edges incident with vertices of T_1 and stored in $\mathcal{E}_i$. For each edge xy, $x \in T_1$, do:
5. If $y \in T_2$, remove xy from $\mathcal{E}_i$ and return it to the caller.
6. Otherwise increase $\ell(xy)$ by one.
   This includes deleting xy from $\mathcal{E}_i$ and inserting it to $\mathcal{E}_{i+1}$.
7. If $i > 0$, call *Replace*$(xy, i-1)$.
8. Otherwise return *null*.

Output: The replacement edge.

As foretold, time complexity will be analysed by amortization on the levels.

5.3.5. Theorem. *(Fully dynamic connectivity, Holm et al. [HdLT01])*
Dynamic connectivity can be maintained in time $\mathcal{O}(\log^2 n)$ amortized per *Insert* and *Delete* and in time $\mathcal{O}(\log n / \log\log n)$ per *Connected* in the worst case.

Proof. The direct cost of an *Insert* is $\mathcal{O}(\log n)$ for the operations on the ET-trees (by Theorem 5.2.7). We will also have the insertion pre-pay all level increases of the new edge. Since the levels never decrease, each edge can be brought a level up at most $L = \lfloor \log n \rfloor$ times. Every increase costs $\mathcal{O}(\log n)$ on the ET-tree operations, so we pay $\mathcal{O}(\log^2 n)$ for all of them.

A *Delete* costs $\mathcal{O}(\log^2 n)$ directly, as we might have to update all L ET-trees. Additionally, we call *Replace* up to L times. The initialization of *Replace* costs $\mathcal{O}(\log n)$ per call, the rest is paid for by the edge level increases.

To bring the complexity of the operation *Connected* from $\mathcal{O}(\log n)$ down to $\mathcal{O}(\log n / \log\log n)$, we apply the trick from Example 5.2.8 and store F_0 in an ET-tree with $a = \max(\lfloor \log n \rfloor, 2)$. This does not hurt the complexity of insertions and deletions, but allows for faster queries. ♠

5.3.6. Remark. An $\Omega(\log n/\log\log n)$ lower bound for the amortized complexity of the dynamic connectivity problem has been proven by Henzinger and Fredman [HF98] in the cell probe model with $\mathcal{O}(\log n)$-bit words. Thorup has answered by a faster algorithm [Tho00b] that achieves $\mathcal{O}(\log n \log^3 \log n)$ time per update and $\mathcal{O}(\log n/\log^{(3)} n)$ per query on a RAM with $\mathcal{O}(\log n)$-bit words. (He claims that the algorithm runs on a Pointer Machine, but it uses arithmetic operations, so it does not fit the definition of the PM we use. The algorithm only does not need direct indexing of arrays.) So far, it is not known how to extend this algorithm to fit our needs, so we omit the details.

5.4. Dynamic spanning forests

Let us turn our attention back to the dynamic MSF. Most of the early algorithms for dynamic connectivity also imply $\mathcal{O}(n^\varepsilon)$ algorithms for dynamic maintenance of the MSF. Henzinger and King [HK97b, HK99] have generalized their randomized connectivity algorithm to maintain the MSF in $\mathcal{O}(\log^5 n)$ time per operation, or $\mathcal{O}(k \log^3 n)$ if only k different values of edge weights are allowed. They have solved the decremental version of the problem first (which starts with a given graph and only edge deletions are allowed) and then presented a general reduction from the fully dynamic MSF to its decremental version. We will describe the algorithm of Holm, de Lichtenberg and Thorup [HdLT01], who have followed the same path. They have modified their dynamic connectivity algorithm to solve the decremental MSF in $\mathcal{O}(\log^2 n)$ and obtained the fully dynamic MSF working in $\mathcal{O}(\log^4 n)$ per operation.

5.4.1. *Decremental MSF.* Turning the algorithm from the previous section to the decremental MSF requires only two changes: First, we have to start with the forest F equal to the MSF of the initial graph. As we require to pay $\mathcal{O}(\log^2 n)$ for every insertion, we can use almost arbitrary MSF algorithm to find F. Second, when we search for an replacement edge, we need to pick the lightest possible choice. We will therefore use the weighted version of the ET-trees (Corollary 5.2.10) and scan the lightest non-tree edge incident with the examined tree first. We must ensure that the lower levels cannot contain a lighter replacement edge, but fortunately the light edges tend to "bubble up" in the hierarchy of levels. This can be formalized in form of the following invariant:

I3 On every cycle, the heaviest edge has the smallest level.

This immediately implies that we always select the right replacement edge:

5.4.2. Lemma. Let F be the minimum spanning forest and e any its edge. Then among all replacement edges for e, the lightest one is at the maximum level.

Proof. Let us consider any two edges f_1 and f_2 replacing e. By minimality of F and the Cycle rule (Lemma 1.3.6), each f_i is the heaviest edge on the cycle $C_i = F[f_i]+f_i$. In a moment, we will show that the symmetric difference C of these two cycles is again a cycle. This implies that if f_1 is heavier than f_2, then f_1 is the heaviest edge on C, so $\ell(f_1) \le \ell(f_2)$ by I3. Therefore the lightest of all replacement edges must have the maximum level.

Why is C a cycle: Let F^a and F^b be the trees of $F - e$ in which the endpoints of e lie, and for every edge g going between F^a and F^b let g^a and g^b be its respective

endpoints. We know that C_i consists of the path $F[f_i^a, e^a]$, the edge e, the path $F[e^b, f_i^b]$, and the edge f_i. Thus C must contain the paths $F[f_1^a, f_2^a]$ and $F[f_1^b, f_2^b]$ and the edges f_1 and f_2, which together form a simple cycle. ♠

We now have to make sure that the additional invariant is really observed:

5.4.3. Lemma. After every operation, the invariant I3 is satisfied.

Proof. When the structure is freshly initialized, I3 is obviously satisfied, as all edges are at level 0. Sole deletions of edges (both tree and non-tree) cannot violate I3, so we need to check only the replaces, in particular the place when an edge e gets its level increased.

For the violation to happen for the first time, e must be the heaviest on some cycle C, so by the Cycle rule, e must be non-tree. The increase of $\ell(e)$ must therefore take place when e is considered as a replacement edge incident with some tree T_1 at level $\ell = \ell(e)$. We will pause the computation just before this increase and we will prove that all other edges of C already are at levels greater than ℓ, so the violation cannot occur.

Let us first show that for edges of C incident with T_1. All edges of T_1 itself already are at the higher levels as they were moved there at the very beginning of the search for the replacement edge. The other tree edges incident with T_1 would have lower levels, which is impossible since the invariant would be already violated. Non-tree edges of C incident with T_1 are lighter than e, so they were already considered as candidates for the replacement edge, because the algorithm always picks the lightest candidate first. Such edges therefore have been already moved a level up.

The case of edges of C that do not touch T_1 is easy to handle: Such edges do not exist. If they did, at least one more edge of C besides e would have to connect T_1 with the other trees of level ℓ. We already know that this could not be a tree edge. If it were a non-tree edge, it could not have level greater than ℓ by I1 nor smaller than ℓ by I3. Therefore it would be a level ℓ edge lighter than e, and as such it would have been selected as the replacement edge before e was. ♠

We can conclude:

5.4.4. Theorem. *(Decremental MSF, Holm et al. [HdLT01])*
When we start with a graph on n vertices with m edges and we perform a sequence of edge deletions, the MSF can be initialized in time $\mathcal{O}((m+n) \cdot \log^2 n)$ and then updated in time $\mathcal{O}(\log^2 n)$ amortized per operation.

5.4.5. *Fully dynamic MSF.* The decremental MSF algorithm can be turned to a fully dynamic one by a blackbox reduction whose properties are summarized in the following theorem:

5.4.6. Theorem. *(MSF dynamization, Holm et al. [HdLT01])*
Suppose that we have a decremental MSF algorithm with the following properties:

1. For any a, b, it can be initialized on a graph with a vertices and b edges.
2. Then it executes an arbitrary sequence of deletions in time $\mathcal{O}(b \cdot t(a,b))$, where t is a non-decreasing function.

Then there exists a fully dynamic MSF algorithm for a graph on n vertices, starting

with no edges, that performs m insertions and deletions in amortized time:

$$\mathcal{O}\left(\log^3 n + \sum_{i=1}^{\log m} \sum_{j=1}^{i} t(\min(n, 2^j), 2^j)\right) \quad \text{per operation.}$$

Proof sketch. The reduction is very technical, but its essence is the following: We maintain a logarithmic number of decremental structures $A_0, \ldots, A_{\lfloor \log n \rfloor}$ of exponentially increasing sizes. Every non-tree edge is contained in exactly one A_i, tree edges can belong to multiple structures.

When an edge is inserted, we union it with some of the A_i's, build a new decremental structure and amortize the cost of the build over the insertions. Deletes of non-tree edges are trivial. Delete of a tree edge is performed on all A_i's containing it and the replacement edge is sought among the replacement edges found in these structures. The unused replacement edges then have to be reinserted back to the structure.

The original reduction of Henzinger et al. [HK97b] handles these reinserts by a mechanism of batch insertions supported by their decremental structure, which is not available in our case. Holm et al. have replaced it by a system of auxiliary edges inserted at various places in the structure. We refer to the article [HdLT01] for details. ♠

5.4.7. Corollary. *(Fully dynamic MSF)*
There is a fully dynamic MSF algorithm that works in time $\mathcal{O}(\log^4 n)$ amortized per operation for graphs on n vertices.

Proof. Apply the reduction from the previous theorem to the decremental algorithm we have developed. This results in an algorithm of amortized complexity $\mathcal{O}(\log^4 \max(m, n))$ where m is the number of operations performed. This could exceed $\mathcal{O}(\log^4 n)$ if m is very large, but we can rebuild the whole structure after n^2 operations, which brings $\log m$ down to $\mathcal{O}(\log n)$. The $\mathcal{O}(n^2 \log^4 n)$ cost of the rebuild then incurs only $\mathcal{O}(\log^4 n)$ additional cost on each operation. ♠

5.4.8. Remark. The limitation of MSF structures based on the Holm's algorithm for connectivity to only edge deletions seems to be unavoidable. The invariant I3 could be easily broken for many cycles at once whenever a very light non-tree edge is inserted. We could try increasing the level of the newly inserted edge, but we would quite likely hit I1 before we managed to skip the levels of all the heaviest edges on the particular cycles.

On the other hand, if we decided to drop I3, we would encounter different obstacles. The ET-trees can bring the lightest non-tree incident with the current tree T_1, but the lightest replacement edge could also be located in the super-trees of T_1 at the lower levels, which are too large to scan and both I1 and I2 prevent us from charging the time on increasing levels there.

An interesting special case in which insertions are possible is when all non-tree edges have the same weight. This leads to the following algorithm for dynamic MSF on graphs with a small set of allowed edge weights. It is based on an idea similar to the $\mathcal{O}(k \log^3 n)$ algorithm of Henzinger and King [HK99], but have adapted it to use the better results on dynamic connectivity we have at hand.

5.4.9. *Dynamic MSF with limited edge weights.* Let us assume for a while that our graph has edges of only two different weights (let us say 1 and 2). We will forget our rule that all edge weights are distinct for a moment and we recall the observation in 1.6.3 that the basic structural properties of the MST's from Section 1.2 still hold.

We split the graph G to two subgraphs G_1 and G_2 according to the edge weights. We use one instance $\mathcal{C}_1$ of the dynamic connectivity algorithm to maintain an arbitrary spanning forest F_1 of G_1, which is obviously minimum. Then we add another instance $\mathcal{C}_2$ to maintain a spanning forest F_2 of the graph $G_2 \cup F_1$ such that all edges of F_1 are forced to be in F_2. Obviously, F_2 is the MSF of the whole graph G — if any edge of F_1 were not contained in $\mathrm{msf}(G)$, we could use the standard exchange argument to create an even lighter spanning tree.

When a weight 2 edge is inserted to G, we insert it to $\mathcal{C}_2$ and it either enters F_2 or becomes a non-tree edge. Similarly, deletion of a weight 2 edge is a pure deletion in $\mathcal{C}_2$, because such edges can be replaced only by other weight 2 edges.

Insertion of edges of weight 1 needs more attention: We insert the edge to $\mathcal{C}_1$. If F_1 stays unchanged, we are done. If the new edge enters F_1, we use a Sleator-Tarjan tree kept for F_2 to check if the new edge covers some tree edge of weight 2. If this is not the case, we insert the new edge to $\mathcal{C}_2$ and hence also to F_2 and we are done. Otherwise we exchange one of the covered weight 2 edges f for e in $\mathcal{C}_2$. We note that e can inherit the level of f and f can become a non-tree edge without changing its level. This adjustment can be performed in time $\mathcal{O}(\log^2 n)$, including paying for the future level increases of the new edge.

Deletion of weight 1 edges is once more straightforward. We delete the edge from $\mathcal{C}_1$. If it has no replacement, we delete it from $\mathcal{C}_2$ as well. If it has a replacement, we delete the edge from $\mathcal{C}_2$ and insert the replacement on its place as described above. We observe than this pair of operations causes an insertion, deletion or a replacement in $\mathcal{C}_2$.

This way, we can handle every insertion and deletion in time $\mathcal{O}(\log^2 n)$ amortized. This construction can be iterated in an obvious way: if we have k distinct edge weights, we build k connectivity structures $\mathcal{C}_1, \ldots, \mathcal{C}_k$. The structure $\mathcal{C}_i$ contains edges of weight i together with the MSF edges from $\mathcal{C}_{i-1}$. Bounding the time complexity is then easy:

5.4.10. Theorem. *(MSF with limited edge weights)*
There is a fully dynamic MSF algorithm that works in time $\mathcal{O}(k \cdot \log^2 n)$ amortized per operation for graphs on n vertices with only k distinct edge weights allowed.

Proof. A change in the graph G involving an edge of weight w causes a change in $\mathcal{C}_w$, which can propagate to $\mathcal{C}_{w+1}$ and so on, possibly up to $\mathcal{C}_k$. In each $\mathcal{C}_i$, we spend time $\mathcal{O}(\log^2 n)$ by updating the connectivity structure according to Theorem 5.3.5 and $\mathcal{O}(\log n)$ on operations with the Sleator-Tarjan trees by Theorem 5.1.7. ♠

5.5. Almost minimum trees

In some situations, finding the single minimum spanning tree is not enough and we are interested in the K lightest spanning trees, usually for some small value of K. Katoh, Ibaraki and Mine [KIM81] have given an algorithm of time complexity $\mathcal{O}(m \log \beta(m,n) + Km)$, building on the MST algorithm of Gabow et al. [GGST86]. Subsequently, Eppstein [Epp92] has discovered an elegant preprocessing step which allows to reduce the running time to $\mathcal{O}(m \log \beta(m,n) + \min(K^2, Km))$ by eliminating edges which are either present in all K trees or in none of them. We will show a variant of their algorithm based on the MST verification procedure of Section 3.4.

In this section, we will require the edge weights to be numeric, because comparisons are certainly not sufficient to determine the second best spanning tree. We will assume that our computation model is able to add, subtract and compare the edge weights in constant time.

Let us focus on finding the second lightest spanning tree first.

5.5.1. *Second lightest spanning tree.* Suppose that we have a weighted graph G and a sequence $T_1, \ldots, T_z$ of all its spanning trees. Also suppose that the weights of these spanning trees are distinct and that the sequence is ordered by weight, i.e., $w(T_1) < \ldots < w(T_z)$ and $T_1 = \mathrm{mst}(G)$. Let us observe that each tree is similar to at least one of its predecessors:

5.5.2. Lemma. *(Difference lemma)*
For each $i > 1$ there exists $j < i$ such that T_i and T_j differ by a single edge exchange.

Proof. We know from the Monotone exchange lemma (1.2.5) that T_1 can be transformed to T_i by a sequence of edge exchanges which never decrease tree weight. The last exchange in this sequence therefore obtains T_i from a tree of the desired properties. ♠

5.5.3. This lemma implies that the second best spanning tree T_2 differs from T_1 by a single edge exchange. It remains to find which exchange it is. Let us consider the exchange of an edge $f \in E \setminus T_1$ with an edge $e \in T_1[f]$. We get a tree $T_1 - e + f$ of weight $w(T_1) - w(e) + w(f)$. To obtain T_2, we have to find e and f such that the difference $w(f) - w(e)$ is the minimum possible. Thus for every f, the edge e must be always the heaviest on the path $T_1[f]$. We can apply the algorithm from Corollary 3.4.14 and find the heaviest edges (peaks) of all such paths and thus examine all possible choices of f in linear time. So we get:

5.5.4. Lemma. For every graph H and a MST T of H, linear time is sufficient to find edges $e \in T$ and $f \in H \setminus T$ such that $w(f) - w(e)$ is minimum.

5.5.5. Notation. We will call this procedure *finding the best exchange in* (H, T).

5.5.6. Corollary. Given G and T_1, we can find T_2 in time $\mathcal{O}(m)$.

5.5.7. *Third lightest spanning tree.* Once we know T_1 and T_2, how to get T_3? According to the Difference lemma, T_3 can be obtained by a single exchange from either T_1 or T_2. Therefore we need to find the best exchange for T_2 and the second best exchange for T_1 and use the better of them. The latter is not easy to find directly, so we will make a minor side step.

We know that T_2 equals $T_1 - e + f$ for some edges e and f. We define two auxiliary graphs: $G_1 := G/e$ and $G_2 := G - e$. The tree T_1/e is obviously the MST

of G_1 (by the Contraction lemma) and T_2 is the MST of G_2 (all T_2-light edges in G_2 would be T_1-light in G).

5.5.8. Observation. The tree T_3 can be obtained by a single edge exchange in either $(G_1, T_1/e)$ or (G_2, T_2):

- If $T_3 = T_1 - e' + f'$ for $e' \neq e$, then $T_3/e = (T_1/e) - e' + f'$ in G_1.
- If $T_3 = T_1 - e + f'$, then $T_3 = T_2 - f + f'$ in G_2.
- If $T_3 = T_2 - e' + f'$, then this exchange is found in G_2.

Conversely, a single exchange in $(G_1, T_1/e)$ or in (G_2, T_2) corresponds to an exchange in either (G, T_1) or (G, T_2). Even stronger, a spanning tree T of G either contains e and then $T.e$ is a spanning tree of G_1, or T doesn't contain e and so it is a spanning tree of G_2.

Thus we can run the previous algorithm for finding the best edge exchange on both G_1 and G_2 and find T_3 again in time $\mathcal{O}(m)$.

5.5.9. *Further spanning trees.* The construction of auxiliary graphs can be iterated to obtain $T_1, \ldots, T_K$ for an arbitrary K. We will build a *meta-tree* of auxiliary graphs. Each node of this meta-tree carries a graph[3] and its minimum spanning tree. The root node contains (G, T_1), its sons have $(G_1, T_1/e)$ and (G_2, T_2). When T_3 is obtained by an exchange in one of these sons, we attach two new leaves to that son and we let them carry the two auxiliary graphs derived by contracting or deleting the exchanged edge. Then we find the best edge exchanges among all leaves of the new meta-tree and repeat the process. By Observation 5.5.8, each spanning tree of G is generated exactly once. The Difference lemma guarantees that the trees are enumerated in the increasing order.

Recalculating the best exchanges in all leaves of the meta-tree after generating each T_i is of course not necessary, because most leaves stay unchanged. We will rather remember the best exchange for each leaf and keep the weight differences of these exchanges in a heap. In every step, we will delete the minimum from the heap and use the exchange in the particular leaf to generate a new spanning tree. Then we will create the new leaves, calculate their best exchanges and insert them into the heap. The algorithm is now straightforward and so will be its analysis:

5.5.10. Algorithm. *(Finding K best spanning trees)*

Input: A weighted graph G, its MST T_1 and an integer $K > 0$.

1. $R \leftarrow$ a meta tree whose vertices carry triples (G', T', F'). Initially it contains just a root with $(G, T_1, \emptyset)$.
   (G' is a graph, T' is its MST, and F' is a set of edges of G that are contracted in G'.)
2. $H \leftarrow$ a heap of quadruples (δ, r, e, f) ordered on δ, initially empty.
   (Each quadruple describes an exchange of e for f in a leaf r of R and $\delta = w(f) - w(e)$ is the weight gain of this exchange.)
3. Find the best edge exchange in (G, T_1) and insert it to H.
4. $i \leftarrow 1$.

[3] This graph is always derived from G by a sequence of edge deletions and contractions. It is tempting to say that it is a minor of G, but this is not true as we preserve multiple edges.

5. While $i < K$:
6. Delete the minimum quadruple (δ, r, e, f) from H.
7. $(G', T', F') \leftarrow$ the triple carried by the leaf r.
8. $i \leftarrow i + 1$.
9. $T_i \leftarrow (T' - e + f) \cup F'$. *(The next spanning tree)*
10. $r_1 \leftarrow$ a new leaf carrying $(G'/e, T'/e, F' + e)$.
11. $r_2 \leftarrow$ a new leaf carrying $(G' - e, T_i, F')$.
12. Attach r_1 and r_2 as sons of r.
13. Find the best edge exchanges in r_1 and r_2 and insert them to H.

Output: The spanning trees $T_2, \ldots, T_K$.

5.5.11. Lemma. Given G and T_1, we can find $T_2, \ldots, T_K$ in time $\mathcal{O}(Km + K \log K)$.

Proof. Generating each T_i requires finding the best exchange for two graphs and also $\mathcal{O}(1)$ operations on the heap. The former takes $\mathcal{O}(m)$ according to Corollary 3.4.14, and each heap operation takes $\mathcal{O}(\log K)$. ♠

5.5.12. Remark. The meta-tree is not needed for the actual operation of the algorithm — it suffices to keep its leaves in the heap.

5.5.13. *Arbitrary weights.* While the assumption that the weights of all spanning trees are distinct has helped us in thinking about the problem, we should not forget that it is somewhat unrealistic. We could refine the proof of our algorithm and demonstrate that the algorithm indeed works without this assumption, but we will rather show that the ties can be broken easily.

Let δ be the minimum positive difference among the weights of all spanning trees of G and $e_1, \ldots, e_m$ be the edges of G. We observe that it suffices to increase $w(e_i)$ by $\delta_i = \delta/2^{i+1}$. The cost of every spanning tree has increased by at most $\sum_i \delta_i < \delta/2$, so if T was lighter than T', it still is. On the other hand, no two trees share the same weight adjustment, so all tree weights are now distinct.

The exact value of δ is not easy to calculate, but closer inspection of the algorithm reveals that it is not needed at all. The only place where the edge weights are examined is when we search for the best exchange. In this case, we compare the differences of pairs of edge weights with each other. Each such difference is therefore adjusted by $\delta \cdot (2^{-i} - 2^{-j})$ for some $i, j > 1$, which again does not influence comparison of originally distinct differences. If two differences were identical, it is sufficient to look at their values of i and j, i.e., at the identifiers of the edges.

5.5.14. *Invariant edges.* Our algorithm can be further improved for small values of K (which seems to be the common case in most applications) by the reduction of Eppstein [Epp92]. We will observe that there are many edges of T_1 which are guaranteed to be contained in $T_2, \ldots, T_K$ as well, and likewise there are many edges of $G \setminus T_1$ which are excluded from all those spanning trees. The idea is the following (again assuming that the tree weights are distinct):

5.5.15. Definition. For an edge $e \in T_1$, we define its *gain* $g(e)$ as the minimum weight gained by exchanging e for another edge. Similarly, we define the gain $G(f)$ for $f \notin T_1$. Put formally:

$$g(e) := \min\{w(f) - w(e) \mid f \in E, e \in T[f]\},$$
$$G(f) := \min\{w(f) - w(e) \mid e \in T[f]\}.$$

5.5.16. Lemma. When $t_1, \ldots, t_{n-1}$ are the edges of T_1 in order of increasing gain, the edges $t_K, \ldots, t_{n-1}$ are present in all trees $T_2, \ldots, T_K$.

Proof. The best exchanges in T_1 involving $t_1, \ldots, t_{K-1}$ produce $K-1$ spanning trees of increasing weights. Any exchange involving $t_K, \ldots, t_n$ produces a tree which is heavier or equal than all those trees. (We are ascertained by the Monotone exchange lemma that the gain of such exchanges need not be reverted later.) ♠

5.5.17. Lemma. When $q_1, \ldots, q_{m-n+1}$ are the edges of $G \setminus T_1$ in order of increasing gain, the edges $q_K, \ldots, q_{m-n+1}$ are not present in any of $T_2, \ldots, T_K$.

Proof. Similar to the previous lemma. ♠

5.5.18. It is therefore sufficient to find $T_2, \ldots, T_K$ in the graph obtained from G by contracting the edges $t_K, \ldots, t_n$ and deleting $q_K, \ldots, q_{m-n+1}$. This graph has only $\mathcal{O}(K)$ vertices and $\mathcal{O}(K)$ edges. The only remaining hurdle is how to calculate the gains. For edges outside T_1, it again suffices to find the peaks of the covered paths. The gains of MST edges require a different algorithm, but Tarjan [Tar79] has shown how to obtain them in time $\mathcal{O}(m\,\alpha(m,n))$.

When we put the results of this section together, we can conclude:

5.5.19. Theorem. *(Finding K lightest spanning trees)*
For a given graph G with real edge weights and a positive integer K, the K best spanning trees can be found in time $\mathcal{O}(m\,\alpha(m,n) + \min(K^2, Km + K \log K))$.

Proof. First we find the MST of G in time $\mathcal{O}(m\,\alpha(m,n))$ using the Pettie's Optimal MST algorithm (Theorem 4.4.6). Then we calculate the gains of MST edges by the Tarjan's algorithm from [Tar79], again in $\mathcal{O}(m\,\alpha(m,n))$, and the gains of the other edges using our MST verification algorithm (Corollary 3.4.14) in $\mathcal{O}(m)$. We use Lemma 5.5.16 to identify edges that are required, and Lemma 5.5.17 to find edges that are superfluous. We contract the former edges, remove the latter ones and run Algorithm 5.5.10 to find the spanning trees. By Lemma 5.5.11, it runs in the desired time.

If $K \geq m$, this reduction does not pay off, so we run Algorithm 5.5.10 directly on the input graph. ♠

5.5.20. *Improvements.* It is an interesting open question whether the algorithms of Section 3.4 can be modified to calculate all gains in linear time. The main procedure could be, but it requires having the input reduced to a balance tree beforehand and here the Borůvka trees fail. The Buchsbaum's Pointer-Machine algorithm (3.4.15) seems to be more promising.

5.5.21. *Large K.* When K is large, re-running the verification algorithm for every change of the graph is too costly. Frederickson [Fre97] has shown how to find the best swaps dynamically, reducing the overall time complexity of Algorithm 5.5.10 to $\mathcal{O}(Km^{1/2})$ and improving the bound in Theorem 5.5.19 to $\mathcal{O}(m\,\alpha(m,n) + \min(K^{3/2}, Km^{1/2}))$. It is open if the dynamic data structures of this chapter could be modified to bring the complexity of finding the next tree down to polylogarithmic.

5.5.22. *Multiple minimum trees.* Another nice application of Theorem 5.5.19 is finding all minimum spanning trees in a graph that does not have distinct edge weights. We find a single MST using any of the algorithms of the previous chapters and then we use the enumeration algorithm of this section to find further spanning trees as long as their weights are equal to the minimum.

We can even use the reduction of the number of edges from Lemmata 5.5.16 and 5.5.17: we start with some fixed K and when we exhaust all K trees, we double K and restart the whole process. The extra time spent on these restarts is dominated by the time of the final pass.

This finally settles the question that we have asked ourselves in Section 1.2, namely whether we lose anything by assuming that all weights are distinct and by searching for just the single minimum tree.

6. Applications

6.1. Special cases and related problems

Towards the end of our story of the minimum spanning trees, we will now focus our attention on various special cases of the MST problem and also to several related problems that frequently arise in practice.

6.1.1. *Graphs with sorted edges.* When the edges of the given graph are already sorted by their weights, we can use the Kruskal's algorithm to find the MST in time $\mathcal{O}(m\,\alpha(n))$ (Theorem 1.4.21). We can however do better: As the minimality of a spanning tree depends only on the order of weights and not on the actual values (The Minimality Theorem, 1.2.6), we can renumber the weights to $1, \ldots, m$ and find the MST using the Fredman-Willard algorithm for integer weights. According to Theorem 3.2.20 it runs in time $\mathcal{O}(m)$ on the Word-RAM.

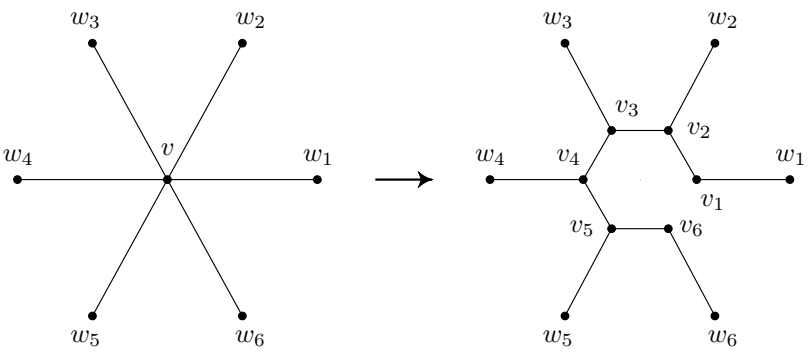
6.1.2. *Graphs with a small number of distinct weights.* When the weights of edges are drawn from a set of a fixed size U, we can sort them in linear time and so reduce the problem to the previous case. A more practical way is to use the Jarník's algorithm (1.4.11), but replace the heap by an array of U buckets. As the number of buckets is constant, we can find the minimum in constant time and hence the whole algorithm runs in time $\mathcal{O}(m)$, even on the Pointer Machine. For large values of U, we can build a binary search tree or the van Emde-Boas tree (see Section 2.3 and [vEB77]) on the top of the buckets to bring the complexity of finding the minimum down to $\mathcal{O}(\log U)$ or $\mathcal{O}(\log \log U)$ respectively.

6.1.3. *Graphs with floating-point weights.* A common case of non-integer weights are rational numbers in floating-point (FP) representation. Even in this case we will be able to find the MST in linear time. The most common representation of binary FP numbers specified by the IEEE standard 754-1985 [IEE85] has a useful property: When the bit strings encoding non-negative FP numbers are read as ordinary integers, the order of these integers is the same as of the original FP numbers. We can therefore once again replace the edge weights by integers and use the linear-time integer algorithm. While the other FP representations (see [Gol91] for an overview) need not have this property, the corresponding integers can be adjusted in $\mathcal{O}(1)$ time to the format we need using bit masking. (More advanced tricks of this type have been employed by Thorup [Tho00a] to extend his linear-time algorithm for single-source shortest paths to FP edge lengths.)

6.1.4. *Graphs with bounded degrees.* For graphs with vertex degrees bounded by a constant Δ, the problem is either trivial (if $\Delta < 3$) or as hard as for arbitrary graphs. There is a simple linear-time transform of arbitrary graphs to graphs with maximum degree 3 which preserves the MST:

6.1.5. Lemma. *(Degree reduction)*
For every graph G there exists a graph G' with maximum degree at most 3 and a function $\pi : E(G) \to E(G')$ such that $\mathrm{mst}(G) = \pi^{-1}(\mathrm{mst}(G'))$. The graph G' and the embedding π can be constructed in time $\mathcal{O}(m)$.

Proof. We show how to eliminate a single vertex v of degree $d > 3$ and then apply induction.



Degree reduction in Lemma 6.1.5

Assume that v has neighbors $w_1, \ldots, w_d$. We replace v and the edges vw_i by d new vertices $v_1, \ldots, v_d$, joined by a path $v_1 v_2 \ldots v_d$, and edges $v_i w_i$. Each edge of the path will receive a weight smaller than all original weights, the other edges will inherit the weights of the edges vw_i they replace. The edges of the path will therefore lie in the MST (this is obvious from the Kruskal's algorithm) and as G can be obtained from the new G' by contracting the path, the rest follows from the Contraction lemma (1.5.10).

This step can be carried out in time $\mathcal{O}(d)$. It replaces a high-degree vertex by vertices of degree 3 and it does not change degrees of any other vertices. So the whole procedure stops in at most n such steps, so it takes time $\mathcal{O}(\sum_{v \in V} \deg(v)) = \mathcal{O}(m)$, including the time needed to find the high-degree vertices at the beginning. ♠

6.1.6. *Euclidean MST.* The MST also has its counterparts in the realm of geometric algorithms. Suppose that we have n points $x_1, \ldots, x_n$ in the plane and we want to find the shortest system of segments connecting these points. If we want the segments to touch only in the given points, this is equivalent to finding the MST of the complete graph on the vertices $V = \{x_1, \ldots, x_n\}$ with edge weights defined as the Euclidean distances of the points. Since the graph is dense, many of the MST algorithms discussed run in linear time with the size of the graph, hence in time $\mathcal{O}(n^2)$.

There is a more efficient method based on the observation that the MST is always a subgraph of the Delaunay's tesselation for the given points (this was first noted by Shamos and Hoey [SH75]). The tesselation is a planar graph, which guarantees that it has $\mathcal{O}(n)$ edges, and it is a dual graph of the Voronoi diagram of the given points, which can be constructed in time $\mathcal{O}(n \log n)$ using for example the Fortune's algorithm [For87]. We can therefore reduce the problem to finding the MST of the tesselation for which $\mathcal{O}(n \log n)$ time is more than sufficient.

This approach fails for non-Euclidean metrics, but in some cases (in particular for the rectilinear metric) the $\mathcal{O}(n \log n)$ time bound is also achievable by the algorithm of Zhou et al. [ZSN02] based on the sweep-line technique and the Red rule. For other variations on the geometric MST, see Eppstein's survey paper [Epp96].

6.1.7. *Steiner trees.* The constraint that the segments in the previous example are allowed to touch each other only in the given points looks artificial and it is indeed uncommon in practical applications (including the problem of designing electrical transmission lines originally studied by Borůvka). If we lift this restriction, we get

the problem known by the name Steiner tree.[1] We can also define it in terms of graphs:

6.1.8. Definition. A *Steiner tree* of a weighted graph (G, w) with a set $M \subseteq V$ of *mandatory vertices* is a tree $T \subseteq G$ that contains all the mandatory vertices and its weight is minimum possible.

When $M = V$, the Steiner tree is identical to the MST, but if we allow an arbitrary choice of the mandatory vertices, it is NP-hard. This has been proven by Garey and Johnson [GGJ77, GJ77] for not only the graph version with weights $\{1, 2\}$, but also for the planar version with Euclidean or rectilinear metric. There is a polynomial-time approximation algorithm with ratio 5/3 for graphs due to Prömel and Steger [PS00] and a polynomial-time approximation scheme for the Euclidean Steiner tree in an arbitrary dimension by Arora [Aro98].

6.1.9. *Approximating the weight of the MST.* Sometimes we are not interested in the actual edges forming the MST and only the weight matters. If we are willing to put up with a randomized approximation, we can even achieve sub-linear complexity. Chazelle et al. [CRT05] have shown an algorithm which, given $0 < \varepsilon < 1/2$, approximates the weight of the MST of a graph with average degree d and edge weights from the set $\{1, \ldots, w\}$ in time $\mathcal{O}(dw\varepsilon^{-2} \cdot \log(dw/\varepsilon))$, producing a weight which has relative error at most ε with probability at least 3/4. They have also proven a close lower bound $\Omega(dw\varepsilon^{-2})$.

For the d-dimensional Euclidean case, there is a randomized approximation algorithm by Czumaj et al. [CEF+03] which with high probability produces a spanning tree within relative error ε in $\widetilde{\mathcal{O}}(\sqrt{n} \cdot \mathrm{poly}(1/\varepsilon))$[2] queries to a data structure containing the points. The data structure is expected to answer orthogonal range queries and cone approximate nearest neighbor queries. There is also an $\widetilde{\mathcal{O}}(n \cdot \mathrm{poly}(1/\varepsilon))$ time approximation algorithm for the MST weight in arbitrary metric spaces by Czumaj and Sohler [CS04]. (This is still sub-linear since the corresponding graph has roughly n^2 edges.)

6.2. Practical algorithms

So far, we were studying the theoretical aspects of the MST algorithms. How should we find the MST on a real computer?

Moret and Shapiro [MS94] have conducted an extensive experimental study of performance of implementations of various MST algorithms on different computers. They have tested the algorithms on several sets of graphs, varying in the number of vertices (up to millions) and in edge density (from constant to $n^{1/2}$. In almost all tests, the winner was an ordinary Prim's algorithm implemented with pairing heaps [FSST86]. The pairing heaps are known to perform surprisingly well in practice, but they still elude attempts at complete theoretical analysis. So far the best results are those of Pettie [Pet05], who has proven that deletion of the minimum takes $\mathcal{O}(\log n)$ time and all other operations take $\mathcal{O}(2^{2\sqrt{\log\log n}})$; both bounds are amortized.

[1] It is named after the Swiss mathematician Jacob Steiner who studied a special case of this problem in the 19th century.

[2] $\widetilde{\mathcal{O}}(f) = \mathcal{O}(f \cdot \log^{\mathcal{O}(1)} f)$ and $\mathrm{poly}(n) = n^{\mathcal{O}(1)}$.

The Moret's study however completely misses variants of the Borůvka's algorithm and many of those have very promising characteristics, especially for planar graphs and minor-closed classes.

Also, Katriel et al. [KST03] have proposed a new algorithm based on the Red rule. It is a randomized algorithm which uses a simplified form of the idea of edge filtering from the algorithm of Karger, Klein and Tarjan (see Section 3.5). The expected time complexity is slightly worse: $\mathcal{O}(n \log n + m)$. However, for dense graphs it performs very well in practice, even in comparison with the Moret's results.

6.2.1. *Parallel algorithms.* Most of the early parallel algorithms for the MST are variants of the Borůvka's algorithm. The operations on the individual trees are independent of each other, so they can be carried out in parallel. There are $\mathcal{O}(\log n)$ steps and each of them can be executed in $\mathcal{O}(\log n)$ parallel time using standard PRAM techniques (see [JáJ92] for the description of the model).

Several better algorithms have been constructed, the best of which run in time $\mathcal{O}(\log n)$. Chong, Han and Lam [CHL01] have recently discovered an algorithm that achieves this time complexity even on the EREW PRAM — the weakest of the parallel RAM's which does not allow concurrent reading nor writing to the same memory cell by multiple processors. In this model, the $\mathcal{O}(\log n)$ bound is clearly the best possible.

As in the sequential models, the basic question still remains open: Is it possible to compute the MST in parallel on EREW PRAM, spending only linear work? This would of course imply existence of a linear-time sequential algorithm, so none of the known parallel algorithms achieve that. Algorithms with linear work can be obtained by utilizing randomness, as shown for example by Pettie and Ramachandran [PR02a], but so far only at the expense of higher time complexity.

7. Ranking Combinatorial Structures

7.1. Ranking and unranking

The techniques for building efficient data structures on the RAM, which we have described in Chapter 2, can be also used for a variety of problems related to ranking of combinatorial structures. Generally, the problems are stated in the following way:

7.1.1. Definition. Let C be a set of objects and $\prec$ a linear order on C. The *rank* $R_{C,\prec}(x)$ of an element $x \in C$ is the number of elements $y \in C$ such that $y \prec x$. We will call the function $R_{C,\prec}$ the *ranking function* for C ordered by $\prec$ and its inverse $R^{-1}_{C,\prec}$ the *unranking function* for C and $\prec$. When the set and the order are clear from the context, we will use plain $R(x)$ and $R^{-1}(x)$. Also, when $\prec$ is defined on a superset C' of C, we naturally extend $R_C(x)$ to elements $x \in C' \setminus C$.

7.1.2. Example. Let us consider the set $C_k = \{\mathbf{0}, \mathbf{1}\}^k$ of all binary strings of length k ordered lexicographically. Then $R^{-1}(i)$ is the i-th smallest element of this set, that is the number i written in binary and padded to k digits (i.e., $\langle i \rangle_k$ in the notation of Section 2.4). Obviously, $R(x)$ is the integer whose binary representation is the string x.

7.1.3. In this chapter, we will investigate how to compute the ranking and unranking functions for different sets efficiently. Usually, we will observe that the ranks (and hence the input and output of our algorithm) are large numbers, so we can use the integers of a similar magnitude to represent non-trivial data structures.

7.1.4. Until the end of the chapter, we will always assume that our model of computation is the Random Access Machine (more specifically, the Word-RAM).

7.2. Ranking of permutations

One of the most common ranking problems is ranking of permutations on the set $[n] = \{1, 2, \ldots, n\}$. This is frequently used to create arrays indexed by permutations: for example in Ruskey's algorithm for finding Hamilton cycles in Cayley graphs (see [RJW95] and [RS93]) or when exploring state spaces of combinatorial puzzles like the Loyd's Fifteen [SD06]. Many other applications are surveyed by Critani et al. [CDDB97] and in most cases, the time complexity of the whole algorithm is limited by the efficiency of the (un)ranking functions.

The permutations are usually ranked according to their lexicographic order. In fact, an arbitrary order is often sufficient if the ranks are used solely for indexing of arrays. The lexicographic order however has an additional advantage of a nice structure, which allows various operations on permutations to be performed directly on their ranks.

Naïve algorithms for lexicographic ranking require time $\Theta(n^2)$ in the worst case [Rei77] and even on average [Lie97]. This can be easily improved to $O(n \log n)$ by using either a binary search tree to calculate inversions, or by a divide-and-conquer technique, or by clever use of modular arithmetic (all three algorithms are described in Knuth [Knu98]). Myrvold and Ruskey [MR01] mention further improvements to $O(n \log n / \log \log n)$ by using the RAM data structures of Dietz [Die89].

Linear time complexity was reached by Myrvold and Ruskey [MR01] for a non-lexicographic order, which is defined locally by the history of the data structure — in fact, they introduce a linear-time unranking algorithm first and then they derive an inverse algorithm without describing the order explicitly. However, they leave the problem of lexicographic ranking open.

We will describe a general procedure which, when combined with suitable RAM data structures, yields a linear-time algorithm for lexicographic (un)ranking.

7.2.1. Notation. We will view permutations on a finite set $A \subseteq \mathbb{N}$ as ordered $|A|$-tuples (in other words, arrays) containing every element of A exactly once. We will use square brackets to index these tuples: $\pi = (\pi[1], \ldots, \pi[|A|])$, and sub-tuples: $\pi[i \ldots j] = (\pi[i], \pi[i+1], \ldots, \pi[j])$. The lexicographic ranking and unranking functions for the permutations on A will be denoted by $L(\pi, A)$ and $L^{-1}(i, A)$ respectively.

7.2.2. Observation. Let us first observe that permutations have a simple recursive structure. If we fix the first element $\pi[1]$ of a permutation π on the set $[n]$, the elements $\pi[2], \ldots, \pi[n]$ form a permutation on $[n] - \{\pi[1]\} = \{1, \ldots, \pi[1]-1, \pi[1]+1, \ldots, n\}$. The lexicographic order of two permutations π and π' on the original set is then determined by $\pi[1]$ and $\pi'[1]$ and only if these elements are equal, it is decided by the lexicographic comparison of permutations $\pi[2 \ldots n]$ and $\pi'[2 \ldots n]$. Moreover, when we fix $\pi[1]$, all permutations on the smaller set occur exactly once, so the rank of π is $(\pi[1]-1) \cdot (n-1)!$ plus the rank of $\pi[2 \ldots n]$.

This gives us a reduction from (un)ranking of permutations on $[n]$ to (un)ranking of permutations on a $(n-1)$-element set, which suggests a straightforward algorithm, but unfortunately this set is different from $[n-1]$ and it even depends on the value of $\pi[1]$. We could renumber the elements to get $[n-1]$, but it would require linear time per iteration. To avoid this, we generalize the problem to permutations on subsets of $[n]$. For a permutation π on a set $A \subseteq [n]$ of size m, similar reasoning gives a simple formula:

$$L((\pi[1], \ldots, \pi[m]), A) = R_A(\pi[1]) \cdot (m-1)! + L((\pi[2], \ldots, \pi[m]), A \setminus \{\pi[1]\}),$$

which uses the ranking function R_A for A. This recursive formula immediately translates to the following recursive algorithms for both ranking and unranking (described for example in [Knu98]):

7.2.3. Algorithm. *Rank*(π, i, n, A): Compute the rank of a permutation $\pi[i \ldots n]$ on A.

1. If $i \geq n$, return 0.
2. $a \leftarrow R_A(\pi[i])$.
3. $b \leftarrow$ *Rank*$(\pi, i+1, n, A \setminus \{\pi[i]\})$.
4. Return $a \cdot (n-i)! + b$.

We can call *Rank*$(\pi, 1, n, [n])$ for ranking on $[n]$, i.e., to calculate $L(\pi, [n])$.

7.2.4. Algorithm. *Unrank*(j, i, n, A): Return an array π such that $\pi[i \ldots n]$ is the j-th permutation on A.

1. If $i > n$, return $(0, \ldots, 0)$.

2. $x \leftarrow R_A^{-1}(\lfloor j/(n-i)! \rfloor)$.
3. $\pi \leftarrow \mathit{Unrank}(j \bmod (n-i)!, i+1, n, A \setminus \{x\})$.
4. $\pi[i] \leftarrow x$.
5. Return π.

We can call $\mathit{Unrank}(j, 1, n, [n])$ for the unranking problem on $[n]$, i.e., to calculate $L^{-1}(j, [n])$.

7.2.5. *Representation of sets.* The most time-consuming parts of the above algorithms are of course operations on the set A. If we store A in a data structure of a known time complexity, the complexity of the whole algorithm is easy to calculate:

7.2.6. Lemma. Suppose that there is a data structure maintaining a subset of $[n]$ under a sequence of deletions, which supports ranking and unranking of elements, and that the time complexity of a single operation is at most $t(n)$. Then lexicographic ranking and unranking of permutations can be performed in time $\mathcal{O}(n \cdot t(n))$.

Proof. Let us analyse the above algorithms. The depth of the recursion is n and in each nested invocation of the recursive procedure we perform a constant number of operations. All of them are either trivial, or calculations of factorials (which can be precomputed in $\mathcal{O}(n)$ time), or operations on the data structure. ♠

7.2.7. Example. If we store A in an ordinary array, we have insertion and deletion in constant time, but ranking and unranking in $\mathcal{O}(n)$, so $t(n) = \mathcal{O}(n)$ and the algorithm is quadratic. Binary search trees give $t(n) = \mathcal{O}(\log n)$. The data structure of Dietz [Die89] improves it to $t(n) = O(\log n / \log\log n)$. In fact, all these variants are equivalent to the classical algorithms based on inversion vectors, because at the time of processing $\pi[i]$, the value of $R_A(\pi[i])$ is exactly the number of elements forming inversions with $\pi[i]$.

7.2.8. To obtain linear time complexity, we will make use of the representation of vectors by integers on the RAM as developed in Section 2.4, but first of all, we will make sure that the ranks are large numbers, so the word size of the machine has to be large as well:

7.2.9. Observation. $\log n! = \Theta(n \log n)$, therefore the word size must be $\Omega(n \log n)$.

Proof. We have $n^n \geq n! \geq \lfloor n/2 \rfloor^{\lfloor n/2 \rfloor}$, so $n \log n \geq \log n! \geq \lfloor n/2 \rfloor \cdot \log \lfloor n/2 \rfloor$. ♠

Thus we get the following theorem:

7.2.10. Theorem. *(Lexicographic ranking of permutations)*
When we order the permutations on the set $[n]$ lexicographically, both ranking and unranking can be performed on the RAM in time $\mathcal{O}(n)$.

Proof. We will store the elements of the set A in a sorted vector. Each element has $\mathcal{O}(\log n)$ bits, so the whole vector takes $\mathcal{O}(n \log n)$ bits, which by the above observation fits in a constant number of machine words. We know from Algorithm 2.4.7 that ranks can be calculated in constant time in such vectors and that insertions and deletions can be translated to ranks and masking. Unranking, that is indexing of the vector, is masking alone. So we can apply the previous Lemma 7.2.6 with $t(n) = \mathcal{O}(1)$. ♠

7.2.11. Remark. We can also easily derive the non-lexicographic linear-time algorithm of Myrvold and Ruskey [MR01] from our algorithm. We will relax the requirements on the data structure to allow the order of elements to depend on the history of the structure (i.e., on the sequence of deletes performed so far). We can

observe that although the algorithm no longer gives the lexicographic ranks, the unranking function is still an inverse of the ranking function, because the sequence of deletes from A is the same during both ranking and unranking.

The implementation of the relaxed structure is straightforward. We store the set A in an array α and use the order of the elements in α determine the order on A. We will also maintain an "inverse" array α^{-1} such that $\alpha[\alpha^{-1}[x]] = x$ for every $x \in A$. Ranking and unranking can be performed by a simple lookup in these arrays: $R_A(x) = \alpha^{-1}[x]$, $R^{-1}(i) = \alpha[i]$. When we want to delete an element, we exchange it with the last element in the array α and update α^{-1} accordingly.

7.3. Ranking of k-permutations

The ideas from the previous section can be also generalized to lexicographic ranking of *k-permutations,* that is of ordered k-tuples of distinct elements drawn from the set $[n]$. There are $n^{\underline{k}} = n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$ such k-permutations and they have a recursive structure similar to the one of the permutations. We will therefore use the same recursive scheme as before (algorithms 7.2.3 and 7.2.4), but we will modify the first step of both algorithms to stop after the first k iterations. We will also replace the number $(n-i)!$ of permutations on the remaining elements by the number of $(k-i)$-permutations on the same elements, i.e., by $(n-i)^{\underline{k-i}}$. As $(n-i)^{\underline{k-i}} = (n-i) \cdot (n-i-1)^{\underline{k-i-1}}$, we can precalculate all these values in linear time.

Unfortunately, the ranks of k-permutations can be much smaller, so we can no longer rely on the same data structure fitting in a constant number of word-sized integers. For example, if $k = 1$, the ranks are $\mathcal{O}(\log n)$-bit numbers, but the data structure still requires $\Theta(n \log n)$ bits.

We do a minor side step by remembering the complement of A instead, that is the set of the at most k elements we have already seen. We will call this set H (because it describes the "holes" in A). Let us prove that $\Omega(k \log n)$ bits are needed to represent the rank, so the vector representation of H certainly fits in a constant number of words.

7.3.1. Lemma. The number of k-permutations on $[n]$ is $2^{\Omega(k \log n)}$.

Proof. We already know that there $n^{\underline{k}}$ such k-permutations. If $k \leq n/2$, then every term in the product is $n/2$ or more, so $\log n^{\underline{k}} \geq k \cdot (\log n - 1)$. If $k \geq n/2$, then $n^{\underline{k}} \geq n^{\underline{n/2}}$ and $\log n^{\underline{k}} \geq (n/2)(\log n - 1) \geq (k/2)(\log n - 1)$. ♠

7.3.2. It remains to show how to translate the operations on A to operations on H, again stored as a sorted vector $\mathbf{h}$. Insertion to A correspond to deletion from H and vice versa. The rank of any $x \in [n]$ in A is x minus the number of holes that are smaller than x, therefore $R_A(x) = x - R_H(x)$. To calculate $R_H(x)$, we can again use the vector operation *Rank* from Algorithm 2.4.7, this time on the vector $\mathbf{h}$.

The only operation, which we cannot translate directly, is unranking in A. We will therefore define an auxiliary vector $\mathbf{r}$ of the same size as $\mathbf{h}$, containing the ranks of the holes: $r_i = R_A(h_i) = h_i - R_H(h_i) = h_i - i$. To find the j-th smallest element of A, we locate the interval between holes to which this element belongs: the interval is bordered from below by a hole h_i such that i is the largest index satisfying $r_i \leq j$. In other words, $i = \mathit{Rank}(r, j+1) - 1$. Finding the right element in the interval is then easy: $R_A^{-1}(j) = h_i + 1 + j - r_i$.

7.3.3. Example. If $A = \{2, 5, 6\}$ and $n = 8$, then $\mathbf{h} = (1, 3, 4, 7, 8)$ and $\mathbf{r} = (0, 1, 1, 3, 3)$. When we want to calculate $R_A^{-1}(2)$, we find $i = 2$ and the wanted element is $h_2 + 1 + 2 - r_2 = 4 + 1 + 2 - 1 = 6$.

7.3.4. The vector $\mathbf{r}$ can be updated in constant time whenever an element is inserted to $\mathbf{h}$. It is sufficient to shift the fields apart (we know that the position of the new element in $\mathbf{r}$ is the same as in $\mathbf{h}$), insert the new value using masking operations and decrease all higher fields by one in parallel by using a single subtraction. Updates after deletions from $\mathbf{h}$ are analogous.

We have replaced all operations on A by the corresponding operations on the modified data structure, each of which works again in constant time. Therefore we have just proven the following theorem, which brings this section to a happy ending:

7.3.5. Theorem. *(Lexicographic ranking of k-permutations)*
When we order the k-permutations on the set $[n]$ lexicographically, both ranking and unranking can be performed on the RAM in time $\mathcal{O}(k)$.

Proof. We modify algorithms 7.2.3 and 7.2.4 for k-permutations as shown at the beginning of this section. We use the vectors $\mathbf{h}$ and $\mathbf{r}$ described above as an implicit representation of the set A. The modified algorithm uses recursion k levels deep and as each operation on A can be performed in $\mathcal{O}(1)$ time using $\mathbf{h}$ and $\mathbf{r}$, every level takes only constant time. The time bound follows. ♠

7.4. Restricted permutations

Another interesting class of combinatorial objects that can be counted and ranked are restricted permutations. An archetypal member of this class are permutations without a fixed point, i.e., permutations π such that $\pi(i) \neq i$ for all i. These are also called *derangements* or *hatcheck permutations.*[1] We will present a general (un)ranking method for any class of restricted permutations and derive a linear-time algorithm for the derangements from it.

7.4.1. Definition. We will fix a non-negative integer n and use $\mathcal{P}$ for the set of all permutations on $[n]$. A *restriction graph* is a bipartite graph G whose parts are two copies of the set $[n]$. A permutation $\pi \in \mathcal{P}$ satisfies the restrictions if $(i, \pi(i))$ is an edge of G for every i.

7.4.2. *Boards and rooks.* We will follow the path unthreaded by Kaplansky and Riordan [KR46] and charted by Stanley in [Sta00]. We will relate restricted permutations to placements of non-attacking rooks on a hollow chessboard.

7.4.3. Definition.

- A *board* is the grid $B = [n] \times [n]$. It consists of n^2 *squares.*
- A *trace* of a permutation $\pi \in \mathcal{P}$ is the set of squares $T(\pi) = \{(i, \pi(i)); i \in [n]\}$.

7.4.4. Observation. The traces of permutations (and thus the permutations themselves) correspond exactly to placements of n rooks at the board in a way such that the rooks do not attack each other (i.e., there is at most one rook in every row and

[1] As the story in [MN98] goes, once upon a time there was a hatcheck lady who was so confused that she was giving out the hats completely at random. What is the probability that none of the gentlemen receives his own hat?

likewise in every column; as there are n rooks, there must be exactly one of them in every row and column). When speaking about *rook placements,* we will always mean non-attacking placements.

Restricted permutations then correspond to placements of rooks on a board with some of the squares removed. The *holes* (missing squares) correspond to the non-edges of G, so $\pi \in \mathcal{P}$ satisfies the restrictions iff $T(\pi)$ avoids the holes.

7.4.5. Definition. Let $H \subseteq B$ be any set of holes in the board. Then:

- N_j denotes the number of placements of n rooks on the board such that exactly j of the rooks stand on holes. That is:

$$N_j := \left| \left\{ \pi \in \mathcal{P} \;\middle|\; |H \cap T(\pi)| = j \right\} \right|.$$

- r_k is the number of ways how to place k rooks on the holes. In other words, this is the number of k-element subsets of H such that no two elements share a common row or column.
- N is the generating function for the N_j's:

$$N(x) = \sum_{j \geq 0} N_j x^j.$$

As $N_j = 0$ for $j > n$, this function is in fact a finite polynomial.

7.4.6. Theorem. *(The number of restricted permutations, Stanley [Sta00])* The function N can be expressed in terms of the numbers r_k as:

$$N(x) = \sum_{k=0}^{n} r_k \cdot (n-k)! \cdot (x-1)^k.$$

Proof. If two polynomials of degree n coincide at more than n points, they are identical, therefore it is sufficient to prove that the equality holds for all $x \in \mathbb{N}^+$. The $N(x)$ counts the ways of placing n rooks on the board and labeling each of them which stands on a hole with an element of $[x]$. The right-hand side counts the same: We can obtain any such configuration by placing k rooks on H first, labeling them with elements of $\{2, \ldots, x\}$, placing additional $n-k$ rooks on the remaining rows and columns (there are $(n-k)!$ ways how to do this) and labeling those of the new rooks standing on a hole with 1. ♠

7.4.7. Corollary. When we substitute $x = 0$ in the above equality, we get a formula for the number of rook placements avoiding the holes altogether:

$$N_0 = N(0) = \sum_{k=0}^{n} (-1)^k \cdot (n-k)! \cdot r_k.$$

7.4.8. Example. Let us apply this theory to the hatcheck lady problem. The set H of holes is the main diagonal of the board: $H = \{(i,i) \mid i \in [n]\}$. When we want

to place k rooks on the holes, we can do that in $r_k = \binom{n}{k}$ ways. By the previous corollary, the number of derangements is:

$$N_0 = \sum_{k=0}^{n} (-1)^k \cdot (n-k)! \cdot \binom{n}{k} = \sum_{k=0}^{n} (-1)^k \cdot \frac{n!}{k!} = n! \cdot \sum_{k=0}^{n} \frac{(-1)^k}{k!}.$$

As the sum converges to $1/e$ when n approaches infinity, we know that the number of derangements is asymptotically $n!/e$.

7.4.9. *Matchings and permanents.* Placements of n rooks (and therefore also restricted permutations) can be also equated with perfect matchings in the restriction graph G. The edges of the matching correspond to the squares occupied by the rooks, the condition that no two rooks share a row nor column translates to the edges not touching each other, and the use of exactly n rooks is equivalent to the matching being perfect.

There is also a well-known correspondence between the perfect matchings in a bipartite graph and non-zero summands in the formula for the permanent of the bipartite adjacency matrix M of the graph. This holds because the non-zero summands are in one-to-one correspondence with the placements of n rooks on the corresponding board. The number N_0 is therefore equal to the permanent of the matrix M.

We will summarize our observations by the following lemma:

7.4.10. Lemma. The following sets have the same cardinality:

- permutations that obey a given restriction graph G,
- non-attacking placements of rooks on a $n \times n$ board avoiding holes that correspond to non-edges of G,
- perfect matchings in the graph G,
- non-zero summands in the permanent of the adjacency matrix of G.

Proof. Follows from 7.4.4 and 7.4.9. ♠

7.4.11. The diversity of the characterizations of restricted permutations brings both good and bad news. The good news is that we can use the plethora of known results on bipartite matchings. Most importantly, we can efficiently determine whether there exists at least one permutation satisfying a given set of restrictions:

7.4.12. Theorem. There is an algorithm which decides in time $\mathcal{O}(n^{1/2} \cdot m)$ whether there exists a permutation satisfying a given restriction graph. The n and m are the number of vertices and edges of the restriction graph.

Proof. It is sufficient to verify that there exists a perfect matching in the given graph. By a standard technique, this can be reduced in linear time to finding a maximum flow in a suitable unit-capacity network. This flow can be then found using the Dinic's algorithm in time $\mathcal{O}(\sqrt{n} \cdot m)$. (See Dinic [Din70] for the flow algorithm, Even and Tarjan [ET75] for the time bound and Schrijver [Sch03] for more references on flows and matchings.) ♠

7.4.13. The bad news is that computing the permanent is known to be #P-complete even for zero-one matrices (as proven by Valiant [Val79]). As a ranking function for a set of matchings can be used to count all such matchings, we obtain the following theorem:

7.4.14. Theorem. If there is a polynomial-time algorithm for lexicographic ranking of permutations with a set of restrictions which is a part of the input, then P = #P.

Proof. We will show that a polynomial-time ranking algorithm would imply a polynomial-time algorithm for computing the permanent of an arbitrary zero-one matrix, which is a #P-complete problem.

We know from Lemma 7.4.10 that non-zero summands in the permanent of a zero-one matrix M correspond to permutations restricted by a graph G whose bipartite adjacency matrix is M. The permanent is therefore equal to the number of such permutations, which is one more than the rank of the lexicographically maximum such permutation. It therefore remains to show that we can find the lexicographically maximum permutation permitted by G in polynomial time.

We can determine $\pi[1]$ by trying all the possible values permitted by G in decreasing order and stopping as soon as we find $\pi[1]$ which can be extended to a complete permutation. This can be verified using the previous theorem on the graph of the remaining restrictions, i.e., on G with the vertices 1 on one side and $\pi[1]$ on the other side removed. Once we have $\pi[1]$, proceed by finding $\pi[2]$ in the same way, using the reduced graph. This way we construct the whole maximum permutation π in $\mathcal{O}(n^2)$ calls to the verification algorithm. ♠

7.4.15. *Recursive structure.* However, the hardness of computing the permanent is the only obstacle. We will show that whenever we are given a set of restrictions for which the counting problem is easy (and it is also easy for subgraphs obtained by deleting vertices), ranking is easy as well. The key will be once again a recursive structure, similar to the one we have seen in the case of plain permutations in 7.2.2.

7.4.16. Notation. As we will work with permutations on different sets simultaneously, we have to extend our notation accordingly. For every finite set of elements $A \subset \mathbb{N}$, we will consider the set $\mathcal{P}_A$ of all permutations on A, as usually viewed as ordered $|A|$-tuples. The restriction graph will be represented by its adjacency matrix $M \in \{0,1\}^{|A|\times|A|}$ and a permutation $\pi \in \mathcal{P}_A$ satisfies M (conforms to the restrictions) iff $M[i,j] = 1$ whenever $j = R_A(\pi[i]) + 1$.[2] The set of all such π will be denoted by $\mathcal{P}_{A,M}$ and their number (which obviously does not depend on the choice of A) by $N_0(M) = \operatorname{per} M$.

We will also frequently need to delete a row and a column simultaneously from M. This operation corresponds to deletion of one vertex from each part of the restriction graph. We will write $M^{i,j}$ for the matrix M with its i-th row and j-th column removed.

7.4.17. Observation. Let us consider a permutation $\pi \in \mathcal{P}_A$ and $n = |A|$. When we fix the value of the element $\pi[1]$, the remaining elements form a permutation $\pi' = \pi[2 \ldots n]$ on the set $A' = A \setminus \{\pi[1]\}$. The permutation π satisfies the restriction matrix M if and only if $M[1,a] = 1$ for $a = R_A(\pi[1])$ and π' satisfies a restriction matrix $M' = M^{1,a}$. This translates to the following counterparts of algorithms 7.2.3 and 7.2.4:

[2] The +1 is added because matrices are indexed from 1 while the lowest rank is 0.

7.4.18. Algorithm. *Rank*(π, i, n, A, M): Compute the lexicographic rank of a permutation $\pi[i \ldots n] \in \mathcal{P}_{A,M}$.

1. If $i \geq n$, return 0.
2. $a \leftarrow R_A(\pi[i])$.
3. $b \leftarrow C_a = \sum_k N_0(M^{1,k})$ over all k such that $1 \leq k \leq a$ and $M[1,k] = 1$. *(C_a is the number of permutations in $\mathcal{P}_{A,M}$ whose first element lies among the first a elements of A.)*
4. Return $b +$ *Rank*$(\pi, i+1, n, A \setminus \{\pi[i]\}, M^{1,a+1})$.

To calculate the rank of $\pi \in \mathcal{P}_{A,M}$, we call *Rank*$(\pi, 1, |A|, A, M)$.

7.4.19. Algorithm. *Unrank*(j, i, n, A, M): Return an array π such that $\pi[i, \ldots, n]$ is the j-th permutation in $\mathcal{P}_{A,M}$.

1. If $i > n$, return $(0, \ldots, 0)$.
2. Find minimum a such that $C_a > j$ (where C_a is as in *Rank* above).
3. $x \leftarrow R_A^{-1}(a-1)$.
4. $\pi \leftarrow$ *Unrank*$(j - C_{a-1}, i+1, n, A \setminus \{x\}, M^{1,a})$.
5. $\pi[i] \leftarrow x$.
6. Return π.

To find the j-th permutation in $\mathcal{P}_{A,M}$, we call *Unrank*$(j, 1, |A|, A, M)$.

7.4.20. The time complexity of these algorithms will be dominated by the computation of the numbers C_a, which requires a linear amount of calls to N_0 on every level of recursion, and by the manipulation with matrices. Because of this, we do not need any sophisticated data structure for the set A, an ordinary sorted array will suffice. (Also, we cannot use the vector representation blindly, because we have no guarantee that the word size is large enough.)

7.4.21. Theorem. *(Lexicographic ranking of restricted permutations)* Suppose that we have a family of matrices $\mathcal{M} = \{M_1, M_2, \ldots\}$ such that $M_n \in \{0,1\}^{n \times n}$ and it is possible to calculate the permanent of M' in time $\mathcal{O}(t(n))$ for every matrix M' obtained by deletion of rows and columns from M_n. Then there exist algorithms for ranking and unranking in $\mathcal{P}_{A,M_n}$ in time $\mathcal{O}(n^4 + n^2 \cdot t(n))$ if M_n and an n-element set A are given as a part of the input.

Proof. We will combine the algorithms 7.4.18 and 7.4.19 with the supplied function for computing the permanent. All matrices constructed by the algorithm are submatrices of M_n of the required type, so all computations of the function N_0 can be performed in time $\mathcal{O}(t(n))$ each.

The recursion is n levels deep. Every level involves a constant number of (un)ranking operations on A and computation of at most n of the C_a's. Each such C_a can be derived from C_{a-1} by constructing a submatrix of M (which takes $\mathcal{O}(n^2)$ time) and computing its N_0. We therefore spend time $\mathcal{O}(n^2)$ on operations with the set A, $\mathcal{O}(n^4)$ on matrix manipulations and $\mathcal{O}(n^2 \cdot t(n))$ by the computations of the N_0's. ♠

7.4.22. *Approximation.* In cases where efficient evaluation of the permanent is out of our reach, we can consider using the fully-polynomial randomized approximation scheme for the permanent described by Jerrum, Sinclair and Vigoda [JSV04]. They have described a randomized algorithm that for every $\varepsilon > 0$ approximates the value

of the permanent of an $n \times n$ matrix with non-negative entries. The output is within relative error ε from the correct value with probability at least $1/2$ and the algorithm runs in time polynomial in n and $1/\varepsilon$. From this, we can get a similar approximation scheme for the ranks.

7.4.23. *Special restriction graphs.* There are also deterministic algorithms for computing the number of perfect matchings in various special graph families (which imply polynomial-time ranking algorithms for the corresponding families of permutations). If the graph is planar, we can use the Kasteleyn's algorithm [Kas67] based on Pfaffian orientations which runs in time $\mathcal{O}(n^3)$. It has been recently extended to arbitrary surfaces by Yuster and Zwick [YZ07] and sped up to $\mathcal{O}(n^{2.19})$. The counting problem for arbitrary minor-closed classes (cf. Section 3.1) is still open.

7.5. Hatcheck lady and other derangements

The time bound for ranking of general restricted permutations shown in the previous section is obviously very coarse. Its main purpose was to demonstrate that many special cases of the ranking problem can be indeed computed in polynomial time. For most families of restriction matrices, we can do much better. One of the possible improvements is replacing the matrix M by the corresponding restriction graph and instead of copying the matrix at every level of recursion, we can perform local operations on the graph and undo them later. Another useful trick is to calculate the N_0's of the smaller matrices using information already computed for the larger matrices.

These speedups are hard to state formally in general (they depend on the structure of the matrices), so we will concentrate on a specific example instead. We will show that for the derangements one can achieve linear time complexity.

7.5.1. Notation. As we already know, the hatcheck permutations correspond to restriction matrices that contain zeroes only on the main diagonal, and to graphs that are complete bipartite with the matching $\{(i,i) \mid i \in [n]\}$ deleted. For a given order n, we will call this matrix D_n and the graph G_n and we will show that the submatrices of D_n share several nice properties:

7.5.2. Lemma. Let D be a submatrix of D_n obtained by deletion of rows and columns. Then the value of the permanent of D depends only on the size of D and on the number of zero entries in D.

Proof. We know from Lemma 7.4.10 that the permanent counts matchings in the graph G obtained from G_n by removing the vertices corresponding to the deleted rows and columns of D_n. Therefore we can prove the lemma for the number of matchings instead.

As G_n is a complete bipartite graph without edges of a single perfect matching, the graph G must be also complete bipartite with some non-touching edges missing. Two such graphs G and G' are therefore isomorphic if and only if they have the same number of vertices and also the same number of missing edges. As the number of matchings is an isomorphism invariant, the lemma follows. ♠

7.5.3. Remark. There is a clear combinatorial intuition behind this lemma: if we are to count permutations with restrictions placed on z elements and these restrictions are independent, it does not matter how exactly they look like.

7.5.4. Definition. Let $n_0(z, d)$ be the permanent shared by all submatrices as described by the above lemma, which have $d \times d$ entries and exactly z zeroes.

7.5.5. Lemma. The function n_0 satisfies the following recurrence:

$$\begin{aligned} n_0(0, d) &= d!, \\ n_0(d, d) &= d! \cdot \sum_{k=0}^{d} \frac{(-1)^k}{k!}, \qquad (*) \\ n_0(z, d) &= z \cdot n_0(z-1, d-1) + (d-z) \cdot n_0(z, d-1) \quad \text{for } z < d. \end{aligned}$$

Proof. The base cases of the recurrence are straightforward: $n_0(0, d)$ counts the unrestricted permutations on $[d]$, and $n_0(d, d)$ is equal to the number of derangements on $[d]$, which we have already computed in Example 7.4.8. Let us prove the third formula.

We will count the permutations π restricted by a matrix M of the given parameters z and d. As $z < d$, there is at least one position in the permutation for which no restriction applies and by Lemma 7.5.2 we can choose without loss of generality that it is the first position.

If we select $\pi[1]$ from the z restricted elements, the rest of π is a permutation on the remaining elements with one restriction less and there are $n_0(z-1, d-1)$ such permutations. On the other hand, if we use an unrestricted element, all restrictions stay in effect, so there are $n_0(z, d-1)$ ways how to do so. ♠

7.5.6. Lemma. The function n_0 also satisfies the following recurrence:

$$n_0(z-1, d) = n_0(z, d) + n_0(z-1, d-1) \quad \text{for } z > 0,\ d > 0. \qquad (✠)$$

Proof. We will again take advantage of having proven Lemma 7.5.2, which allows us to choose arbitrary matrices with the given parameters. Let us pick a matrix M_z containing z zeroes such that $M_z[1, 1] = 0$. Then define M_{z-1} which is equal to M_z everywhere except $M_{z-1}[1, 1] = 1$.

We will count the permutations $\pi \in \mathcal{P}_d$ satisfying M_{z-1} in two ways. First, there are $n_0(z-1, d)$ such permutations. On the other hand, we can divide the them to two types depending on whether $\pi[1] = 1$. Those having $\pi[1] \neq 1$ are exactly the $n_0(z, d)$ permutations satisfying M_z. The others correspond to permutations $(\pi[2], \ldots, \pi[d])$ on $\{2, \ldots, d\}$ that satisfy $M_z^{1,1}$, so there are $n_0(z-1, d-1)$ of them. ♠

7.5.7. Corollary. For a given n, a table of the values $n_0(z, d)$ for all $0 \leq z \leq d \leq n$ can be precomputed in time $\mathcal{O}(n^2)$.

Proof. Use either recurrence and induction on $z + d$. ♠

7.5.8. Corollary. For every $0 \leq z < d$ we have $n_0(z, d) - n_0(z+1, d) \leq n_0(z, d)/d$.

Proof. According to the recurrence (✠), the difference $n_0(z, d) - n_0(z+1, d)$ is equal to $n_0(z, d-1)$. We can bound this by plugging the trivial inequality $n_0(z, d-1) \leq n_0(z-1, d-1)$ to $(*)$, from which we obtain $n_0(z, d) \geq d \cdot n_0(z, d-1)$. ♠

7.5.9. *The algorithm.* Let us show how to modify the ranking algorithm (7.4.18) using the insight we have gained into the structure of derangements.

The algorithm uses the matrix M only for computing N_0 of its submatrices and we have shown that this value depends only on the order of the matrix and

the number of zeroes in it. We will therefore replace maintenance of the matrix by remembering the number z of its zeroes and the set Z that contains the elements $x \in A$ whose locations are restricted (there is a zero anywhere in the $(R_A(x)+1)$-th column of M). In other words, every $x \in Z$ can appear at all positions in the permutation except one (and these forbidden positions are different for different x's), while the remaining elements of A can appear anywhere.

As we already observed (7.4.8) that the number of derangements on $[n]$ is $\Theta(n!)$, we can again use word-sized vectors to represent the sets A and Z with insertion, deletion, ranking and unranking on them in constant time.

When the algorithm selects a submatrix $M' = M^{1,k}$ for an element x whose rank is $k-1$, this matrix it is described by either by the choice of $z' = z-1$ and $Z' = Z \setminus \{x\}$ (if $x \in Z$) or $z' = z$ and $Z' = Z$ (if $x \notin Z$). All computations of N_0 in the algorithm can be therefore replaced by looking up the appropriate $n_0(z', |A|-1)$ in the precomputed table. Moreover, we can calculate a single C_a in constant time, because all summands are either $n_0(z, |A|-1)$ or $n_0(z-1, |A|-1)$, depending on the set Z. We get:

$$C_a = r \cdot n_0(z-1, |A|-1) + (a-r) \cdot n_0(z, |A|-1),$$

where $r = R_Z(R_A^{-1}(a))$, that is the number of restricted elements among the a smallest ones in A.

All operations at a single level of the *Rank* function now run in constant time, but *Unrank* needs to search among the C_a's to find the first of them which exceeds the given rank. We could use binary search, but that would take $\Theta(\log n)$ time. There is however a clever trick: the value of C_a does not vary too much with the set Z. Specifically, by Corollary 7.5.8 the difference between the values for $Z = \emptyset$ and $Z = A$ is at most $n_0(z-1, |A|-1)$. It is therefore sufficient to just divide the rank by $n_0(z-1, |A|-1)$ and we get either the correct value of a or one more. Both possibilities can be checked in constant time.

We can therefore conclude this section by the following theorem:

7.5.10. Theorem. *(Ranking of derangements)*
For every n, the derangements on the set $[n]$ can be ranked and unranked according to the lexicographic order in time $\mathcal{O}(n)$ after spending $\mathcal{O}(n^2)$ on initialization of auxiliary tables.

Proof. We modify the general algorithms for (un)ranking of restricted permutations (7.4.18 and 7.4.19) as described above (7.5.9). Each of the n levels of recursion will then run in constant time. The values n_0 will be looked up in a table precalculated in quadratic time as shown in Corollary 7.5.7. ♠

8. Epilogue

We have seen the many facets of the minimum spanning tree problem. It has turned out that while the major question of the existence of a linear-time MST algorithm is still open, backing off a little bit in an almost arbitrary direction leads to a linear solution. This includes classes of graphs with edge density at least $\lambda_k(n)$ for an arbitrary fixed k, minor-closed classes, and graphs whose edge weights are integers. Using randomness also helps, as does having the edges pre-sorted.

If we do not know anything about the structure of the graph and we are only allowed to compare the edge weights, we can use the Pettie's MST algorithm. Its time complexity is guaranteed to be asymptotically optimal, but we do not know what it really is — the best what we have is an $\mathcal{O}(m\,\alpha(m,n))$ upper bound and the trivial $\Omega(m)$ lower bound.

One thing we however know for sure. The algorithm runs on the weakest of our computational models —the Pointer Machine— and its complexity is linear in the minimum number of comparisons needed to decide the problem. We therefore need not worry about the details of computational models, which have contributed so much to the linear-time algorithms for our special cases. Instead, it is sufficient to study the complexity of MST decision trees. However, aside from the properties mentioned in Section 4.3, not much is known about these trees so far.

As for the dynamic algorithms, we have an algorithm which maintains the minimum spanning forest within poly-logarithmic time per operation. The optimum complexity is once again undecided — the known lower bounds are very far from the upper ones. The known algorithms run on the Pointer machine and we do not know if using a stronger model can help.

For the ranking problems, the situation is completely different. We have shown linear-time algorithms for three important problems of this kind. The techniques, which we have used, seem to be applicable to other ranking problems. On the other hand, ranking of general restricted permutations has turned out to balance on the verge of $\#P$-completeness. All our algorithms run on the RAM model, which seems to be the only sensible choice for problems of inherently arithmetic nature. While the unit-cost assumption on arithmetic operations is not universally accepted, our results imply that the complexity of our algorithm is dominated by the necessary arithmetics.

Aside from the concrete problems we have solved, we have also built several algorithmic techniques of general interest: the unification procedures using pointer-based bucket sorting (Section 2.2) and the vector computations on the RAM (Section 2.4). We hope that they will be useful in many other algorithms.

A. Notation

A.1. Symbols

$A(x, y)$ ..... Ackermann's function *(A.3.1)*
$A(x)$ ..... diagonal Ackermann's function *(A.3.1)*
AND ..... bitwise conjunction: $(x \text{ AND } y)[i] = 1$ iff $x[i] = 1 \wedge y[i] = 1$
C_k ..... cycle on k vertices
$\mathcal{D}(G)$ ..... optimal MSF decision tree for a graph G *(4.3.1)*
$D(G)$ ..... depth of $\mathcal{D}(G)$ *(4.3.1)*
$D(m, n)$ .... decision tree complexity of MSF for m edges and n vertices *(4.3.1)*
D_n ..... $n \times n$ matrix with 0's on the main diagonal and 1's elsewhere *(7.5.1)*
$\deg_G(v)$ .... degree of vertex v in graph G; we omit G if it is clear from context
$E(G)$ ..... set of edges of a graph G
E ..... $E(G)$ when the graph G is clear from context
$\mathbb{E}X$ ..... expected value of a random variable X
K_k ..... complete graph on k vertices
$L(\pi, A)$ .... lexicographic ranking function for permutations on a set $A \subseteq \mathbb{N}$ *(7.2.1)*
$L^{-1}(i, A)$ ... lexicographic unranking function, the inverse of L *(7.2.1)*
$\log n$ ..... binary logarithm of the number n
$\log^* n$ ..... iterated logarithm: $\log^* n := \min\{i \mid \log^{(i)} n \leq 1\}$; the inverse of $2 \uparrow n$
$LSB(x)$ .... position of the lowest bit set in x *(2.4.9)*
$MSB(x)$ ... position of the highest bit set in x *(2.4.9)*
MSF ..... minimum spanning forest *(1.1.2)*
$\text{msf}(G)$ .... the unique minimum spanning forest of a graph G *(1.2.8)*
MST ..... minimum spanning tree *(1.1.2)*
$\text{mst}(G)$ .... the unique minimum spanning tree of a graph G *(1.2.8)*
$m(G)$ ..... number of edges of a graph G, that is $|E(G)|$
m ..... $m(G)$ when the graph G is clear from context
$\mathbb{N}$ ..... set of all non-negative integers
$\mathbb{N}^+$ ..... set of all positive integers
$N_0(M)$ ..... number of permutations satisfying the restrictions M *(7.4.16)*
$n(G)$ ..... number of vertices of a graph G, that is $|V(G)|$
n ..... $n(G)$ when the graph G is clear from context
NOT ..... bitwise negation: $(\text{NOT } x)[i] = 1 - x[i]$
$\mathcal{O}(g)$ ..... asymptotic O: $f = \mathcal{O}(g)$ iff $\exists c > 0 : f(n) \leq g(n)$ for all $n \geq n_0$
$\widetilde{\mathcal{O}}(g)$ ..... $f = \widetilde{\mathcal{O}}(g)$ iff $f = \mathcal{O}(g \cdot \log^{\mathcal{O}(1)} g)$
OR ..... bitwise disjunction: $(x \text{ OR } y)[i] = 1$ iff $x[i] = 1 \vee y[i] = 1$
$\mathcal{P}_A$ ..... set of all permutations on a set A *(7.4.16)*
$\mathcal{P}_{A,M}$ ..... set of all permutations on A satisfying the restrictions M *(7.4.16)*
$\text{per}\, M$ ..... permanent of a square matrix M
$\text{poly}(n)$ .... $f = \text{poly}(n)$ iff $f = \mathcal{O}(n^c)$ for some c
$\Pr[\varphi]$ ..... probability that a predicate φ is true
$\mathbb{R}$ ..... set of all real numbers
$R_{C,\prec}(x)$ .... rank of x in a set C ordered by $\prec$ *(7.1.1)*

$R^{-1}_{C,\prec}(i)$ .... unrank of i: the i-th smallest element of a set C ordered by $\prec$ *(7.1.1)*
$V(G)$ ..... set of vertices of a graph G
V ..... $V(G)$ when the graph G is clear from context
W ..... word size of the RAM *(2.1.2)*
$w(e)$ ..... weight of an edge e
XOR ..... bitwise non-equivalence: $(x \text{ XOR } y)[i] = 1$ iff $x[i] \neq y[i]$
$\alpha(n)$ ..... diagonal inverse of the Ackermann's function *(A.3.4)*
$\alpha(m,n)$ .... $\alpha(m,n) := \min\{x \geq 1 \mid A(x, 4\lceil m/n \rceil) > \log n\}$ *(A.3.4)*
$\beta(m,n)$ .... $\beta(m,n) := \min\{i \mid \log^{(i)} n \leq m/n\}$ *(3.2.16)*
$\delta_G(U)$ ..... the cut separating $U \subset V(G)$ from $V(G) \setminus U$ *(1.3.8)*
$\delta_G(v)$ ..... edges of a one-vertex cut, i.e., $\delta_G(\{v\})$ *(1.3.8)*
$\Theta(g)$ ..... asymptotic Θ: $f = \Theta(g)$ iff $f = \mathcal{O}(g)$ and $f = \Omega(g)$
$\lambda_i(n)$ ..... inverse of the i-th row of the Ackermann's function *(A.3.4)*
$\varrho(\mathcal{C})$ ..... edge density of a graph class $\mathcal{C}$ *(3.1.9)*
$\Omega(g)$ ..... asymptotic Ω: $f = \Omega(g)$ iff $\exists c > 0 : f(n) \geq g(n)$ for all $n \geq n_0$
$T[u,v]$ ..... the path in a tree T joining vertices u and v *(1.2.1)*
$T[e]$ ..... the path in a tree T joining the endpoints of an edge e *(1.2.1)*
$A \,\Delta\, B$ ..... symetric difference of sets: $(A \setminus B) \cup (B \setminus A)$
$G - e$ ..... graph G with edge e removed
$G + e$ ..... graph G with edge e added
$G[U]$ ..... subgraph induced by a set $U \subset V(G)$
$\binom{X}{k}$ ..... the set of all k-element subsets of a set X
G/e ..... multigraph contraction *(A.2.3)*
$G \,.\, e$ ..... simple graph contraction *(A.2.4)*
G/X, $G \,.\, X$ contraction by a set X of vertices or edges *(A.2.5)*
$f[X]$ ..... function applied to a set: $f[X] := \{f(x) \mid x \in X\}$
$f[e]$ ..... as edges are two-element sets, $f[e]$ maps both endpoints of an edge e
$f^{(i)}$ ..... function f iterated i times: $f^{(0)}(x) := x$, $f^{(i+1)}(x) := f(f^{(i)}(x))$
$2 \uparrow n$ ..... the tower function (iterated exponential): $2\uparrow 0 := 1$, $2\uparrow(n+1) := 2^{2\uparrow n}$
$\langle x \rangle$ ..... number $x \in \mathbb{N}$ written in binary *(2.4.2)*
$\langle x \rangle_b$ ..... $\langle x \rangle$ zero-padded to exactly b bits *(2.4.2)*
$x[i]$ ..... when $x \in \mathbb{N}$: the value of the i-th bit of x *(2.4.2)*
$x[B]$ ..... when $x \in \mathbb{N}$: the values of the bits at positions in the set B *(2.5.13)*
$\pi[i]$ ..... when π is a sequence: the i-th element of π, starting with $\pi[1]$ *(7.2.1)*
$\pi[i \ldots j]$ .... the subsequence $\pi[i], \pi[i+1], \ldots, \pi[j]$
σ^k ..... the string σ repeated k times *(2.4.2)*
$\mathbf{0}$, $\mathbf{1}$ ..... bits in a bit string *(2.4.2)*
$\equiv$ ..... congruence modulo a given number
$\mathbf{x}$ ..... vector with elements $x_1, \ldots, x_d$; x is its bitwise encoding *(2.4.4)*
$x << n$ ..... bitwise shift of x by n positions to the left: $x << n = x \cdot 2^n$
$x >> n$ ..... bitwise shift of x by n positions to the right: $x >> n = \lfloor x/2^n \rfloor$
$[n]$ ..... the set $\{1, 2, \ldots, n\}$ *(7.2)*
$n^{\underline{k}}$ ..... k-th falling factorial power: $n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$ *(7.3)*
$H \preccurlyeq G$ ..... H is a minor of G *(3.1.1)*
$G \Uparrow R$ ..... graph G with edges in R corrupted *(4.2.4)*
R^C ..... $R^C = R \cap \delta(C)$ *(4.2.4)*
$M^{i,j}$ ..... the matrix M with i-th row and j-th column deleted *(7.4.16)*

A.2. Multigraphs and contractions

Since the formalism of multigraphs is not fixed in the literature, we will better define it carefully, following [Die05]:

A.2.1. Definition. A *multigraph* is an ordered triple (V, E, M), where V is the set of vertices, E is the set of edges, taken as abstract objects disjoint with the vertices, and M is a mapping $E \to V \cup \binom{V}{2}$ which assigns to each edge either a pair of vertices or a single vertex (if the edge is a loop).

A.2.2. Notation. When the meaning is clear from the context, we use the standard graph notation even for multigraphs. For example, $xy \in E(G)$ becomes a shorthand for $\exists e \in E(G)$ such that $M(G)(e) = \{x, y\}$. Also, we consider multigraphs with no multiple edges nor loops and simple graphs to be the same objects, although they formally differ.

A.2.3. Definition. Let $G = (V, E, M)$ be a multigraph and $e = xy$ its arbitrary edge. The *(multigraph) contraction of e in G* produces a multigraph $G/e = (V', E', M')$ such that:

$$\begin{aligned} V' &= (V(G) \setminus \{x, y\}) \cup \{v_e\}, \quad \text{where } v_e \text{ is a new vertex,} \\ E' &= E(G) - \{e\}, \\ M'(f) &= \{m(v) \mid v \in M(f)\} \quad \text{for every } f \in E', \text{ and} \\ m(x) &= \begin{cases} v_e & \text{for } v = x, y, \\ v & \text{otherwise.} \end{cases} \end{aligned}$$

We sometimes also need to contract edges in simple graphs. It is equivalent to performing the multigraph contraction and then unifying parallel edges and deleting loops.

A.2.4. Definition. Let $G = (V, E)$ a simple graph and $e = xy$ its arbitrary edge. The *(simple graph) contraction of e in G* produces a graph $G \,.\, e = (V', E')$ such that:

$$\begin{aligned} V' &= (V(G) \setminus \{x, y\}) \cup \{v_e\}, \quad \text{where } v_e \text{ is a new vertex,} \\ E' &= \{\{m(x), m(y)\} \mid xy \in E \wedge m(x) \neq m(y)\}, \\ m(x) &= \begin{cases} v_e & \text{for } v = x, y, \\ v & \text{otherwise.} \end{cases} \end{aligned}$$

A.2.5. Definition. We can also extend the above definitions to contractions of a set of vertices or edges. For $F \subseteq E(G)$, the graph G/F is defined as $(G/f_1)/f_2/\ldots/f_k$ where $f_1, \ldots, f_k$ are the elements of F (the result obviously does not depend on the order of edges). For $U \subseteq V(G)$, we define G/U as the graph with all vertices of U merged to a single vertex, that is $(G \cup U^*)/U^*$, where U^* is the complete graph on U. Similarly for $G \,.\, F$ and $G \,.\, U$.

A.3. Ackermann's function and its inverses

The Ackermann's function is an extremely quickly growing function which has been introduced by Ackermann [Ack28] in the context of computability theory. Its original purpose was to demonstrate that not every recursive function is also primitive recursive. At the first sight, it does not seem related to efficient algorithms at all. Its various inverses however occur in analyses of algorithms and mathematical structures surprisingly often: We meet them in Section 1.4 in the time complexity of the Disjoint Set Union data structure and also in the best known upper bound on the decision tree complexity of minimum spanning trees in Section 4.4. Another important application is in the complexity of Davenport-Schinzel sequences (see Klazar's survey [Kla02]), but as far as we know, these are not otherwise related to the topic of our study.

Various sources differ in the exact definition of both the Ackermann's function and its inverse, but most of these differences are in factors that are negligible in the light of the giant asymptotic growth of the function.[1] We will use the definition by double recursion given by Tarjan [Tar75], which is predominant in the literature on graph algorithms.

A.3.1. Definition. The *Ackermann's function* $A(x, y)$ is a function on non-negative integers defined as follows:

$$\begin{aligned}
&A(0, y) := 2y, \\
&A(x, 0) := 0, \\
&A(x, 1) := 2 \quad \text{for } x \geq 1, \\
&A(x, y) := A(x-1, A(x, y-1)) \quad \text{for } x \geq 1,\ y \geq 2.
\end{aligned}$$

The functions $A(x, \cdot)$ are called the *rows* of $A(x, y)$, similarly $A(\cdot, y)$ are its *columns.*

Sometimes, a single-parameter version of this function is also used. It is defined as the diagonal of the previous function, i.e., $A(x) := A(x, x)$.

A.3.2. Example. We can try evaluating $A(x, y)$ in some points:

$$\begin{aligned}
&A(x, 2) = A(x-1, A(x, 1)) = A(x-1, 2) = A(0, 2) = 4, \\
&A(1, y) = A(0, A(1, y-1)) = 2A(1, y-1) = 2^{y-1}A(1, 1) = 2^y, \\
&A(2, y) = A(1, A(2, y-1)) = 2^{A(2,y-1)} = 2 \uparrow y \ \text{ (the tower of exponentials)}, \\
&A(3, y) = \text{the tower function iterated } y \text{ times}, \\
&A(4, 3) = A(3, A(4, 2)) = A(3, 4) = A(2, A(3, 3)) = A(2, A(2, A(3, 2))) = \\
&\qquad = A(2, A(2, 4)) = 2 \uparrow (2 \uparrow 4) = 2 \uparrow 65536.
\end{aligned}$$

A.3.3. *Inverses.* As common for functions of multiple parameters, there is no single function which could claim the title of the only true Inverse Ackermann's function. The following three functions related to the inverse of the function A are often considered:

[1] To quote Pettie [Pet06]: "In the field of algorithms & complexity, Ackermann's function is rarely defined the same way twice. We would not presume to buck such a well-established precedent. Here is a slight variant."

A.3.4. Definition. The *i-th row inverse* $\lambda_i(n)$ of the Ackermann's function is defined by:

$$\lambda_i(n) := \min\{y \mid A(i,y) > \log n\}.$$

The *diagonal inverse* $\alpha(n)$ is defined by:

$$\alpha(n) := \min\{x \mid A(x) > \log n\}.$$

The two-parameter *alpha function* $\alpha(m,n)$ is defined for $m \geq n$ by:

$$\alpha(m,n) := \min\{x \geq 1 \mid A(x, 4\lceil m/n \rceil) > \log n\}.$$

A.3.5. Example. $\lambda_1(n) = \mathcal{O}(\log \log n)$, $\lambda_2(n) = \mathcal{O}(\log^* n)$, $\lambda_3(n)$ grows even slower and $\alpha(n)$ is asymptotically smaller than $\lambda_i(n)$ for any fixed i.

A.3.6. Observation. It is easy to verify that all the rows are strictly increasing and so are all columns, except the first three columns which are constant. Therefore for a fixed n, $\alpha(m,n)$ is maximized at $m = n$. So $\alpha(m,n) \leq 3$ when $\log n < A(3,4)$, which covers all values of m that are likely to occur in practice.

A.3.7. Lemma. $\alpha(m,n) \leq \alpha(n) + 1$.

Proof. We know that $A(x, 4\lceil m/n \rceil) \geq A(x,4) = A(x-1, A(x,3)) \geq A(x-1, x-1)$, so $A(x, 4\lceil m/n \rceil)$ rises above $\log n$ no later than $A(x-1, x-1)$ does. ♠

A.3.8. Lemma. When i is a fixed constant and $m \geq n \cdot \lambda_i(n)$, then $\alpha(m,n) \leq i$.

Proof. The choice of m guarantees that $A(x, 4\lceil m/n \rceil) \geq A(x, \lambda_i(n))$, which is greater than $\log n$ for all $x \geq i$. ♠

B. Bibliography

[Ack28] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99(1):118–133. Springer Verlag, 1928.

[AGKR02] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest Common Ancestors: A Survey and a new Distributed Algorithm. *Proceedings of the 14th annual ACM Symposium on Parallel algorithms and architectures*, pages 258–264, 2002.

[AHR98] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *FOCS '98: Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 534–543, 1998.

[AK02] V. Arvind and P. P. Kurur. Graph isomorphism is in SPP. In *FOCS 2002: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 743–750, 2002.

[AMT99] A. Andersson, P. B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theoretical Computer Science*, 215(1-2):337–344, 1999.

[Aro98] S. Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753–782, 1998.

[BA95] A. M. Ben-Amram. What is a "Pointer Machine"? *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 26, 1995.

[BFC00] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Theoretical INformatics*, pages 88–94, 2000.

[BFP+73] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[BGS72] M. Beeler, R. W. Gosper, and R. Schroeppel. HAKMEM. Memo 239, Massachusetts Institute of Technology, 1972.

[BKRW98] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *STOC 1998: Proceedings of the 30th annual ACM Symposium on Theory of Computing*, pages 279–288, 1998.

[Bor26a] O. Borůvka. O jistém problému minimálním (About a Certain Minimal Problem). *Práce moravské přírodovědecké společnosti v Brně*, III:37–58, 1926. In Czech with German summary.

[Bor26b] O. Borůvka. Příspěvek k řešení otázky ekonomické stavby elektrovodných sítí (Contribution to the solution of a problem of economical construction of electric networks). *Elektronický obzor*, 15:153–154, 1926. In Czech.

[Bro93] A. Brodnik. Computation of the least significant set bit. In *Proceedings of the 2nd Electrotechnical and Computer Science Conference, Portoroz, Slovenia*, 1993.

[Cay89] A. Cayley. A theorem on trees. *Quarterly Journal of Mathematics*, 23:376–378, 1889.

[CDDB97] F. Critani, M. Dall'Aglio, and G. Di Biase. Ranking and unranking permutations with applications. In *Innovation in Mathematics: Proceedings of Second International Mathematica Symposium*, pages 99–106, 1997.

[CEF+03] A. Czumaj, F. Ergün, L. Fortnow, A. Magen, I. Newman, R. Rubinfeld, and C. Sohler. Sublinear-time approximation of Euclidean minimum spanning tree. In *SODA 2003: Proceedings of the 14th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 813–822, 2003.

[Cha97] B. Chazelle. A faster deterministic algorithm for minimum spanning trees. In *FOCS '97: Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, page 22, 1997.

[Cha98] T. M. Chan. Backwards analysis of the Karger-Klein-Tarjan algorithm for minimum spanning trees. *Information Processing Letters*, 67(6):303–304, 1998.

[Cha00a] B. Chazelle. A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[Cha00b] B. Chazelle. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM*, 47(6):1012–1027, 2000.

[Che52] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations. *The Annals of Mathematical Statistics*, 23(4):493–507. JSTOR, 1952.

[CHL01] K. W. Chong, Y. Han, and T. W. Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *Journal of the ACM*, 48(2):297–323, 2001.

[Cho38] G. Choquet. Etude de certains réseaux de routes. *Comptes-rendus de l'Acadmie des Sciences*, 206:310, 1938. In French.

[CR72] S. A. Cook and R. A. Reckhow. Time-bounded random access machines. In *STOC '72: Proceedings of the fourth annual ACM Symposium on Theory of Computing*, pages 73–80, 1972.

[CRT05] B. Chazelle, R. Rubinfeld, and L. Trevisan. Approximating the Minimum Spanning Tree Weight in Sublinear Time. *SIAM Journal on Computing*, 34:1370, 2005.

[CS04] A. Czumaj and C. Sohler. Estimating the weight of metric minimum spanning trees in sublinear-time. In *STOC 2004: Proceedings of the 36th annual ACM Symposium on Theory of Computing*, pages 175–183, 2004.

[Die89] P. F. Dietz. Optimal algorithms for list indexing and subset rank. In *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, pages 39–46, 1989.

[Die05] R. Diestel. *Graph Theory*. Springer Verlag, 2005.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271. Springer Verlag, 1959.

[Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[DIR99] Y. Dinitz, A. Itai, and M. Rodeh. On an algorithm of Zemlyachenko for subtree isomorphism. *Information Processing Letters*, 70(3):141–146, 1999.

[DRT92] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal of Computing*, 21(6):1184–1192, 1992.

[EGIN97] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification — a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, 1997.

[Eis97] J. Eisner. State-of-the-Art Algorithms for Minimum Spanning Trees: A Tutorial Discussion. Manuscript available online (78 pages), University of Pennsylvania, 1997, *http://cs.jhu.edu/~jason/papers/#ms97*.

[Epp92] D. Eppstein. Finding the k Smallest Spanning Trees. *BIT*, 32(2):237–248. Springer Verlag, 1992.

[Epp96] D. Eppstein. Spanning Trees and Spanners. Technical Report 96-16, Information and Computer Science, University of California, Irvine, 1996.

[ET75] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4(4):507–518, 1975.

[For87] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(1):153–174, 1987.

[Fre85] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal of Computing*, 14:781–798, 1985.

[Fre97] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal on Computing*, 26(2):484–538, 1997.

[FS89] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *STOC '89: Proceedings of the 21st annual ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[FSST86] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[FW90] M. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *FOCS '90: Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 719–725, 1990.

[FW93] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[GGJ77] M. R. Garey, R. L. Graham, and D. S. Johnson. The Complexity of Computing Steiner Minimal Trees. *SIAM Journal on Applied Mathematics*, 32(4):835–859, 1977.

[GGST86] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[GH61] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of SIAM*, 9(4):551–570, 1961.

[GH85] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[GJ77] M. R. Garey and D. S. Johnson. The Rectilinear Steiner Tree Problem is NP-Complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977.

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[Gus98] J. Gustedt. Minimum spanning trees for minor-closed graph classes in parallel. In *STACS '98: Proceedings of the 15th annual Symposium on Theoretical Aspects of Computer Science*, pages 421–431, 1998.

[Hag98] T. Hagerup. Sorting and Searching on the Word RAM. In *STACS '98: Proceedings of the 15th annual Symposium on Theoretical Aspects of Computer Science*, pages 366–398, 1998.

[Hag00] T. Hagerup. Improved Shortest Paths on the Word RAM. In *ICALP 2000: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 61–72, 2000.

[Han02] Y. Han. Deterministic sorting in $\mathcal{O}(n \log\log n)$ time and linear space. In *STOC 2002: Proceedings of the 34th annual ACM Symposium on Theory of Computing*, pages 602–608, 2002.

[HdLT01] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

[HF98] M. R. Henzinger and M. L. Fredman. Lower Bounds for Fully Dynamic Connectivity Problems in Graphs. *Algorithmica*, 22(3):351–362, 1998.

[HK97a] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 594–604, 1997.

[HK97b] M. R. Henzinger and V. King. Fully dynamic 2-edge-connectivity algorithm in polylogarithmic time per operation. Technical note 1997-004, Digital Equipment Corp., Systems Research Center, 1997.

[HK99] M. R. Henzinger and V. King. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *Journal of the ACM*, 46(4):502–516, 1999.

[HKPB07] R. Hariharan, T. Kavitha, D. Panigrahi, and A. Bhalgat. An $\widetilde{\mathcal{O}}(mn)$ Gomory-Hu tree construction algorithm for unweighted graphs. In

STOC 2007: Proceedings of the 39th annual ACM Symposium on Theory of Computing, pages 605–614, 2007.

[HMN+07] K. Horák, M. Mareš, P. Novotný, J. Šimša, J. Švrček, P. Töpfer, and J. Zhouf. *54. ročník matematické olympiády na středních školách (The 54th Year of the Math Olympiad at High Schools)*. Jednota českých matematiků a fyziků, Praha, 2007. In Czech.

[HMP01] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic Dictionaries. *Journal of Algorithms*, 41(1):69–85, 2001.

[Hoa61] C. A. R. Hoare. Algorithm 65: find. *Communications of the ACM*, 4(7):321–322, 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16. British Computer Society, 1962.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[HT02] Y. Han and M. Thorup. Integer Sorting in $\mathcal{O}(n\sqrt{\log\log n})$ Expected Time and Linear Space. In *FOCS 2002: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 135–144, 2002.

[IEE85] IEEE. *IEEE Standard 754-1985 for Binary Floating-point Arithmetic*. IEEE, 1985.

[Int07] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel Corp., 2007.

[ITY95] M. Ito, N. Takagi, and S. Yajima. Efficient Initial Approximation and Fast Converging Methods for Division and Square Root. In *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 2–9, 1995.

[JáJ92] J. JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., 1992.

[Jar30] V. Jarník. O jistém problému minimálním (About a Certain Minimal Problem). *Práce moravské přírodovědecké společnosti v Brně*, VI:57–63, 1930. In Czech.

[JS03] P. Jones and L. Simon. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[JSV04] M. Jerrum, A. Sinclair, and E. Vigoda. A Polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Nonnegative Entries. *Journal of the ACM*, 51(4):671–697, 2004.

[Kar93] D. R. Karger. Random sampling in matroids, with applications to graph connectivity and minimum spanning trees. In *FOCS '93: Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pages 84–93, 1993.

[Kas67] P. W. Kasteleyn. Graph theory and crystal physics. In *Graph Theory and Theoretical Physics*, pages 43–110. Academic Press, London, 1967.

[KIM81] N. Katoh, T. Ibaraki, and H. Mine. An algorithm for finding k minimum spanning trees. *SIAM Journal on Computing*, 10(2):247–255, 1981.

[Kin97] V. King. A Simpler Minimum Spanning Tree Verification Algorithm. *Algorithmica*, 18:263–270, 1997.

[KKT95] D. R. Karger, P. N. Klein, and R. E. Tarjan. A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. *Journal of the ACM*, 42(2):321–328, 1995.

[Kla02] M. Klazar. Generalized Davenport–Schinzel Sequences: Results, problems, and applications. *INTEGERS: The Electronic Journal of Combinatorial Number Theory*, 2(A11), 2002.

[Knu97a] D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.

[Knu97b] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.

[Knu98] D. E. Knuth. *The Art of Computer Programming, Volume 3 (2nd ed.): Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[Kom85] J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[Kos84] A. V. Kostochka. Lower bound of the Hadwiger number of graphs by their average degree. *Combinatorica*, 4(4):307–316, 1984.

[KR46] I. Kaplansky and J. Riordan. The problem of the rooks and its applications. *Duke Mathematical Journal*, 13(2):259–268. Duke University Press, 1946.

[Kru56] J. B. Kruskal, Jr. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[KST03] I. Katriel, P. Sanders, and J. Träff. A practical minimum spanning tree algorithm using the cycle property. In *Algorithms — ESA 2003: 11th European Symposium*, volume 2832 of *Lecture Notes in Computer Science*, pages 679–690. Springer Verlag, 2003.

[Lie97] J. Liebehenschel. Ranking and Unranking of Lexicographically Ordered Words: An Average-Case Analysis. *Journal of Automata, Languages and Combinatorics*, 2(4):227–268, 1997.

[Lov05] L. Lovász. Graph Minor Theory. *Bulletin of the American Mathematical Society*, 43(1):75–86, 2005.

[LRCS01] C. E. Leiserson, R. L. Rivest, T. H. Cormen, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[Mad67] W. Mader. Homomorphieeigenschaften und mittlere Kantendichte von Graphen. *Mathematische Annalen*, 174(4):265–268. Springer Verlag, 1967. In German.

[Mar00] M. Mareš. Dynamické grafové algoritmy (Dynamic Graph Algorithms). Master's thesis, Charles University in Prague, Faculty of Math and Physics, 2000. In Czech.

[Mar04] M. Mareš. Two linear time algorithms for MST on minor closed graph classes. *Archivum Mathematicum*, 40:315–320. Masaryk University, Brno, Czech Republic, 2004.

[Mar07] M. Mareš. Krajinou grafových algoritmů (Through the Landscape of Graph Algorithms). ITI series 2007–330, Institut Teoretické Informatiky, Praha, Czech Republic, 2007. In Czech.

[Mat95] T. Matsui. The Minimum Spanning tree Problem on a Planar Graph. *Discrete Applied Mathematics*, 58:91–94, 1995.

[MN98] J. Matoušek and J. Nešetřil. *Invitation to Discrete Mathematics*. Oxford University Press, 1998.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MR01] W. J. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.

[MS94] B. M. E. Moret and H. D. Shapiro. An Empirical Assessment of Algorithms for Constructing a Minimum Spanning Tree. *Computational Support for Discrete Mathematics: DIMACS Workshop, March 12-14, 1992*. American Mathematical Society, 1994.

[MS07] M. Mareš and M. Straka. Linear-time ranking of permutations. In *Algorithms — ESA 2007: 15th Annual European Symposium*, volume 4698 of *Lecture Notes in Computer Science*, pages 187–193. Springer Verlag, 2007.

[NdM03] J. Nešetřil and P. O. de Mendez. Colorings and Homomorphisms of Minor Closed Classes. In B. Aronov, S. Basu, J. Pach, and M. Sharir, editors, *Discrete and Computational Geometry: The Goodman-Pollack Festschrift*, pages 651–664. Springer Verlag, 2003.

[Neš97] J. Nešetřil. Some remarks on the history of MST-problem. *Archivum Mathematicum*, 33:15–22. Masaryk University, Brno, Czech Republic, 1997.

[NMN01] J. Nešetřil, E. Milková, and H. Nešetřilová. Otakar Borůvka on Minimum Spanning Tree Problem. *Discrete Mathematics*, 233(1–3):3–36, 2001.

[Oka99] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[Oxl92] J. G. Oxley. *Matroid Theory*. Oxford University Press, 1992.

[Pet99] S. Pettie. Finding minimum spanning trees in $\mathcal{O}(m\,\alpha(m,n))$ time. Tech Report TR99-23, University of Texas at Austin, 1999.

[Pet05] S. Pettie. Towards a final analysis of pairing heaps. In *FOCS 2005: Proceedings of the 46rd Annual IEEE Symposium on Foundations of Computer Science*, pages 174–183, 2005.

[Pet06] S. Pettie. An inverse-Ackermann type lower bound for online minimum spanning tree verification. *Combinatorica*, 26(2):207–230, 2006.

[PR02a] S. Pettie and V. Ramachandran. Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms. In *SODA 2002: Proceedings of the 13th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 713–722, 2002.

[PR02b] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:567–574, 1957.

[PS00] H. J. Promel and A. Steger. A new approximation algorithm for the Steiner tree problem with performance ratio 5/3. *Journal of Algorithms*, 36:89–101, 2000.

[Rei77] E. M. Reingold. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall College Div., 1977.

[RJW95] F. Ruskey, M. Jiang, and A. Weston. The Hamiltonicity of directed-Cayley graphs (or: A tale of backtracking). *Discrete Applied Mathematics*, 57:75–83, 1995.

[RS93] F. Ruskey and C. D. Savage. Hamilton Cycles that Extend Transposition Matchings in Cayley Graphs of S_n. *SIAM Journal on Discrete Mathematics*, 6(1):152–166, 1993.

[RS04] N. Robertson and P. D. Seymour. Graph minors: XX. Wagner's Conjecture. *Journal of Combinatorial Theory Series B*, 92(2):325–357, 2004.

[Sch03] A. Schrijver. *Combinatorial Optimization — Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer Verlag, 2003.

[SD06] J. Slocum and S. D. *The 15 Puzzle Book*. The Slocum Puzzle Foundation, Beverly Hills, CA, USA, 2006.

[SH75] M. I. Shamos and D. Hoey. Closest-point problems. *FOCS '75: Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–162, 1975.

[Sol65] M. Sollin. Le trace de canalisation. In C. Berge and A. Ghouilla-Houri, editors, *Programming, Games, and Transportation Networks*. Wiley, New York, 1965. In French.

[ST83] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[Sta00] R. P. Stanley. *Enumerative Combinatorics. Vol. 1 (2nd ed.)*. Cambridge University Press, 2000.

[Tak00] T. Takaoka. Theory of Trinomial Heaps. In *Computing and Combinatorics: 6th Annual International Conference, COCOON 2000*, volume 1858 of *Lecture Notes in Computer Science*, pages 362–372. Springer Verlag, 2000.

[Tar75] R. E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[Tar79] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[Tar83] R. E. Tarjan. *Data structures and network algorithms*, volume 44 of *CMBS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, 1983.

[TC99] T. Takaoka and N. Z. Christchurch. Theory of 2-3 Heaps. In *Computing and Combinatorics: 5th Annual International Conference, COCOON '99*, volume 1627 of *Lecture Notes in Computer Science*, pages 41–50. Springer Verlag, 1999.

[Tho84] A. Thomason. An extremal function for contractions of graphs. *Mathematical proceedings of the Cambridge Philosophical Society*, 95(2):261–265. Cambridge University Press, 1984.

[Tho99] M. Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *Journal of the ACM*, 46(3):362–394, 1999.

[Tho00a] M. Thorup. Floats, Integers, and Single Source Shortest Paths. *Journal of Algorithms*, 35(2):189–201, 2000.

[Tho00b] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *STOC 2000: Proceedings of the 32nd annual ACM Symposium on Theory of Computing*, pages 343–350, 2000.

[Tho03] M. Thorup. On AC^0 Implementations of Fusion Trees and Atomic Heaps. In *SODA 2003: Proceedings of the 14th annual ACM-SIAM Symposium on Discrete algorithms*, pages 699–707, 2003.

[Tho04] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences*, 69(3):330–353, 2004.

[Tur84] G. Turán. Succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.

[Val79] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[vEB77] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[Vui78] J. Vuillemin. A Data Structure for Manipulating Priority Queues. *Communications of the ACM*, 21(4):309–315, 1978.

[YZ07] R. Yuster and U. Zwick. Maximum matching in graphs with an excluded minor. In *SODA 2007: Proceedings of the 18th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 108–117, 2007.

[Zem73] V. N. Zemlayachenko. Determining tree isomorphism. In *Voprosy Kibernetiki, Proceedings of the Seminar on Combinatorial Mathematics*, pages 54–60. Scientific Council of the Complex Problem "Kibernetika", Akad. Nauk SSSR, 1973. In Russian.

[ZSN02] H. Zhou, N. Shenoy, and W. Nicholls. Efficient minimum spanning tree construction without Delaunay triangulation. *Information Processing Letters*, 81(5):271–276, 2002.