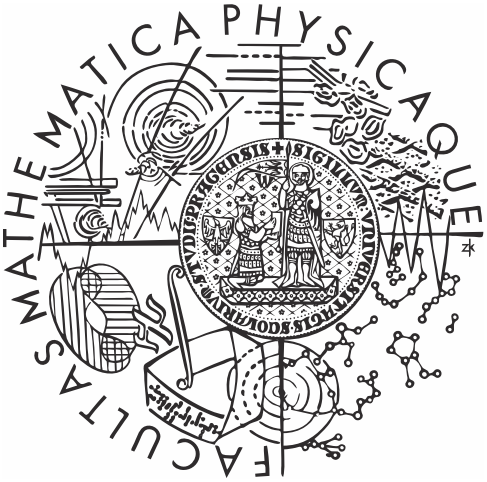
Charles University in Prague
Faculty of Mathematics and Physics



Abstract of 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

1. Introduction

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.

We have tried to cover all known important results on both problems and unite them in a single coherent theory. At many places, we have attempted to contribute our 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.

We 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. 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 we 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.

2. Minimum Spanning Trees

2.1. 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 [Neš97] 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:

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

2.1.2. 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* $w(H) := \sum_{e \in E(H)} w(e)$.
- 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, and he has discovered another efficient algorithm. In the next 50 years, several significantly faster algorithms were published, 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 [PR02] whose time complexity is provably optimal.

Before we discuss the algorithms, let us review the basic properties of spanning trees. We will mostly follow the theory developed by Tarjan in [Tar83] and show that the weights on edges are not necessary for the definition of the MST.

2.1.3. Definition. *(Heavy and light edges)*
Let G be a connected graph with edge weights w and T its spanning tree. Then:

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

- 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.

2.1.4. Theorem. A spanning tree T is minimum iff there is no T-light edge.

2.1.5. Theorem. If all edge weights are distinct, then the minimum spanning tree is unique.

2.1.6. When G is a graph with distinct edge weights, we will use $\mathrm{mst}(G)$ to denote its unique minimum spanning tree. 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, the ties can be broken arbitrarily.

2.2. Classical algorithms

The characterization of MST's in terms of light edges makes it easy to develop the Tarjan's Red-Blue meta-algorithm, which is based on the following properties:

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

2.2.2. 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.

The algorithm repeatedly colors lightest edges of cuts blue and heaviest edges of cycles red. We prove that no matter which order of the colorings we use, the algorithm always stops and the blue edges form the MST.

All three classical MST algorithms (Borůvka's, Jarník's and Kruskal's) can be then obtained as specializations of this procedure. We also calculate the time complexity of standard implementations of these algorithms.

2.2.3. 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, perform a *Borůvka step:*
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.

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

2.2.5. 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.

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

2.2.7. 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.

2.2.8. Theorem. The Kruskal's algorithm finds the MST of the graph given as input in time $\mathcal{O}(m \log n)$. If the edges are already sorted by their weights, the time drops to $\mathcal{O}(m\,\alpha(m,n))$, where $\alpha(m,n)$ is a certain inverse of the Ackermann's function.

2.3. 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. A contractive version of the Borůvka's algorithm is easy to formulate and also to analyse:

2.3.1. 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 edge 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.

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

We also show that this time bound is tight — we construct an explicit family of graphs on which the algorithm spends $\Theta(m \log n)$ steps. Given a planar graph, the algorithm however runs much faster (we get a linear-time algorithm much simpler than the one of Matsui [Mat95]):

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

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:

2.3.4. Lemma. *(Contraction lemma)*
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 $\mathrm{mst}(G) = \pi^{-1}[\mathrm{mst}(G/e)] + e$.

3. Fine Details of Computation

3.1. 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, so we need to state our computation models carefully and develop a repertoire of basic data structures tailor-made for the fine details of the models. In recent decades, most researchers in the area of combinatorial algorithms have been considering the following two computational models, and we will do likewise.

The *Random Access Machine (RAM)* is not a single coherent model, but rather a family of closely related machines (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.) We will consider the variant usually called the *Word-RAM*. It allows the "C-language operators", i.e., arithmetics and bitwise logical operations, running in constant time on words of a specified size.

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

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 formal definition is closely related to the *linking automaton* proposed by Knuth in [Knu97].

3.2. Bucket sorting and related techniques

In the Contractive Borůvka's algorithm, we needed to contract a given set of edges in the current graph and then flatten the graph, all this in time $\mathcal{O}(m)$. This can be easily handled on both the RAM and the PM by bucket sorting. We develop a bunch of pointer-based sorting techniques which can be summarized by the following lemma:

3.2.1. Lemma. 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 $\mathcal{O}(n + \sum_i |S_i|)$.

3.2.2. A direct consequence of this unification is a linear-time algorithm for subtree isomorphism, significantly simpler than the standard one due to Zemlayachenko (see [Zem73] and also Dinitz et al. [DIR99]). When we apply a similar technique to general graphs, we get the framework of topological graph computation of Buchsbaum et al. [BKRW98].

3.2.3. 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.

3.2.4. 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., $\|\mathcal{C}\| = \sum_{G \in \mathcal{C}} n(G) + m(G)$.

3.2.5. Theorem. 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.

3.3. 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 represents 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 (2.2.5) 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 further improved by Han and Thorup [Han02, HT02].

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, which will later lead to a linear-time MST algorithm for arbitrary integer weights. 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. First of all, we observe that we can encode vectors in integers:

3.3.1. 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.

3.3.2. 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.*)

3.3.3. If we want to fit the whole vector in a single machine word, the parameters b and d must satisfy the condition $(b+1)d\le W$ (where W is the word size of the machine). By using multiple-precision arithmetics, we can encode all vectors satisfying $bd=\mathcal{O}(W)$. We describe how to translate simple vector manipulations to sequences of $\mathcal{O}(1)$ RAM operations on their codes. For example, we can handle element-wise comparison of vectors, insertion in a sorted vector or shuffling elements of a vector according to a fixed permutation, all in $\mathcal{O}(1)$ time. This also implies that several functions on numbers can be performed in constant time, most notably binary logarithms. The vector operations then serve as building blocks for construction of the Q-heaps. We get:

3.3.4. Theorem. 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 we can perform Q-heap operations on Q (insertion, deletion, search for a given value and search for the i-th smallest element) 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.

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

4. Advanced MST Algorithms

4.1. Minor-closed graph classes

The contractive algorithm given in Section 2.3 has been found to perform well on planar graphs, but in general its time complexity was not linear. Can we find any broader class of graphs where the linear bound holds? The right context turns out to be the minor-closed classes, which are closed under contractions and have bounded density.

4.1.1. 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.

4.1.2. 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 $\mathcal{C}$.

4.1.3. 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, and graphs of bounded tree-width or path-width.

4.1.4. 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). For analysis of the contractive algorithm, we will make use of the bounded density of minor-closed classes:

4.1.5. 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 $G\in\mathcal{C}$.

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

4.1.7. 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 (2.3.1)

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.)

4.1.8. *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 $\mathcal{O}(\deg(u))$, so we have to make sure that there is a steady supply of low-degree vertices. It indeed is in minor-closed classes:

4.1.9. 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.

This leads to the following algorithm:

4.1.10. 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.

4.1.11. 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.)

4.2. Iterated algorithms

We have seen that the Jarník's Algorithm 2.2.5 runs in $\Theta(m \log n)$ time. Fredman and Tarjan [FT87] have shown a faster implementation using their Fibonacci heaps, which runs in time $\mathcal{O}(m + n \log n)$. This is $\mathcal{O}(m)$ whenever the density of the input graph reaches $\Omega(\log n)$. This suggests that we could combine the algorithm with another MST algorithm, which identifies a subset of the MST edges and contracts them to increase the density of the graph. For example, if we perform several Borůvka steps and then we run the Jarník's algorithm, we find the MST in time $\mathcal{O}(m \log \log n)$.

Actually, there is a much better choice of the algorithms to combine: use the Jarník's algorithm with a Fibonacci heap 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. This improves the time complexity significantly:

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

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

4.2.3. *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 (we can use the Q-heap trees from Section 3.3).

4.2.4. 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.

4.3. 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 the next section.

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].

To verify that a spanning tree T is minimum, it is sufficient to check that all edges outside T are T-heavy. For each edge $uv \in E \setminus T$, we will find the heaviest edge of the tree path $T[u,v]$ (we will call it the *peak* of the path) and compare its weight to $w(uv)$. We have therefore transformed the MST verification to the problem of finding peaks for a set of *query paths* on a given tree. By a sequence of further transformations, we can even assume that the given tree is *complete branching* (all vertices are on the same level and internal vertices always have outdegree 2) and that the query paths join a vertex with one of its ancestors.

Komlós has given a simple algorithm that traverses the complete branching tree recursively. At each moment, it maintains an array of peaks of the restrictions of the query paths to the subtree below the current vertex. If we account for the comparisons performed by this algorithm carefully and express the bound in terms of the size of the original problem (before all the transformations), we get:

4.3.1. 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.

It remains to demonstrate that the overhead of the algorithm needed to find the required comparisons and to infer the peaks from their results can be decreased, so that it gets bounded by the number of comparisons and therefore also by $\mathcal{O}(m)$. We will follow the idea of King from [Kin97], but as we have the power of the RAM data structures from Section 3.3 at our command, the low-level details will be easier. Still, the construction is rather technical, so we omit it from this abstract and state only the final theorem:

4.3.2. 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 $\mathcal{O}(m)$.

4.4. A randomized algorithm

When we analysed the Contractive Borůvka's algorithm in Section 2.3, 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 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:

4.4.1. 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[1] edges in G is at most n/p.

4.4.2. 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.

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

Input: A graph G with an edge comparison oracle.

1. Remove isolated vertices from G. If no vertices remain, stop and return an empty forest.
2. Perform two Borůvka steps (iterations of Algorithm 2.3.1) on G and remember the set B of the edges having been contracted.

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

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.

A careful analysis of this algorithm, based on properties of its recursion tree and on the peak-finding algorithm of the previous section, yields the following time bounds:

4.4.4. Theorem. The KKT algorithm runs in time $\mathcal{O}(\min(n^2, m \log n))$ in the worst case on the RAM. The expected time complexity is $\mathcal{O}(m)$.

5. Approaching Optimality

5.1. Soft heaps

A vast majority of MST algorithms that we have encountered so far is based on the Tarjan's Blue rule (Lemma 2.2.1), the only exception being the randomized KKT algorithm, which also used the Red rule (Lemma 2.2.2). 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 [PR02] 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.

In the thesis, we describe the exact mechanics of the soft heaps and analyse its complexity. The important properties are characterized by the following theorem:

5.1.1. 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.

5.2. 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. We can for example try implanting the soft heap in the Jarník's algorithm, preferably in the earlier version without Fibonacci heaps as the soft heaps lack the *Decrease* operation. This brave, but somewhat simple-minded attempt is however doomed to fail because of corruption of items inside the soft heap. While the basic structural properties of MST's no longer hold in corrupted graphs, 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.

5.2.1. Definition. A subgraph $C \subseteq G$ is *contractible* iff for every pair of edges $e, f \in \delta(C)$[1] 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.

For example, 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. We can now easily reformulate the Contraction lemma (2.3.4) in the language of contractible subgraphs:

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

Let us bring corruption back to the game and state a "robust" version of this lemma.

5.2.3. 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. 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., $R^C = R \cap \delta(C)$).

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

5.2.4. 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 $\mathrm{msf}(G) \subseteq \mathrm{msf}(C) \cup \mathrm{msf}((G/C) \setminus R^C) \cup R^C$.

5.2.5. We will now mimic the Iterated 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.

5.2.6. Theorem. *(Partitioning to contractible clusters, Chazelle [Cha97])*
Given a weighted graph G and parameters ε $(0 < \varepsilon \leq 1/2)$ and t, we can construct 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. $\mathrm{msf}(G) \subseteq \bigcup_i \mathrm{msf}(C_i) \cup \mathrm{msf}\big((G/\bigcup_i C_i) \setminus R^{\mathcal{C}}\big) \cup R^{\mathcal{C}}$.
7. The construction takes $\mathcal{O}(n + m \log(1/\varepsilon))$ time.

5.3. 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. Let us define them first:

5.3.1. 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. 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.

5.3.2. 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.

The downside is that we do not know any explicit construction of the optimal decision trees, nor even 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. Also, we can construct an optimal decision tree using brute force:

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

5.4. 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:

5.4.1. Algorithm. *(Optimal MST algorithm, Pettie and Ramachandran [PR02])*

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 the partitioning procedure (5.2.6) 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 (see Section 4.2).
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, 2.3.1) 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 (5.2.6) and from the proofs of the respective algorithms used as subroutines. As for time complexity, we prove:

5.4.2. Theorem. The time complexity of the Optimal algorithm is $\Theta(D(m,n))$.

5.4.3. *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 has been discovered by Chazelle [Cha00a] and independently by Pettie [Pet99]. It achieves complexity $\mathcal{O}(m\,\alpha(m,n))$. Using any of these results, we can prove an Ackermannian upper bound on the optimal algorithm:

5.4.4. Theorem. The time complexity of the Optimal algorithm is $\mathcal{O}(m\,\alpha(m,n))$.

6. Dynamic Spanning Trees

6.1. 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:

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

- *Init*(n) — Create a graph with n isolated vertices $\{1, \ldots, n\}$. (It is possible to modify the structure to support dynamic addition and removal of vertices, too.)
- *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.

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. This suggests that a dynamic MSF algorithm could be obtained by modifying the mechanics of the data structure to keep the forest minimum. 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. It is easy to prove that $\mathcal{O}(1)$ modifications always suffice, so we can formulate our problem as follows:

6.1.2. 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 msf(G).
- *Delete*(G, e) — Delete an edge specified by its identifier from G. Return the list of additions and deletions of edges in msf(G).

6.1.3. *Incremental MSF.* In case only edge insertions are allowed, the problem reduces to finding the heaviest edge (peak) on the tree path covered by the newly inserted edge and replacing the peak if needed. This can be handled quite efficiently by using the Link-Cut trees of Sleator and Tarjan [ST83]. We obtain logarithmic time bound:

6.1.4. Theorem. *(Incremental MSF)*
When only edge insertions are allowed, the dynamic MSF can be maintained in time $\mathcal{O}(\log n)$ amortized per operation.

6.2. 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.

The implementation uses the Eulerian Tour trees of Henzinger and King [HK99] to represent the forests F_ℓ together with the non-tree edges at each particular level. A simple amortized analysis using the levels yields the following result:

6.2.1. 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.

6.2.2. 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 [Tho00] 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.

6.3. 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.

6.3.1. *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 a weighted version of the ET-trees. 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:

6.3.2. 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.

A brief analysis also shows that the invariant I3 is observed by all operations on the structure. We can conclude:

6.3.3. 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.

6.3.4. *Fully dynamic MSF.* The decremental MSF algorithm can be turned to a fully dynamic one by a blackbox reduction of Holm et al.:

6.3.5. 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:

$$\mathcal{O}\left(\log^3 n + \sum_{i=1}^{\log m} \sum_{j=1}^{i} t(\min(n, 2^j), 2^j)\right) \quad \text{per operation.}$$

6.3.6. 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.

6.3.7. *Dynamic MSF with limited edge weights.* If the set from which the edge weights are drawn is small, we can take a different approach. If only two values are allowed, we split the graph to subgraphs G_1 and G_2 induced by the edges of the respective weights and we maintain separate connectivity structures (together with a spanning tree) for G_1 and $G_2 \cup T_1$ (where T_1 is a spanning tree of G_1). We can easily modify the structure for $G_2 \cup T_1$ to prefer the edges of T_1. This ensures that the spanning tree of $G_2 \cup T_1$ will be the MST of the whole G.

If there are more possible values, we simply iterate this construction: the i-th structure contains edges of weight i and the edges of the spanning tree from the $(i-1)$-th structure. We get:

6.3.8. 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.

6.4. 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 4.3.

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.

6.4.1. *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:

6.4.2. Lemma. *(Difference lemma)*
For each $i > 1$ there exists $j < i$ such that T_i and T_j differ by a single edge exchange.

6.4.3. 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, but this can be reduced to finding peaks of the paths covered by the edges outside T_1, which we already are able to solve efficiently by the methods of Section 4.3. Therefore:

6.4.4. 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. (We will call this procedure *finding the best exchange in* (H,T).)

6.4.5. Corollary. Given G and T_1, we can find T_2 in time $\mathcal{O}(m)$.

6.4.6. *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 observe:

6.4.7. 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.

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 $\mathcal{O}(m)$.

6.4.8. *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 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 6.4.7, each spanning tree of G is generated exactly once. The Difference lemma guarantees that the trees are enumerated in the increasing order. So we get:

6.4.9. Lemma. Given G and T_1, we can find $T_2, \ldots, T_K$ in time $\mathcal{O}(Km + K \log K)$.

6.4.10. *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]. He has proven 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. When we combine this with the previous construction, we get the following theorem:

6.4.11. 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 $\mathcal{O}(m\,\alpha(m,n) + \min(K^2, Km + K \log K))$.

7. Ranking Combinatorial Structures

7.1. Ranking and unranking

The techniques for building efficient data structures on the RAM, which we have described in Section 3.3, 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 3.3). Obviously, $R(x)$ is the integer whose binary representation is the string x.

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. 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. $\pi \leftarrow$ *Unrank*$(j \bmod (n-i)!, i+1, n, A \setminus \{x\})$.
4. $\pi[i] \leftarrow x$.
5. Return π.

We can call *Unrank*$(j, 1, n, [n])$ 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 $\mathcal{O}(n \cdot t(n))$.

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 $\pi[i]$.

To obtain linear time complexity, we will make use of the representation of vectors by integers on the RAM as developed in Section 3.3. We observe that since the words of the RAM need to be able to hold integers as large as $n!$, the word size must be at least $\log n! = \Theta(n \log n)$. Therefore the whole set A fits in $\mathcal{O}(1)$ words and we get:

7.2.7. 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 $\mathcal{O}(n)$.

7.2.8. *The case of k-permutations.* Our algorithm 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. 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). Since $\Omega(k \log n)$ bits are needed to represent the rank, the vector representation of H certainly fits in a constant number of words. When we translate the operations on A to operations on H, again stored as a vector, we get:

7.2.9. 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 $\mathcal{O}(k)$.

7.3. 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.* 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.

7.3.1. 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.

7.3.2. *Equivalent formulations.* 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.

7.3.3. 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 $T(\pi) = \{(i, \pi(i)); i \in [n]\}$.

7.3.4. 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 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.

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 of restricted permutations is therefore equal to the permanent of the matrix M.

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:

7.3.5. 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.

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:

7.3.6. 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.

However, the hardness of computing the permanent is the only obstacle. We 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. We get:

7.3.7. 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.

Our time bound for ranking of general restricted permutations 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. These speedups are hard to state formally in general (they depend on the structure of the matrices), but we demonstrate them on the specific case of derangements. We show that each matrix can be sufficiently characterized by two numbers: the order of the matrix and the number of zeroes in it. We find a recurrent formula for the permanent, based on these parameters, which we use to precalculate all permanents in advance. When we plug it in the general algorithm, we get:

7.3.8. 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.

8. Conclusions

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 $\lambda_k(n)$ (the k-th row inverse of the Ackermann's function) 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, 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 and the vector computations on the RAM. We hope that they will be useful in many other algorithms.

9. Bibliography

[BA95] A. M. Ben-Amram. What is a "Pointer Machine"? *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 26, 1995.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[Fre85] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal of Computing*, 14:781–798, 1985.

[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.

[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.

[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.

[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.

[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.

[Jar30] V. Jarník. O jistém problému minimálním (About a Certain Minimal Problem). *Práce moravské přírodovědecké společnosti v Brně*, VI:57–63, 1930. In Czech.

[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.

[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.

[Knu97] D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental 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.

[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.

[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.

[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.

[Mat95] T. Matsui. The Minimum Spanning tree Problem on a Planar Graph. *Discrete Applied Mathematics*, 58:91–94, 1995.

[MR01] W. J. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.

[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.

[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.

[PR02] 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.

[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.

[SD06] J. Slocum and S. D. *The 15 Puzzle Book*. The Slocum Puzzle Foundation, Beverly Hills, CA, USA, 2006.

[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.

[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.

[Tho99] M. Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *Journal of the ACM*, 46(3):362–394, 1999.

[Tho00] 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.

[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.

[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.

[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.