FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Marius-Florin Cristian

The Minimum Spanning Tree Problem

The Road to Linear Complexity

Supervisor: Jyrki Katajainen

Submission: August 2018; last update: December 13, 2018

Abstract. In this thesis we review the evolution of the *minimum-spanning-tree problem* with the aim of solving it. In order to solve the problem, one must reduce the running time from $O(m\alpha(m,n))$ to $O(m)$ in a deterministic fashion. On the RAM model we manage to show that the disjoint sets operations performed in computing the minimum spanning tree can be made to run in linear time by allowing a certain degree of error that must be accounted for. We propose a linear-time density-partition procedure with the aim of producing a sparse graph in which the minimum spanning tree of the original graph lies. We also explore how to classify the cycles of a sparse graph on levels such that the small cycles lie at low levels, and the longer ones lie at high level. We need such a cycle classification in order to trim the components of low degree of the graph in $O(n)$ time, instead of $O(n^2)$. Finally we discuss an algorithm based on the density-partition, the cycle hierarchy, and the disjoint sets with error that seems to run in linear time on the RAM model, without any assumptions on the edge weights, and without the use of randomization. However, this discussion has little to offer if one wishes to solve the problem on a pointer machine, as the bottleneck will represent the operations performed by the disjoint-sets data structure without using table look-up, which is $O(m\alpha(m,n))$.

Contents

Chapter I. Introduction

In this chapter we define the problem, the playground, and the tools needed for investigation. In the second chapter we review the data structures that facilitate the implementation of the algorithms. In the third chapter we highlight notable ideas from the problems evolution, with respect to the complexity of the solutions. In the fourth chapter we document our ideas and propose new approaches; we manage to show that the disjoint sets operations in the computation of the minimum spanning tree can be bounded by $O(m+n)$ running time; given a sparse graph, we explore how to trim the components that are trees in $O(n)$ time. In the fifth chapter we summarize our conclusions.

1. Problem Definition

Assume that we are given an undirected, connected graph $G = (V, E, C)$, where $V = \{v_1, v_2, ...v_n\}$ is the vertex set, $E = \{e_1, e_2, ..., e_m\}$ the edge set, and each edge e_i has a cost (or weight) associated $c_i \in C$, where $C = \{c_1, c_2, ..., c_m\}$. Let $|V| = n$ and $|E| = m$. A *spanning tree* $T = (V, E')$ of G connects its vertices and the edge set $E' \subseteq E$ has $n - 1$ edges. The *weight* of that spanning tree is the sum of the cost of its edges, i.e. $\sum_{e_i \in E'} c_i$. In the *minimum-spanning-tree problem* the task is to find a spanning tree of G for which the cost is minimum. If the graph is not necessarily connected then we can talk about its *minimum spanning forest* which is a union of the minimum spanning trees of its connected components. We will use the abbreviations MST and MSF to refer to the minimum spanning tree and respectively to the minimum spanning forest of a graph.

The particularity of the defined graph is that we do not allow duplicate edges, that is to say each vertex can be connected to another distinct vertex by only one edge and hence $m \leq n^2$ holds. Throughout this thesis, we will use $O(m \log m)$ and $O(m \log n)$ interchangeable as a consequence of not allowing duplicates and logarithm properties ($\log_2 m \leq \log_2 n^2 \cong \log_2 m \leq 2 \log_2 n$).

2. Models of Computation

We will tackle the problem from the perspective of two computational models: a *pointer machine* (PM) and a *word random access machine* (RAM).

As defined by Penttonen and Katajainen in their paper [15], a *pointer machine* (PM) has an input tape which is scanned by a one-way read-only head, and an output tape with one-way write-only head. The memory of a PM is a graph-like structure, bearing resemblance with a dynamic record structure which is available in many programming languages. A memory cell consists of k pointers labelled with the letters from a pointer alphabet P, where k is a fixed number, $k \geq 2$. There are k letters in the pointer alphabet, and different pointers of a cell are labelled with different letters. Thus, there is a one-to-one correspondence between pointers and labels. The memory structure can be considered as a digraph whose edges are labelled and which has a constant fan out k.

A word RAM operates on words of size *wordsize* bits, it has a memory of $2^{wordsize}$ and a processor that manipulates these words. A word can store an integer or an address to another word. Each word can be accessed at unit cost. The processor can execute arithmetic, logical and, bit-wise operations at unit cost.

The only restriction is that the operations performed on the cost of the edges are comparisons. Even more, we do not have any assumptions on the weight of the edges of our graph. If instead we add some restrictions on the weights of the edges (denoted by small integers) some algorithms or steps presented become faster and could yield in improvements by using the full power of the RAM model, "but there is little to learn from such models, however, if one's goal is to resolve the MST question and settle what truly is one of the most remarkable open problems in computer science" [3, Section 11.5].

3. Ackermann's Function

One of the variants of the Ackermann's function as used by Chazelle[1]:

$$A(i,j) = \begin{cases} A(1,j) = 2j, & \text{for any } j > 0, \\ A(i,1) = 2, & \text{for any } i > 0, \\ A(i,j) = A(i,j-1) \times A(i-1, A(i,j-1)), & \text{for any } i,j > 1. \end{cases}$$

The utility of this function lies in its inverse denoted by $\alpha(m,n) = \min\{i \geq 1 : A(i, 4\lceil m/n \rceil) > \log n\}$ as it is used to asymptotically bound the depth of tree-based data structures. A variation of it is used in van Leeuwen's and Tarjan's [18] analysis of disjoint sets data structures and in Fredman's and Tarjan's [6] analysis of *Fibonacci heaps* with implicit key.

4. General Approach

Every method seems to exploit in one way or another the following properties of the minimum spanning tree (taken from [10]):

Cycle property: For any cycle C in a graph, the heaviest edge in C does not appear in the minimum spanning forest.
Cut property: For any proper non-empty subset X of the vertices, the lightest edge with exactly one endpoint in X belongs to the minimum spanning

forest.

Terminology used:
A *fragment* is a subtree of the minimum spanning tree.
A subgraph C of G is *contractible* if its intersection with $MST(G)$ is connected [1].
A *minor* is a graph derived from a sequence of edge contractions and their implied vertex deletions [1].

Chapter II. Data Structures

In this chapter we will present three tree-based data structures that are used as the backbone of the MST algorithms.

1. Disjoint Sets

Two versions of the data structure are presented in Cormen et al. [4, Chapter 21]: an implementation based on linked lists and one based on forests. The data structure is used to group n distinct elements into a collection of disjoint sets. The following operations must be supported:

- Make-Set(x) - creates a new set whose only member is x;
- Find(x) - returns a pointer to the set containing x;
- Union(x, y) - unites the sets that contain x and y.

One of the applications of the data structure is to check if two components are connected. For our problem, we will use it exactly for this purpose: to answer if by adding a new edge to the minimum spanning tree construction two components (fragments) get connected or if one is extended.

In disjoint-set implementation using a linked list, each set is represented by its own linked list. The element for each set has two attributes: *head* (pointing to the first object in the list) and *tail* (pointing to the last object in the list). Each element in the list contains a set member, a pointer to the next object and a pointer back to the set of contained elements. The Make-Set and Find operations take $O(1)$ but unfortunately a sequence of n Union operations take $O(n \log n)$ time in the worst case. If we keep track of the size of each list, during a union we can update the pointers of the shorter list, but if the lists have the same size, then the running time will be $\Omega(s)$, where s denotes the size of the lists.

Theorem 1. ([4, Section 21.2]) *Using the linked-list representation of disjoint sets, a sequence of m* Make-Set, Union *and* Find *operations, of which n are* Make-Set *operations take $O(m + n \lg n)$ time.*

In a *disjoint-set forest*, each set is represented by a tree, and each node of the tree represents an element that points only to its parent. The root of each tree represents the set and it is its own parent. A straightforward implementation is no better than the linked-list one, but we can achieve an asymptotically optimal disjoint-set data structure using two heuristics to improve it: *union by rank* and *path compression.* The *union by rank* heuristic method is similar to the improvement done in the linked-list case; for each node we keep a rank which is an upper bound on the height of that node. The root with a smaller rank will point to the root with a higher rank. The *path compression* is used during Find operations to make each node on the find path point directly to the root. Using the above two heuristics the following theorem holds (where α represents the inverse Ackermann function):

Theorem 2. ([4, Section 21.4]) *A sequence of m* Make-Set*,* Union*, and* Find *operations, of which n are* Make-Set *operations, can be performed on a disjoint-set forest with union by rank and path compression in worst-case time $O(m\alpha(m, n))$.*

2. Fibonacci Heaps

The data structure has been invented, by Fredman and Tarjan [6] (see also [4, Chapter 19] for a more detailed explanation). The strength of the *Fibonacci heap* lies in the ability to perform Extract-Min operations in $O(\log n)$ amortized time (where n is the number of elements in the heap) and Decrease-Key operations in $O(1)$ amortized time, while the cost of the other heap operations is constant.

A *heap* is an abstract data structure consisting of a set of elements, each with a real-valued key, that supports the following operations:

- Make-Heap() - returns a new empty heap;
- Insert(x, H) - inserts a new element x into the heap H;
- Find-Min(H) - returns an element of minimum key in heap H;
- Delete-Min(H) - returns an element of minimum key in heap H and deletes it;
- meld(H_1, H_2) - returns a heap formed by the element disjoint heaps H_1, H_2 and destroys H_1, H_2;
- Decrease-Key(d, x, H) - decreases the key of the element x in heap H by setting it to be d;
- Delete(i, H) - deletes element i from the heap H.

A *Fibonacci heap* is a collection of element-disjoint min-heap ordered trees (the key of each node is greater or equal to the key of its parent). The rank of an element represents the number of its children; nodes can be marked or unmarked; a node is marked if it loses one children. Nodes have one pointer to the parent and one pointer to its children. Children of each node are doubly linked in a circular list. We maintain a pointer to the root of the minimum key in the heap. The roots of all the trees in the heap are doubly linked in a circular list.

3. Soft Heaps

The soft heap implementation can be found in Chazelle's book [3, Section 11.2]. Chazelle's implementation is based on binomial trees, however, this subsection will present the key elements of the implementation based on binary trees by Kaplan and Zwick [9] as it is more intuitive.

The soft heap is a priority queue composed of a collection of binary trees at most one of each rank. The rank of a binary tree (rank of the root) is represented by its depth (the number of nodes until the furthest away leaf). Every node has such a rank that never changes. A node in a binary tree contains a list of keys (elements), a *ckey* which represents an upper bound on the keys of the list elements, and a target size which bounds the number of elements in this list. Let ϵ be the corruption parameter, and $r = \lceil \log_2 \frac{1}{\epsilon} \rceil + 5$. The target size is defined as $s_k = 1$, if *rank* $k \leq r$ or $s_k = \lceil \frac{3}{2} s_{k-1} \rceil$, otherwise. Ideally we want the list size to be close to the target size, if it is lower than half of the target size, the *sift* operation is called recursively in order to populate the current list with more elements from nodes with lower rank (thus higher *ckeys*), this is known as *carpooling.*

At any time the data structure may increase the key value of certain elements, but it may never decrease them. The heap property is always maintained regarding the current key values. We call an item *corrupted* if its key value has been increased. Once an item has been corrupted, it remains forever corrupted!

Theorem 3. ([9, 1, 2]) *Given an empty soft heap with error rate* $0 < \epsilon \leq \frac{1}{2}$ *and a mixed sequence of operations that include* n *inserts, the data structure supports the operations* MAKE-HEAP*,* MELD*,* DELETE*,* FIND-MIN *in amortized constant time and* INSERT *in* $O(\log 1/\epsilon)$ *amortized time. The number of corrupted elements will be at most* ϵn *(this number being optimal in a comparison-based model).*

Simplified, for a *soft heap* with n elements, in the worst case, a FIND-MIN operation will extract a fraction of n of the elements in partial or full order amongst whom the the minimum *ckey* lies.

Chapter III. The Road

In this chapter we discuss a selection of earlier algorithmic results and our reasoning and remarks. We classify the algorithms based on the approach used, thus the chapter is split in four sections each covering the details of implementation, running times and a conclusion. In the first section we summarize the greedy algorithms, in the second section the randomized algorithm, and in the third section the verification algorithm, in the fourth section the best-known deterministc solution, and in the fifth section the optimal algorithm. We end the chapter with a section containing our remarks and ideas.

1. Classical Algorithms

In this section we present three algorithms from the paper by Graham and Hell [8]. The terminology might be rephrased in order to keep a fluent presentation.

1.1 Joining Two Nearest Fragments

Attributed to Kruskal, the algorithm outlined in Figure 1 uses a *sort*-based method to order the edges by weight, thus giving an asymptotic time complexity of $O(m \log n)$ under the assumption that the sorting method is comparison-based. However, for graphs in which the universe of weights is defined over small integers, *radix sort* can be used to obtain nearly $O(m)$ time. There is also a variant of this algorithm where we remove from the graph the edges with the highest cost such that we do not isolate any vertex. In order to have an efficient implementation one should use a disjoint-set data structure to store the fragments; the $\exists$ PATH FROM u TO v is equivalent to FIND(u)==FIND(v) in the disjoint-set MST. A major drawback of this algorithm is that it performs its comparisons on all of the edges, even on those who are for sure heavy and outside of the MST. As we will see later in Chazelle's method [1], the idea is improved by searching for minimum "bridges" between such fragments from a restricted set of edges, not the total. In the same manner, without giving up on the sorting of the edges, Osipov, Sanders and Singler's [14] explore (and benchmark) practical improvements of Kruskal's idea by avoiding to sort the edges that are "obviously" not in the MST, and achieve a worst-case bound of $O(m + n \log n \log \frac{m}{n})$, which is linear for not too sparse graphs.

In a comparison-based model the initial sorting cost of the edges dominates the disjoint-set operation costs, thus the running cost is $O(m \log m+n)$ in the worst case. If we are allowed a more powerful computation model where we can manipulate bits and the edge weights are represented by small integers, the sorting can be done in linear time and the running time will

```
procedure Kruskal-MST
input $G = (V, E, C)$: undirected weighted graph
output minimum spanning tree
for $v \in V$
  | MAKE-SET($v$)
INITIALLY $T = \emptyset$
SORT THE EDGES $E$ BY WEIGHT
for $e = \{u, v\} \in E,\ u, v \in V$
  | if FIND($u$) $\neq$ FIND($y$)
  |   | ADD $e$ INTO $T$
  |   | UNION($x, y$)
return $T$
```

Figure 1. Two nearest fragments; Kruskal's algorithm using a disjoint-set data structure

be $O(m\alpha(m, n) + n)$ (because of the disjoint sets operations performed). It is to note that if the structure of the union operations is known in advance, the worst-case running time will be $O(m + n)$ (the disjoint set operations performed will have the same bound) [7].

1.2 Expanding Single Fragment

Usually attributed to Prim and Dijkstra but discovered by Jarnik, the algorithm outlined in Figure 2 has some similarities with the one proposed by Kruskal in the sense that both of them incrementally construct the minimal spanning tree. The running time is $O(n^2)$ if a table is used to represent the graph or $O(m \log n)$ when heaps are used to store adjacent edges. For dense graphs implementations of the algorithm yield a running time of $O(m)$, where a dense graph satisfies $m \geq c \cdot n^p$, $c > 0, p > 1$ constants. However, in the heap implementation, the algorithm relies heavily on the decrease-key operation in order to keep track of the vertex nearest to the fragment being expanded. We can improve the bound by exploiting the fact that a *Fibonacci heap* supports the DECREASE-KEY operation in $O(1)$; therefore, the algorithm will yield an $O(m + n \log n)$ complexity. Fredman and Tarjan presented [6] the data structure and how to use it on top of Prim's idea to get a total running time of $O(m + n \log n)$.

However, the running time can be improved to $m\beta(m, n)$, where $\beta(m, n) = \min\{i | \log^{(i)} n \leq \frac{m}{n}\}$, which is linear if $\frac{m}{n} = \log^{(3)} n$. This improvement is based on expanding single fragments until the heap of neighbours reaches a critical size, then move to another fragment, stopping again when the same criteria is met. Once every vertex is part of a tree, we condense each tree in a single vertex and start a new pass. After a sufficient number of passes, only one supervertex will remain; the MST can be extracted by expanding

```
procedure Prim-MST
input $G = (V, E, C)$: undirected weighted graph
output minimum spanning tree
 $T = \{\emptyset\}$
for $v \in V$
|  $key(v) = +\infty$
$FHeap = \{v \in V\}$
SELECT AN INITIAL VERTEX $s$ AND SET $key(s) = 0$
DECREASE-KEY$(0, s, FHeap)$
for $v \in V, s.t.\ \exists\{s, v\}$
|  KEY$(v)$=$c(s, v)$;
 while $\exists v \in V$ $with$ $key(v) \neq \pm\infty$
    $v$ =EXTRACT-MIN$(FHeap)$
    $light\_edge$=NULL
    $min\_weight$=$+\infty$
    for $u \in V, s.t.\ \exists\{v, u\}$
        if $w(v, u)$ <$key(u)$
            $key(u)$=$w(v, u)$
            if $min\_key > key(u)$
                $min\_key = key(u)$
                $light\_edge = \{v, u\}$
    ADD $light\_edge$ INTO T
return $T$
```

Figure 2. Expanding single fragment; Prim's algorithm using a *Fibonacci heap*

the supervertices.

1.3 Joining All Nearest Fragments

Attributed to Borůvka, a straight implementation of the algorithm presented in Figure 3 yields an asymptotic complexity of $O(m \log n)$. The idea behind this algorithm is to construct Borůvka steps in order to reduce the number of vertices of the graph. The procedure is used in other MST algorithms such as the randomized version of Karger, Klein and Tarjan [10] or the deterministic versions of Chazzelle [2], or the optimal version of Pettie and Ramachandran's [16] in order to reduce the number of vertices by a series of vertex contractions and increase the density (edge to vertex ratio) of the graph. Let v be the number of vertices in the graph before executing such a phase and n the number of vertices in the graph after executing such a phase, then the newly formed graph has at most half of the original vertices ($n \leq \frac{v}{2}$). We will use a disjoint set to keep track of the fragments we connect.

```
procedure BorůvkaPhase
input $F$, $v$: forest $F$ and a vertex $v$
output updates the forest
 for $v \in V$
  | PICK $u$ s.t. min$(v,u)$, AND FIND$(u) \neq$ FIND$(v)$
  | $F = F \cup \{v,u\}$
  | UNION$(v,u)$
  | for $\{u,z\}$
  |  | if FIND(u)==FIND(z)
  |  |  | DELETE$(\{u,z\})$
end;


procedure Borůvka-MST
input $G = (V,E,C)$: undirected weighted graph
output minimum spanning tree
 $F = \emptyset$
for $v \in V$
 | MAKE-SET$(v)$
while $|F| < |V| - 1$
 | BORŮVKAPHASE(F,V)
return F;
```

Figure 3. Joining All Nearest Fragments; Borůvka's algorithm using a disjoint-set data structure

2. Randomized Approach

In this section we present the randomized algorithm from the paper by Karger et. al. [10]. A similar variant of the algorithm is also covered in Motwani and Raghavan's book [13, Section 10.3]. The following is split into three subsections. In the first subsection we explain the Borůvka Phase, in the second subsection we explain the random-sampling phase in relation to random graphs (Erdős and Rényi observed similarities in the evolution of such graphs [5]), and in the third subsection we state the algorithm.

We assume that there are no isolated vertices and the edge weights are distinct.

We define F as being any forest in our graph G, now considering two vertices u, v who lie in the same connected component (tree) of F, there exists a unique path $F(u, v)$ from u to v. Let $w_F(u, v)$ be the maximum weight of an edge in such path ($w_F(u, v) = max_{e \in P(u,v)} w(e)$ or ∞ if there is no such path P in the the forest F). An edge is called F-heavy if $w(u, v) > w_F(u, v)$, $u, v \in V$, $u, v \in F$, otherwise it is called F-light. Because the components in the minimum spanning tree are acyclic, for any forest F, no F-heavy edge can be added to the output, (recall that an F-heavy edge is the heaviest edge in a cycle. Thus, it cannot appear in the MST).

2.1 Expanded Borůvka Phase

The procedure *BorůvkaPhase* (presented in the previous chapter) can be modified to also perform the edge contraction and elimination of self loops. The steps of the implementation are: first we mark the edges to be contracted, then we determine the connected components formed by the marked edges and replace each such component by a vertex. Finally, we eliminate multiple edges and self loops created by the contraction. A good implementation is done in $O(n + m)$ time, where n denotes the number of vertices in the graph, and m the number of edges.

2.2 Random Sampling

In order to understand the need for the random sampling and the connection with Borůvka phases, first we briefly state the five phases of random graph construction devised by Erdős and Rényi [5].

Let $\{v_1, v_2, ..., v_n\}$ be the vertex set. Therefore, we have $\binom{n}{2}$ possible distinct edges that can connect two of the vertices. In order to construct a random graph, at time $t = 1$ we choose an edge uniformly at random (with probability $\frac{1}{\binom{n}{2}}$) from the set of possible edges, at time $t = k$, we choose an edge uniformly at random from the remaining edge set (with probability $\frac{1}{\binom{n}{2}-k}$), the graph $G_{n,m}$ is the graph consisting of the vertices $v_1, ..., v_n$ and the edges $e_1, ..., e_m$. Let's have n a fixed large integer, and m increasing from 1 to $\binom{n}{2}$. It then $G_{n,m}$ passes through the following five phases (each corresponding to a range of growth of m, this range being defined in terms of the number of vertices n):

Phase 1: When $m(n) = o(n)$ the graph consists almost entirely of components which are trees.

Phase 2: When $m(n) = cn,\ 0 < c < 1/2$, $G_{n,m}$ already contains cycles. Almost all components are trees or consist of exactly one cycle. The greatest component is a tree and with probability tending to 1 has $\frac{1}{a}(\log n - \frac{5}{2}\log\log n)$ vertices, where $a = 2c - 1 - \log 2c$.

Phase 3: When $m(n) \geq cn,\ c \geq 1/2$, structure changes abruptly after $m(n)$ passes $\frac{n}{2}$. For $c = \frac{1}{2}$ the largest component has $n^{2/3}$ vertices and is quite complex; for $c > \frac{1}{2}$ the largest component has with high probability $n \cdot (1 - 2c \sum_{k=1}^{+\infty} \frac{k^{k-1}}{k!}(2ce^{-2c})^k)$ vertices. The other components are relatively small and most of them being trees.

Phase 4: When $m(n) = cn\log n,\ c \leq 1/2$, the graph almost surely is connected (detailed description of this phase can be found in the paper by Erdős and Rényi [5], as quite a lot of things are going on and it is out of the scope of this section).

Phase 5: When $m(n) = (n\log n)w(n),\ w(n) \to +\infty$ The graph is most surely connected and the degrees of all vertices are asymptotically equal.

Now back to the random sampling: the idea is to discard edges that cannot be in the minimum spanning tree; this can be done in linear time using King's verification algorithm [11].

Theorem 4. (Theorem 10.18 [13]) *Given a graph G and a forest F, all F-heavy edges in G can be identified in time O(n + m).*

Lemma 1. (Karger et. al. [10]) *Let H be a subgraph of G obtained by including each edge with probability p, and let F be the minimum spanning forest of H. The expected number of F-light edges in G is at most $\frac{n}{p}$, where n is the number of vertices of G.*

The proof of this lemma is skipped, but a few remarks must be included:

Remark 1. To compute H and F at the same time, we use a variant of Kruskal's algorithm. Thus, edges must be processed in increasing order of weight.

Remark 2. When processing an edge e, if its endpoints are in the same component, skip it as it is F-heavy for sure.

Edges are included in H with probability $p = \frac{1}{2}$. Edges from H are included in F if they also are F-light.

Remark 3. The number of F-light edges is bounded by the number of coin flips used to include them in H. The number of such edges is stochastically dominated by a variable with a negative binomial distribution.

2.3 The Randomized Algorithm

The idea is to intermesh Borůvka steps with random sampling steps in order to reduce the graph and the density of edges. The algorithm is recursive and generates two subproblems that are ideally at most a constant fraction less than the original problem.

Step 1: Apply two successive Borůvka steps to the graph.

Step 2: Choose a subgraph H by selecting each edge with probability $1/2$ independently at random. Apply the algorithm recursively to H producing minimum spanning forest F of H, find all F-heavy edges (both in H and not in H) and delete them from the contracted graph.

Step 3: Apply the algorithm recursively to the remaining graph to produce a spanning forest F'. Return the edges contracted in Step 1 and those in F'.

Theorem 5. *After one execution of the recursion step of the algorithm, the graph will be classified as a phase 3 graph in the worst case.*

Proof. The worst case, when we are talking about the density of the graph, is in the case when the graph has $< n\log^{(3)} n$ edges.

Now consider the application of two Borůvka steps followed by a random selection of the remaining edges with a probability of $1/2$. We reduce the number of edges to $\frac{n}{8}$. We must find the F-light edges in a graph with $\frac{7n}{8}\log n$ edges. Out of this at most $2n$ edges are F-light, thus the rest are discarded. The reduction is from $\frac{7n}{8}\log^{(3)} n$ to $2n$ edges after one step. Thus the graph is classified as a phase 3 graph ($\frac{n}{2} \leq m \leq \frac{n\log n}{2}$). □

Theorem 6. **([10])** *The worst case running time of the minimum-spanning-forest algorithm is $O(\min\{n^2, m\log n\})$, the same as the bound for Borůvka's algorithm.*

Theorem 7. **([10])** *The minimium-spanning-forest algorithm runs in $O(m)$ time with probability $1 - e^{-\Omega(m)}$.*

3. Verification Algorithm

Given a connected, undirected, weighted graph G and a spanning tree T of G verify that T is the minimum spanning tree of G. In this section we present King's linear-time verification algorithm [11] and data structure. The algorithm is built on top of Komlós' [12] verification idea for a full branching trees. A *full branching tree* is a rooted tree with all of the leaves on the same level and every internal node has at least two children. Even if Komlós doesn't provide an implementation, King offers one based on a novel data structure.

The verification method exploits the fact that a spanning tree is a minimum spanning tree if the weight of each non-tree edge $\{u, v\}$ is at least the weight of the heaviest edge in the path in the tree from u to v. Komlós observes that given a spanning tree T, a full-branching tree B can be built in $O(n)$ with no more than $2n$ edges and the following property: Let $T(x, y)$ be the set of edges in the path in T from x to y, and let $B(x, y)$ denote the set of edges in the path in B from x to y; Then the weight of the heaviest edge in $T(x, y)$ is the weight of the heaviest edge in $B(x, y)$. In the following we explore on how to efficiently implement Komlós' algorithm on a RAM machine of word size $\Theta(\log n)$.

3.1 Deriving a Full-Branching Tree

Given a tree T, we construct a Borůvka tree B as follows: Initially all of the nodes in T are leaves in B, then we apply successive BorůvkaPhases to construct B level by level by a series of contractions until we reach the root. B will be a full branching tree. Because we apply BorůvkaPhases to a tree, it is constructed in linear time with respect to the total number of edges in the graph. The edges of B are labelled with the weight selected at time of contraction.

Theorem 8. ([11]) *Let T be a spanning tree and B the tree resulted from applying successive BorůvkaPhases to T. Then for any pair x, y, the heaviest edge in the path in T from x to y is equal to the weight of the heaviest edge in the path from x to y in B.*

3.2 Komlós' Algorithm

For a full branching tree of weighted edges with n nodes and m query paths between pair of leaves, Komlós' algorithm computes the heaviest edge on the path between each pair with $O(n \log((m + n)/n)$ comparisons. The main idea is to break up each path in the tree, into two half paths, from the leaves to the lowest common ancestor of the pair and from the lowest common ancestor to the root. The heaviest edge in each path is found as follows:

For a node $v \in B$, let $A(v)$ be the set of paths that contain v restricted to the interval $[root, v]$. Now start from the *root* and descend by level and for each node v encountered find the heaviest edge in the path $A(v)$ as follows: If p is the parent of v, we already know the heaviest edge in $A(p)$. Let $A(v|p)$ be the set of restrictions of each path in $A(v)$ to the interval $[root, p]$. Since $A(v|p) \in A(v)$, the ordering of the heavy weights is known ($\forall s, t \in A(v)$, path s includes path t or the other way around). To determine the heaviest edge in $A(v)$ we compare only $w(\{v, p\})$ with each of the heaviest edge in $A(v|p)$ (using binary search). Komlós shows that $\sum_{v \in T} \lg |A(v)| = O(n \log((m + n)/n)$, which is the upper bound on the number of comparisons needed to find the heaviest edge in each half path. The heaviest edge in each query path is determined with one extra comparison per path.

3.3 Data Structure

In order to implement Komlós' algorithm in linear time we need the following data structure implemented on a *RAM* machine:

Let *wordsize* be the size of the word, represented on $\lceil \lg n \rceil$ bits, where n is the number of leaves. The nodes are labelled with a *wordsize*-bit label

as encountered in the depth-first traversal of the tree as follows: the leaves will be labelled $0, 1, 2, ..., n$ and any internal node will be labelled with the label of the leaf in its subtree which has the longest 0's suffix. The edges are tagged with a $2 \cdot \log_2 \log_2 n$ x-bit tag, such that: the first half of the tag represents the distance from the *root* to the lower endpoint of the edge in the tree and the second half represents the index of the rightmost 1 bit in the label of the aforementioned node (e.g. if $label(v) = 00001000_{(2)}$ and it is situated at depth $6_{(10)} = 00110_{(2)}$, then $tag(e) = \{0110, 0101\}$).
The way the labels and tags are embedded allows us to locate an edge e, given its tag in constant time (nodes with the same label are connected by a path in the tree; once the lower endpoint v of e is found, then e is the unique edge connecting v to its parent). We name this property the *label property.*

For each node v, let $LCA(v)$ be a single word of length $wordsize$ whose i'th bit is 1 if there is a path in $A(v)$ whose upper endpoint is at distance i from the *root* (there is one query path with exactly one endpoint in the subtree rooted at v, such that the lowest common ancestor of its two endpoints is at distance i from the *root*).

For all nodes v let $A_i(v)$ the i'th longest path in $A(v)$. A node is called *big* if $|A(v)| > \frac{wordsize}{tagsize}$, (we do not keep track of the infix zeroes in the $tagsize$ and $wordsize$). If a node is not *big*, then it is *small.*

For each *big* node v, we keep an ordered list called $bigList(v)$ of size $O(\log \log n)$ words, whose i'th element is the tag of the heaviest edge in $A_i(v)$, for $i = 1, 2, ..., |A(v)|$. For each *small* node v let a be the nearest *big* ancestor of v. We keep an ordered list called $smallList(v)$, whose i'th element is either the tag of the heaviest edge $e_i \in A_i(v)$ or if e is in the interval $[root, a]$, then j such that $A_i(v|a) = A_j(a)$; that is j is a pointer to the entry in $bigList(a)$ that contains the tag for e. Once a tag appears in $smallList$, all the later entries are tags. For each *small* v keep a pointer to the first tag in its $smallList$.

Let $swnum = \lfloor wordsize/tagsize \rfloor$. This denotes the maximum number of subwords stored in a word.
The extra operations needed besides bit-wise OR, XOR, AND, addition and multiplication are:

– $select_r(I, J)$: two strings I, J represented on r bits each, it outputs a list of bits of J which have been "selected" by I (e.g. let $(k_1, k_2, ...)$ be the ordered list of indices of those bits of I whose value is 1, then the list $(j_{k_1}, j_{k_2}, ...)$, where j_{k_i} is the value of the k_i'th bit of J);

– $selectS_r(I, J)$: two strings I, J, where I is a string of no more than r bits, no more than $swnum$ of which are 1, and J is a list of $swnum$

subwords. It outputs a list of subwords of J which have been "selected" (e.g. let $(k_1, k_2, ...)$ be the ordered list of indices of those bits of I whose value is 1, then the list $(j_{k_1}, j_{k_2}, ...)$, where j_{k_i} is the value of the k_i'th subword of J);

- *weight*$_r$: takes as input a string of length r and outputs the number of bits set to 1;
- *index*$_r$: takes an r-bit vector with no more than h 1's and outputs a list of subwords containing the indices of the 1's in the vector;
- *subword*1: is a constant such that for $i = 1, 2, ...swnum$ the $(i * tagsize)$'th bit is 1 and the remaining bits are 0.

Figures 4, 5 and 6 represent how the data structure encodes the input tree.

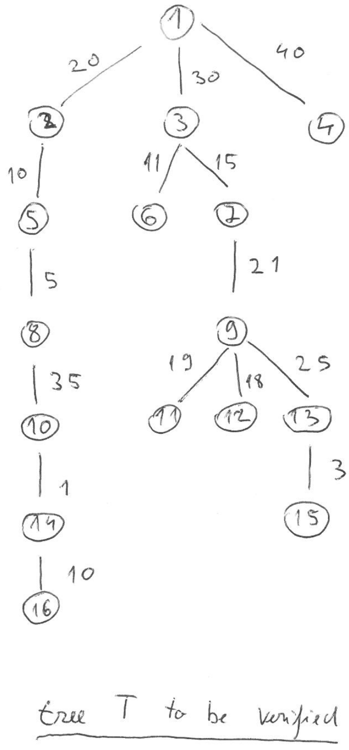
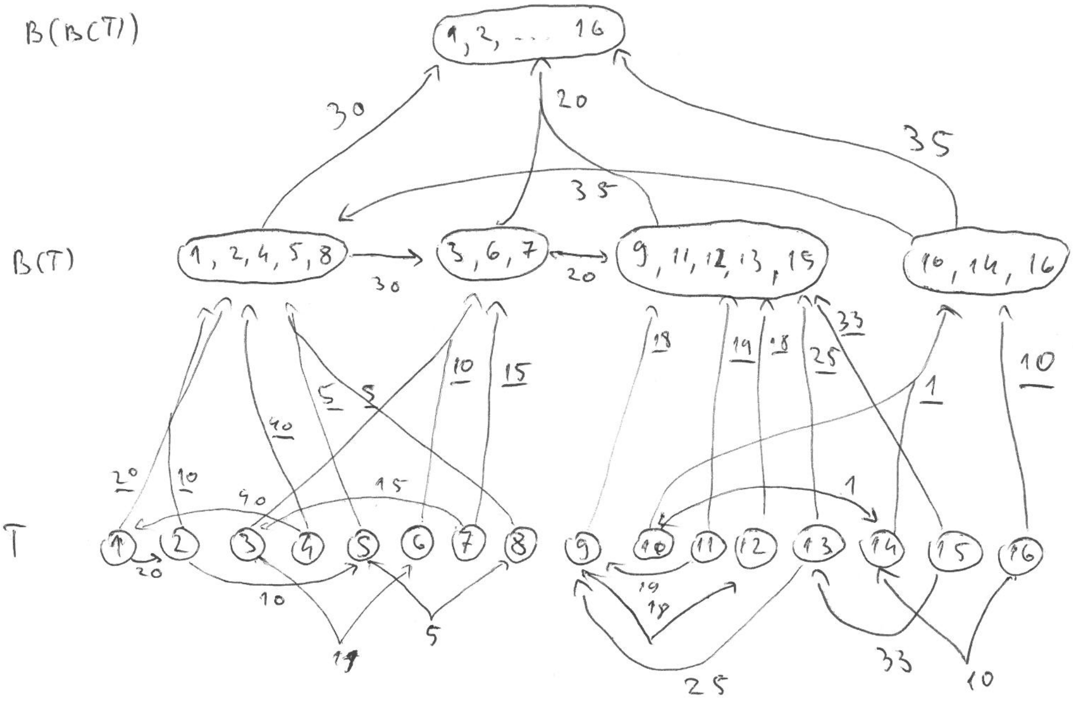


Figure 4. Spanning tree T to be verified. Here the vertices are labelled with distinct integers and the edges have a conveniently assigned weight. In the next three figures, we work with the same node labels and weights in order to exemplify how the transformations take place.



full branching tree B obtained from
tree T after 2 Borůvka Phases

Figure 5. Full branching tree B

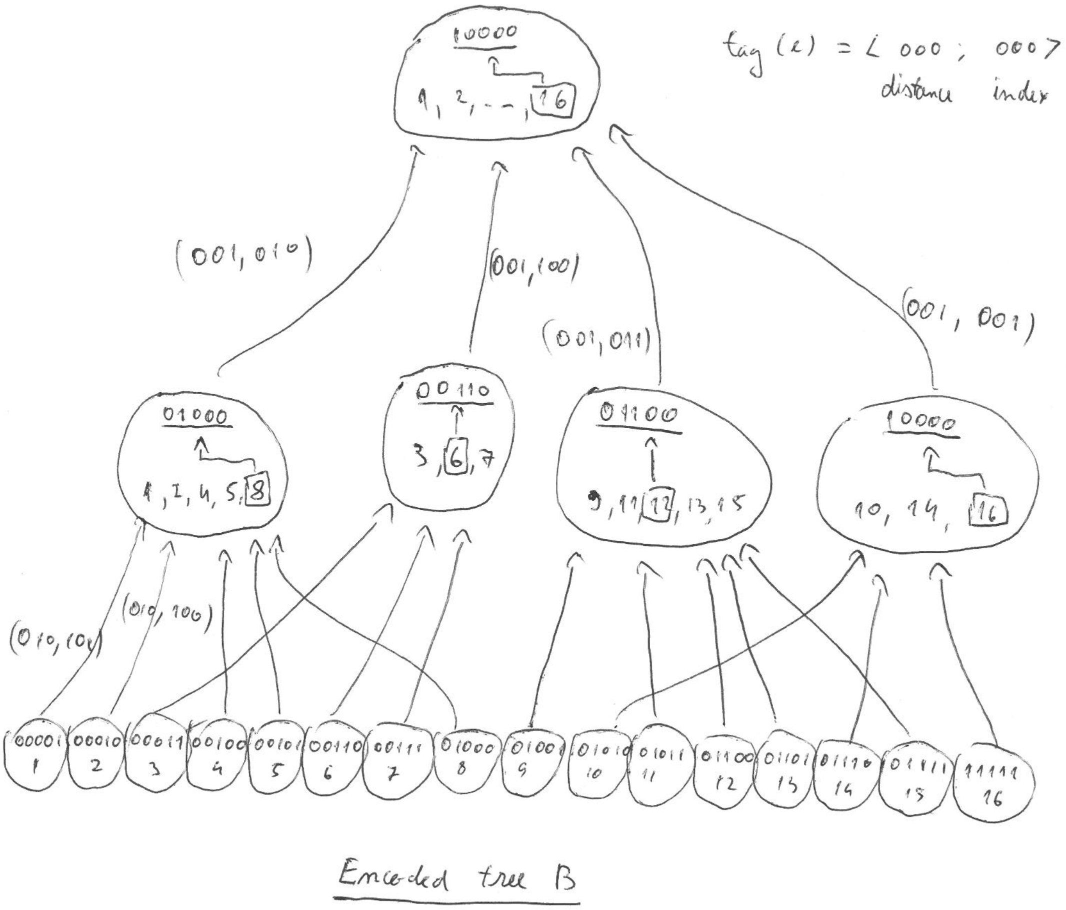


Figure 6. Encoded tree B

3.4 King's Algorithm

Precompute all LCA in time $O(m+n)$ (we do not follow the references on how to build it) for each pair of endpoints of the m query paths. Form the vector $LCA(l)$ for each leaf l, and then form the vector $LCA(v)$ for each node at distance i from the *root* by bit-wise ORing the LCA of its children and setting the j'th bits to 0 for all $j \geq i$.
Generate *bigList*(v) or *smallList*(v) in time proportional to $\log |A(v)|$. If v is *big* the implementation time is $O(\log \log v)$; $O(1)$ otherwise.
Initially $A(root) = \emptyset$. Proceed down the tree from the parent p to each of its children v. Generate *bigList*$(v|p)$ or *smallList*$(v|p)$ depending on $|A(v)|$, compare $w(\{v,p\})$ to the weights in the generated lists and insert the edge in the appropriate place to form *bigList*(v) or *smallList*(v). Repeat until the leaves are reached.
Let v be any node, p its parent and a the nearest big ancestor.
If v is *small*: if p is *small*, we create *smallList*$(v|p)$ from *smallList*(p) in constant time; if p is *big*, we create *smallList*$(v|p)$ from $LCA(v)$ and $LCA(p)$ in constant time.
If v is *big*: if v has a *big* ancestor, then we create *bigList*$(v|a)$ from *bigList*(a), $LCA(v)$ and $LCA(a)$ in $O(\lg \lg n)$ time (if $p \neq a$, then we create *bigList*$(v|p)$ from *bigList*$(v|a)$ and *smallList*(p) in $O(\lg \lg n)$ time); if v doesn't have a *big* ancestor, *bigList*$(v|p)$ = *smallList*(p).
To insert a tag in its appropriate place in the list: let $e = \{v,p\}$, and let i be the rank of $w(e)$ compared with the heaviest edge of $A(v|p)$. Insert e into positions i through $|A(v)|$ into our list data structure.

Must include example, as illustrated in the original paper [11].

4. Building the Tree of Contractible Subgraphs

In this section we present the best deterministic solution for the problem as presented in Chazelle's paper [1] and his book [3, Chapter 11]. In the next two subsections we briefly present how to build the tree of contractible subgraphs and how to use it to achieve the best deterministic bound.

Theorem 9. ([1]) *The* MST *of a connected graph with n vertices and m edges can be computed in $O(m\alpha(m,n))$ worst-case time.*

4.1 Building the Tree T of Contractible Subgraphs

T represents a hierarchy of contractible subgraphs of G. The root of T is a G_0 represented by a single vertex, and the leaves of T are the initial contractible components that form G before any contraction. A node in the tree at depth z represents a subgraph after contracting a series of such $C_1 ... C_z$. We will use Ackermann's function to provide a trade-off between the depth (d_z) of T and the size (n_z) of each component C_z.

$$n_z = \begin{cases} A(t,1)^4 = 8, & \text{if } d_z > 0 \\ A(t-1, A(t, d_z - 1))^3, & \text{if } d_z > 1. \end{cases}$$

We will bound the depth of the tree to be $d = c\lceil (m/n)^{1/3} \rceil$, where c denotes a constant large enough (we will see later as c is the number of Borůvka steps we execute in one recursion; it also denotes the error rate of the soft heaps: $1/c$, when building T). In order to compute T in $O(m + d^3 n)$ time, we will choose the target size of the soft heap to be $t = min\{i > 0 | n \leq A(i,d)^3\}$, where A is the Ackermann function presented in the introduction.

We must build T maintaining a connected order, thus we will build it in *postorder* (left-right-node). Let z be the current node visited in T and $z_1 ... z_k$ the active path. We are currently assembling $C_{z_1} ... C_{z_k}$. When C_{z_k} is ready (contracted into one vertex), the vertex is added to $C_{z_{k-1}}$ (the previous component). In order to ensure that each of the components being contracted has at least one vertex which is a vertex of G or a contraction of a connected graph, if z_{i+1} is the leftmost child of z_i then C_{z_i} has no vertex, and thus z_i is omitted from the active path.

If an edge $e = \{u, v\} \in G$ satisfies $u \in \cup_{i=1}^{k} C_{z_i}$ and $v \notin \cup_{i=1}^{k} C_{z_i}$, then we call it a *border edge.* Because we insert into the soft heap only edges of the border type, if a *border edge* gets corrupted when it is incident to the component C_{z_i} during a contraction, then we call it *bad.* Only the bad edges affect us, thus we must consider them differently. The number of bad edges can be limited to $\frac{m}{2} + d^3 n$.

Theorem 10. ([1]) *The total number of bad edges produced while building T is $|B| \leq \frac{m_0}{2} + d^3 n_0$.*

In mid action, t will represent the target size n_z of each C_z: $n_z = A(t-1, A(t, d_{z_k} - 1))^3$ (number of vertices of each subgraph at level z), where d_{z_k} is the height of $z \in T$. A subgraph C_z includes all of the non-discarded edges of G_0, thus it suffices to specify only the vertices. We will define the *working-cost* of an edge e as follows: if e is *bad*, then the working cost is the same as the current cost (the corrupted weight), otherwise it is the original cost of the edge. Thus we have three types of edge cost (weight): original, current and working. While building T we maintain the following two invariants:

INV1: $\forall i < k$, we keep an edge (*chain-link*) joining C_{z_i} to C_{z_k} whose **current cost** is: **at most** that of any border edge incident to $\cup_{l=1}^{i} C_{z_l}$ and **less than** the working cost of any edge joining two distinct C_{z_j}, $(j \leq i)$. In order to enforce the latter condition we maintain a *min-link*, if it exits, for each pair $i < j$: it is an edge of minimum working cost joining C_{z_i} and C_{z_j}.

INV2: $\forall j$, the border edges $\{u, v\}$ (with $u \in C_{z_j}$) are stored in either a soft heap $H(j)$ or in one $H(i,j)$ (but not in both), where $0 \leq i < j$. If $e \in H(j) \implies \exists \{v, w\}$ *s.t.* $w \in H(i,j)$. If $e \in H(i,j) \implies \exists \{v, w\}$ *s.t* $w \in C_{z_j}$, but $w \notin$ *any* C_{z_l}, $(i < l < j)$. If $i = 0$ then v is incident to no such C_{z_l}. All the soft heaps have error rate $\frac{1}{c}$.

Just a small note to clear any confusion: border edges connect subgraphs in the same level z of the tree, and the working-path connects edges in a vertical path from the root to the leaf. We need separate soft heaps in order to create a buffer zone and fight the *badness* of the edges.

In Figure 7 it is exemplified the difference between the working path and the border edges, and how the tree of contractible subgraphs is being built.

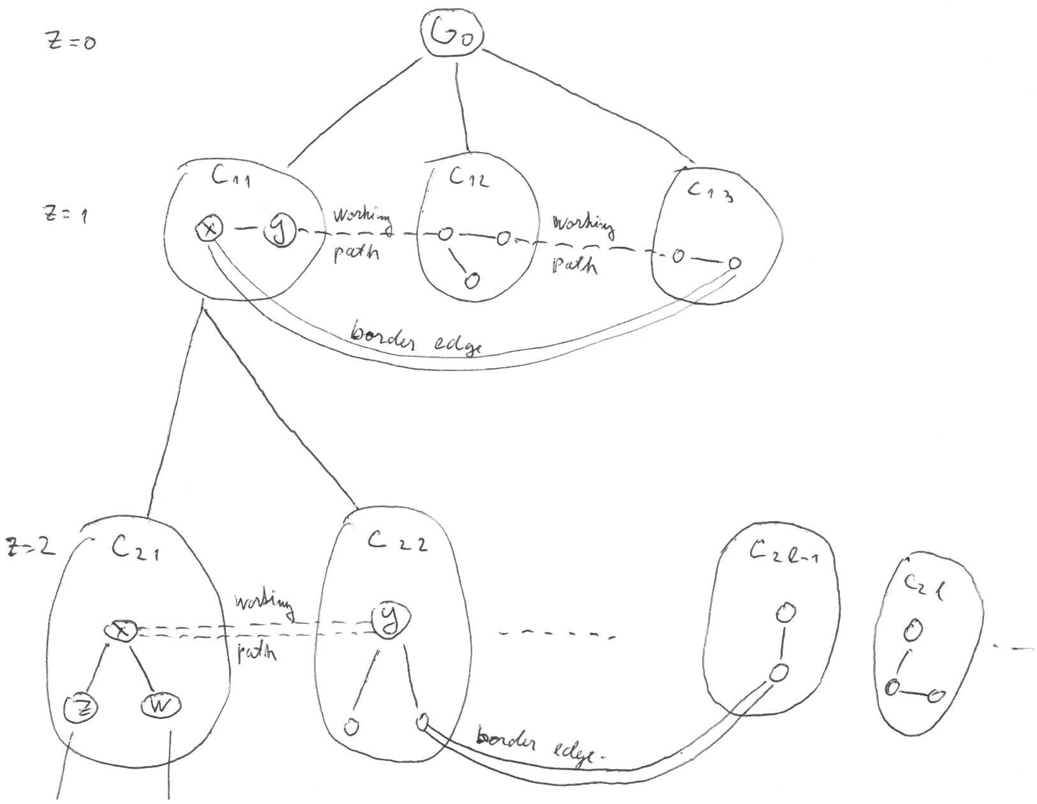


Figure 7. Tree T of contractible subgraphs

When building T, the postorder traversal is driven by a stack whose two operations pop and push which translate into a *retraction* and an *extension* of the active path.

The *retraction* rule: A retraction happens for depth $k \geq 2$, when C_{z_k} has reached its target size $n_z = A(t, d_{z_k} - 1)^3$ (the number of vertices of C_z is n_z). For the case when we are working with leaves, $d_{z_k} = 1$ and thus we just have 8 vertices($n_k = A(t,1)^3 = 8$). Now C_{z_k} is contracted and becomes a vertex in $C_{z_{k-1}}$ (a vertex joined to $C_{z_{k-1}}$ by the chain-link and maybe other edges between $C_{z_{k-1}}$ and C_{z_k}). As a result $C_{z_{k-1}}$ gains one more vertex and maybe more edges; the end of the active path is now z_{k-1}. Because in *INV1* we avoid zero vertices components, if $|d_{z_{k-1}} - d_{z_k}| > 1$ then we add a new node; the heights are understood as heights in the fully build T. Thus *INV1* is maintained in the worst case in $O(k)$ time. In order to maintain *INV2* we use the following subroutine: First the heaps $H(k)$ and $H(k-1,k)$ are destroyed and the corrupted edges are discarded. The remaining elements are partitioned in *clusters* of edges that share the same endpoint outside the chain-link. This can be viewed as: we might not have picked the smallest weight edge in a cycle, thus keep the edges that might be on a cycle formed by component C_{z_k}. Now for each cluster, select an edge $\{r,s\}$ of minimum current cost and discard the others. We must insert this edge in the heap implied by *INV2* as follows: If the edge comes from $H(k)$ and shares s with with $H(k-1)$ or if it comes from $H(k-1,k)$ then it also shares s with $H(i,k-1)$ and can be inserted in $H(k-1)$; otherwise the edge comes from $H(k)$ and shares s with $H(i,k)$ thus it can be inserted into $H(i,k)$, with $i < k-1$. Finally for each $i < k-1$ MELD $H(i,k)$ to $H(i,k-1)$.

The *extension* rule: Perform a FIND-MIN on all of the heaps in order to get the border edge $\{u,v\}$ of minimum current cost $w(u,v)$. We call such an edge the extension edge. Now from all of the $min-links$ of working cost less than $w(u,v)$, find an edge $\{a,b\}$ incident to C_{z_i} with smallest index i (the leftmost subgraph). If we find it we perform a $fusion$ by contracting $\cup_{j=i+1}^{k} C_{z_j}$ into C_{z_i}. After such a contraction, we update the relevant min-links in $O(k^2)$ time. Similar to the retraction rule, we must update the heaps $H(i+1)...H(k)$ and all $H(j,j')$ with $i \leq j < j'$. We discard all corrupted edges, regroup the remaining ones into clusters and for each cluster insert the edge $\{r,s\}$ of minimum current cost and discard the others. Again when inserting $\{r,s\}$ we must consider two cases: If $\{r,s\}$ comes from $H(j,j')$ or $H(j')$ and shares an edge with $H(j,j')$ $(i \leq j < j')$ then by *INV2* it also shares an edge $\{r',s\}$ in some $H(h,l)$ $(h < i \leq l)$, thus $\{r',s\}$ will migrate into $H(h,i)$ through melding of the heaps if it was not there, and we can insert $\{r,s\}$ into $H(i)$; Otherwise $\{r,s\}$ comes from $H(j)$ $(i < j)$ and shares an edge in some $H(h,j)$ $(h < i)$, thus we can insert $\{r,s\}$ to $H(h,j)$. Now after the insertion, for each h,j $(h < i < j)$ meld $H(h,j)$ into $H(h,i)$. Because of *INV1* vertex a cannot be further down the chain than u. Regardless if a fusion took place, we made v into a single-vertex C_{z_k} and the extension edge $\{u, v\}$ into the chain link. If a fusion took place we increment k by one and the end of the active path is now z_k. Old border edges incident to v are no longer of border type and should be deleted from their respective heap. Find the min-link between v and $\cup_{i=1}^{k} C_{z_i}$ and insert the new border edges into their appropriate $H(i, k)$.

To construct T, at any given node z_k of height at least 1, we perform extensions as long as we can (stack push). We stop when the condition for a retraction has been met. For the root z_1 there is no retraction condition. The algorithm stops when we run out of border edges and extensions are no longer possible. If no fusions take place, for any internal z distinct from the root, the expansion C_z has $n_z A(t, d_z - 1)^3 = A(t, d_z)^3$ vertices. The expansion of C_z consists of vertices of G_0 mapping into C_z and the edges of G_0 joining pairs of them. Fusions force contractions before the target size is reached and by creating arbitrary large expansions. Any C_z whose expansion exceeeds the allowed size is broken in subgraphs of the right size.

One must remark that a *fusion* is not a *retraction* as the edge under the fusion is also contracted. The entire algorithm can be imagined as one giant "fusion" and computing MST in linear time.

4.2 Best Deterministic Algorithm

The idea of the algorithm is to verify if a component C of the graph is contractible before computing its MST. We decompose G in vertex disjoint contractible subgraphs C_i of decent size. Contract each such element into a single vertex and decompose the suitable minor into a set of C_i' contractible subraphs. Repeat until G becomes one single vertex.

```
procedure msf
input $G, t$: undirected graph $G$, and the target size $t$
output minimum spanning forest
if $t == 1$ or $n == O(1)$
  | return Prim-MST(G) by direct computation
Perform $c$ consecutive BorůvkaPhases
Build $T$ and form $B$ graph of bad edges
$F = \cup_{z \in T} msf(C_z - B, t - 1)$
return $msf(F \cup B, t) \cup \{edges\ contracted\ in\ boruvka\ phase\}$;
```

Figure 8. Chazzelle's algorithm

5. Optimal Approach

In this section we summarize the optimal approach considered by Pettie and Ramachandran [16]. The idea revolves around pre-computing optimal decision trees which answer queries to small subproblems that are isomorphic to each other, thus one such decision tree can be used for multiple contractible subgraphs. Throughout this section when we talk about a contractible subgraph we refer to one obtained from a graph G by expanding a single fragment a constant number of times (as explained in Section 4.2).

For graphs that are sufficiently dense ($\frac{m}{n} \geq \log^{(3)} n$) there are algorithms that solve the problem in linear time. We will exploit this in our optimal construction of the algorithm by creating dense enough components (by a series of vertex contractions).

Lemma 2. (Robust Contraction Lemma [16]) *If T is a tree of MSF edges, we can contract T into a single vertex while maintaining that the MSF of the contracted graph $\cup T$ gives the MSF of the graph before contraction.*

We use this lemma to contract certain corrupted trees (MSF trees of certain subgraphs that got corrupted in the soft heap). Let $G \Uparrow M$ be the graph derived from G by corrupting each of the edge weights in M. Let M_C be the corrupted edges with exactly one endpoint in C. Let G/C the graph obtained from G by contracting all connected components in C.

Lemma 3. ([16]) *Let M be a set of edges in a graph G. If C is a subgraph of G that is contractible with respect to $G \Uparrow M$, then $MSF(G) \subset MSF(C) \cup MSF(G/C - M_C) \cup M_C$.*

We use the above two lemmas to create a partition procedure which splits the graph in edge disjoint subgraphs.

5.1 The Partition Procedure

Pick a vertex v, mark it, add it to the set V_i and create a soft heap with all of its edges. Repeat until V_i reaches a certain size or there are no more unmarked vertices: let $x \in V_i$, find and delete the minimum cost edge $\{x, y\}$ from the soft heap. Repeat this procedure until y is no longer part of V_i, then add y to V_i. If y was not visited, then insert all of its edges in the soft heap. Set all vertices in V_i as marked. All of the corrupted vertices with one endpoint in V_i are added in M_C (thus M_C is the set of corrupted edges that connect different components C_i). Remove from the graph G the corrupted border edges (the one added in M_C). Dismantle the soft heap and move to another vertex that was not visited in order to build a new component. $C = \cup_i C_i$ where C_i is the subgraph of G induced by V_i. The partition procedure is similar to the algorithm presented in Section 4.2 ([6]) with the particularity that we use a soft heap instead of a Fibonacci heap. A component stops growing if it attaches itself to another component or it reaches a maximum size. Each edge is inserted into the soft heap no more than twice and extracted no more than once.

The partition procedure presented in this section is expanding one component at a time while the partition procedure used in the previous section is based on contracting components. They can be viewed as complementary procedures.

Lemma 4. ([16]) *Give a graph G, any $0 < \epsilon < 1/2$ and a parameter maxisze, the partition procedure finds edge-disjoint subgraphs $M_C, C_1, C_2, ..., C_k$ in time $O(E_G \cdot \log(1/\epsilon))$, while satisfying several conditions:*

- $\forall v \in E_G, \exists i$ *s.t.* $v \in E_{C_i}$,
- $\forall i, |V_{C_i}| \leq maxsize$,
- *for each conglomerate* $P \in \cup_i C_i$, $|V_P| \geq maxsize$ *(in the partition procedure, the first component being expanded will reach* maxsize*, then the following components being expanded will either reach* maxsize *or connect to the maxsize component; we call a* conglomerate *a component that has reached* maxsize *and all of the components being connected to it),*
- $|E_{M_C}| \leq 2\epsilon \cdot |E_G|$
- $MSF(G) \subseteq \cup_i MSF(C_i) \cup MSF(G - \cup_i MSF(C_i) - M_C) \cup M_C$

5.2 Decision Trees

A MSF decision tree is a rooted tree having and edge-weight comparison associated with each internal node. Each internal node has two children, one representing that the comparison is true and the other one false. The leaves list off the edges in some spanning tree of the graph. A MSF decision tree for G is *correct* if the edge comparisons encountered at any path from the root to a leaf uniquely identify the spanning tree at the leaf as the MSF. A decision tree for G is optimal if it is correct and there exists no other correct decision tree of lesser depth. The total time to find an optimal decision tree by brute force search for all graphs on fewer than r edges is bounded by $2^{r^2} r^{4 \cdot 2^{r^2}} \cdot (r^2 + 1)! \leq 2^{2^{r^2} \cdot r^{4 \cdot 2^{r^2}} (r^2+1)!}$ (there are $2^{\binom{r}{2}}$ graphs on r vertices; Prim-MST uses at most r^2 comparisons, thus the tree has fewer than 2^{r^2} internal nodes; there are $r^{2^{r^2+2}}$ distinct decision trees to check; and to determine if one decision tree is correct, we generate all possible permutations of the edge weights: $(r^2 + 1)!$). Setting $r = log^3 n$ allows to compute all decision trees in $o(n)$ time. For example if $n = 2^{4^{512}}$ the size $r = 10$, thus one decision tree answers querries for a 10 edged subgraph.

5.3 Optimal Algorithm

Precompute all of the optimal decision trees for all graphs of $\leq \log^3 n$ vertices. Call the partition procedure and get the set of subgraphs $\{C_i\}$ while discarding the corrupted edges, as each component is completed. For each C_i, compute the MST F_i using the pre-computed trees. Contract each connected component spanned by the forests, the resulting graph has $\leq \frac{n}{\log^3 n}$ vertices. Use the dense case algorithm to compute F_0 of such contracted component. The MST is contained in $F_0 \cup F_1 \cup ... \cup F_k \cup M_{C_1} \cup ... \cup M_{C_k}$. Apply two Borůvka steps to this new graph and compute the MST recursively on this new graph.

```
**procedure** Optimal-MSF
**input** $G = (V, E, C)$: undirected weighted graph
**output** minimum spanning tree
**if** $E = \emptyset$
    **return** $\emptyset$
$maxsize = \lceil \log^{(3)} |V| \rceil$
$M, C = Partition(G, maxsize, \epsilon)$
$F = DecisionTree(C)$
$G_a = G/F - M$
$F_0$ =Prim-MST($G_a$)
$G_b = F_0 \cup F \cup M$
$F'$ =BorůvkaPhase($G_b$)
$F' = F' \cup$BorůvkaPhase($G_b$)
$F$ =Optimal-MSF($G_b$)
**return** $F \cup F'$
```

Figure 9. Optimal MSF; The *DecisionTree* procedure takes as input a collection of graphs with at most *maxsize* vertices and returns their MSF using the pre-computed decision trees explained in the previous subsection; The BorůvkaPhase procedure modifies the input graph into a contracted graph.

6. Historical Conclusions

6.1 Greedy Algorithms

The greedy algorithms are decent in practice and intuitive to follow. The implementations are easy, and all of the new algorithms are built on top of subproblems that are solved in a greedy fashion. Kruskal's algorithm works well on sparse graphs, Prim's algorithm suitable for dense graphs, and Borůvka's algorithm is used to increase the density of the graphs.

6.2 Randomized Algorithm

In practice the random bit generation must be accounted for. The idea is good, it is just the repetitive sampling phase that is a bit unnatural as it gets patched with a verification algorithm on every iteration in order to remove the heavy edges.

6.3 Verification Algorithm

The verification is solid, the implementation is complex but sound.

6.4 Best Deterministic Algorithm

The idea is interesting and explained in simple words it goes like this: a greedy algorithm produces a correct solution almost too correct and too expensive; how can we make the steps in the greedy algorithm faster and cheaper, at the cost of a solution that is not too far a way from being correct. We can view this process as a student before an exam: instead of studying constantly through the semester, the student skims through it building a superficial knowledge base and before the exam it makes up for the skipped parts without going over the deadline. This process is referred to as the discrepancy theory; produce a solution that is not too far away from the correct one. In practice the algorithm most probably performs poorly as there are too many heaps with too many pointers. The soft heap is perfect if we do not do any deletions.

6.5 Optimal Algorithm

Because the precomputation of the decision trees is based on the size of the edge weight universe, the answer to the question: "How to precompute the decision trees?" remains open. Furthermore in order to find all decision trees for a graph of r vertices in $o(n)$ time; the graph must be split in components of size $\log^{(3)} n$, this means that for a graph with $n = 2064^{2064^{2064}}$ vertices each component will be of size ≈ 14. This does not seem very practical.

Chapter IV. Documenting Our Trials

By this point we have figured out that in order to solve the problem people tend to attack two approaches: one tries to split the problem in very dense subgraphs (Pettie and Ramachandran's optimal [16]) the other approach in sparse subgraphs (the linear randomized of Karger, Klein and Tarjan [10]). Chazelle seems to understand that in order to solve the problem one must attack on all fronts, and this is exactly what we are going to do throughout this chapter.

The disjoint-set data structure seems to do subtle extra work. Maybe an improvement in that direction would be an approximate disjoint-set data structure that avoids the subtle extra work. Expanding on Chazelle's idea, we can try to use the discrepancy method to construct an almost good disjoint-set data structure. Then maybe solve the problem a constant number of times to get a pair of disjoint decent approximations (similar to the "probabilistic" method of Karger, Klein and Tarjan [10]). Can we generate all pair of approximations with the following property: given c approximations $T_1, ...T_c$ the edges that do not differ in the construction of the trees are for sure in the MST, and the ones that do differ can be used to construct the missing part of the MST. Hopefully this will be close enough in order to correct the output in linear time in order to create the MST.

This chapter is structured in four sections, The Bad, The Ugly, The Good and the algorithm to rule them all. In the first section, we present the bad ideas and thoughts that have little to no value. In the second subsection we present the valuable ideas that are lacking a proper formalization. In the third subsection we present the good and potential good ideas of which we have a solid proof. In the forth section we glue everything together and propose our algorithm.

In Section 1.1 we argue that the running time of a MST algorithm is almost linear if the graph is very sparse ($\frac{m}{n} = c$ and c is a small constant).

In Section 1.2 we explain how to partition the graph in order to make the disjoint-set operations linear.

In Section 2.1 we explain and try to formalize that it is possible to make the disjoint-set operation linear in the running of a MST algorithm.

In Section 3 we explain how to partition our graph based on the density of its components, how to trim the trees formed, and how to classify the cycles of a sparse graph.

In Section 4 we propose the algorithm that runs in $O(m + n)$ time.

1. The Bad

In this section we debunk a farrago of our judgments that aim to improve on the understanding of the subtleties of the problem. Although the reader might consider this section futile, we view it as an exercise that complements our understanding of the literature.

1.1 Variations of Prim's Algorithm

In this subsection we explore some variations of Prim's MST algorithm, and claim that for very sparse graphs where $\frac{m}{n} = c$ and c is a small constant, the running time is almost linear.

In Prim's algorithm we start with one vertex and grow its smallest incident edge until the whole graph is covered. A Fibonacci heap is used to store the minimum and towards the end of the algorithm it becomes costly to Extract-Min, $O(\log n)$. Fredman and Tarjan [6] propose to pick a vertex and expand until a certain size it is reached, then pick a new one. This will build a collection of trees. Such an algorithm works on phases, and at each new phase, one old tree represents one new vertex. The underlying data structure that is used is still a Fibonacci heap. Pettie and Ramachandran [16] propose in their partition procedure to apply a phase of Fredman and Tarjan's version of the algorithm, but instead of using a Fibonacci heap use a soft heap. By using a soft heap they find contractible subgraphs with an error margin. However, the Insert and the Extract-Min operations run in constant time in their partition procedure.

What if for each vertex we would sort all of its edges? Then the bottleneck would be $O(n \log m)$. What if for each vertex we use an approximate sorting of the vertices via a soft heap? The problem with using soft heaps is that there is too much corruption. If we set the corruption parameter as a factor of the degree of the vertex, then we would not encounter problems with corrupted elements. The problem we encounter is the price of insertion which will lead to running time of $O(n \log m)$. There is not much gain from asking such questions. Let's try to keep the corruption factor constant $\epsilon = \frac{1}{8}$. The first Extract-Min will return a set of items on which the minimum lies, after a second Extract-Min operation we do not know if the minimum is in the same set. Consider two vertices u, v who both point with a corrupted minimum to each other. The chances are that either one has the correct minimum or both have the wrong minimum. Would this approximation be good enough? I would call this approach blindly trusting your neighbours to make up for your lacking. It might be useful in order to get a poor approximation of how the tree structure is formed.

Now let's consider that the graph is very sparse, so sparse that the edge to vertex ratio is bounded by a constant c. We could use a soft heap with the error rate $1/c$ and on average only one edge will be corrupted. The good thing is the first Extract-Min performed will return a set k in which all values $< k$ lie. Thus all the corrupted items in which the minimum lies can be found in one EXTRACT-MIN. The problem is that the constant c might be big enough, and the complexity of inserting c edges per vertex will be $O(c \log c)$. If instead of a soft heap we use a Fibonacci heap then the complexity of extracting k minimums will be $O(k \log c)$, which is better for $c < k$. Let's set $\epsilon = \frac{1}{8}$, and bound the component size to be 8. Considering that on average each vertex has $\approx c$ edges the following analysis takes place in the same idea of Fredman and Tarjan [6]:

We expand trees using Prim-MST until a certain size is reached, then we consider such a tree as a new vertex in a new iteration. Between iterations, the time taken to do a cleanup is $O(m)$; if t is the number of old trees, then the time for growing a new tree is $O(t + m \log 8)$, we need t EXTRACT-MIN operations, one for each tree, and another m insert operations. In order to bound the number of passes we must consider the effect of a pass that begins with t trees and $m' \leq m$ edges, then each tree remaining after the pass has more than 8 incident edges (as trees tend to connect to each other). The number of trees that remain after each pass satisfies $m' \leq m/4$, as some edges might have both of their endpoints in the same component. As with each pass the reduction is by at least a factor of 4, we need $\min\{i | \log^{(i)} c \leq 1\}$ in order to remain with a collection of n vertices and n edges. This analysis sketch follows the structure of Fredmans and Trajans analysis of improved MST with Fibonacci heaps [6, Section 5].

By following this thought process if we have a constant c small enough, for example $c = 2^4$, and $m = c \cdot n$, then the following process will run in $3 \cdot O(n \cdot 2^4)$ time. But if instead $c = 2^{24}$ then the running time will be $4 \cdot O(n \cdot 2^{24})$. We must account for the impractical $\epsilon = \frac{1}{2^{24}}$ which will represent the the running time of insertions in the soft heap. The insertions in the soft heaps is $O(\log \frac{1}{\epsilon})$, and for our example every insert will take $O(24)$ time. Thus the actual running time will be $4 \cdot O(n \cdot 2^{24} \cdot 24)$.

We conclude that the MST of graphs that have very few edges can be computed in linear time, and for dense enough graphs it can be computed in linear time, thus we must either create dense enough subgraphs or subgraphs where $\frac{m}{n} = c$, and c is very low.

1.2 A Reviewed Partition Procedure for Building Disjoint-Sets

In order to build a hierarchy of cycles we wish to first identify the smallest possible cycles in our graph, and then build the longer cycles as an expansion of the small ones.

Vertices with more edges are most likely to be part of more cycles than vertices with fewer edges. The partition procedure proposed in Chapter 3 Section 5.1 makes use of the cut property. It produces a high amount of discrepancy that must be accounted for because of the soft heaps usage. While detecting cycles we do not require to deal with edge weights at first, we can postpone the use of the cut property until later in our algorithm, once we have more order. We would prefer to partition our graph in such a way that the low degree vertices orbit around the high degree vertices.

In this way if we trim or contract the low degree ones, the partitions will represent a dense enough graph. We propose the following modification to the procedure: instead of picking a vertex and growing the components with probably the minimum weight edge, we pick a high degree vertex and grow the components with probably the lowest degree vertices; if the picked vertex has no neighbouring low degree vertices we add it to the set HDV and pick another suitable one. The set HDV will represent the set of high degree vertices. We will postpone the processing of HDV for after the hierarchy is built, but note that HDV will represent an almost fully connected subgraph of G, thus Prim's algorithm can be applied to get in linear time the MST of HDV. However, we need to account for vertices that have near maximal degree; in such a partition procedure they represent dead ends. If we encounter such a vertex, then it will absorb the whole graph.

We can use this version of the partition procedure without affecting Theorem 12, we just need to be careful to account for the discrepancy produced in a different manner. The gain is that we do not work with edge weights, thus we do not need to get the exact minimum degree of a vertex.

A style question that arises: would it be better to pick as starting points vertices of minimal degree and expand the components with other vertices of low degree? The cycle detection procedure is complementary to the partition procedure.

2. The Ugly

In this section we present and try to prove the unorthodox ideas on which we build our algorithm. We acknowledge that some parts might have holes, but we try to postpone patching them until later.

2.1 Tailoring Disjoint Sets

In this subsection we try to adapt the disjoint sets data structure such that for our specific problem a series of m operations and n elements will take no more than $O(m + n)$ time. In the analysis of disjoint-sets with path compression and union by rank [17, Chapter 2, Section 2] a credit system is used to achieve the bound $O(m\alpha(m, n))$, where m represents the number of operations and n the number of elements. For each level $i \in [0, 1, ..., \alpha(m, n) + 1]$, blocks of integers $k \in [0, 1, ..., \log n]$ are partitioned such that the rank of a given element x lies in the same partition as the rank of its parent. We just briefly mention the analysis and we do not expand on it. If the structure of the unions is known in advance, there is a linear-time algorithm for disjoint-set union [7]. This algorithm operates on a RAM model as it needs table look-up for small sets.

How can we exploit the structure of our MST and how the components get formed in order to take advantage of such a table look-up? The verification algorithm for the MST also uses a table look-up; thus we assume this kind of tricks are allowed as long as we do not make any assumptions on the edge weights.

We view the disjoint set as a rooted tree T of n elements. We can use the knowledge of this tree in order to precompute the answers to FIND operations on small sets. Thus the valuable idea is to carefully craft the MST in such a way that we gain enough information in order to use a table look-up and precompute the answers to FIND operations. In the following lemmas b refers to the size of the precomputed tables.

Lemma 5. ([7, Lemma 2]) *If b is $\Omega(\log\log n)$ and each execution of $microfind$ requires $O(1)$ time, then the total time for m intermixed link and $find$ operations is $O(m+n)$.*

Note that b doesn't have to be $\Omega(\log\log n)$, much smaller values can suffice.

Theorem 11. ([7, Theorem 1]) *With an appropriate choice of b, the algorithm for static tree set union runs in $O(m+n)$ time with $O(n)$ preprocessing and uses $O(n)$ space.*

We propose the following theorem and try to sketch a proof. However, we do not account on how it will influence the running time of the algorithm and we postpone such a discussion for later.

Theorem 12. *Given an undirected weighted graph G of n vertices and m edges, in the computation of its minimum spanning tree the disjoint set operations can be bounded to $O(m+n)$ running time, and can allow a degree of error that must be accounted for.*

Proof. We must consider how the union tree will be build in the running of our MST algorithm. If we expand the exact minimum incident edge of each vertex then the union tree has exactly the same structure as the minimum spanning tree. Once we have components of size $\log\log n$ we can create the table in which we answer find queries in $O(1)$ time. The problem that arises in such an expansion is that we use a disjoint-set data structure with n elements and $n\log\log n$ operations in the worst case to keep track of such components, thus before we can build the table the cost is already bounded by $O((n\log\log n)\cdot\alpha(n\log\log n, n))$, which is not good.

Our luck is that we do not require for the union tree to have the exact same structure as the minimum spanning tree, it suffices for the union tree to have the structure of a spanning tree that shares enough common edges (or cycles) with the MST. By doing so we can trust that in the near future the components will be connected.

The only purpose for using the disjoint set is in cycle detection, hence one must use it exactly for that. Consider an approximation of the components that get formed in the MST cycle detection. In our search for cycles, at a point t in time, we do not need to know the exact components we are going to union, if at time $t+c$ the faulty merged ones will be part of the same good component (where c represents a relatively small constant).

Assume we are searching for cycles and at time t the components $c_1, c_2, ..., c_k$, $k \leq n$ are formed. Assume we have a spanning tree T such that $\epsilon n \leq |MST \cap T| < n$, $0 < \epsilon \leq 1/2$. Let $T_{(t)}$ denote the edges added at time t

in the construction of the tree T. In the worst case, at time $t+1$ in the building of T the components are merged two by two, and in the building of MST the exact opposite pairs of components are merged such that $T_{(t)} \cap MST_{(t)} = \emptyset$ this reduces k by a factor of 2. Considering that at most $n - \epsilon n$ edges differ, $\cup_{i=1}^{c}(T_{(t+i)} \cap MST_{(t+i)}) \neq \emptyset$, and the components in which we search for cycles will represent subcycles in both T and MST. A component can be part of many cycles of different lengths. During our running of our algorithm, if we decide to apply the cycle property, it does not matter that much if we start with a long cycle, or with a shorter one that is included in the long one. However because our components can contain errors, we must enforce an ordering of processing the cycles. We start with the longer ones and proceed with the next shortest one. As an unwanted effect we might disconnect components or not connect them at all.

What happens when the answer to queries is incorrect. There are two cases to consider when using the tables:

Case 1: If the components are connected but the answer to Find is false a Union is usually performed, then the components will get connected again which will take $O(1)$ time since there is nothing to do but try to connect. We can correct the value of the table in $O(1)$ to properly reflect the component, and mark the Find as faulty; thus only after a Union we can check if the edge under inspection is part of a cycle or not.

Case 2: If the components are not connected and the answer to Find is true, then the components will not get connected at this point in time. We might consider an edge as part of a cycle even if it isn't. This can create *phantom cycles* in the running of the algorithm. The phantom cycles can be addressed after the hierarchy of cycles has been established. The obvious side-effect is that some edges that are part of the MST might be omitted. It is best to minimize the impact of this case by building an almost correct MST where we allow some cycles and extra edges, as long as the number of edges is in a constant factor of n.

Use a partition procedure similar to the one presented in Chapter 3 Section 5.1 with the parameter $maxsize = \log\log n$. By Lemma 4 the components formed have size $\leq maxsize$. The rest follows from Theorem 11 [7, Theorem 1]. The disjoint set might answer faulty queries that might get corrected in time.

Given an algorithm A based on the classical disjoint set forest, if we substitute the data structure, then algorithm A with the linear time disjoint set will not output correctly. An algorithm B based on the linear time disjoint set can be made to output correctly with extra work. The hard part is to make the extra work be linear as well. □

In order to make use of Theorem 12, the MST algorithm should focus on building a *pseudoforest* of MST edges. We define a pseudoforest as a collection of sparse graphs in which each connected component has at most one cycle. We want the components of our pseudoforest to have the following property: by removing the heaviest edge from a components cycle, the remaining edges are all part of the graphs MST.

Two questions remain to be addressed: how to build the hierarchy of cycles and how to enforce $MST(PF_G) \subseteq MST(G)$ (where given a graph G, PF_G denotes a pseudoforest of G).

2.2 Pseudoforests

The beauty of the pseudoforests lie in the fact that they are a collection of sparse graphs, which are almost a collection of spanning trees. If one can create a minimal spanning forest in constant time from a pseudoforest that is a connected enough, then we just need to expand the connected components with the lowest incident edge and one shall soon have a minimum spanning tree. If we use the faulty disjoint-set data structure, then we will be talking about pseudoforests, probably more than one cycle can find its way in a component.

The pseudoforests are a side product of using the cycle hierarchy with a faulty disjoint-set. We need the cycle hierarchy in order to efficiently trim the trees that are subtrees of the MST. When a trim occurs that has as a root a phantom cycle, either the component will become disconnected or the trim will stop too soon.

3. The Good

In this section we present the ideas of which we are sure they are good.

3.1 Data Structure

For each vertex we need the following attributes:

- $d(v)$ - the degree of the current vertex,
- ld - list of low degree neighbours, vertices of degree $d < c \cdot \log^{(3)} n$, $0 < c \leq 1$
- hd - list of high degree neighbours, vertices of degree $d > c \cdot \log^{(3)} n$, $c > 0$,

We can compute such a split in $O(m + n)$ time by first iterating over the vertices and setting their degree, and then iterating again and setting their neighbours. We visit each edge twice per iteration. Should the lists ld and hd be Fibonacci heaps, or soft heaps, or simply lists?

3.2 Density Partition

We partition our graph based on the density of the components.

We classify the vertices based on their degree and neighbours degree. Thus we can have high degree vertices (HD) with high degree neighbours (HDN), low degree neighbours (LDN) and mixed degree neighbours (MDN); similar for low degree vertices (LD). The ones with mixed degree neighbours will represent the bridge between components.

Considering HD-LDN and their complementary LD-HDN: while working with edge weights, two actions can happen; either the LDN neighbours get absorbed in the HD vertex after a series of contractions, or they will represent border neighbours that will get absorbed in a HD-MDN/LD-MDN component. At this phase we are only allowed to contract the LD neighbour that will end up either directly into the HD or into another LD that lies in the same orbit of the HD under investigation. The contracted edges are part of the MST.

Considering HD-HDN: apply Prim-MST on such components, leaving the edges of border type untouched. For example if a high degree neighbour would expand into a component of low degree, abandon the expansion, move to another unprocessed vertex in the same component under investigation. The expanded edges are part of the MST. This process might lose HD components that can be connected in the MST. If at component level there are vertices that got disconnected, store in a soft heap the minimum amount of edges needed to complete the minimum spanning tree of the component. The collection of such soft heaps might be useful when connecting the disjoint components.

Considering LD-LDN: the components are sparse enough in order to process the edges in the increasing order by weight. Thus we apply a variation of Kruskal's algorithm and discard the F-heavy edges, remaining with only the F-light ones. It is similar to the process of discarding F-heavy edges in the random sampling phase presented in Chapter 3 Section 2, except that we do not need to flip coins; the components are already sparse enough. This components get absorbed in LD-MDN, or HD-LDN quite fast. If the graph is too sparse and the components LD-LDN are too many or too big, we must consider an alternative to the above technique, as the complexity will be $O(m \log n)$; we would explore such a solution in the next section. If the graph is predominantly dense, and the sparse components are few, one can apply some BorůvkaPhases on just the sparse components until they get absorbed in the dense ones. The remaining graph will be dense enough and the problem would be solved.

Considering HD-MDN and their complementary LD-MDN: the idea is to postpone the processing of this components until we get rid of all of the dense and sparse components. Now let's consider the working sets of edges; on the same collection of vertices we will treat the lists ld, and the list hd as two separate graphs. Now let's assume we are working with n_0 vertices, we compute two separate spanning trees such that the MST of the component lies in $MST(G_s(n_0, ld)) \cup MST(G_d(n_0, hd))$. The sparse graph can be computed recursively, as we will explore in the next section or in linear time if it is almost a tree, and the dense one again in linear time. The number of edges after the reunion will be $2n_0$.

If the MST of each density partition is computed in linear time then why is the problem not solved? To answer that, we must account for the border edges. As every partition can have k edges replaced, where k represents the number of disjoint components its border edges are incident to. We do not know the density of the forest created by the border edges. But we know that if we disregard the border edges, every component after being processed is sparse. We now must consider the components as super nodes and remove the parallel border edges between them (if they point to the same component). We can now build again the components around the border edges that have not been processed and recurse the process once in order to have a sparse enough graph. Now the edge to vertex ratio should be small enough. The problem with using super nodes is that they share the complexity of running disjoint-set operations in the long run.

Because the whole graph can be sparse, it is a common practice to make use of BorůvkaPhases to increase the density. We try to postpone such practices, in the construction of the algorithm we shall use it in order to melt cycles and avoid dead ends.

3.3 Trimming Vertices of Low Degree

We can start trimming the graph by contracting all of the vertices with degree 1. There might be none. How can we force such vertices to appear? Let's think about the different types of cycles we can encounter in our graph: there are *simple cycles* (where we do not allow any vertex and edge repetition), there are *induced cycles* (cycles where no two vertices of the cycle are connected by an edge that itself does not belong to the cycle), and *peripheral cycles* (cycles that do not separate any part of the graph from any other part). If we use a disjoint-set data structure, we can expand each vertex with all of the incident edges and find quite fast cycles of small degree. Consider such a partitioning where each subgraph might not be contractible but has only induced cycles in itself and border edges that might be part of a bigger cycle. If we were to build a hierarchy of cycles we would need to treat each subgraph as a single vertex at one iteration and just deal with the border edges from the previous ones.

Theorem 13. *Edges that connect components of degree* 1 *belong to the MST regardless of their weight.*

Proof. By not adding the edge, the component becomes disconnected and we no longer have a tree. □

The follow-up is that we do not need to consider the components with one border edge, thus the MST of such components is a subtree of the MST of the graph.

When we start a sequence of trim operations, they might cascade, thus we first iterate through the whole graph and trim every such component, then check their LDN list in order to remove the newly created low degree components. In one such trim cascade we can lose all of the vertices that are classified as HD-LDN or LD-LDN. The running time is dependent on the graphs saturation with cycles, each components trim-cascade stops once a vertex that is part of a cycle is reached. The trimming ensures that only edges that are part of cycles remain into further consideration. If we start trimming a tree, then we will have as many cascades as the size of the tree. This will give a running time of $O(n^2)$.

In order to get an improvement we must exploit the stop condition of the cascades. If we know the cycle structure beforehand, and given a vertex of degree 1, we know the tree it is part of such that the root of the tree represents a cycle, we no longer require n cascades, and the trim procedure will run in $O(n)$.

We propose to first identify the cycles in the graph, then proceed with removing the heaviest edge, and then trimming any trees created.

3.4 Hierarchy of Simple Cycles

Assume we have built such a hierarchy of minimal cycles. It is a far stretch as we have not shown yet how to build it, we just hypothesize. What would be the depth? In the best case at the first iteration, all of the subgraphs have 3 vertices, and at each such level up in the hierarchy each new subgraph has 3 vertices (the smallest possible cycle), thus the total number of levels would be $\log_3 n$, with a vertex reduction of $n/3$ per level. In reality each subgraphs vertex size is dependent on the number of edges incident to its vertices. We haven't used any edge comparison until now. We can view the problem as a graph expansion, where each level in the hierarchy represents an artificial vertex expansion doubled by an edge distribution amongst the newly created vertices.

While we build the hierarchy of cycles we do not perform any edge comparisons, it would be faulty to use an edge contraction on each individual cycle subgraph as the border edges might be the minimum incident edge, this is why we start building the MST top-down. We are not interested in vertices of degree n or very high degree, thus we can treat them separately and try to split the section of the graph in which they lie as a fully connected subgraph. Vertices of high degree are for sure part of many cycles. We are aiming at working with sparse enough graphs, which can be created in linear time from the original graph.

Start by picking the vertex with the lowest degree and use UNION on all of the incident edges that have a low degree endpoint. Repeat for each vertex incident to the original one until a FIND operation returns the same element. Once we find the first cycle, we continue with the remaining edges and find all cycles that are formed with the same tree. We try to avoid high degree components as they are part of too many cycles. Partition the rest of the edges into two sets: one set of border edges and one of incident edges in the subgraph. Contract the component into one vertex, ignoring the set of border edges and inspecting the inner cycles in order to eliminate the heaviest edge (in such a component we do not grow the MST with the contracted edges, we eliminate the edges that are not in the MST for sure; we can also grow it by adding edges that are incident to vertices that have no border edges). Once finished proceed with a new disjoint-set for a new vertex in the original graph, ignoring any border edge encountered (we keep track of the encountered border edges, once a border edge has been encountered twice add its endpoints to a new set, and increment its level in the hierarchy).

At this point there is a bifurcation in our approach. For each contracted component we can use a soft heap to keep its border edges; and for each level use another soft heap where we add the border edges encountered twice. If we go with this approach, it will be similar to what Chazelle is doing: we start building the hierarchy in postorder, and it will be a contractible component hierarchy with respect to subtrees from left to right. If we continue with the new disjoint-sets we start building a new level disregarding the order in which the edges will be contracted. We do not sin to approximate but we must do extra work. Maybe a combination of the two methods will represent a trade-off or improvement?

Such an approach might be erroneous as if the graph has dense components, the majority of the cycles are formed with the dense components, it would be hard to create such a hierarchy. It would rather create a cut of the dense components using sparse components. Thus we must always work with sparse enough graphs.

3.5 Classifying the Cycles by Levels

Lemma 6. *Given an undirected connected graph $G = (V, E)$, the vertices of high degree are part of more cycles than the vertices of low degree.*

When we classify the cycles of the graph by levels, we ideally want at level 0 to have cycles of length 3, at level 1 cycles of length 4, and so on. We define a vertex $v \in V$ as having a high degree if $degree(v) \approx c_1 \log\log\log |V|$, where $c_1 > 0$ is a constant. If the degree of a vertex is too high $degree(v) \approx c_2|V|$, where $0 < c_2 \leq 1$ then such a vertex will absorb the whole graph. Vertices of very high degree must be treated separately as they are part of almost every cycle. We can consider the grouping of vertices of very high degree as a separate dense graph, or as a binder of the graph under inspection.

Lemma 7. *A direct implication of Lemma 6, vertices of higher degree have influence over far more multiple levels than the edges of low degree.*

Edges should be compared only with edges on the same level.

To build level i, use a disjoint set called $L_{(i)}$ to keep track of the sets formed in this level. This will represent the collection of trees who have as a root a cycle. Each node in the tree will present a cycle-rooted tree from a previous level. To classify the cycles inside the level pick an unvisited vertex of high degree v, and in a disjoint set $N_{(v)}$ add all of its neighbouring vertices of degree higher than 1; add the neighbours of degree 1 in the MST. For each added neighbour call FIND on its unvisited neighbours and check their degree. If a neighbour lands in the same component, then add the edge in the cycle corresponding to that component. If a neighbour doesn't land in the same component but has degree 1 add it to the MST. Otherwise mark the edge as being of border type (outside of the current level scope).

Lemma 8. *At level i, edges of border type will represent active edges at level $i+1$.*

Between level iteration we create parallel edges and to deal with them just the edge of minimum weight counts as long as they point towards the same component (from all the edges that connect components (c_1, c_2) we consider just the minimum), thus treat any number of parallel edges as one. For example if between component u and v there are n parallel edges, discard everyone except the one of minimum weight. We are allowed to do such, because of the cut property. Furthermore, if the number of incident edges of a vertex v is equal to the number of parallel edges, then we consider the degree of v as 1. We compute the degree of a vertex by the following equation:

$$degree(v_{c_j}) = ie - \sum_{i \neq j} (pe_{c_i} - 1), \tag{1}$$

where ie is the number of incident edges, and pe_{c_i} is the number of incident edges in the component c_i.

A small clarification, we are allowed to discard the parallel edges because of how we build the components; if inside component c_1 we have two vertices u, v which get extended (while computing the MST of the component) inside the same component then we discard the parallel edges, but instead if one vertex u gets extended with an edge of border type, and no other vertex in the component gets extended with the minimum towards u, then it would be faulty to disregard the parallel edges as it would violate the cut property. Instead we allow parallel edges and remove the highest weight edge from the cycles they form. We can view this process as a 2-colouring of the edges; in the sense that edges (including the ones of border type) in a component c_i that are for sure part of the MST are colored in blue, and edges that might be in the MST are colored in red. The rest of the edges are greyed out, i.e. parallel edges, or the one that were discarded in the process of building the MST of a component.

Assume that at level i there are n_i vertices, if the selected vertices that lie in the center of our components have an average degree of k, then we have n_i/k components and at the level $i+1$ the edges considered will be at most k^2 after eliminating the parallel ones. Now the question is: do we need to build the whole tree, or we can stop at a level where the components become dense enough? If the disjoint-set operations are correct and produce no error, then we do not need the whole tree. Otherwise we need the whole tree in order to account for the missing possible MST edges.

For level i the computation cost is for each vertex of degree k, and we need $O(k^2 + k)$ comparisons in total to create the new components for the next level.

4. One Algorithm to Bind Them All

In this section we state the algorithm that makes use of the density partition, the disjoint-sets with errors, and the hierarchy of cycles and fill in the missing details.

4.1 Dealing with Disconnected Components

Because of the faulty disjoint-set data structure, there might be trees that are seen as containing cycles or pseudoforests. After all of the processing is done, those trees will have an unknown degree of segmentation, purely dependent on how the structure of cycles was predicted. Because the algorithm is deterministic, we find it hard to bound such an error degree. However, to account for this error it is quite easy, as the components must be expanded with the minimum incident edge that connects them; if it was removed in the process of eliminating cycles, we just add it back, if there was another candidate edge that was missed, we just need compare them and chose the minimum.

4.2 RAM Model Algorithm

We construct the algorithm in Figure 10 around the density partition. We need access to table look-up for cycle detection when calling the Trim procedure.

The DensityPartition presented in Subsection 1.6 returns the graphs containing the low degree components, the mixed degree components and the high degree components. We recurse on the sparse low degree components, and apply Prim-MST on the dense ones. The Trim procedure presented in Subsection 1.7, removes the peripheral components of degree 1, and adds the edges connecting them to the MST. The BorůvkaPhase should melt the the remaining components and border edges. The running time is given by the recurrence:

$$T(m,n) \leq T(E_l,V_l)+T(E_m(low),V_m(low))+T(m-m_0,\frac{n}{2}-n_t)+O(m+n), \tag{2}$$

where E_l, E_m and V_l, V_m represent the edges and vertices of the sparse components, m_0 represents the edges discarded while building the components and after trimming them, which might be anywhere from $n \leq m_0 \leq m$, and n_t represents the vertices trimmed. Because each grouping of vertices loses enough edges in order to become a tree, and the border edges that connect such components lose all of the parallel edges, m_0 should be a fraction of m, as if $m = n\log n$ or above, then there are enough dense components that are reduced to trees, and if $m = c\dot{n}$, then it loses at least n edges. The running time of the trim varies, as in the last recursive step, it will be $O(n_l^2)$, where n_l is the number of vertices that remain. The running time of the algorithm is between $O(m)$ for graphs that are not too sparse, and $O(n^2)$ when we

```
procedure RAM-MST
input $G = (V, E, C)$: undirected weighted graph
output minimum spanning tree
if $E = \emptyset$
  | return $\emptyset$
if $E = O(1)$
  | return Borůvka-MST($G$)
$G_l, G_m, G_h$=$DensityPartition(G)$
$F', G_1$=RAM-MST($G_l$)
$F$”$, G_2$=RAM-MST($G_m(low)$)
$G_0 = G_1 \cup G_2$
$F = F' \cup F$”
$F', G_1$=Prim-MST($G_h$)
$F$”$, G_2$=Prim-MST($G_m(high)$)
$G_0 = G_0 \cup G_1 \cup G_2$
$F = F \cup F' \cup F$”
$F' = Trim(G_0)$
$F' = F' \cup$BorůvkaPhase($G_0$)
$F' = F' \cup Trim(G_0)$
return $F \cup F' \cup$ RAM-MST($G_0$)
```

Figure 10. RAM density partition MST

are dealing with graphs that have very few cycles (because of the many cascades in the trim procedure that works poorly on sparse graphs). In order to improve on such a bound, we must find the cycle structure beforehand, and given a vertex of degree 1, find the tree of which it is part of, such that the root of the tree represents a cycle. This way we can skip the cascades and get a bound of $O(n)$.

Theorem 14. *Given a connected undirected weighted graph of arbitrary density and the cycle structure of the graph, its MST can be computed in linear time on the RAM model making use of table look-up.*

4.3 Pointer Machine MST

We must account for the disjoint set-operations, which we have not covered yet. In the RAM model, we can use the table look-up to make the disjoint-set operations run in linear time. But on a pointer machine the disjoint-set operations run in $O(m\alpha(m, n))$ which is asymptotically optimal.

Theorem 15. *Given a connected undirected weighted graph of arbitrary density and the cycle structure of the graph, its MST can be computed in $O(m\alpha(m, n))$ on a pointer-machine, where the running time is bounded by the disjoint-set operations.*

Proof. We need to account for the the disjoint-set operations needed to identify the cycles and answer queries on vertices of degree 1: what cycle lies at the root of the tree that it is part of? □

Chapter V. Conclusions

We did not manage to solve the problem, but valuable insight was gained. Not surprisingly there are a multitude of techniques with optimal results on special graphs, and different computational models. We conclude this thesis by mentioning the road we have followed and where we have tried to sketch improvements.
From a pointer machine perspective, it all started with a bound of $O(n^2)$, but improvements in data structures such as Fibonacci heaps managed to lower it to $O(m\beta(m,n))$, where $\beta(m,n) = \{\min i | \log^{(i)} n \leq m/n\}$. This means that for dense enough graphs the running time is linear. A lot of effort has been invested in improving the bound for sparse graphs, and with the break-through of soft heaps, Chazelle manged to bring it to $O(m\alpha(m,n))$.

Given a verification algorithm for a problem, an algorithm can be build using the verification procedure and have the same running time. We could not find any proof. If we consider King's verification procedure, it runs in linear-time but on a different computational model. Pettie and Ramachandran [16] seem to try to exploit such folklore using decision trees instead of table look-up for small subproblems, but details seem to be missing, and there are still open questions about how to build them efficiently.

From a random-access-machine perspective, things have gained traction fast; for a while $O(m \log n)$ was the bottleneck because of the sorting used in Kruskal's algorithm. People started making assumption about the edge weights, and sorting was reduced to $O(m + n)$ using bucket-sort. However, the edge weight assumptions are not satisfying as they restrain the problem. The focus on verification algorithms and the popularization of randomization managed to yet again bring the bound to expected $O(m)$ (Karger, Klein and Tarjan [10]). Such a method works poorly on sparse graphs, as it is based on breaking a dense graph into random sparse subgraphs.

We have noticed that in Kruskal's algorithm even if sorting is done in linear time, the bottleneck is represented by the disjoint-set operations $O(m\alpha(m,n))$. Thus we have explored on how can the complexity be reduced from $O(m\alpha(m,n))$ to $O(m + n)$. Gabow and Tarjan [7] show that if the structure of unions are known before hand the operations can be made to run in linear time using table look-up.

We have spent the whole Chapter 4 in trying to sketch a solution around this result. We acknowledge that our discussion is lacking some details, and the strength of our proofs is questionable, we show how to build a deterministic algorithm that runs in $O(m+n)$ for sparse graphs, and a reduction of dense enough graphs to sparse graphs in $O(m + n)$ time without the use of randomization. We are not satisfied with the discussion that revolves around the classification of cycles as there is this persistent feeling that we have omitted some trivial fact that plays a crucial role. Our contribution must be read with a high grain of skepticism.

References

[1] B. Chazelle, A minimum spanning tree algorithm with inverse-Ackermann type complexity, *J. ACM* **47**, 6 (2000), 1028–1047.

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

[3] B. Chazelle, *The Discrepancy Method Randomness and Complexity*, Princeton University, New Jersey (2002).

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd edition, The MIT Press (2009).

[5] P. Erdős and A. Rnyi, On the evolution of random graphs, *Publictation of the Mathematical Institute of the Hungarian Academy of Sciences* (1960), 17–61.

[6] M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* **34**, 3 (1987), 596–615.

[7] H. N. Gabow and R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, STOC '83*, ACM, New York, NY, USA (1983), 246–251.

[8] R. Graham and P. Hell, On the history of the minimum spanning tree problem, *Annals of the History of Computing* **7**, 1 (1985), 43–57.

[9] H. Kaplan and U. Zwick, A simpler implementation and analysis of chazelle's soft heaps, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '09*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2009), 477–485.

[10] D. R. Karger, P. N. Klein, and R. E. Tarjan, A randomized linear-time algorithm to find minimum spanning trees, *J. ACM* **42**, 2 (1995), 321–328.

[11] V. King, A simpler minimum spanning tree verification algorithm, *Algorithmica* **18**, 2 (1997), 263–270.

[12] J. Komlós, Linear verification for spanning trees, *Combinatorica* **5**, 1 (1985), 57–65.

[13] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press (1995).

[14] V. Osipov, P. Sanders, and J. Singler, The filter-Kruskal minimum spanning tree algorithm, *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, Society for Industrial and Applied Mathematics (2009), 52–61.

[15] M. Penttonen and J. Katajainen, Notes on the complexity of sorting in abstract machines, *BIT* **25**, 4 (1985), 611–622.

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

[17] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1983).

[18] R. E. Tarjan and J. van Leeuwen, Worst-case analysis of set union algorithms, *J. ACM* **31**, 2 (1984), 245–281.