Jagiellonian University
Department of Theoretical Computer Science

Hubert Bernacki

Implementation and comparison of minimum spanning tree algorithms

Bachelor Thesis

Supervisor: dr hab. Krzysztof Turowski

August 2025

Contents

1 Introduction **2**

1.1 Problem history 2

1.2 Preliminaries 4

1.2.1 Graphs 4

1.2.2 Minimum spanning tree 6

2 Chazelle's $O(m\alpha(m,n))$ algorithm **8**

2.1 Idea and pseudocode 8

2.1.1 Contractible subgraphs 8

2.1.2 T-hierarchy 9

2.1.3 Reverse the construction schedule 13

2.1.4 Pseudocodes 15

2.2 Correctness 19

2.3 Running time complexity 20

3 Chazelle-Rubinfeld-Trevisan sublinear weight finding algorithm **23**

3.1 Definitions 23

3.2 Idea and pseudocode 24

3.3 Correctness and running time analysis 26

4 Experiments **29**

4.1 Introduction 29

4.2 Results 29

4.2.1 Dataset 29

4.2.2 Chazelle's algorithm 29

4.2.3 Chazelle-Rubinfeld-Trevisan algorithm 30

4.3 Finishing thoughts 33

Chapter 1

Introduction

The minimum spanning problem is one of the simplest and most researched problems in graph algorithms. The task is to for a given connected and weighted graph is to find a spanning tree with a minimum sum of edge weights.

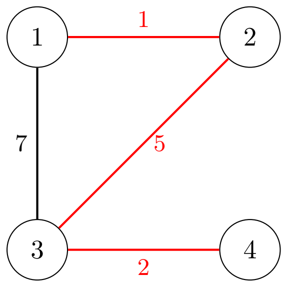

Figure 1.1: Example graph. Red edges form the minimum spanning tree

It arises naturally when dealing with any kinds of networks like constructing efficient electricity grid [2] or water supply [7]. It is also used in other algorithms – image registration [19], handwriting recognition [22] or in the approximate solution for the traveling salesman problem [6].

Algorithms solving the problem and their analysis are presented in many popular resources such as Cormen et al. textbook “Introduction to Algorithms” [8], cp-algorithms website [1] and a comprehensive overview by Eisner [10].

1.1 Problem history

The minimum spanning tree problem has long and rich history going as far back as 1926 when Borůvka used his algorithm to compute the most efficient electricity network for Moravia [2].

The problem was studied for years since and the most widely known contributions were done by Kruskal in 1956 [18] and Prim in 1957 [21]. These algorithms solve the problem in a greedy manner. The basic argument behind them is that for each vertex its smallest incident edge has to be the part of the minimum spanning tree. Otherwise, such edge would create a cycle with the minimum spanning tree and on this cycle there would be a larger edge also incident to the same vertex, switching these edges in the spanning tree would make its total weight smaller.

That said each algorithm proceeds differently:

- **Borůvka's algorithm**: This algorithm works in phases, in each phase the smallest incident edge to each vertex in the graph is found and these edges are contracted all at once at the end of the phase. Consecutive phases are run until the graph contracts to one vertex. It is proven to work in $O(m \log n)$ time.

- **Kruskal's algorithm**: All of the edges are sorted. A disjoint set union is created with each vertex being a different set. Sorted edges are checked in an ascending order when an edge connects two different sets it get contracted and added to the spanning tree under construction. It is proven to work in $O(m \log n)$ time if the sorting algorithm is comparison based.

- **Prim's algorithm**: One vertex is picked. Until all vertices are visited the smallest edge joining visited and unvisited vertices is found. It's unvisited end is visited and it's neighboring edges are added to the priority queue on which the unprocessed edges lie. With a standard heap this algorithm runs in $O(m \log n)$ time and using more complex structures like Fibonacci heap [11] its complexity decreases to $O(m + n \log n)$.

Further contributions include Yao [23] and Cheriton with Tarjan [5] who independently found different algorithms with $O(m \log \log n)$ complexity. Later Fredman and Tarjan [11] lowered the complexity to $O(m\beta(m, n))$ where $\beta(m, n)$ is defined as the number of log iterations necessary to map n to number smaller then $\frac{m}{n}$ and in the worst case for $m = O(n)$ it is $O(m \log^* n)$. It was shortly after improved to $O(m \log \beta(m, n))$ by Gabow et al. [13]. In 2000 Chazelle [3] found an algorithm with $O(m\ \alpha(m, n))$ complexity, where the α is the inverse Ackermann function.

The optimal algorithm has been discovered by Pettie and Ramachandrdan in 2002 [20]. However the running time complexity of their algorithm is unknown. That said for the practical purposes the problem of finding the mst has been solved for many years as the textbook algorithms have really good complexities and are easy to implement.

The algorithms presented above all work in a comparison-based model in which the edges are comparable in constant time and no bit manipulation is allowed. If the edge costs are integers and the model allows for bucketing and bit manipulation then the problem is possible to solve in $O(m + n)$ time deterministically, as presented by Fredman and Willard [12].

It is also possible to verify in linear time that given tree is the minimum spanning tree as shown by Komlós [17], King [16] and Hagerup [14].

All of the above algorithms were deterministic, however there also exist randomized algorithms for this problem. Karger, Klein and Tarjan showed in 1995 [15] one that runs in $O(m + n)$ expected time. Furthermore in 2005 Chazelle, Rubinfeld and Trevisan [4] devised a sublinear randomized algorithm which does not find the minimum spanning tree, but rather approximates its total weight.

Table 1.1: Minimum spanning tree algorithms. Implemented algorithms highlighted in **bold**.

Name	**Year**	**Complexity**	**Note**
Borůvka	1926	$O(m \log n)$	
Kruskal	1956	$O(m \log n)$	
Prim	1957	$O(m \log n)$	
Yao	1975	$O(m \log \log(n))$	
Fredman, Tarjan	1984	$O(m\beta(m, n))$	
Komlós	1984	$O(m + n)$	verification
Galil et al.	1986	$O(m \log \beta(m, n))$	
Fredman, Willard	1993	$O(m + n)$	bit manipulation
Karger, Klein, Tarjan	1995	$O(m + n)$	randomized
King	1997	$O(m + n)$	verification
Chazelle	**2000**	$O(m\alpha(m, n))$	
Pettie, Ramachandran	2002	optimal	unknown complexity
Chazelle et al.	**2005**	sublinear	approximate weight
Hagerup	2009	$O(m + n)$	verification

1.2 Preliminaries

1.2.1 Graphs

Definition 1 (Graph [9])**.** A graph is a pair $G = (V, E)$ of sets such that $E \subseteq \binom{V}{2}$ and $V \cap E = \emptyset$.

Throughout the paper for $G = (V, E)$ we write $V(G)$ to denote V – the set of vertices (nodes or points) and $E(G)$ to denote E – the set of edges.

For any graphs G and H we define the following operations:

- $G \cup H = (V', E')$, with $V' = V(G) \cup V(H)$ and $E' = E(G) \cup E(H)$
- $G \cap H = (V', E')$, with $V' = V(G) \cap V(H)$ and $E' = E(G) \cap E(H)$
- $G \setminus H = (V', E')$, with $V' = V(G) \setminus V(H)$ and $E' = E(G) \setminus E(H)$

Definition 2 (Graph with loops)**.** A graph with loops is a pair $G = (V, E)$ of sets such that $E \subseteq V^2$.

Definition 3 (Weighted graph)**.** Graph $G = (V, E)$ is weighted if there is a function $w : E \to \mathbb{R}_+$ defined on the edges of G.

For a given edge $e = \{u, v\}$ we denote its weight (also called cost) $w(e) = w_e = w_{uv}$.

Definition 4 (Incident [9])**.** A vertex v is **incident** with an edge e if $v \in e$.

We denote by $E(v)$ set of all edges in E incident with a vertex v.

Definition 5 (Degree [9])**.** The degree $d_G(v) = |E(v)|$ of a vertex v is the number of edges incident with v.

Since G is often obvious from the context, we will use the notation $d(v)$ or even d_v.

For graphs with loops we define the vertex degree as $d(v) = |E(v)| + |\{v\colon \{v\} \in E\}|$. Loops are counted twice.

Definition 6 (Average degree [9])**.** The number

$$d(G) = \frac{1}{|V|} \sum_{v \in V} d(v)$$

is the **average degree** of G.

If it is clear from the context we shorten $d(G)$ to d.

Now let us proceed with a few auxiliary lemmas and definitions

Definition 7 (Subgraph [9])**.** For graphs $G = (V, E), G' = (V', E')$ If $V' \subseteq V$ and $E' \subseteq E$, then G' is a subgraph of G.

We use the notation $G' \subseteq G$ to denote that G' is a subgraph of G.

Definition 8 (Induced subgraph [9])**.** If $G' \subseteq G$ and G' contains all the edges $\{x, y\} \in E$ with $x, y \in V$, then G' is an **induced subgraph** of G.

We also say that V' induces G' in G.

Definition 9 (Spanning [9])**.** $G' \subseteq G$ is a **spanning** subgraph of G if V' spans all of G, i.e. if $V' = V$.

Definition 10 (Cut of a graph)**.** For a given graph $G = (V, E)$ if we have nonempty sets $V_1 \cup V_2 = V$ such that $V_1 \cap V_2 = \emptyset$. We call it a **cut** and the cut set is the set of edges with one endpoint in V_1 and one in V_2.

Definition 11 (Edge contraction)**.** Given a graph $G = (V, E)$ and an edge uv we say that a graph $G' = (V', E')$, where $V' = V(G) \setminus \{u, v\} \cup \{uv\}$ and $E' = \{e \in E : e \in V' \vee (\{w, uv\} \text{ if } wu \in E \vee wv \in E)\}$ has been created through a contraction of edge uv.

Definition 12 (Graph minor)**.** We say that graph H is a minor of graph G if it can be obtained through a sequence of edge contractions beginning with graph G.

Definition 13 (Path [9])**.** A path is a non-empty graph $P = (V, E)$ of the form $V = \{x_0, x_1, ..., x_k\}, E = \{x_0x_1, x_1x_2, ..., x_{k-1}x_k\}$, where x_i are all distinct.

The vertices x_0 and x_k are **connected** by P and are called **ends**. Throughout the paper we use the notation $P = x_0...x_{k-1}$.

Definition 14 (Cycle [9])**.** If $P = x_0...x_{k-1}$ is a path and $k \geq 3$, then the graph $C := P + x_{k-1}x_0$ is called a **cycle**.

Throughout the paper we use the notation $C = x_0...x_{k-1}x_0$.

Definition 15 (Connected graph [9])**.** A graph is called **connected** if it is non-empty and any two of its vertices are linked by a path in G.

Definition 16 (Connected component [9])**.** Let $G = (V, E)$ be a graph. A maximal connected subgraph of G is a **connected component** of G.

The components are induced subgraphs and their vertex sets partition V.

Definition 17 (Forest, tree [9])**.** An acyclic graph, one not containing any cycles, is called a **forest**. A connected forest is called a **tree**.

Definition 18 (Leaves, inner vertices [9])**.** The vertices of degree 1 in a tree are its **leaves**, the others are its **inner vertices**.

Definition 19 (Root, rooted tree [9])**.** A tree T with one vertex marked as **root** is called a **rooted tree**.

Definition 20 (Tree cost)**.** Cost (also called weight) $w(T)$ of a weighted tree T is the sum of costs of its edges. That is $w(T) = \sum\limits_{e \in E(T)} w_e$.

1.2.2 Minimum spanning tree

Definition 21 (Minimum spanning tree)**.** Minimum spanning tree of a graph G, denoted as $MST(G)$, is a spanning tree with the minimum cost of all of the spanning trees of G.

We will also write MST when the G is known from the context.

Definition 22 (Minimum spanning tree weight)**.** For a given graph G let $M(G) = w(MST(G))$ be the weight of its minimum spanning tree.

Problem 1 (Minimum spanning tree problem – optimization)**.** For a given graph G find and output the $MST(G)$.

Problem 2 (Minimum spanning tree problem – decision)**.** For a given graph G and a number k find a spanning tree T of G such that $w(T) \leq k$.

Following properties assume that the graph G has distinct edge weights:

Theorem 1 (Strong cut property [10])**.** *For every graph G it holds that $e \in MST(G) \iff e$ is the lightest edge across some cut of G.*

Theorem 2 (Strong cycle property [10])**.** *For every graph G it holds that $e \notin MST(G) \iff e$ is the heaviest edge on some cycle of G.*

Here is the proof of both properties at once due to Eisner [10].

Proof. ($\Rightarrow$) Every e when removed from $MST(G)$ determines a cut across which it is lightest as if there was a lighter edge e' in the cut-set then $MST(G)-e+e'$ would have a smaller weight. Every $e \notin MST(G)$, when added to the MST(G) creates a cycle on which it is heaviest as if there was a heavier edge e' on the cycle, then once again we would have a contradiction with $MST(G) - e + e'$.

($\Leftarrow$) We derive contrapositives from the cases just proved. If $e \notin MST(G)$ then it is a heaviest edge on some cycle and it cannot be the lightest in any cut-set as the cut set would have to intersect the cycle. If $e \in MST(G)$ then it is the lightest in some cut set, so it cannot be the heaviest on any cycle as it would have to have a non empty intersection with the cut-set on which e is the lightest. $\square$

Theorem 3 (Minimum spanning tree uniqueness)**.** *If a connected graph G has distinct edge costs then it has exactly one minimum spanning tree.*

Proof. Both strong cut property and strong cycle property classify edges into the ones in MST and outside the MST. $\square$

Chapter 2

Chazelle's $O(m\alpha(m,n))$ algorithm

Chazelle [3] proposed in 2000 a new algorithm which solves the minimum spanning tree problem. He has proven that the algorithm runs in $O(m\alpha(m,n))$ time. It has not been improved on since by any other algorithm with known complexity.

2.1 Idea and pseudocode

The main idea of the algorithm is to partition a graph into smaller subgraphs for which we compute the minimum spanning trees recursively. In order to make it fast the smaller minimum spanning trees together form a spanning tree that is not necessarily minimal. However in the process some edges are proven not to be in the MST and other's have to be reprocessed in a final recursive call.

There are several ideas that work together in this algorithm and this section is devoted to explaining them.

2.1.1 Contractible subgraphs

It is important to have a certain mindset when considering the MST problem – that is, one has to think about it in terms of edge contractions. In particular, let us reacall an importanat duality between to properties:

- cut property (Theorem 1) – allows us to prove that edge is in the MST and can be contracted.
- cycle property (Theorem 2) – allows us to prove that edge is outside of the MST and can be discarded.

These properties are crucial both for the construction of algorithms and for the proofs of certain properties like strong contractibility – which we will consider now.

Definition 23 (Contractible subgraph [3])**.** A connected induced subgraph $H \subseteq G$ is contractible if and only if $MST(H) \subseteq MST(G)$.

The notion of contractible subgraphs is helpful when creating divide and conquer algorithms for the MST problem as we shall see below.

Let's say we have a partition of the vertices of a graph G into sets $V_1, V_2, ..., V_k$ and corresponding induced subgraphs $G_1, G_2, ..., G_k$. Denote G' as a graph G after contracting each of the V_i sets of vertices. Now we can see that thanks to the definition of contractible subgraphs we have that

$$MST(G) = \left(\bigcup_{i=1}^{k} MST(V_i) \right) \cup MST(G'). \tag{2.1}$$

Having that we could recursively apply our MST algorithm and just sum the results.

While contractibility is extremely useful, it is a global property. That is, it depends on the $MST(G)$. Thus proving it from the definition requires computing the $MST(G)$ first.

Definition 24 (Strongly contractible subgraph)**.** A connected induced subgraph $H \subseteq G$ is strongly contractible if and only if for all $e, f \in E(G)$ with exactly one endpoint in H, there exists path in H which connects e and f and no edge along this path exceeds the cost of both e and f.

Lemma 1. *If a subgraph H of G is strongly contractible then it is contractible.*

Proof. Let's assume H is not contractible, thus $MST(G) \cap H$ has to have multiple connected components. Let's take a shortest path in $MST(G)$ that connects these components. It is the shortest so it cannot have edges in H, let e, f be the first and last edge i.e. the edges which connect H to the G/H.

But the subgraph H is connected and strongly contractible, so there is a path that completes the cycle and lies fully inside H and has all of the edge costs smaller than $\max\{c(e), c(f)\}$. So the largest edge on this cycle lies outside the path in H, and by the cycle property it cannot be in the $MST(G)$ – a contradiction. □

The notion of strongly contractible subgraphs is local. It allows us to certify contractibility of a given subgraph by looking only at it and its neighborhood.

2.1.2 T-hierarchy

Borůvka

To introduce the concept of T-hierarchy let's first consider Borůvka's algorithm:

Algorithm 1 Borůvka's Algorithm

```
function BORUVKA($G$: Graph)
    $F \leftarrow \emptyset$
    while $|E| > 0$ do
        $F \leftarrow F \cup$ BORUVKA-STEP($G$)
    end while
    return $F$
end function
```

Algorithm 2 BORUVKA-STEP

```
function BORUVKA-STEP($G$: Graph)
    $F \leftarrow \emptyset$
    $marked \leftarrow \emptyset$
    for $v \in V(G)$ do
        $e \leftarrow \arg\min \{c(e) : e \in E(v)\}$
        $marked \leftarrow marked \cup \{e\}$
    end for
    for $e \in marked$ do
        CONTRACT($G, e$)
        $F$.INSERT($e$)
    end for
    return $F$
end function
```

Let's consider what happens to a connected graph G during the Algorithm 2. Let n, m be number of vertices and edges in G. We can see that the number of vertices in the resultant graph will be at most $\frac{n}{2}$, i.e. it drops by a factor of 2. So the Algorithm 1 calls Algorithm 2 only $O(\log n)$ times.

Unfortunately the number of edges m doesn't always drop by a multiplicative factor. In the worst case only $\frac{n}{2}$ edges will be contracted. Given the fact that BORUVKA-STEP can be performed in $O(n + m)$ time gives the total running time $O(m \log n)$.

Consider what would happen if the number of edges decreased in every iteration by a multiplicative factor q, then the whole algorithm would run in time

$$(n + m) + \left(\frac{n}{2} + qm\right) + \left(\frac{n}{4} + q^2m\right) ... \leq 2n + \frac{1}{1-q}m = O(n + m).$$

So if we found a way to remove a factor of edges in linear or almost linear time, then we could also perform a single Algorithm 2 in linear time to also decrease the number of vertices by some factor and have a routine that works in linear or slightly larger time.

Borůvka hierarchy

Consider the sequence of graphs produced during consecutive Boruvka-step invocations. Each consecutive graph is a minor of the previous one, as Boruvka-step only performs edge contractions. Being a minor is a transitive property so a graph obtained after any number of Algorithm 2 will be a minor of every previous one.

Now looking at just one Algorithm 2 we can also think about it as of identifying multiple vertices as one, in this spirit we could create a forest which has two sets of vertices – first set corresponds to vertices in original graph and the second one has vertices from the resultant minor. We can connect vertices v from the first set and w from the second set if w was created by contracting v and some other vertices. See Figure 2.1 and Figure 2.3.

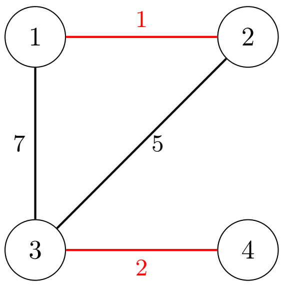

Figure 2.1: Example graph. Red edges are contracted in the first run of Algorithm 2.

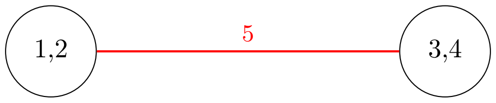

Figure 2.2: The example graph minor after the Algorithm 2 – also the $C_{1,2,3,4}$ graph.

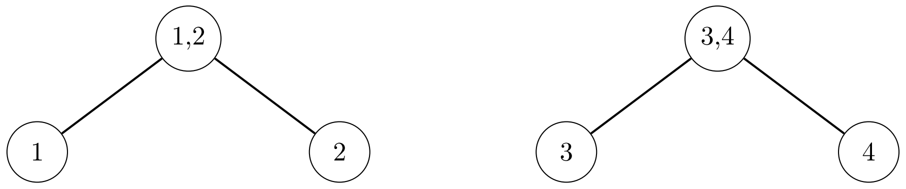

Figure 2.3: Forest corresponding to a single iteration of Algorithm 2.

If we extend that forest by adding sets of vertices corresponding to minor's in each consecutive Algorithm 2 iteration we will get a tree T, which root represents a contraction of all vertices into a single node. Each layer below it corresponds to the minor from the previous iteration, and leafs of this tree T are just the vertices of the original graph G for which we want to find the minimum spanning tree. See Figure 2.4 below.

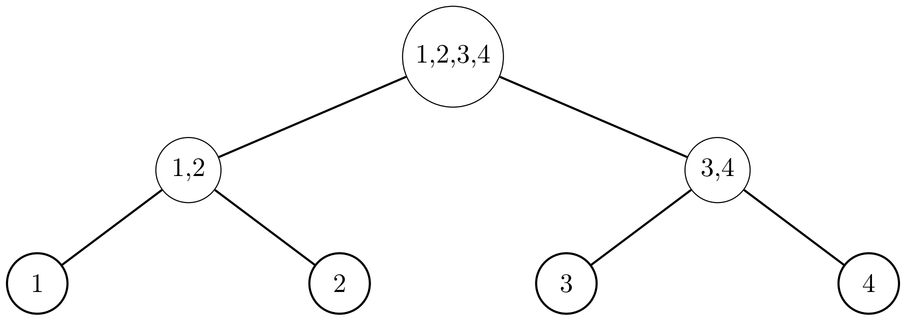

Figure 2.4: T-hierarchy constructed by the Algorithm 1.

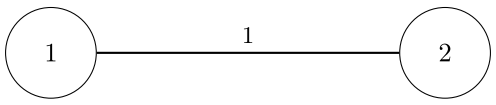

Figure 2.5: The $C_{1,2}$ graph

Definition 25 (T-hierarchy)**.** T-hierarchy is a rooted tree T, whose leaves correspond to the vertices of a given graph G and the internal nodes represent contractions of all of the leaves below them.

In particular root corresponds to the contraction of the whole graph G and a subtree of T corresponds to a minor of the graph G.

Definition 26 (z_i `and` C_{z_i})**.** We introduce the notation that z_i is an internal node of the tree T and C_{z_i} is a graph whose vertices are the children of z_i. The edge between vertices u and v in this graph corresponds to the smallest edge (if it exists) joining sets of vertices of G – the leaves below u and v in the T-hierarchy respectively.

Now the T-hierarchy constructed by the Borůvka's algorithm has an important property. If you look at any node in the hierarchy its correspondent set of vertices of G induces a contractible subgraph in G. This property follows directly from the fact that the hierarchy was created by contracting the edges which belong to the $MST(G)$.

If we were given a T-hierarchy with this property, we could compute the minimum spanning tree using the divide and conquer method with the following lemma:

Lemma 2 (MST construction using T-hierarchy)**.** *Given a graph G and corresponding T-hierarchy such that for every $z \in V(T)$ its C_z is contractible we* *can find the $MST(G)$ using the following identity:*

$$MST(G) = \bigcup_{z \in T} MST(C_z).$$

Proof. We repeatedly apply (2.1) until the G' is fully contracted. Consider what happens after one application of (2.1) to the original graph G and a partition into the contractible subgraphs defined by the bottom layer of T (z such that all of z's children are leaves).

For example for Figure 2.4, after the first application of (2.1) we are left with a G' and its T-hierarchy T' which was obtained from T by pruning all of the leaves $1, 2, 3$ and 4. □

2.1.3 Reverse the construction schedule

One of the main ideas behind the algorithm is to reverse the MST construction schedule. First we could create a hierarchy T such that each layer represents a partition of the graph G into contractible subgraphs. Then we would calculate the $MST(G)$ recursively using the Lemma 2.

Corruption

Nevertheless, calculating the T-hierarchy proves to be a difficult task. To make this step fast Chazelle decided to loosen the requirements put on the T-hierarchy. During its creation some edges will become corrupted, that is their weight will be purposefully made higher then original. The resultant T-hierarchy will have its internal nodes contractible, but in terms of the **working cost** of the edges and not counting the **discarded** edges (both notions will be defined later).

It is important to mention that all of the **discarded** edges and also the edges for which the **working cost** is different than their original cost will be reprocessed later with their costs reverted back to the original.

Remark 1 (Edge corruption)**.** Each edge of the graph G has its original cost and current cost associated with it. The current cost can change during a construction of T, but will never be lower than the original cost. We call edges **corrupted** when their current and original costs differ.

Prim

In order to understand why and how the corruption happens and what are the working costs of the edges referred to in the previous subsection we need to first understand how the T-hierarchy is built.

Let's first consider Prim's algorithm as it will be the basis for the T-hierarchy construction. We can think about it as picking a vertex v and sequentially contracting entire graph to it edge by edge. It also can be thought of as a Dijkstra analogue: there are visited and unvisited nodes and we always visit the closest one, but not the closest one to the origin v like in Dijkstra.

Algorithm 3 Prim's Algorithm

```
1:  function PRIM($G$: Graph)
2:      $v \leftarrow$ PICK-ANY-VERTEX($G$)
3:      $F \leftarrow \emptyset$
4:      while $E(G) \neq \emptyset$ do
5:          $e \leftarrow$ SMALLEST-EDGE-ADJACENT-TO($v$)
6:          CONTRACT-EDGE($e, G$)            ▷ Contracts other vertex $u$ into $v$
7:          $F \leftarrow F \cup \{e\}$
8:      end while
9:      return $F$
10: end function
```

Building the T-hierarchy

The Prim's algorithm does not care about the already visited vertices: when a vertex v is processed its edges are pushed on a priority queue and the vertex can be forgotten of.

To build the T-hierarchy we will perform a Prim's algorithm but with two important modifications:

- we maintain a structure on the visited vertices – this structure is the T-hierarchy being built.
- as a priority queue we will use soft heaps, a heap-like data structure devised by Chazelle.

The exact construction schedule and soft heaps will be discussed thoroughly later. For the purpose of this introduction let us mention their most important properties:

- Soft heaps are responsible for the edge corruption.
- T-hierarchy is created **bottom-up** – when a C_z is assembled all of its children are constructed too and the current costs of edges inside them are frozen. When the C_z has been constructed it gets contracted to a vertex and put inside its parent.
- In the process of construction some edges are **discarded** – these edges were necessarily bad.

Let us finally introduce the crucial auxiliary notions for this algorithm:

Definition 27 (Border edge)**.** During the Prim's algorithm if the edge joins a visited and an unvisited node, then its a **border edge**.

In the Chazelle's algorithm context during the construction of T-hierarchy we maintain a structure on visited nodes, that is, the C_z graphs. We use the same name border edge for the edges joining visited and unvisited vertices. We also say that a border edge $\{u, v\}$ joins a C_z to an unvisited vertex v when the visited vertex u is under the z node in the T-hierarchy.

Definition 28 (Bad edge)**.** A border edge e becomes bad if it is corrupted and has one endpoint in a current C_z in the moment of its contraction.

We will sometimes reset the current costs of the edges, but once the edge becomes bad it always remains bad, even after the reset.

Definition 29 (Working cost)**.** Working cost of an edge e is its current cost if it is bad, otherwise it is its original cost.

Soft heap

Soft heap is a minimum heap devised by Chazelle, which is a workhorse for the algorithm. It breaks the traditional $O(\log n)$ bound for heaps for all of its operations, but at the cost of corruption: some elements may have their keys increased.

Each element inserted into the soft heaps has its original key, and also a ckey - they are equal in the beginning. The soft heap maintains an ordering using the ckeys and ckeys cannot be lower then keys.

Soft heap allows for `make-heap`, `insert`, `meld`, `pop-min` and `delete` operations. Each of them runs in $O(1)$ amortized time, but for the `insert` operation which runs in $O(\log \frac{1}{\epsilon})$ amortized time. The ϵ parameter is set in the `make-heap`.

Soft heap maintains an invariant that there are at most ϵn corrupted elements on the heap, where n is the number of inserted elements – not to confuse with the number of elements currently stored in the heap.

2.1.4 Pseudocodes

Algorithm 4 Recursive MST Algorithm

```
1: function MST($G$: Graph, $t$: int)
2:     if $t = 1 \lor n = O(1)$ then
3:         return BORUVKA($G$)                              ▷ STEP 1
4:     end if
5:     $G_0, F_0 \leftarrow$ BORUVKA-STEPS($G, c$)          ▷ STEP 2
6:     $T, B \leftarrow$ CONSTRUCT-$T$-HIERARCHY($G_0$)     ▷ STEP 3
7:     $F \leftarrow \bigcup_{C_z \in T}$ MST($C_z/B$, $t-1$)   ▷ STEP 4
8:     return $F_0 +$ MST($F + B$, $t$)                     ▷ STEP 5
9: end function
```

Algorithm 4 presents a basic pseudocode for the whole algorithm. It takes a graph G and a number t as inputs and returns $MST(G)$. The parameter t is used in creation of T-hierarchy for tuning the recursion so that we have the double induction necessary for the complexity bound involving the inverse Ackermann function.

- **STEP 1** This is the base case: we just run the Algorithm 1 if the graph is small enough or $t = 1$. As the algorithm is recursive we need a base case.

- **STEP 2** We apply c consecutive BORUVKA-STEPs to the graph G and return the resultant minor G_0 and the forest of contracted edges F_0.
- **STEP 3** This step is the hardest and contains most of the actual algorithm. Pseudocode and its analysis is provided below.
- **STEP 4** We recursively call Algorithm 4 on C_z's of all the internal nodes of T-hierarchy with bad edges removed from them. We collect the returned trees into a forest F.
- **STEP 5** We recursively call Algorithm 4 on a graph created from bad edges B and forest F as we need to reprocess the bad edges.

Let's get back to the most important part: the construction of T-hierarchy. Here is the pseudocode:

Algorithm 5 T-hierarchy construction

```
function CONSTRUCT-$T$-HIERARCHY($G_0$: Graph)
    Cz ← ∅
    H ← ∅                                         ▷ set of soft heaps
    B ← ∅                                         ▷ set of bad edges
    min-links ← ∅
    PICK-STARTING-VERTEX()
    while EDGES-ON-HEAPS() do
        if MET-DESIRED-SIZE() then
            RETRACTION()
        else
            $e$ ← H.POP-MIN()
            if NEED-FUSION() then
                FUSION()
            end if
            EXTENSION()
        end if
    end while
    while Cz.LENGTH() $> 1$ do
        RETRACTION()
    end while
    return Cz[1], B
end function
```

Remark 2. All of the functions in the pseudocode above have the access to the state variables defined at the beginning. Thus, they were omitted from the function calls as they would clutter the pseudocode.

As stated previously, the T-hierarchy construction mimics the Prim's algorithm: in each iteration of the loop we pop the smallest edge from the heaps and process it.

Let's first consider the desired properties of T:

- each C_z has to be strongly contractible in terms of working costs and ignoring discarded edges,
- each C_z has a desired size which it should attain before being contracted – however, the exact sizing is only important for the runtime complexity analysis and will be discussed there.

Let's consider the variables used in Algorithm 5.

- `Cz` – Is a stack of graphs which represent C_z's **under construction** – the so called active path. At the bottom of the stack lies the $C_{z_{root}}$ which corresponds to the root of T. Each consecutive stack item corresponds to the chld in T of the previous one. We use the notation $C_{z_1}, C_{z_2} ... C_{z_k}$ to enumerate all of the C_z under construction.
- `H` is a minimum heap containing all of the border edges. For the complexity reasons (discussed later) it has to be a 2-dimensional array of soft heaps: if the active path is $C_{z_1}, C_{z_2} ... C_{z_k}$ then we have a `H[i][j]` for all $i, j \in \{0, 1, ..., k\}, i \leq j$.
- `B` – a graph of the bad edges.

In order not to produce to many bad edges and enforce contractibility of the C_z's following invariants will be maintained during the construction of the T-hierarchy.

Invariant 1 ([3]). *We keep an edge called a chain-link joining C_{z_i} and $C_{z_{i+1}}$ for all $i \in \{1, 2, ..., k-1\}$. Its current cost is:*

(i) at most that of any border edge incident to $C_{z_1} \cup \cdots \cup C_{z_i}$

(ii) less than the working cost of any edge joining 2 distinct C_{z_j} with $j \leq i$

To enforce the latter condition efficiently we maintain a min-link (if it exists) for each pair $i < j$ – an edge of minimum working cost joining C_{z_i} and C_{z_j}.

Invariant 2 ([3]). *For all j, the border edges $\{u, v\}$ with $u \in C_{z_j}$, the edges are stored in a soft heap – denoted either `H[j][j]` or in one `H[i][j]`, where $0 \leq i < j$. No edge appears in more than one soft heap.*

Membership in `H[j][j]` implies that v is incident to at least one edge stored in some `H[i][j]`.

Membership in `H[i][j]` implies that v is also adjacent to C_{z_i} but not to any C_{z_l} in between ($i < l < j$). It is extended to `H[0][j]` and it means that v is not incident to any C_{z_l} with ($l < j$).

As shown in the pseudocode the construction of T-hierarchy works in a loop with three main operations: RETRACTION, FUSION and EXTENSION.

RETRACTION

When the C_{z_k} has attained its desired size we contract it to 1 vertex and add this vertex to its parent – $C_{z_{k-1}}$. We need to maintain a valid state for:

- `H` – we meld `H[k][*]` to appropriate `H[k-1][*]` – implied by Invariant 2. `H[k-1][k]` and `H[k][k]` have to be destroyed and each edge has to be examined again – that's were the corrupted edges (bad now) are **discarded**. Some edges will now have the same origin – only the smallest one of them is retained in a heap implied by Invariant 2.
- `min-links` – we have to pop `min-links[k]` and update `min-links[k-1]` using it.
- `Cz[k-1]` edges – we need to update the edges to the newly added vertex.

FUSION

We have a border edge e (called **extension edge**) that we want to use for the extension but doing so would violate the Invariant 1. Out of all pairs i, j we need to find one with the lowest i that `min-links[i][j]` working cost is smaller or equal to the current cost of e. Let's say we found the edge $e = \{a, b\}$ for some $a \in C_{z_i}, b \in C_{z_j}$.

We contract all of the edges with both endpoints in $\{a\} \cup C_{z_{i+1}} \cup ... \cup C_{z_k}$ into a and now the edge e has one of its endpoints in a. Maintaining valid state for `min-links` and `H` is analogous to the one in retraction, but now we do multi-pops instead of normal pops. That said this operation is not a retraction – no new vertex is created and added to C_{z_i}.

Remark 3. The edges contracted during the fusion have to be somehow returned from the CONSTRUCT-T-HIERARCHY function and added to the F accumulator. It can be done explicitly or they can be stored inside a and added during the construction of $MST(a)$ in **STEP 4**.

EXTENSION

We are given an extension edge e which connects the end of the active path C_{z_k} (guaranteed by the earlier fusion) to an unvisited vertex v. We need to:

- process all of the edges of v – old border edges need to be removed from the heaps and used to update min-links. New border edges have to be pushed onto the heaps implied by Invariant 2.
- push v onto the `Cz`.

That's an in-depth overview of what happens during the construction of T-hierarchy. It is similar to the Prim's algorithm – we process edges until there are no more edges left (i.e. we have processed the whole graph). If the lowest node has met its desired size we perform retraction and otherwise we find the minimum border edge among all of the heaps and perform the extension (along with fusion if necessary).

2.2 Correctness

Let's first consider the invariants which are maintained in the **STEP 3**.

Lemma 3. *During the **STEP 3** Invariants 1 and 2 are maintained.*

Proof. Consider Invariant 1 – retraction obviously maintains it as it only removes the last chain-link and only changes the now last C_z, so it won't introduce problems neither with (i) nor with (ii). For the extension remember that we first call fusion if the (ii) cannot be enforced with a given border edge. The sole purpose of fusion is to maintain (ii) invariant. After the fusion we will have that the extension edge can become a new chain link as thanks to the fusion it satisfies (ii).

Invariant 2 is self explanatory – it gives us the recipe of how to store the border edges and we follow it. □

These invariant make sure that if we maintain them then we will produce contractible C_z's which is important as it allows us to use divide and conquer in **STEP 4**.

To prove this we will use the notion of strong contractibility, the fact that it can certify contractibility of a subgraph by only looking at it and its incident edges is paramount. We want to prove the following lemma:

Lemma 4 (Lemma 3.2 in [3])**.** *With respect to working costs and excluding discarded edges, C_z is strongly contractible at the time of its contraction (during retraction), and the same holds for every fusion edge (a, b).*

Proof. Consider the fact that C_z grows monotonically, since retractions add one vertex to it and fusions may add new neighboring edges to it. As only border edges can be discarded, therefore C_z will also not loose any edges inside it. So we only need to consider the moment right before the retraction.

Let's only consider the case when no fusion occurred, consider any two edges e, f which are incident to C_z right before the retraction. As no fusion ever happened the C_z was created solely by retracting chain links, thus it has a spanning tree made out of only chain-links. Now we can find the path π between e and f that is entirely inside this spanning tree. Consider the edge g with the largest current cost (in time of its selection as chain link) in case of ties pick one that was selected for extension last. Now let's say that $g = (u, v)$ where at time of the extension u was inside C_z and v was not. Without loss of the generality u is between e and v on the path π. Now if e is not a border edge than it has to link to a C_z earlier on the active path, so by the Invariant 1(ii) it has to have its working cost larger than g. In the case it is a border edge, if it had a smaller working cost, then we would have picked it before g for the extension edge. □

For the correctness proof we will need a following lemma:

Lemma 5 (Lemma 3.1 in [3])**.** *If the edge of G_0 is not bad and it lies outside of F, then it lies outside of $MST(G_0)$*

The proof of this lemma relies heavily on the Lemma 4.

Theorem 4 (Correctness proof)**.** *The Algorithm 4 when given a connected graph G as an input returns the $MST(G)$.*

Proof. The proof is an induction on t and n.

For the base case – small n or $t = 1$ – the algorithm runs just the **STEP 1** e.g. runs the BORUVKA algorithm on the given graph which is proven to work.

Now consider the inductive step with sufficiently large t and n (so that we do not fall into **STEP 1**). First let G_0 and F_0 denote respectively a minor of G which is the result of **STEP 2** and the forest of contracted edges during **STEP 2**. By the correctness of the BORUVKA and the BORUVKA-STEPS algorithms we know that $F_0 \subseteq MSF(G)$ and $MSF(G_0) \cup F_0 = MSF(G)$. As the whole function returns MST$(F \cup B, t)$ we need to prove that $MST(F \cup B) = MST(G_0)$ which is directly implied by Lemma 5. □

2.3 Running time complexity

Let us first define few functions for bounding the complexity.

Definition 30 (Ackermann function)**.** For all $i, j \in \mathbb{N}$ let Ackermann function A be defined as:

$$A(i,j) = \begin{cases} 2j & \text{if } i = 0, \\ 0 & \text{if } j = 0, \\ 2 & \text{if } j = 1, \\ A(i-1, A(i, j-1)) & \text{otherwise.} \end{cases}$$

Definition 31 (Inverse Ackermann function)**.** Define inverse Ackermann function as $\alpha(m,n) = \min\{i \geq 1 \colon A(i, 4\lceil \frac{m}{n} \rceil) > \log n\}$

Definition 32 (S function)**.** For all $i, j \in \mathbb{N}$ let function S be defined as:

$$S(i,j) = \begin{cases} 2j & \text{if } i = 1, \\ 2 & \text{if } j = 1, \\ S(i, j-1) S(i-1, S(i, j-1)) & \text{otherwise.} \end{cases}$$

As the algorithm is recursive the complexity proof will be inductive. To bound the running time of each step of the algorithm following lemmas will be used.

Lemma 6 (Decay lemma, Lemma 4.1. in [3])**.** *The total number of bad edges produced while building T is $|B| \leq \frac{m_0}{2} + d^3 n_0$.*

Lemma 7 (Lemma 4.2. in [3])**.** *Total number of inserts in all the heaps is at most $4m_0$.*

The proof of the lemma above is based on Invariant 2 and uses amortized analysis.

Theorem 5 (Lemma 5.1. in [3])**.** *If $d = c\lceil(\frac{m}{n})^{\frac{1}{3}}\rceil$ and $t = \min\{i > 0\colon n \leq S(i,d)^3\}$, then $t = O(\alpha(m,n))$*

The theorem below along with Lemmas 9 and 10 are proven by one induction and they reference each other in the induction step.

Theorem 6. *There exists a constant b that for any graph G and positive integer t such that* MST(G,t) *runs in time $bt(m + d^3(n-1))$. Here d is an integer large enough so that $n \leq S(t,d)^3$.*

Lemma 8 (Bound on **STEP 2** and **STEP 3**)**.** *There exists a constant b such that for any graph G and positive integer t **STEP 2** and **STEP 3** take at most $\frac{b}{2}(n + m + d^2n_0)$ time.*

Proof. **STEP 2** takes $O(n+m)$ time we can hide it in the b constant and we are left with the **STEP 3**.

The construction of the T-hierarchy is dominated by the soft heap operations and changes to `min-links`. Let's first consider the soft heap operations:

- `inserts` – From Lemma 7 we know that these operations take $O(m_0) = O(m)$ time.
- `make-heaps` – These operations only happen at the start and during the extension, which happens exactly $n_0 - 1$ times (Consider Prim's algorithm). Depth of the T-hierarchy is d thus in the worst case we could create d new heaps during 1 extension. It follows that there are at most $O(dn_0)$ of them
- `melds` – There cannot be more melds than `make-heaps`, thus there is at most $O(dn_0)$ of them.
- `pop-mins` – There are exactly $n_0 - 1$ of them.

Now for the `min-links` operations we can see that there are always less then d^2 elements in `min-links` array. And all updates and checks during the extension, fusion and retraction can be done in linear time of the size of the array. Thus there are $O(d^2n_0)$ operations on `min-links`.

Summing these values up and picking a sufficiently large b we get the $\frac{b}{2}(n + m + d^2n_0)$ bound. □

Lemma 9 (Bound on **STEP 4**)**.** *There exists a constant b such that for any graph G and positive integer t **STEP 4** takes at most $b(t-1)(m_0 - |B| + dn_0)$ time.*

The proof follows from the induction.

Lemma 10 (Bound on **STEP 5**)**.** *There exists a constant b such that for any graph G and positive integer t **STEP 5** takes at most $bt(n_0 - 1 + |B| + d^3(n_0 - 1))$ time.*

The proof follows from the induction.

Proof of Theorem 6. The proof is by induction on t and n.

Base case: for n and t small enough we only call **STEP 1** i.e. compute the $MST(G)$ in constant time as the n and t are less than some constant. We need to pick b large enough so that the theorem holds.

Induction step: Fix n and t and consider a graph G with n vertices. From Lemmas 8 to 10 we get the bound for each step of the algorithm. We can now finish the proof by summing them up:

$$\begin{aligned}\frac{b}{2}\left(n+m+d^2n_0\right)+b(t-1)\left(m_0-|B|+dn_0\right)+bt\left(n_0-1+|B|+d^3(n_0-1)\right)\\ \leq btm_0+b\left(\frac{m}{2}-m_0+|B|\right)+2btd^3n_0+\frac{bn}{2}\\ \leq btm-b(m-m_0)\left(t-\frac{1}{2}\right)+3btd^3n_0+\frac{bn}{2}.\end{aligned}$$

The last inequality follows from Lemma 6.

Now remember that $n_0 \leq \frac{n}{2^c}$ because we performed c times BORUVKA-STEPS at the beginning. Moreover, even after a single BORUVKA-STEP we get $m-m_0 > \frac{n}{2}$ and since we are in the inductive case $t > 1$ holds. Using these inequalities we can finally show that it is bounded by $bt\left(m+d^3(n-1)\right)$. □

Theorem 7. *The MST of a connected graph with n vertices and m edges can be computed in $O(m\alpha(m,n))$ time.*

Proof. It follows directly from Theorems 5 and 6. □

Chapter 3

Chazelle-Rubinfeld-Trevisan sublinear weight finding algorithm

The problem of finding the minimum spanning tree for a given graph requires at least linear time in terms of edges and vertices. However there might be a case in which only the weight of the MST is needed. This still would require linear time for a deterministic algorithm as each edge potentially can be in the MST, but it opens a possibility for faster approximate algorithms.

Problem 3 (Minimum spanning tree weight approximation)**.** Given a graph G and parameter ϵ return an approximation $M^*(G)$ of the $M(G)$ such that $\mathbb{P}(|M^*(G) - M(G)| \leq \epsilon M(G)) \geq \frac{3}{4}$.

Chazelle, Rubinfeld and Trevisan devised an algorithm that solves this problem [4]. To achieve this result they put additional constraint on the input graph – all edge weights have to be in the set $\{1, 2, \ldots, w\}$. Their algorithm's expected running time is $O(dw\epsilon^{-2} \log \frac{dw}{\epsilon})$ so it depends on ϵ and the average degree d of the input graph. Clearly e.g. for constant ϵ and constant upper bound w on edge weights the algorithm runs in time $O(d \log d)$ which is sublinear for sparse graphs.

3.1 Definitions

Definition 33. For a given graph G we denote by $G^{(i)}$ a subgraph of G which contains only the edges with weight at most i. That is, $E(G^{(i)}) = \{e \in E(G) : w_e \leq i\}$

Definition 34. For a given graph G and its minimum spanning tree T we denote by α_i the number of edges in T with weight equal to i.

Lemma 11. *For a given graph G with all edge weights from $\{1, 2, \ldots, w\}$ each MST of G has the same sequence $(\alpha_1, \alpha_2, \ldots \alpha_w)$.*

Proof. Let's assume that the claim is false, and take the smallest i for which there are two MST's T and T' with $\alpha_i < \alpha'_i$ respectively. Consider the connected components of $T^{(i)}$ and $T'^{(i)}$ – as the number of edges in each T differs and each edge removes 1 connected component it has to be that their connected components are different.

Now as $T^{(i)}$ has less connected components it has to have at least one connected component which is not a subset of any connected component in $T'^{(i)}$. And thus inside it has to be an edge uv such that in $T'^{(i)}$ u and v are in a different connected component. That is a contradiction with cycle property (see Theorem 2) as they will be connected by an edge with a larger cost, completing a cycle in which it will be the largest. □

3.2 Idea and pseudocode

The main idea behind the algorithm is to reduce the problem to counting connected components in subgraphs of the given graph. The reduction is based on the Theorem 8 (defined later) which states that:

$$M(G) = n - w + \sum_{i=1}^{w-1} c(G^{(i)}) \tag{3.1}$$

So given a function APPROXIMATE-CCS which returns an estimate of the number of connected components in a graph we can find an approximation for the $M(G)$ like this:

Algorithm 6 Approximate MST Weight

```
1: function APPROXIMATE-MST-WEIGHT($G$: Graph, $\epsilon$: float)
2:     $ans \leftarrow |V(G)| - w$
3:     for $i \leftarrow 1$ to $w - 1$ do
4:         $ans \leftarrow ans +$ APPROXIMATE-CCS($G^{(i)}$, $\epsilon$, $\frac{4w}{\epsilon}$, $d^*$)
5:     end for
6:     return $ans$
7: end function
```

The arguments with which the APPROXIMATE-CCS is called are tuned so that the complexity and running time of the algorithm is optimal. They will be discussed in the next section.

Approximating number of connected components

We denote by m_u the number of edges in the connected component of u.

The estimation of the connected component count is driven by the modified handshaking lemma:

Lemma 12 (Handshaking lemma)**.** *Let $G = (V, E)$ be a graph and let $\frac{d_u}{|E|} = 2$ if $|E| = 0$. Then the following equality holds:*

$$\sum_{u \in V(G)} \frac{1}{2} \frac{d_u}{|E|} = 1.$$

Proof. Each edge has two ends and contributes to two vertex degrees thus:

$$2|E| = \sum_{u \in V} d_u$$

Dividing both sides by $2|E|$ we get the claim. □

Theorem 8. *Let $G = (V, E)$ be a graph with c connected components. Assume that for its isolated vertices $\frac{d_u}{m_u} = 2$.*

$$\sum_{u \in V} \frac{1}{2} \frac{d_u}{m_u} = c. \tag{3.2}$$

Proof. For every connected component $G_i \subseteq V$ we have from the handshaking lemma (see Lemma 12):

$$\sum_{u \in V(G_i)} \frac{1}{2} \frac{d_u}{m_u} = 1.$$

Now if $V(G) = \bigcup_{i=1}^{c} V(G_i)$ and each G_i is a connected component then we have that:

$$\sum_{u \in V} \frac{1}{2} \frac{d_u}{m_u} = \sum_{i=1}^{c} \sum_{u \in V(G_i)} \frac{1}{2} \frac{d_u}{m_u} = c.$$

□

This theorem may be used as an estimator for the number of connected components. Additionally, we cannot compute $\frac{d_u}{m_u}$ for a fixed u directly, as it could require linear time, so we need to find its estimator β_u:

Definition 35 ($\frac{d_u}{m_u}$ estimator)**.** Given a vertex $u \in V(G)$ and constants W, d^* denote $S \subseteq V(G)$ the set of vertices which lie in connected components with fewer than W vertices which all have degrees $d_v \leq d^*$.

$$\beta_u = \begin{cases} 0 & \text{if } u \notin S, \\ 2 & \text{if } m_u = 0 \text{ with probability } \frac{1}{2}, \\ \dfrac{d_u}{m_u} \cdot 2^{\lceil \log(\frac{m_u}{d_u}) \rceil} & \text{with probability } 2^{-\lceil \log(\frac{m_u}{d_u}) \rceil}, \\ 0 & \text{otherwise.} \end{cases}$$

Calculation of the β estimator will use BFS, constant W and an estimate d^* of d to bound the number of vertices visited in the BFS.

The whole approximation scheme with the β estimator is:

Algorithm 7 Approximate the number of connected components

```
1: function APPROX-NUMBER-CCS(
2:     $G$: Graph
3:     $\epsilon$: float
4:     $W$: int    ▷ Upper bound on the number of visited vertices in the BFS
5:     $d^*$: float                                  ▷ Approximation of $d(G)$
   )
6:     $r \leftarrow O(\frac{1}{\epsilon^2})$
7:     $V^* \leftarrow \emptyset, E^* \leftarrow \emptyset$                      ▷ visited vertices and edges.
8:     for $i = 1, 2, \ldots, r$ do
9:         $u_i \leftarrow$ uniformly and independently drawn vertex from $V(G)$
10:        $\beta_i \leftarrow 0$
11:        coin-flips $\leftarrow 0$
12:        BFS-VISIT-VERTEX($u_i, V^*, E^*$)             ▷ visits all adjacent edges.
13:        (*) bit $\leftarrow$ RANDOM-BIT()
14:        coin-flips $\leftarrow$ coin-flips $+1$
15:        if bit $= 1$ and $|V^*| \leq W$ and $\forall_{v \in V^*} d_v \leq d^*$ then
16:            target-edges $\leftarrow 2 \cdot |E^*|$
17:            while $|E^*| <$ target-edges and not BFS-TERMINATED() do
18:                BFS-VISIT-NEXT-VERTEX(visited-edges)
19:            end while
20:            if BFS-TERMINATED() then
21:                if $E^* = \emptyset$ then
22:                    $\beta_i \leftarrow 2$
23:                else
24:                    $\beta_i \leftarrow 2^{\texttt{coin-flips}} \cdot \frac{d_{u_i}}{|E^*|}$
25:                end if
26:            else
27:                goto 13: (*)
28:            end if
29:        end if
30:    end for
31:    return $\hat{c} = \frac{n}{2r} \sum_{i=1}^{r} \beta_i$
32: end function
```

3.3 Correctness and running time analysis

Theorem 9 (Theorem 2 in [4]). *Let c be the number of connected components in a graph G with n vertices and average degree d. Then APPROXIMATE-NUMBER-CCS runs in time $O(d\epsilon^{-2}log(\frac{d}{\epsilon}))$ and returns $\hat{c}$ such that $\mathbb{P}(|c - \hat{c}| \leq \epsilon n) \geq \frac{3}{4}$.*

Proof. Let's first bound the estimator β_i for a fixed i.
By the choice $W = \frac{4}{\epsilon}$ there are at most $\frac{\epsilon n}{2}$ components with vertices not in S thus:

$$c - \frac{\epsilon n}{2} \leq \mathbb{E}[\hat{c}] \leq c$$

As $\beta_i \leq 2$ we can bound its variance by:

$$\mathrm{Var}(\beta_i) \leq \mathbb{E}[\beta_i^2] \leq 2\mathbb{E}[\beta_i] = \frac{2}{n}\sum_{u \in S} \frac{d_u}{m_u} \leq \frac{4c}{n}$$

Then looking at the algorithm we can bound the variance of $\hat{c}$ like this:

$$\mathrm{Var}(\hat{c}) = \mathrm{Var}\left(\frac{n}{2r}\sum_{i=1}^{r}\beta_i\right) = \frac{n^2}{4r^2}\cdot r \cdot \mathrm{Var}(\beta_i) \leq \frac{nc}{r}$$

Now from Chebyshev inequality and our choice $r = A\epsilon^{-2}$ for some constant A we get:

$$\mathbb{P}(|\hat{c} - c| > \epsilon n) \leq \mathbb{P}\left(|\hat{c} - \mathbb{E}[\hat{c}]| > \frac{\epsilon n}{2}\right) \leq \frac{4c}{\epsilon^2 r n} = \frac{Ac}{n}$$

By modifing A constant constant we can bound the probability of the event $|c - \hat{c}| > \epsilon n$ by an arbitrarily small constant, even when $c = \Theta(n)$.

The runtime analysis is quite simple. Denote X_i the number of edges visited when considering the vertex u_i and let $X = \sum_{i=1}^{r} X_i$. If $u_i \in S$ and RANDOM-BIT returned 1 exactly k times then we have visited no more than $d_{u_i}2^k$ edges. Such event happens with probability 2^{-k}. Note that $k \leq \lceil \log{(Wd^*)} \rceil$ as there are at most Wd^* edges in the connected component containing u_i.

So we have the following:

$$\mathbb{E}[X_i] \leq \sum_{i=1}^{\lceil \log Wd^* \rceil} \frac{1}{2^i} d_{u_i} \cdot 2^i \leq \lceil \log Wd^* \rceil d_{u_i}$$

Now we can use the linearity of expectations together with our choices of $r = O(\epsilon^{-2})$ `and` $Wd^* = O(\frac{d}{\epsilon})$ to get that:

$$\mathbb{E}[X] = \sum_{i=1}^{r} \mathbb{E}[X_i] \leq r \cdot d \lceil \log{(Wd^*)} \rceil = O\left(d\epsilon^{-2} \cdot \log \frac{d}{\epsilon}\right).$$

□

Theorem 10 (Claim 5 in [4]). *For integer $w \geq 2$ it holds that*

$$M(G) = n - w + \sum_{i=1}^{w-1} c^{(i)}. \tag{3.3}$$

Proof. Let α_i be the number of edges of weight i in an MST of G (From Lemma 11 we know that each MST has the same α_i).
Note that because each tree edge reduces the number of connected components by one, then for any $l \in \{0, 1, \ldots, w-1\}$ we have

$$\sum_{i>l} \alpha_i = c^{(l)} - 1. \tag{3.4}$$

Now the MST weight is just:

$$M(G) = \sum_{i=1}^{w} i\alpha_i = \sum_{i=1}^{w}\sum_{k=1}^{i} \alpha_i = \sum_{l=0}^{w-1}\sum_{i=l+1}^{w} \alpha_i =$$

$$= \sum_{l=0}^{w-1}(c^{(l)} - 1) = -w + \sum_{l=0}^{w-1} c^{(l)} = n - w + \sum_{l=1}^{w-1} c^{(l)}$$

□

Theorem 11 (Theorem 6 in [4]). *Let $\frac{w}{n} < \frac{1}{2}$. For any graph G the algorithm APPROXIMATE-MST-WEIGHT runs in time $O(dw\epsilon^{-2}\log\frac{dw}{\epsilon})$ and returns a value $M^*(G)$ such that for $M(G)$ we have $\mathbb{P}(|M^*(G) - M(G)| \leq \epsilon M(G)) \geq \frac{3}{4}$.*

Proof. Repeating the analysis from Theorem 9, but with $W = \frac{4w}{\epsilon}$ we get that the running time of the algorithm is $w \cdot O(d\epsilon^{-2}log(\frac{dw}{\epsilon}))$.

Let $c = \sum_{i=1}^{w-1} c^{(i)}$. To prove the correctness we need to first repeat the analysis of each $c^{(i)}$ estimation but with the new W value:

$$c^{(i)} - \frac{\epsilon n}{2w} \leq \mathbb{E}[\hat{c}^{(i)}] \leq c^{(i)} \text{ and } \mathrm{Var}(\hat{c}^{(i)}) \leq \frac{nc^{(i)}}{r}$$

Then after summing it over all i's we get:

$$c - \frac{\epsilon n}{2} \leq \mathbb{E}[\hat{c}] \leq c \text{ and } \mathrm{Var}(\hat{c}) \leq \frac{nc}{r}. \tag{3.5}$$

Once again apply the Chebyshev inequality to get:

$$\mathbb{P}\left(|\hat{c} - c| > (n - w + c)\frac{\epsilon}{3}\right) < \frac{9nc}{r\epsilon^2(n-w+c)^2} \tag{3.6}$$

which is arbitrarily small for $r = \Omega(\epsilon^{-2})$. Now combining (3.12) and (3.13) we get the bound on the error of the whole estimate:

$$|M^*(G) - M(G)| = |\hat{c} - c| \leq \frac{\epsilon n}{2} + \frac{\epsilon(n - w + c)}{3} \leq \epsilon M(G).$$

□

Chapter 4

Experiments

4.1 Introduction

The algorithms presented in this paper were implemented in `C++` using NetworKit and Koala libraries. The source code is submitted as a part of the Koala-NetworKit GitHub repository: `https://github.com/krzysztof-turowski/koala-networkit/pull/45`.

The algorithms and the soft heap are implemented in these files:

- `include/mst/MinimumSpanningTree.hpp`
- `cpp/mst/MinimumSpanningTree.cpp`
- `include/structures/heap/SoftHeap.hpp`

4.2 Results

4.2.1 Dataset

Three primary datasets were used for testing purposes were road maps of the New York city, the Great Lakes Area and Florida respectively, where the edge weights represent distances.

For the purpose of testing the randomized Chazelle-Rubinfeld-Trevisan algorithm five new graphs each were created from them with edge weights preprocessed so that $w'_e = \max\{w, \lceil \frac{w_e}{200} \rceil\}$ for $w \in \{15, 31, 63, 127, 255\}$. The runtime of that algorithm is dependent on the maximum edge weight in the graph and these datasets have large edge values.

For the purpose of testing Chazelle's determinisitc algorithm the three graphs were preprocessed so that they are connected and the all edges have distinct costs.

4.2.2 Chazelle's algorithm

Graph G	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	Chazelle	Borůvka	Time difference
New York	264346	733846	957	646	48%
Florida	1070376	2712798	4157	3126	33%
Lakes	2758119	6885658	13803	10292	34%

Table 4.1: Comparison of Chazelle's and Borůvka's algorithms. Time in ms.

As we can see from the Table 4.1 my implementation of the Chazelle's algorithm is slower then the Borůvka's algorithm. It is to be expected as Chazelle's algorithm uses the Borůvka's algorithm as a subroutine and in the time it takes to construct the T hierarchy several Boruvka-steps could be performed. Notice that the graphs are small enough so that the number of Boruvka-steps necessary to compute the minimum spanning tree is at most 22.

4.2.3 Chazelle-Rubinfeld-Trevisan algorithm

On each of the 3 datasets the algorithm was run with all possible pairs of values $\epsilon \in \{0.1, 0.05, 0.03, 0.02, .01\}$ and $x \in \{15, 31, 63, 127, 255\}$. Each configuration was run 5 times and the time and MST weight values used to create tables and plots were averaged over these 5 runs.

I will first show the tables and plots and analyze them at the end.

$x \backslash \epsilon$	0.1	0.05	0.03	0.02	0.01	Borůvka
New York City						
15	2.1	7.6	19.2	45.0	175.0	1129.0
31	5.8	17.7	51.7	115.2	492.2	1125.4
63	10.9	40.7	112.2	269.7	1175.8	1121.4
127	21.7	80.2	239.6	565.3	2503.0	1100.3
255	46.1	175.4	525.9	1257.3	5323.3	1136.4
Florida						
15	2.2	7.1	17.6	38.0	151.6	4579.7
31	5.3	18.2	49.2	109.5	470.7	4528.7
63	12.2	43.9	124.2	280.1	1241.8	4731.2
127	23.9	92.3	268.5	631.7	2807.5	4722.7
255	49.8	195.1	578.4	1383.7	5964.4	4708.8
Great Lake Area						
15	3.1	8.2	20.0	41.0	156.7	12620.3
31	6.7	21.8	63.0	129.7	631.7	13855.9
63	14.5	52.5	155.2	382.2	1449.4	14445.3
127	30.5	112.9	333.3	768.9	3303.6	14053.5
255	60.4	246.2	704.1	1663.2	7228.6	13955.5

Table 4.2: Average time in ms.

$x\backslash\epsilon$	0.1	0.05	0.03	0.02	0.01
New York City					
15	0.0144	0.0071	0.0055	0.0048	0.0021
31	0.0090	0.0063	0.0027	0.0046	0.0019
63	0.0141	0.0084	0.0028	0.0037	0.0010
127	0.0148	0.0058	0.0065	0.0049	0.0015
255	0.0142	0.0073	0.0039	0.0024	0.0021
Florida					
15	0.0224	0.0121	0.0068	0.0036	0.0021
31	0.0095	0.0107	0.0077	0.0027	0.0014
63	0.0222	0.0096	0.0017	0.0020	0.0017
127	0.0166	0.0084	0.0032	0.0031	0.0013
255	0.0169	0.0119	0.0089	0.0051	0.0014
Great Lake Area					
15	0.0108	0.0067	0.0035	0.0021	0.0013
31	0.0173	0.0070	0.0101	0.0039	0.0016
63	0.0198	0.0160	0.0012	0.0033	0.0014
127	0.0124	0.0043	0.0049	0.0016	0.0027
255	0.0216	0.0043	0.0029	0.0027	0.0013

Table 4.3: Average absolute relative error

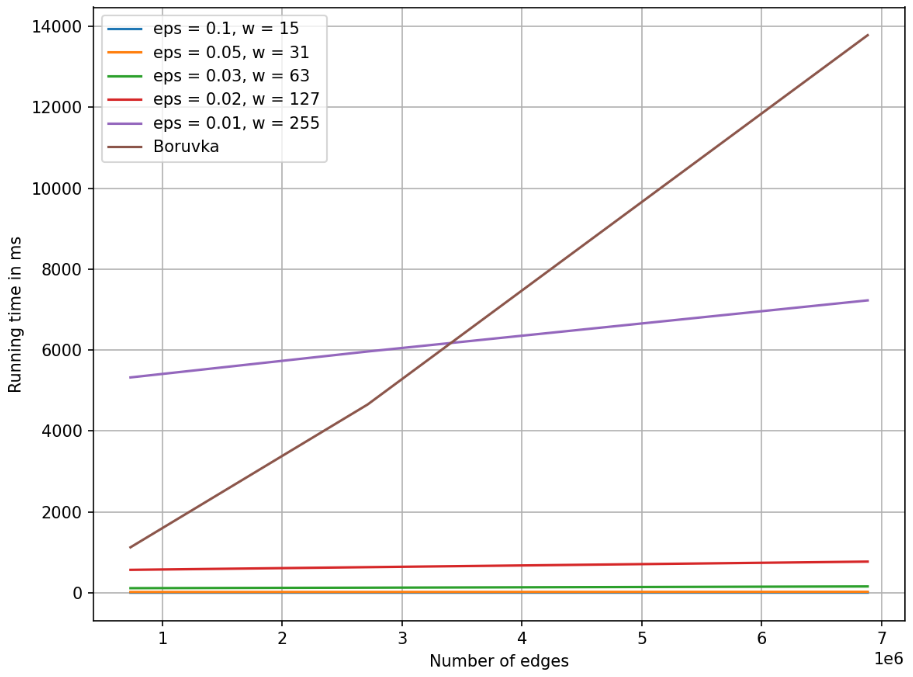

Figure 4.1: Comparison of the CRT and Borůvka algorithms across all datasets.

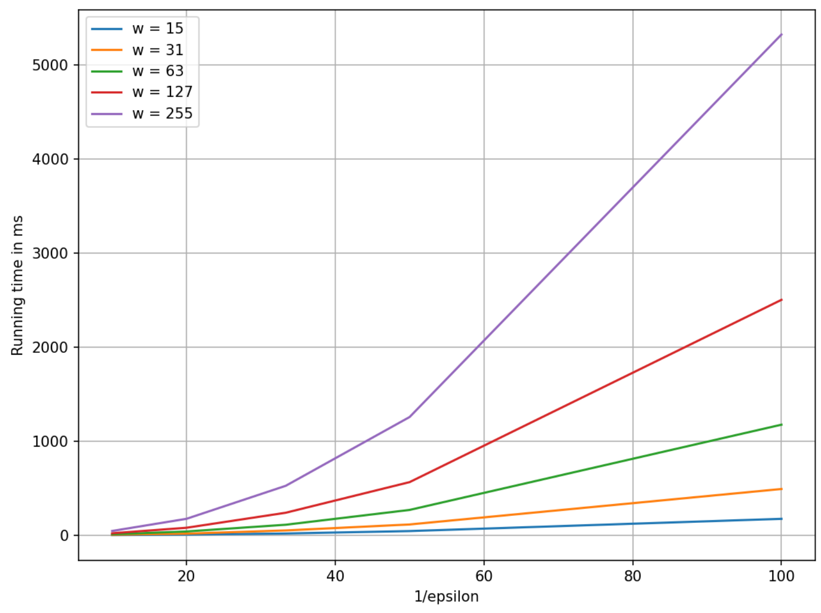

Figure 4.2: Comparison of the CRT algorithm run on the New York City road map graph processed with different w values.

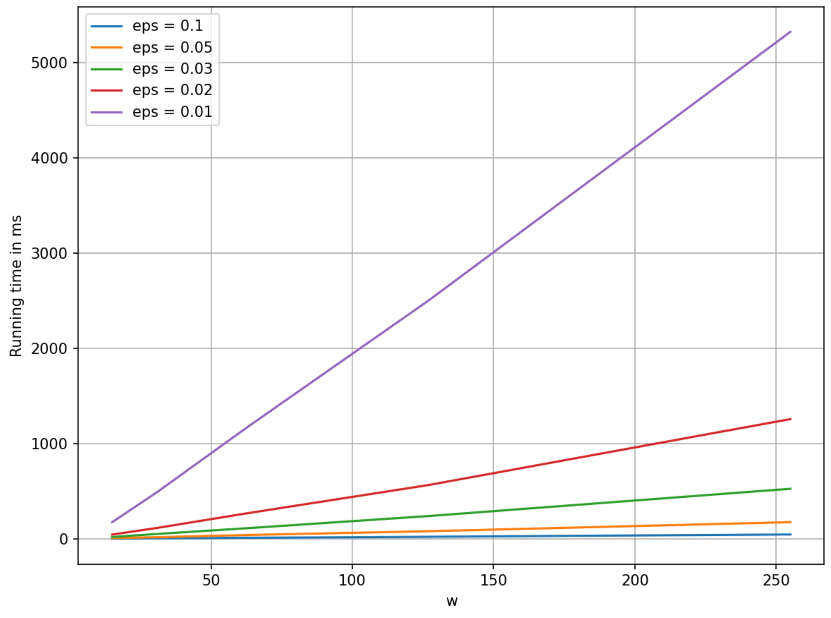

Figure 4.3: Comparison of the CRT algorithm run with different ϵ values on the Florida road map graph processed with different w values.

Looking at the results it is evident that Chazelle-Rubinfeld-Trevisan algorithm clearly outperform Borůvka's on most of the test cases. Focusing on the New York City graph we can see that the randomized algorithm provided accurate (relative error is much smaller then ϵ) results and finished much faster then the deterministic one, but for the most computationally intensive cases with $(\epsilon, w) \in \{(0.01, 255), (0.01, 127), (0.01, 63), (0.02, 255)\}$.

Remember that the expected time complexity of the Chazelle-Rubinfeld-Trevisan algorithm is $O(dw\epsilon^{-2} \log \frac{dw}{\epsilon})$. We will consider what happens when we fix the ϵ or the w parameter.

Fixing the ϵ value for a given graph shows that the algorithm running times grow slightly faster than a linear function in terms of w, that is, the maximum edge cost. This is illustrated by the columns of the average times tables or Figure 4.2.

On the other hand fixing the w for a given graph clearly shows a superlinear growth when plotted against the ϵ^{-1} that might indeed be $O(\epsilon^{-2} \log \epsilon^{-1})$ – as Figure 4.3 shows.

Figure 4.1 shows how the Chazelle-Rubinfeld-Trevisan runtime is affected by increasing the graph size. It is important to note that the average degrees in the input graphs are $5.55, 5.07, 4.99$ respectively, so while the number of edges rises, the average degree falls however slightly. Based on the complexity alone we should expect the running time to be slightly better for the bigger graphs. One possible explanation for the runtime growing slowly could be the hardware. Larger graph means that more calls to the memory result in cache misses as lower fraction of the data will fit in the caches.

4.3 Finishing thoughts

Chazelle's deterministic algorithm turned out to be more complex and slower in practice then Borůvka's algorithm. This was expected, since despite its superior theoretical complexity, the algorithm relies on multiple costly passes and recursive steps that make it less practical.

Chazelle, Rubinfeld and Trevisan algorithm displays promising results for computing the MST weight of graphs with small edge costs and with low precision. It outperformed Borůvka's algorithm significantly in most of the test cases. Modifying the graph by scaling the edge costs down or capping them at some maximum value can be useful to reduce the running time of the approximate algorithm, but it may change the MST weight drastically, far more then in the ϵ relative error range.

Bibliography

[1] CP-Algorithms. *Minimum Spanning Tree – Prim's Algorithm*. URL: `https://cp-algorithms.com/graph/mst_prim.html`.

[2] Otokar Borůvka. "Über ein Minimalproblem". In: *Práce moravské přírodovědecké společnosti* 3 (1926), pp. 37–58.

[3] Bernard Chazelle. "A minimum spanning tree algorithm with inverse-Ackermann type complexity". In: *Journal of the ACM (JACM)* 47.6 (2000), pp. 1028–1047.

[4] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. "Approximating the Minimum Spanning Tree Weight in Sublinear Time". In: *SIAM Journal on Computing* 34.6 (2005), pp. 1370–1379.

[5] David Cheriton and Robert Tarjan. "Finding Minimum Spanning Trees". In: *SIAM Journal on Computing* 5.4 (1976), pp. 724–742.

[6] Nicos Christofides. "Bounds for the travelling-salesman problem". In: *Operations Research* 20.5 (1972), pp. 1044–1056.

[7] Fengxia Cong and Ying Zhao. "The application of minimum spanning tree algorithm in the water supply network". In: *2015 International Industrial Informatics and Computer Engineering Conference*. Advances in Computer Science Research. Atlantis Press. 2015, pp. 52–55.

[8] Thomas Cormen et al. *Introduction to Algorithms*. 4th ed. Cambridge, MA: The MIT Press, 2022. ISBN: 9780262046305.

[9] Reinhard Diestel. *Graph Theory*. 6th ed. Vol. 173. Graduate Texts in Mathematics. Springer, 2025. ISBN: 9783662701065.

[10] Jason Eisner. *State-of-the-Art Algorithms for Minimum Spanning Trees: A Tutorial Discussion*. Tech. rep. Technical Report. University of Pennsylvania, 1997.

[11] Michael Fredman and Robert Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[12] Michael Fredman and Dan Willard. "Trans-dichotomous algorithms for minimum spanning trees and shortest paths". In: *Journal of Computer and System Sciences* 48.3 (1994), pp. 533–551.

[13] Harold Gabow, Zvi Galil, and Robert Tarjan. "Efficient Algorithms for Finding Minimum Spanning Tree in Undirected and Directed Graphs". In: *Combinatorica* 6.2 (1986), pp. 109–122.

[14] Torben Hagerup. "An even simpler linear-time algorithm for verifying minimum spanning trees". In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Ed. by Christophe Paul and Michel Habib. Vol. 5911. Lecture Notes in Computer Science. 2009, pp. 178–189.

[15] David Karger, Philip Klein, and Robert Tarjan. "A randomized linear-time algorithm to find minimum spanning trees". In: *Journal of the ACM (JACM)* 42.2 (1995), pp. 321–328.

[16] Valerie King. "A simpler minimum spanning tree verification algorithm". In: *Algorithmica* 18.2 (1997), pp. 263–270.

[17] János Komlós. "Linear verification for spanning trees". In: *Combinatorica* 5.1 (1985), pp. 57–65.

[18] Joseph Kruskal. "On the shortest spanning subtree of a graph and the traveling salesman problem". In: *Proceedings of the American Mathematical Society* 7.1 (1956), pp. 48–50.

[19] Bing Ma et al. "Image Registration With Minimum Spanning Tree Algorithm". In: *Proceedings 2000 International Conference on Image Processing (Cat. No.00CH37101)*. 2000, pp. 481–484.

[20] Seth Pettie and Vijaya Ramachandran. "An Optimal Minimum Spanning Tree Algorithm". In: *Journal of the ACM (JACM)* 49.1 (2000), pp. 49–60.

[21] Robert Prim. "Shortest connection networks and some generalizations". In: *Bell System Technical Journal* 36.6 (1957), pp. 1389–1401.

[22] Ernesto Tapia and Raúl Rojas. "Recognition of On-line Handwritten Mathematical Expressions Using a Minimum Spanning Tree Construction and Symbol Dominance". In: *Graphics Recognition. Recent Advances and Perspectives*. Ed. by Josep Lladós and Young-Bin Kwon. Vol. 3088. Lecture Notes in Computer Science. Springer, 2003, pp. 329–340.

[23] Andrew Chi-Chih Yao. "An $O(E \log\log V)$ Algorithm for Finding Minimum Spanning Trees". In: *Information Processing Letters* 4.1 (1975), pp. 21–23.