Course ”Trees - the ubiquitous structure in computer science and mathematics“, JASS’08

Minimum Spanning Trees

Maximilian Schlund

April 20, 2008

Contents

1 Preliminaries

Definition 1. Given a Graph $G = (V, E)$, together with a weight function $w : E \to \mathbb{R}$. A spanning acyclic Subgraph F with minimum total weight is called a minimum spanning forest (MSF). If F is connected (thus a tree) it is called minimum spanning tree (MST)

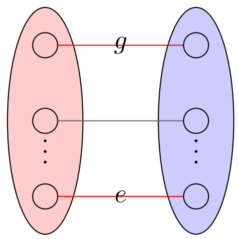
Definition 2.

- $F(a, b)$ denotes the path (if exists) in the graph F from node a to node b.
- by default we set $m := |E|$ and $n := |V|$

For simplicity, we assume that all weights in the Graph G are distinct, therefore the MST of G is unique.

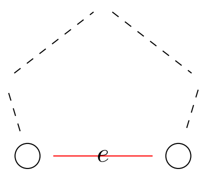
Theorem 3 (Cut property)**.** *Let $C = (V_1, V_2)$ be a cut in G, then the lightest edge e crossing the cut belongs to the MST of G.*

Theorem 4 (Cycle property)**.** *For any cycle C in G the heaviest edge e in C does not belong to the MST of G*



Proof.

1) Assume $e \notin MST(G)$. Let T be the MST and e be the lightest edge crossing the cut. Then $T \cup e$ yields a circle that includes at least two edges e and g crossing the cut. Exchanging e for g gives us a MST with lower weight and therefore a contradiction.



2) Assume $e \in MST$. Deleting $e = (v, w)$ splits the Tree in two parts which can be reconnected using an edge from the circle. This edge is lighter than e and so is the resulting spanning tree which yields a contradiction. □

2 Simple Algorithms

2.1 Dijkstra-Jarnik-Prim Algorithm (DJP)

Dijkstra-Jarnik-Prim Algorithm (DJP) Grows a tree T, one edge per step, starting with a tree consisting of one arbitrary vertex. Augment T by choosing an edge incident to T having the least weight. By the cut property this edge belongs to the MST of the Graph.

Algorithm 1 DJP Algorithm

$V(T) := \{v\}$; $E(T) = \emptyset$
for $n - 1$ times **do**
  choose lightest edge (x, y) indicent to T with $x \in T$ and $y \notin T$
  $V(T) := V(T) \cup \{y\}$
  $E(T) := E(T) \cup (x, y)$
end for

Runtime: $\mathcal{O}(m + n \log n)$ if implemented using Fibonacci-Heaps

2.2 Kruskal's Algorithm

- Sort all edges according to their weight in ascending order.

- Include edges successively in the MSF if they do not complete a circle.

Correctness: follows directly from cut and cycle properties.

2.3 Borůvka's Algorithm

Algorithm 2 Borůvka's Algorithm

```
make every vertex a singleton blue tree
repeat
  for each blue tree do
    select minimum weight edge incident to it
    color all selected edges blue
  end for
until there is one red tree
```

Each inner execution of the repeat-loop is called a Borůvka step. Each step reduces the number of vertices by at least 2 and takes $\mathcal{O}(m)$ time, therefore the total running time is in $\mathcal{O}(m \log n)$

3 Advanced Algorithms

3.1 MST Verification[Kin97]

MST Verification is the problem of checking whether a given tree is a MST of a given graph. The algorithm studied here will also be of use later, when we look at the randomized MST algorithm.

Theorem 5. *A spanning tree is a MST iff the weight of each nontree edge (u, v) is at least the weight of the heaviest edge in the path in the tree between u and v.*

Definition 6. Tree path problem: finding the heaviest edges in the paths between certain pairs of nodes ("query paths").

This Problem was solved by Komlòs in 1984. His algorithm requires a linear number of comparisons but nonlinear overhead for preprocessing.

King [Kin97] presented a simpler algorithm using Komlòs' algorithm on a full branching tree B which gives linear runtime (on a unit cost RAM).

Theorem 7. *If T is a spanning tree then there is an $\mathcal{O}(n)$ algorithm that constructs a full branching tree B s.t.*

- *B has not more than $2n$ nodes*
- *For any pair of nodes x and y in T, the weight of the heaviest edge in $T(x, y)$ equals the weight of the heaviest edge in $B(x, y)$*

Proof. see [Kin97]. □

Construction of B: We run Borůvka's algorithm on a **tree** T

- for each node v in V we create a leaf $f(v)$ for B
- let $A = \{v \in V|$ v contracted into t by Borůvka step$\}$. Add new node $f(t)$ to B and add $\Big\{(f(a), f(t))|\forall a \in A\Big\}$ to the set of edges in B

The runtime of a Borůvka step is proportional to the number of "uncolored" edges. This number drops by a factor of 2 after each step (because T is a tree) Therefore the runtime is in $\mathcal{O}(n)$.

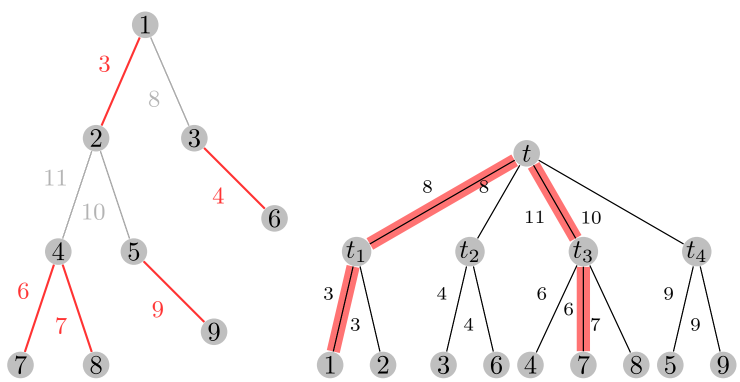


Figure 1: Borůvka's algorithm applied to the left tree yields the tree on the right. The red marked path is an example for a query path for the pair of leaves $(1, 7)$

Komlòs algorithm on full branching trees:

Goal: Find the heaviest edge between every pair of leaves (query path)

Idea: Break each path in two half-paths, from each leaf to the lowest common ancestor.

$\rightsquigarrow$ Then finding the heaviest edge in a query path then requieres one additional comparison.

Definition 8.

- $A(v)$ denotes the set of paths which contain v and are restricted to the "interval" $[root, v]$
- $A(v|a)$ denotes the set of restrictions of each path in $A(v)$ to the interval $[root, a]$
- $H(p)$ denotes the weight of the heaviest edge in a path p

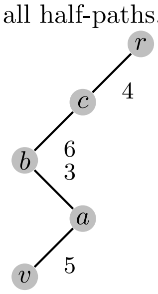
Example:

- $A(v) = \{(v,a),(v,a,b),(v,a,b,c),(v,a,b,c,root)\}$
- $A(v|a) = \{(a,b),(a,b,c),(a,b,c,root)\}$
- $length(s) > length(t) \Rightarrow H(s) \geq H(t)$ for any two paths s, t in $A(v)$ therefore the order of $H(A(v))$ is determined by length of paths.

Algorithm:

- ▶ Starting with the root descend the Tree level by level
- ▶ At each node v, the heaviest edge in the Set $A(v)$ is determined:
  - – Let p be the parent of v and assume we know $H(A(p))$.
  - – Then $H(A(v))$ can be found by comparing v, p to each weight in $H(A(p)$ using binary search.

Komlòs showed, that $\sum_{v \in T} \log |A(v)| \in \mathcal{O}(n \log((m+n)/n)) \subset \mathcal{O}(m)$, which is an upper bound on the number of comparisons needed to find the heaviest edge in all half-paths.



$A(r) = \emptyset$

$A(c) = \{(c,r)\}$

$A(b) = \{(b,c),(b,c,r)\}$

$A(a) = \{(a,b),(a,b,c),(a,b,c,r)\}$

$A(v) = \{(v,a),(v,a,b),(v,a,b,c),(v,a,b,c,r)\}$

$H(A(c)) = \{4\}$

$H(A(b)) = \{6,6\}$

$H(A(a)) = \{3,6,6\}$

$H(A(v)) = \{5,5,6,6\}$

Summary Verify that T is a MST

- generate full-branching tree B via Borůvka algorithm applied to T

- precompute the heaviest edge of all half-paths in B
- precompute all lowest common ancestors in B for the leaves x and y that form a non-tree edge (x, y) in T
- for every non-tree edge $e = (x, y)$ in T compare $w(e)$ to heaviest edge in half-paths $B(x, lca)$ and $B(lca, y)$

Remark: the LCA of all pairs can be computed in $\mathcal{O}(m + n)$, therefore the total running time of the algorithm is in $\mathcal{O}(m)$

3.2 A Randomized Linear-Time Algorithm for MST[DRK95]

Here we will study a randomized algorithm that runs in expected linear time on a unit cost RAM. One can also show(see [DRK95]) that it runs in linear time with very high probability.

Definition 9. Let G be a weighted graph and F be a forest in G.

- $w_F(x, y)$ denotes the maximum weight of an edge on $F(x, y)$
- An edge (x, y) is called F-heavy if $w(x, y) > w_F(x, y)$ and F-light otherwise

Note:

- All edges of F are F-light
- For any forest F, no F-heavy edge can be in the MSF of G by the cycle property.

Algorithm

Step(1) Apply two Borůvka steps to the graph, reducing the number of vertices by at least a factor of 4

Step(2) In the contracted graph choose a subgraph H by selecting each edge with probability $1/2$. Apply the algorithm **recursively** on H, to get a MSF F of H. Find all the F-heavy edges (both those in H and not in H) and delete them.

Step(3) Apply the algorithm **recursively** to the remaining graph to compute a spanning forest F'. Return the edges contracted in Step(1) together with the edges of F'

Remarks:

- all edges in $H - F$ are F-Heavy, but there may be more in the rest of G
- only the edges that are in F appear in **both** recursions

Correctness is proved by induction and cycle property (every edge that is deleted in Step(2) cannot be in the MSF)

Analysis:

Step(1) takes time $\mathcal{O}(m)$ (2 Borůvka steps).

Step(2) finding F-heavy edges takes time $\mathcal{O}(m)$ using Komlòs algorithm as described in [Kin97]

so for some constant c we can describe the runtime as:

$$T(n,m) = cm + \underbrace{T(n_2, m_2)}_{\text{recursion in Step(2)}} + \underbrace{T(n_3, m_3)}_{\text{recursion in Step(3)}}$$

- imagine recursion tree: left child of a node is seen as the recursion in Step(2), right child as the recursion in Step(3)

Theorem 10. *Wort-case Runtime of the algorithm is in* $\mathcal{O}(\min\{n^2, m\log n\})$

Proof. First we proof the $\mathcal{O}(n^2)$ Bound

- consider subproblem in depth d: the number of nodes at depth d is at most $n/4^d \Rightarrow$ num. of edges $\leq (n/4^d)^2/2$
- sum over all subproblems: the total number of edges is not greater than $\sum_{d=0}^{\infty} 2^d \frac{n^2}{2\cdot 4^{2d}} \leq \frac{n^2}{2}\sum_{d=0}^{\infty} \frac{2^d}{2^{2d}} = n^2$

Now the $\mathcal{O}(m\log n)$ bound:

- ▶ Consider a parent problem in the recursion tree with v vertices. Edges of F are in both subproblems, edges revomed in Step(1) are in none. All other edges are in exactly one subproblem.
- ▶ After Step(1) there are $v' \leq v/4$ nodes left in $G \Rightarrow F$ has at most $v' - 1 \leq v/4$ edges. But at least $v/2$ edges are removed in Step(1) therefore the total number of edges in both subproblems does not increase.
- ▶ $\Rightarrow$ Number of edges in all subproblems at depth d in recursion tree is $\leq m$ $\Rightarrow$ as the recursion tree has depth $\mathcal{O}(\log n)$ the runtime is in $\mathcal{O}(m\log n)$

□

Theorem 11. *The expected runtime of the algorithm is in* $\mathcal{O}(m)$

Proof. Let $T(n,m)$ now denote the expected runtime of the algorithm. We define two random variables:

- $X :=$ number of edges in the parent problem
- $Y :=$ number of edges in the left subproblem (i.e. the recursion in Step(2))

In Step(2) each edge that was not removed in Step(1) is included with probability $1/2 \Rightarrow E[Y|X = k] \leq k/2 \Rightarrow E[Y|X] \leq X/2 \Rightarrow E[Y] \leq E[X]/2$

$$T(n, m) = cm + \underbrace{T(n_2, m_2)}_{\text{recursion in Step(2)}} + \underbrace{T(n_3, m_3)}_{\text{recursion in Step(3)}}$$

$$T(n, m) \leq cm + T(n/4, m/2) + T(n/4, \underbrace{n/2)}_{\text{by Lemma 12}}$$

$$\Rightarrow T(n, m) \in \mathcal{O}(m + n) \subset \mathcal{O}(m)$$

□

Lemma 12. *Let H be a subgraph of G obtained by including each edge independently with probability p and let F be the MSF of H. Then the expected number of F-light edges in G is $\leq n/p$.*

Proof. Using the mean of the negative binomial distribution. see [DRK95] □

3.3 An Optimal MST Algorithm[SP01]

3.3.1 Overview

This algorithm developed by Pettie and Ramachandran[SP01] is asymptotically optimal on a pointer machine, although its runtime is unknown. This is due to the use of decision trees for edge weight comparisons, which use a minimum number of comparisons. The Problem is, that the exact depth of these decision trees is not known.

An improved version of Prim's algorithm discovered by Fredman and Tarjan [MLF87] can compute the MST of a graph in time $\mathcal{O}(m\beta(n, m))$ with $\beta(n, m) = \min\{i | \log^{(i)} n < n/m\}$. So for graphs with density $\Omega(\log^{(3)} n)$ this yields a linear-time algorithm. We will refer to this as the DenseCase algorithm.

The idea is now to partition the graph in components that are "small", computing the MSF for these components using decision trees and then contracting the components into single vertices making the graph sufficiently dense so that the DenseCase algorithm can be applied. Then we use Borůvka steps to reduce the number of edges and recurse.

3.3.2 Soft Heaps

A central data structure used by the algorithm is the Soft Heap (see [Cha00]), an approximate priority queue with a fixed error rate ϵ It supports all heap operations (Insert, FindMin, Delete, Union) in constant armotized time except for Insert which takes time $\mathcal{O}(\log(\frac{1}{\epsilon})$.

This is achieved by using "car pooling" for datastructures: Items are grouped together sharing the same key. To achieve this, Items can adopt larger keys from other items and are called corrupted as a result. It is however guaranteed that after n Inserts, no more than ϵn corrupted Items are in the heap. This is shown in [Cha00]:

Lemma 13. *For any $0 < \epsilon \leq 1/2$, a soft heap with error rate ϵ supports each operation in constant amortized time, except for insert, which takes $\mathcal{O}(\log(\frac{1}{\epsilon}))$ time. The data structure never contains more than ϵn corrupted items at any given time.*

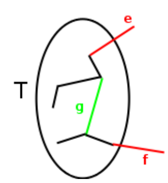
3.3.3 Decision Trees

A MST decision tree for a graph is a binary tree with edge weight comparisions associated with each node (e.g. $w(e_1) < w(e_5)$). The left child represents that the comparison is true, the right child that it is false. The leaves of a decision tree are annotated with the edges that are in some spanning tree. The tree is called *correct*, if the edge weight comparisons decending from the root identify the MST of the input tree uniquely. The decision tree is *optimal* if there is no correct decision tree with lesser depth.

By combinatorical arguments one can bound the time needed by a brute force algorithm to generate, check and find the optimal one among all possible decision trees of a graph with $\log^{(3)}$ vertices by $o(n)$ (see [SP01]).

3.3.4 Important lemmas

Lemma 14 (DJP Lemma)**.** *Let T be a tree formed after some number of steps of the DJP algorithm. Let e and f be two arbitrary edges with exactly one endpoint in T and let g be the maximum weight edge on the path from e to f in T. Then g cannot be heavier than both e and f.*

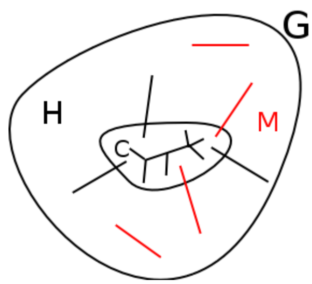


Proof. Let $\mathcal{P}$ be the path connecting e and f, assume the contrary, that g is the heaviest edge in $\mathcal{P} \cup \{e, f\}$. At the moment g is selected by DJP there are two edges eligible one of which is g. If the other edge is in $\mathcal{P}$ then it must be lighter than g. If it is either e or f then by the assumption it must be lighter than g. In both cases g could not be chosen next by DJP so we have a contradiction. □

Definition 15. Let F be a subgraph of G. $G \setminus F$ denotes the graph that results from G by contracting all connected components formed by F.

Definition 16. Let M and C be Subgraphs of G.

- $G \Uparrow M$ the graph obtained from G when raising the weight of every edge in M by an arbitrary amount (these edges are *corrupted*)
- M_C is the set of edges in M with exactly one endpoint in C
- C is said to be DJP-contractable if after some steps of the DJP algorithm with start in C the resulting tree is a MST of C



Lemma 17 (Contraction lemma)**.** *Let M be a set of edges in a graph G. If C is a subgraph of G that is DJP-contractable w.r.t. $G \Uparrow M$, then*

$$MSF(G) \subset MSF(C) \cup MSF(G \setminus C - M_C) \cup M_C$$

Proof [SP01]. We prove $MSF(G)^C \supset \underbrace{MSF(C)^C}_{(1)} \cap MSF(G \setminus C - M_C)^C \cap M_C^C$

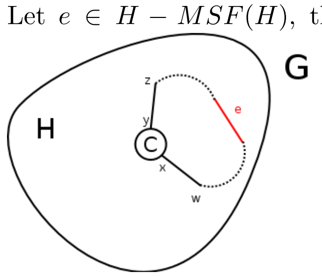
where A^C denotes the complement of the set A (concerning the edges, so $MSF(C)^C = C - MSF(C)$)

(1) Every edge in C that is not in $MSF(C)$ is the heaviest edge on a cycle in C (because C has a MST). This cycle exists in G as well, so this edge is also not in the MSF of G.

It remains to show that $MSF(G)^C \supset MSF(G \setminus C - M_C)^C \cap M_C^C$. Set $H := G \setminus C - M_C$. Then we are left with

$$MSF(G)^C \supset H - MSF(H) \cap \underbrace{G \setminus C - M_C}_{=H} = H - MSF(H)$$

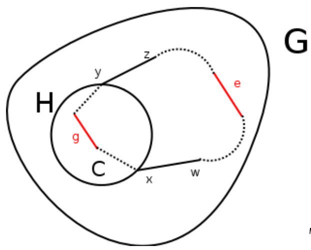
Let $e \in H - MSF(H)$, then e is the heaviest edge on some cycle χ in H.



1) If χ does not involve the super-node C then it exists in G as well and $e \notin MSF(G)$.

2) Otherwise χ includes a path $\mathcal{P} = (x, w, \ldots, z, y)$ in H with $x, y \in C$. Since H includes no corrupted edges with one endpoint in C, the G-weight of the end edges (x, w) and (z, y) is the same as their $(G \Uparrow M)$-weight.

Let T be the spanning tree of $C \Uparrow M$ that was found by the DJP algorithm, $\mathcal{Q}$ be the path in T connecting x and y, and g be the heaviest edge in $\mathcal{Q}$.



Then $\mathcal{P} \cup \mathcal{Q}$ forms a circle with e being heavier than both (x, w) and (y, z). By the DJP-Lemma 14 the heavier of these both edges is heavier than the $G \Uparrow M$-weight of g which is an upper bound on the G-weigths of all edges in $\mathcal{Q}$. So w.r.t. G-weights, e is the heaviest edge on the cycle $\mathcal{P} \cup \mathcal{Q}$ and thus cannot be in $MSF(G)$ □

Corollary 18. *by applying Lemma 17 i times we get*

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

3.3.5 The Algorithm

Overview of the optimal algorithm:

1) find DJP-contractable subgraphs $C_1, C_2, \ldots, C_k$ with their associated sets $M = \bigcup_i M_{C_i}$, where M_{C_i} consists of corrupted edges with exactly one endpoint in C.

2) Find MSF F_i of each C_i by using precomputed decision trees for edge weight comparisons. Also find the MSF F_0 of the contracted graph $G \setminus (\bigcup_i C_i) - \bigcup_i M_{C_i}$. By Lemma 17 the MSF of G is contained within $F_0 \cup \bigcup_i (F_i \cup M_{C_i})$.

3) Find some edges of the MSF of G via two Borůvka steps and recurse on the contraced graph

Note

- in Step 1) we make sure that each C_i is extremely small ($< \log^{(3)} n$ vertices) so we can apply the decision trees in Step2)
- until Step 3) no edges of the MSF of G have been identified - we only have discarded lots of edges.
- F_0 in Step can be found by the DenseCase algorithm

The procedure Partition (Algorithm 3) – taken from [SP01] – finds the DJP-contractable subgraphs and the set M.

- We partition the Graph into DJP contractable components that are very small i.e. have less than $\log^{(3)} n$ vertices.

Algorithm 3 Partition (G, $maxsize$, ϵ) returns $M, \mathcal{C}$

```
All vertices are initially "live"
M := ∅; i := 0
while there is a live vertex do
  i := i + 1
  Let V_i := {v} where v is any live vertex
  Create a Soft Heap consisting of v's edges
  while all vertices in V_i are live and |V_i| < maxsize do
    repeat
      delete min-weight edge (x, y) from Soft Heap
    until y ∉ V_i
    V_i := V_i ∪ y
    If y is live then insert each of y's edges into the Soft Heap
  end while
  Set all vertices in V_i to be dead
  Let M_{V_i} be the corrupted edges with one endpoint in V_i
  M := M ∪ M_{V_i}      G := G − M_{V_i}
  Dismantle the Soft Heap
end while
Let C := {C_1, ..., C_i} where C_k is the subgraph of G induced by V_k
Return M, C
```

- The growing of a component stops if it has reached its maximum size, or it attaches to an existing component with at least $\log^{(3)} n$ vertices
- Then we delete all corrupted edges M_c and contract all remaining connected components into single vertices
- As each connected component consists of at least $\log^{(3)} n$ vertices the resulting graph has not more than $n/\log^{(3)} n$ vertices and we can apply the DenseCase algorithm to the remaining graph

3.3.6 Analysis

Apart from recursive calls the computation is clearly linear. Partition takes $\mathcal{O}(m \log(1/\epsilon))$ time and because of the reductions in vertices DenseCase also takes linear time. For $\epsilon = \frac{1}{8}$ the number of edges passed to the recursive calls is $\leq m/4 + n/4 \leq m/2$ which gives a geometric reduction in the number of edges. The lower bound for any MSF algorithm is $\mathcal{O}(m)$, so the only bottleneck, if any, must lie in the decision trees, which are optimal by construction. In [SP01] it is shown that

$$T(m, n) \in \mathcal{O}(\mathcal{T}^*(m, n))$$

if T is the runtime of our algorithm and T^* is the optimal number of comparisions needed for determining the MSF of an arbitrary graph.

Algorithm 4 **OptimalMSF**(G)

if $E(G) = \emptyset$ **then**
  Return $(\emptyset)$
end if
$r := \log^{(3)} |V(G)|$
$M, \mathcal{C} :=$ **Partition**(G, r, ϵ)
$\mathcal{F} :=$ **DecisionTrees**$(\mathcal{C})$
Let $k := |C|$, let $\mathcal{C} = \{C_1, \dots, C_k\}$ and $\mathcal{F} = \{F_1, \dots, F_k\}$
$G_a := G \setminus (F_1 \cup \dots \cup F_k) - M$
$F_0 :=$**DenseCase**(G_a)
$G_b := F_0 \cup F_1 \cup \dots \cup F_k \cup M$
$F', G_c :=$ **Boruvka2**(G_b)
$F :=$ **OptimalMSF**(G_c)
Return $(F \cup F')$

4 Summary

- We can verify a MST in linear time on a RAM with wordsize $\log n$
- There is an randomized algorithm that runs in expected linear time and w.h.p. in "real" linear time
- The MST can be computed optimally on a pointer machine - but we do not know the worst case runtime

Open problems:

- Is there a linear time MST algorithm that runs on pointer machines?
- Is there an optimal algorithm that does not use precomputed decision trees?
- Can we find optimal parallel algorithms for the MST problem?
- Optimal MST algorithms for euclidean spanning trees in three or more dimensions

References

[Cha00] B. Chazelle. The soft heap: An approximate priority queue with optimal error rate. *Journal of the ACM*, 47(6):1012–1027, 2000.

[DRK95] R. E. Tarjan D. R. Karger, P. N. Klein. A Randomized linear-time algorithm to find Minimum Spanning Trees. *Journal of the ACM*, 42(2):321–328, 1995.

[Kin97] V. King. A simpler Minimum Spanning Tree Verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[MLF87] R. E. Tarjan M. L. Fredman. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[SP01] V. Ramachandran S. Pettie. An Optimal Minimum Spanning Tree algorithm. Technical report, Department of Computer Sciences, The University of Texas at Austin, 2001. URL: http://www.cs.utexas.edu/users/vlr/papers/optmsf.pdf.