Linear-Time Minimum Spanning Tree Algorithms

LOUKAS DROSOS

Department of Physics, National Kapodistrian University of Athens, Greece

Abstract. The problem considered here is that of finding a minimum spanning tree in a connected graph. Two algorithms with linear-time time complexity have been presented in the minimum spanning tree problem history, a randomized linear-time algorithm was presented in 1995 by Karger, Klein and Tarjan, and a linear-time deterministic algorithm was presented in 2000 by Chazelle.

1. *Introduction*

The minimum spanning tree problem is classic: We have a connected, undirected graph $G = (V, E)$ that has n vertices and m edges, as well as a weight function $w: E \rightarrow R$ on the edges. For all edges $(u, v) \in E$, we assume that their weights are pairwise distinct. The goal is to compute a minimum spanning tree of G, i.e., an acyclic subset $T \subseteq E$ that connects all of the vertices, with the minimum total edge-weight [1], [2].

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

In case the graph G is not connected, the goal is to find a minimum spanning forest, which is a minimum spanning tree in each connected component of G. Both problems will be referred to as the minimum spanning tree problem [3].

The origins of the minimum spanning tree problem extend deep into history, reaching back to Borůvka's influential contributions in 1926, which is the earliest known minimum spanning tree algorithm and runs in $O(\min\{n^2, m \log n\})$ steps. Two further classic algorithms are Kruskal's algorithm, a greedy algorithm that runs in $O(m \log n)$ time, and the Prim's algorithm, that requires $O(m + n \log n)$ time and can be improved over the binary-heap implementation if $|n|$ is much smaller than $|m|$. An improved faster general deterministic algorithm that runs in linear-time is due to Chazelle. It runs in $O(m\, \alpha(m, n))$ time, where $\alpha(m, n)$ is the slow-growing inverse Ackermann function. If randomness is taken into account, there is a simple linear-time algorithm due to Karger, Klein, and Tarjan that runs in $O(m)$ time with high probability in the restricted random-access model [2], [4].

2. *The KKT Algorithm*

The Karger-Klain-Tarjan (or KKT) randomized minimum spanning tree algorithm was first introduced in 1995. It solves the slightly more general problem of finding a minimum spanning forest in a possibly disconnected graph that has no isolated vertices [4].

2.1 *Cut and Cycle Properties*

The algorithm relies on two rules: the *cut rule* and the *cycle rule* (also known as the *blue rule* and the *red rule* respectively in Bob Tarjan's monograph).

Lemma 2.1 (Cut Rule). *The cut rule states that for any cut of the graph, the minimum-weight edge that crosses the cut must be in the minimum spanning tree. This rule helps us determine what to add to our minimum spanning tree* [5].

Proof. Let $S \subsetneq V$ be any nonempty proper subset of vertices, let $e = \{u, v\}$ be the minimum-weight edge that crosses the cut defined by $(S, \bar{S})$, and let T be a spanning tree not containing e. Then $T \cup \{e\}$ contains a unique cycle C. Since C crosses the cut $(S, \bar{S})$ once at the edge e, it must also cross at another edge e'. But $w(e') > w(e)$, so $T' = (T - \{e'\}) \cup \{e\})$ is a lower-weight tree than T, so T is not the minimum spanning tree. Since T was an arbitrary spanning tree not containing e, the minimum spanning tree must contain e [5].

Lemma 2.2 (Cycle Rule). *For any cycle in the graph G, if the weight of an edge of a cycle is larger than any of the individual weights of all other edges of that cycle, then this edge cannot belong in the minimum spanning tree* [6].

Proof. Let C be any cycle and let e be the heaviest edge in C. For a contradiction, let T be a minimum spanning tree that contains e. Dropping e from T gives two components. Now there must be some edge e' in $C \setminus \{e\}$ that crosses between these two components, and hence $T' = (T - \{e'\}) \cup \{e\}$ is a spanning tree. By the choice of e we have $w(e') < w(e)$, so T' is a lower-weight spanning tree than T, a contradiction [5].

2.2 *Light and Heavy Edges*

For the randomized algorithm it is crucial to define that an edge e can be heavy or light with respect to some forest F of a graph G. An edge $e \in E$ is *F-light* if $F \cup \{e\}$ either continues to be a forest of G, or the heaviest edge in the cycle containing e is *not* e. An edge in G that is not F-light is *F -heavy*. Because of the cycle property, if an edge is F-heavy edge then it cannot be in the minimum spanning tree of G, no matter what forest F is used [3].

That means that if an edge is F-light, then it belongs to the minimum spanning tree F. The opposite is true as well, if F is a minimum spanning tree of graph G, then edge e is *F-light* if and only if it belongs to the minimum spanning tree F. For any forest F, the *F-light* edges contain the minimum spanning tree of the underlying graph G. In other

words, any *F-heavy* edge is also heavy with respect to the minimum spanning tree of the entire graph [5].

Given a forest *F* of *G*, the set of *F-light* edges can be determined in linear time O(m + n) using a minimum spanning tree verification algorithm.

Lemma 2.3. *Let H be a subgraph obtained from G by including each edge independently 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 n/p, where n is the number of vertices of G* [4].

Proof. Let *F* be a minimum spanning tree of graph *G*, and *G* has n vertices. An edge can be *F-light* if a coin flip turns up heads. If the probability of getting heads is *p*, to obtain one head the expected number of coin flips is $1/p$. In order to obtain *n* heads, the expected number of coin flips, from linearity of expectation, is $\sum_{i=1}^{n} 1/p = n/p$. Since *F* can only have *n* − 1 edges, the number of *F-light* edges in *G* is at most *n/p* [4], [5].

2.3 *The Algorithm*

The fundamental idea behind the algorithm involves a pivotal step of random sampling. This particular step plays a crucial role in partitioning a graph into two distinct subgraphs by selecting edges in a random manner for inclusion in each subgraph. These steps are called Borůvka steps and are a part of Borůvka's algorithm. In a Borůvka step we select for each vertex the minimum-weight edge incident to the vertex. Then we contract all the selected edges, replacing by a single vertex each connected component defined by the selected edges and deleting all resulting isolated vertices, loops (edges both of whose endpoints are the same), and all but the lowest-weight edge among each set of multiple edges. By the cut property of minimum spanning trees, all edges contracted are in the minimum spanning tree. Each Borůvka step runs in $O(m)$ time where m is the number of edges in G, and reduces the number of vertices by at least a factor of two, while each random-sampling step discards enough edges to reduce the density (ratio of edges to vertices) to fixed constant with high probability [4], [7].

The algorithm recursively finds the minimum spanning forest of the first subproblem and uses the solution in conjunction with the linear time verification algorithm mentioned previously to discard edges in the graph that cannot be in the minimum spanning tree [8]. The algorithm gets an undirected and weighted graph *G = (V, E)* with no isolated vertices as input.

MST – KKT (G, w)

1. if $G = \emptyset$
2. return an empty forest
3. G' = BORUVKA(G)
4. G' = BORUVKA(G')
5. E = Contracted edges from the Borůvka steps
6. H = Subgraph of G' by selecting each edge with probability 1/2

7. F = Minimum Spanning Forest of H
8. Remove all *F-heavy* edges from G' using a linear time minimum spanning tree verification algorithm
9. F' = Minimum Spanning Forest of G'
10. $T = F' \cup E$
11. return T

The first two lines of the algorithm return an empty forest if the input graph is empty. Lines 3-4 apply two successive Borůvka steps to the graph, thereby identifying many minimum spanning tree edges and reducing the number of vertices by at least a factor of four, while line 5 contains the contracted edges from lines 3-4. Line 6 creates a subgraph H by selecting each edge independently with probability 1/2, while line 7 applies the algorithm recursively to H, producing a minimum spanning forest F of H. Line 8 finds all the *F-heavy* edges (both those in H and those not in H) using a linear time minimum spanning tree verification algorithm, and deletes them from the contracted graph. Line 9 applies the algorithm recursively to the remaining graph to compute a spanning forest F'. Lines 10-11 return those edges contracted in the Borůvka steps together with the edges of F' [4].

We establish the correctness of the algorithm through an inductive argument. Utilizing the cut property, it can be affirmed that each edge contracted in the two successive Borůvka steps is inherently part of the minimum spanning forest. Consequently, the remaining edges, constituting the minimum spanning forest of the original graph, also form a minimum spanning forest for the contracted graph. As per the cycle property, the edges removed using a linear time minimum spanning tree verification algorithm do not belong to the minimum spanning forest. Leveraging the inductive assumption, we can assert that the final recursive call accurately determines the minimum spanning forest for the remaining graph. This iterative process reinforces the algorithm's reliability in computing the minimum spanning forest for the entire graph [4].

2.4 *Analysis*

The running time of Karger's, Klain's and Tarjan's algorithm for a given graph $G = (V, E)$, with $|V| = n$ and $|E| = m$, is represented as T (m, n). The two Borůvka's steps run in $O(m)$ time. The recursive call on G' runs in $T(m_1, n_1)$ time, while the *F-heavy* edges identification runs in $O(m_1)$ time. The second recursive call on G' runs in $T(m_2, n_2)$ time. Finally, the concatenation of edges from the previous Borůvka's steps runs in $O(m)$ time [9]. The number of edges in sub-graph, m_1 and m_2 is bounded by the number of edges m, therefore the overall running time is $O(m)$, which only depends on the number edges in graph G. The probability of this minimum spanning forest algorithm running in $O(m)$ time is $1 - exp(-\Omega(m))$, while in the worst case, where all the edges are added to either the left or right subproblem on each invocation, this randomized algorithm simply becomes Borůvka's algorithm and runs in $O(\min\{n^2, m \log n\})$ time on a graph with n vertices and m edges [8], [9].

3. *Chazelle's Algorithm*

Chazelle's minimum spanning tree algorithm is an efficient deterministic algorithm for finding the minimum spanning tree of a connected, undirected graph. It was first introduced in 2000 and it achieves near-linear time complexity, specifically $O(m\ \alpha(m, n))$, where α is the inverse Ackermann function. The minimum spanning tree problem asks for a spanning acyclic subgraph of the input graph that has the least total cost [10].

3.1 *Edge Corruption*

The previous algorithm highlighted the identification of edges belonging to the minimum spanning tree through the cut property and cycle property. While computing the minimum spanning tree, certain edges of the input graph *G* become corrupted. Edge corruption means that the working cost of an edge is increased by a small amount, typically a constant factor, and can happen more than once. An edge can be corrupted when it is added to a soft heap [10].

Edge corruption can lead to the creation of bad edges, which are edges that become corrupted and then turn bad when their incident subgraph is contracted into a single vertex. Once an edge becomes bad, it can no longer be part of the minimum spanning tree, and this can lead to incorrect results. However, Chazelle's algorithm limits the number of bad edges to within $m/2 + d^3n$, where m is the number of edges in the graph, n is the number of vertices and d is the height of the minimum spanning tree, while the edges that are corrupted but never bad is irrelevant [10].

3.2 *The Soft-Heap*

The soft heap is a simple priority queue used for selecting good candidate edges in the process of constructing a minimum spanning tree of a connected graph. It allows some degree of error in the selection of edges, but at the same time it maintains overall correctness and efficiency, and it supports the operations [11]:

-*create*(*F*): Creates an empty soft heap *F*

-*insert*(*F, x*): Inserts the item *x* to heap *F*

-*delete*(*F, x*): Removes the item *x* from heap *F*

-*findmin*(*F*): Returns the item with the current smallest key in heap *F*

-*meld*(*F, F'*): Creates a new heap containing the union of items stored in *F* and *F'* , destroying *F* and *F'* in the process

A soft heap may, at any time, increase the value of certain keys, making the items of these keys corrupted. A soft heap is parameterized by an error tolerance ε and supports each operation in constant amortized time, except for insert, which takes $O(log(1/\varepsilon))$ time. The data structure never contains more than εn corrupted items at any given time [10].

3.3 *The Algorithm*

The input of this algorithm is a connected, undirected graph $G = (V, E)$, where $|V| = n$ and $|E| = m$, with no self-loops, and each edge e is assigned a distinct real-valued cost $c(e)$. To compute the minimum spanning tree of graph G, the input graph has to be decomposed into vertex-disjoint contractible subgraphs of suitable size. A subgraph C of G is contractible if its intersection with the minimum spanning tree of graph G is connected. The algorithm uses Borůvka steps to decompose G until it becomes a single vertex. This forms a hierarchy of contractible subgraphs, which can be modeled by a perfectly balanced tree F. The leaves of the tree are the vertices of G, while an internal node z and its children are associated with a graph C_z whose vertices are the contractions of the graphs of those children. Each level of the tree F represents a certain minor of the input graph G, and each subgraph C_z is a contractible subgraph of the minor associated with the level of its children. The leaf level of the tree corresponds to G while the root of the tree corresponds to the whole graph G contracted into a single vertex. Because the subgraphs C_z*'s* are contractible, if we compute the minimum spanning tree of each C_z recursively and unite them as one, the minimum spanning tree of graph G will be produced [10].

The algorithm is based on recursion. The Inverse Ackermann function can be used to optimize recursion. Since the Ackermann function grows very rapidly, the Inverse Ackermann function grows very slowly. The Ackermann function *S(i, j)* is defined as follows:

$$S(i,j) = \begin{cases} j+1 & if\ i = 0 \\ S(i-1,1) & if\ j = 0\ and\ i \neq 0 \\ S\big(i-1, S(i,j-1)\big) & otherwise \end{cases}$$

Using the Ackermann function, we can define two different types of inverse functions:

$$a(m,n) = \min\{i \geq 1 \mid S(i,[m/n]) > lg\ n\}$$

$$b(x,y) = min\{z \geq 0 \mid S(x,z) > y\}$$

Note the relationship between these two functions: if $m = n \cdot b(t, lg\ n)$, then $\alpha(m, n) = t$ [12].

Let d_z be the height of an internal node z in the minimum spanning tree F, which is defined as the maximum number of edges from z to a leaf below. Based on the Ackermann function, if $d_z = 1$ then $n_z = S(t, 1)^3 = 8$, where as if $d_z > 1$ then $n_z = S(t\text{-}1, S(t, d_z - 1))^3 = 8$, where $t > 0$ is minimum such that $n \leq S(t, d)^3$, with $d = c[(m/n)^{1/3}]$, for a large enough constant c. The algorithm takes as input an integer $t \geq 1$ and a graph with distinct edge costs and returns that graph's minimum spanning forest. It is now mandated by the recursion invariants that the input graph is not connected [10].

MST – CHAZELLE (G, t)

1. if $t = 1$ or $n = O(1)$
2. return minimum spanning forest of G by direct computation
3. $G_0 = c$ consecutive BORUVKA(G)
4. F = Minimum Spanning Forest of G
5. B = Graph of bad edges
6. $F = \cup_{z \in F}$ minimum spanning forest of $(C_Z \setminus B, t-1)$
7. return minimum spanning forest of $(F \cup B, t) \cup$ {edges contracted in the Borůvka steps

In lines 1 -2, if $t = 1$, the algorithm performs Borůvka steps until G is contracted to a single vertex, while if $n = O(1)$, the algorithm computes the minimum spanning tree. Line 3 simply reduces the number of vertices of the input graph using Borůvka steps, transforming G into a graph G_0, with $n_0 \leq n/2^c$ vertices and $m_0 \leq m$ edges. Lines 4 -5, where $t > 1$, require a bit more analysis. The active path is denoted by the internal nodes $z_1, \ldots, z_k$, while their subgraphs $C_{z1}, \ldots, C_{zk}$ currently under construction form a cost-decreasing chain of edges. Each subgraph includes all the non-discarded edges of G_0 whose endpoints map into it, therefore, to specify a given subgraph it suffices to provide its vertices. At any time, every edge has a working cost, which is its current cost if the edge is bad, and its original cost otherwise. There are three types of cost: original, current, and working cost. There are two invariants that hold at any time with respect to the current graph G_0 [10].

Invariant 1: For all $i < k$, we keep an edge joining C_{zi} to C_{zi+1} whose current cost is at most that of any border edge incident to $C_{z1} \cup \ldots \cup C_{zi}$ and less than the working cost of any edge joining two distinct C_{zj}'s, where $j \leq i$. We call that edge a chain-link. To enforce the latter condition efficiently, for each pair $i < j$ may exist an edge of minimum working cost joining C_{zi} and C_{zj} called a min-link [10].

Invariant 2: For all j, the border edges (u, v) with $u \in C_{zj}$ are stored either in a soft heap $H(j)$, or in a soft heap $H(i, j)$, where $0 \leq i < j$, and no edge appears in more than one heap. An edge stored in $H(j)$ implies that v is incident to at least one edge stored in some $H(i, j)$, while if it is stored in $H(i, j)$ then v is adjacent to C_z but not to any C_{zi} in between $(i < l < j)$. This is also true for $i = 0$, meaning that v is not incident to any C_{zl} where $l < j$. All the soft heaps have error rate $1/c$ [10].

The tree F is built in a postorder traversal driven by a stack whose two operations, pop and push, translate into, respectively, a retraction and an extension of the active path [10].

Retraction happens in $O(k)$ time for $k \geq 2$, when the number of vertices in the last subgraph C_{zk} is $n_{zk} = S(t-1, S(t, d_{zk} - 1))$[3], where d_{zk} is the height z_k in F. The subgraph C_{zk} is contracted and becomes a new vertex that joins C_{zk-1} by the chain-link between C_{zk-1} and C_{zk}. C_{zk-1} gains one vertex and one or several new edges, and the end of the active path is now z_{k-1}. If the heights of z_k and z_{k-1} differ by more than one, a new node is added between them. Every corrupted edge is discarded and the heaps $H(k)$ and $H(k$-

1, k) are destroyed. The remaining items are partitioned into subsets of edges, called clusters, that share the same endpoint outside the chain. For each cluster only the minimum current cost edge (r, s) remains and is inserted into a heap $H(i, k)$ based on the Invariant 2. Along with (r, s), there is at least a second edge pointing to s that is inserted into the heap. After the edges are inserted $H(i, k)$, for each $i < k - 1$, $H(i, k)$ melds into $H(i, k - 1)$ [10].

Extension uses the findmin function in all the heaps to find the extension edge, which is the border edge (u, v) that has the minimum current cost $c(u, v)$. Out of all the min-links of working cost $c(u, v)$ or less, it also finds an edge (a, b) incident to the C_{zi} of smallest index i, and if it exists, the whole subchain $C_{zi+1} \cup ... \cup C_{zk}$ is contracted into a single vertex a, which is called a fusion. The update of all relevant min-links is easily done in $O(k^2)$ time. All corrupted edges are discarded from the $H(i+1)$, …, $H(k)$ and all $H(j, j')$ heaps, where $i \leq j < j'$, since these edges are now bad. The remaining edges are grouped into clusters, and the minimum current cost edge (r, s) is reinserted into a heap $H(i)$ or $H(h, j)$ by the Invariant 2, while the rest of the edges are discarded. For each h, j with $h < i < j$, $H(h, j)$ is melded into $H(h, i)$. Vertex u now belongs to the last C_z in the chain, even if it didn't originally belong there. The chain is extended by making v into the single-vertex C_{zk} and the extension edge (u, v) into the chain-link incident to it, regardless of whether a fusion took place or not. The active path now ends at the new z_k, since without fusion, the new value of k is the old one plus one, while with fusion, it is $i + 1$. The old border edges incident to v are deleted from their respective heaps. The min-link between v and each of C_{z1}, …, C_{zk-1} is inserted incident to v into the appropriate $H(i, k)$. If multiple edges exist, all the edges but the cheapest in each group are discarded [10].

The construction of F is based on retractions and extensions. At any given node z_k of height at least 1, extensions are performed until the size condition for retraction at that node has been met. The algorithm stops only when border edges run out and extensions are no longer possible. During this process, the algorithm keeps track of the bad edges in order to form the graph of bad edges B [10].

In line 6, after the tree F has been built, the cost f each edge is reset to its original value, the parameter t is decremented by one and for each node z the algorithm is applied recursively to $C_z \setminus B$, which is the subgraph C_z with no bad edges. The output tree F is a set of edges joining vertices in C_z, which are treated as edges of G_0 in line 7. Line 7 creates the subgraph $F \cup B$ of graph G_0 by adding the edges of F to graph B [10].

The correctness of the algorithm is proved by induction on t and n, where we know that $t > 1$ and n is larger than a suitable constant. The Boruůvka steps contract only minimum spanning tree edges, therefore, by induction on the correctness of the minimum spanning forest, the output of the algorithm is in fact the minimum spanning forest of the input graph G, if and only if every edge e of G_0 outside of $F \cup B$ is also outside of the minimum spanning forest of G_0 [10].

Lemma 3.1. *Consider the subgraph C_z at the time of its contraction. With respect to working costs, C_z is strongly contractible and the same holds of every fusion edge (a, b)* [10].

Proof. A subgraph C_z of graph G_0 is strongly contractible if for every pair of edges e, f in G_0, each with exactly one vertex in C_z, there exists a path in C_z that connects e to f along which no edge exceeds the cost of both e and f. The lemma refers only to the edges present in C_z and in its neighborhood at the time of its contraction. The subgraph C_z is formed by incrementally adding vertices via retractions. Some neighboring edges might be added by fusion into some $a \in C_z$. C_z grows monotonically because it contains no border edges, so edge discarding never takes place within it. There are two cases, either fusion occurs, or it does not [10].

If no fusion occurs, each retraction has a unique chain-link associated with it, and together they form a spanning tree of C_z. The tree has a unique path π that joins any two edges e, f that have exactly one endpoint in C_z. If edge $g = (u,v)$ has the highest current cost of all edges of π and is the last edge selected for extension chronologically, and vertex v is the endpoint outside the chain at the time of extension, then vertex u lies between v and one of the two edges. If e is that edge, then the working cost of e must be at least the current cost of g. The term "working" refers to the time right after C_z is contracted, while "current" refers to the time when g is selected. The proof is based on Invariant 1. If e currently joins C_z to some other $C_{z'}$, then it is the second case of Invariant 1. If not, then let an edge e' be the first border edge encountered along the path π from g to e. By the first case of Invariant 1, e''s current cost is at least that of g, which means that e' does not belong to the path π, so $e = e'$. Because e is a border edge when C_z is contracted, if e is in a corrupted state, then it becomes bad after C_z's contraction and its working cost is at least its current cost, otherwise both costs coincide with the original one [10].

If a fusion occurs into C_z, it is dealt the same way as the no-fusion case until the contraction of (a, b) into a happens. Now a number of edges are contracted within C_z, which keeps C_z strongly contractible. Before the contraction of (a, b) into a, the min-link (a, b) is fused into C_z. The edge (a, b) is the cheapest edge incident to b in terms of working costs, since all corrupted border edges incident to b become bad, and so their working and current costs agree. This means that the edge (a, b) is strongly contractible [10].

Lemma 3.2. *If an edge of G_0 is not bad and lies outside F, then it lies outside of the minimum spanning tree of G_0* [10].

Proof. G_0 is transformed into a single vertex due to the contractions of minors $S_1, S_2, \ldots$, with each minor either being of the form C_z or consisting multiple edges of some fusion edge (a, b). Let G_0^* be the graph G_0 without the edged that were discarded during the construction of tree F, with G_0^* spanning all the vertices of G_0. Because of the Lemma 3.1, the working costs of all edges within C_z or with one endpoint in C_z are frozen. For a minor S_i, a vertex of S_i is either a vertex of G_0 or the contraction of some

S_j ($j < i$). The same applies to the minor S_j its vertices are either vertices of G_0 or contractions of S_k ($k < j$). By repeated applications of Lemma 3.2, the minimum spanning tree of G_0^* is the union of all the minimum spanning trees S_i's, which is called the composition property. There are two cases, either the edge e under consideration belongs to G_0^*, or it does not [10].

If edge e belongs to G_0^*, since e is not in F, then it does not belong in the minimum spanning tree of $S_i \setminus B$, where S_i is the unique minor that contains both endpoints of e among its vertices. Also, since e is not a bad edge it means that it outside of the minimum spanning tree of S_i, and by the composition property, outside of the minimum spanning tree of G_0^*. If we switch all edge costs to their original values, any changes that occur can only be downward. Since edge e is not bad, its value does not change and, with respect to original costs, it still is the most expensive edge on a cycle of G_0^*. This proves that the edge e is not in the minimum spanning tree of G_0^*, and thus it is not in the minimum spanning tree of G_0 [10].

If edge e does not belong to G_0^*, it is discarded. Before being discarded, $e = (u, v)$ shared a common endpoint v with a cheaper edge $e' = (u', v)$. Either through retraction or while in a fusion, u and u' end up in a vertex which, by repeated applications of Lemma 3.2, is the contraction of a contractible subgraph of G_0^*relative to working costs. Since e and e' are not corrupted and the former is more expensive than the latter, it proves that e is outside of the minimum spanning tree of G_0 [10].

3.4 *Analysis*

In lines 1 -2, if $t = 1$, the problem is solved in $O(n^2)$ time by performing Boruvka steps until G is contracted to a single vertex. If $n = O(1)$, the algorithm computes the minimum spanning tree in $O(m)$ linear time. The Boruvka steps in line 3, where graph G is transformed into graph G_0, with $n_0 \leq n/2^c$ vertices and $m_0 \leq m$ edges, requires $O(n + m)$ time. Lines 4 – 5 are dominated by the heap operations. There are $O(m_0)$ inserts, which means there are $O(m_0)$ deletes and melds. There are also n_0 - 1 edge extensions, each of them calling up to $O(d^2)$ findmins. Since each heap operation takes constant time, computing F takes $O(m_0 + d^2 n_0)$ time plus the bookkeeping costs of accessing the right heaps at the right time which also take $O(m_0 + d^2 n_0)$ time. Let b be a constant large enough, but arbitrarily smaller than c, then the running time of lines 4 – 5 is upper bounded by $b/2(n + m + d^2 n_0)$. In line 6 the overall recursion time is upper bounded by $b(t - 1)(m_0 - |B| + dn_0)$. Finally, line 7 recurses with respect to the graph $F \cup B$ and is upper bounded by $bt(n_0 - 1 + |B| + d^3(n_0 - 1))$. Adding up all these costs and taking into account that $n_0 \leq n/2^c$ we find that the time complexity of the algorithm is upper bounded by $bt(m + d^3(n - 1))$. The running time of the algorithm when it computes the minimum spanning tree of a connected graph G with n vertices and m edges, $d^3 n = O(m)$ and $t = O(\alpha(m, n))$ by the Ackermann function, is $O(m\alpha(m,n))$ [10].

REFERENCES

[1] C. E. L. R. L. R. C. S. Thomas H. Cormen, Introduction to Algorithms Third Edition, 2009.

[2] Wolfgang Mulzer, "The KKT-Algorithm for Minimum Spanning Trees".

[3] Ming-Yang Kao, Encyclopedia of Algorithms, 2008.

[4] P. N. K. R. E. T. David R. Karger, "A Randomized Linear-Time Algorithm To Find Minimum Spanning Tree," *Journal of the Aswcl.tmn for Computmg Machinery,* pp. 321 - 328, 2 03 1995.

[5] Anupam Gupta, "Advanced Algorithms: Deterministic MSTs," 2018.

[6] "Wikipedia: The Free Encyclopedia," [Online]. Available: https://en.wikipedia.org/wiki/Minimum_spanning_tree. [Accessed 27 12 2023].

[7] V. R. Seth Pettie, "Randomized Minimum Spanning Tree Algorithms Using Exponentially Fewer Random Bits".*ACM Journal Name.*

[8] "Wikipedia: The Free Encyclopedia," [Online]. Available: https://en.wikipedia.org/wiki/Expected_linear_time_MST_algorithm. [Accessed 27 12 2023].

[9] Yvette (I-Ting) Tsai, "A Randomized Algorithm to Find Minimum Spanning Tree," 2013.

[10] Bernard Chazelle, "A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity," *Journal of the ACM Vol. 47,* pp. 1028-1047, 6 11 2000.

[11] V. R. Seth Pettie, "An optimal minimum spanning tree algortihm," 2000.

[12] D. M. R. E. T. S. X. Sarah Eisenstat, "Minimum Spanning Tree in Deterministic Linear Time For Graphs of High Girth".