Lecture notes for "Analysis of Algorithms": **Minimum Spanning Trees**

Lecturer: *Uri Zwick* *

May 22, 2013

Abstract

We present a general framework for obtaining efficient algorithms for computing minimum spanning trees. We use this framework to derive the classical algorithms of Prim, Kruskal and Borůvka. We then describe the randomized linear-time algorithm of Karger, Klein and Tarjan. The algorithm of Karger, Klein and Tarjan uses deterministic linear-time implementations of a *verification* algorithm of Komlós.

1 Minimum Spanning Trees

Let $G = (V, E, w)$ be a weighted undirected graph, where $w : E \to \mathbb{R}$ is a weight (or cost) function defined on its edges. A subgraph $T = (V, E_T)$ of G which is a tree is said to be a *spanning tree* of G. The *weight* of a spanning tree $T = (V, E_T)$ of G is defined to be $w(T) = \sum_{e \in E_T} w(e)$. In the *Minimum Spanning Tree* (MST) problem we are asked to find a spanning tree of minimum weight of a given connected input graph $G = (V, E)$.

2 Preliminaries

Definition 2.1 (Cuts) *A* cut $(S, \bar{S})$ *of a graph* $G = (V, E)$ *is a partition of the vertex set into two nonempty parts* $S \subseteq V$ *and* $\bar{S} = V - S$. *An edge* $e \in E$ crosses *the cut if and only if* $|e \cap S| = 1$, *i.e., if one endpoint of* e *is in* S *and the other in* $\bar{S}$. *We let* $E(S, \bar{S}) = \{e \in E \mid |e \cap S| = 1\}$ *be the set of edges that cross the cut* $(S, \bar{S})$ *and refer to the as the edges of the cut.*

Definition 2.2 (Cycles) *A (simple)* cycle C *in a graph* $G = (V, E)$ *is a sequence of distinct vertices* $C = \langle v_1, v_2, \ldots, v_k \rangle$ *such that* $k \geq 3$ *and* $E(C) = \{(v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, v_1)\} \subseteq E$.

*School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E–mail: zwick@tau.ac.il

The following lemma, whose proof is immediate, states that the intersection between any cut and any cycle contains an even number of edges.

Lemma 2.3 *Let $G = (V, E)$ be an undirected graph. If $(S, \bar{S})$ is a cut of G and C is a cycle in G then $|E(C) \cap E(S, \bar{S})|$ is even.*

3 A greedy framework for finding MSTs

Following Tarjan [Tar83], we describe a general greedy framework that can be used to derive essentially all known MST algorithms.

Let $G = (V, E, w)$ be a weighted undirected graph. We assume in this section that G is *connected*. Initially, all the edges of G are *uncolored*. We describe below two rules that can be used to color certain edges of the graph either *blue* or *red*. Any sequence of applications of these rules would maintain the following invariant:

> **Invariant:** There is a minimum spanning tree of G that contains all blue edges and none of the red edges.

We shall see below that as long as there are still uncolored edges in the graph, at least one new edge can be colored using one of the rules. When all the edges are colored, the invariant says that the set of blue edges forms a minimum spanning tree of G.

The two coloring rules are:

> **Blue rule (cut rule):** Select a *cut* that contains no blue edges. Among the uncolored edges of the cut, select one of minimum weight and color it *blue*.

> **Red rule (cycle rule):** Select a simple *cycle* containing no red edges. Among the uncolored edges on the cycle, select one of maximum weight and color it *red*.

Theorem 3.1 *Any coloring obtained by repeated applications of the cut and cycle rules satisfies the invariant above. If some edges are uncolored, then at least one of the two rules can be applied.*

Proof: We start by proving that any coloring obtained by applying the two rules satisfies the invariant. We do that by induction on the number of applications. Suppose that c is a coloring that satisfies the invariant, and that c' is a coloring obtained from c by coloring an edge e blue using the cut rule, or red using the cycle rule. Let B be the set of edges colored blue by c, and let R be the set of edges colored red by c. By the induction hypothesis, there exists a minimum spanning tree T of G such that $B \subseteq T$ and $T \cap R = \phi$.

Case 1: e is colored blue. If $e \in T$, we are done. Suppose, therefore, that $e \notin T$. Let $(S, \bar{S})$ be the cut to which the cut rule was applied to justify coloring e blue. Then, before the application

of the rule none of the edges in the cut $(S, \bar{S})$ are blue, and among the uncolored edges of the cut e has minimum weight. Consider $T \cup \{e\}$. Adding e to T closes a cycle. This cycle must contain at least one edge $e' \neq e$ that also crosses the cut $(S, \bar{S})$. This edge e' must be uncolored. (It cannot be colored blue, as $(S, \bar{S})$ contains no blue edges, and it cannot be colored red, as it belongs to T.) We also have $w(e) \leq w(e')$. Now, $T' = T \cup \{e\} - \{e'\}$ is a spanning tree of T and $w(T') = w(T) + w(e) - w(e') \leq w(T)$. Thus, T' is also a minimum spanning tree. (We also have $w(e) = w(e')$.) Now, $B \cup \{e\} \subseteq T'$ and $T' \cap R = \phi$, as required.

Case 2: e is colored red. If $e \notin T$, we are done. Suppose, therefore, that $e \in T$. Let C be the cycle to which the cycle rule was applied to justify coloring e red. Let S and $\bar{S}$ be the two connected components of $T - \{e\}$. By Lemma 2.3, the cut $(S, \bar{S})$ must contain at least one other edge $e' \neq e$ that also belongs to C. The edge e' is not colored red and $w(e') \leq w(e)$. Now $T' = T - \{e\} \cup \{e'\}$ is again a spanning tree. As $w(T') \leq w(T)$, it is a minimum spanning tree. (It follows, in fact, that $w(e) = w(e')$.) Now, $B \subseteq T'$ and $T' \cap (R \cup \{e\}) = \phi$, as required.

Finally, suppose that at least one edge of G is still uncolored. The set B of blue edges must form a forest. (By the invariant, there is a tree T that contains B, and hence B cannot contain a cycle.) If this blue forest is not yet a spanning tree, then consider the cut defined by one of the blue trees in this forest. This cut contains no blue edges and not all edges in the cut can be colored red. The cut rule can therefore applied to this cut. If the blue edges form a spanning tree, then any remaining uncolored edge e closes a cycle all whose other edges are colored blue. By applying the cycle rule to this cycle we can color e red. □

Coloring an edge blue selects it for inclusion in the constructed minimum spanning trees. If at some stage the set of blue edges forms a spanning tree of the graph then, by the invariant, this spanning tree must be a *minimum* spanning tree. Coloring an edge red is equivalent to removing it from the graph, and hence not including it in the constructed minimum spanning tree.

4 Three classical algorithms

In this section we describe three classical MST algorithms. Interestingly, they are all rely solely on the blue rule. The linear expected time MST algorithm that we present in Section 7, on the other hand, relies on both the blue and red rules.

4.1 Kruskal's algorithm

Kruskal's algorithm [Kru56] is very simple.

> **Kruskal's algorithm:** Let $e_1, e_2, \ldots, e_m$ be the edges of a connected graph $G = (V, E)$ sorted according to weight, i.e., $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$. For $i = 1, 2, \ldots, m$, color e_i blue if the two endpoints of e_i are not in the same blue tree.

Theorem 4.1 *Kruskal's algorithm finds a minimum spanning tree*

Proof: We prove the correctness of the algorithm in two steps. We first show that all the colorings done by the algorithm can be justified by applications of the blue rule. We then show that at the end of the algorithm the blue edges form a tree, which must then be a minimum spanning tree.

Suppose that e_i is colored blue by the algorithm. Let S be the set of vertices in one of two blue trees that e_i touches. Then, e_i belongs to the cut $(S, \bar{S})$. The cut contains no other blue edges. We next claim that e_i is an edge of minimum weight in this cut, hence coloring e_i blue is justified. Suppose, for the sake of contradiction, that the cut $(S, \bar{S})$ also contains an edge e_j such that $w(e_j) < w(e_i)$. As the edges are sorted according to weight, we get that $j < i$, and hence e_j is examined by the algorithm before e_i is. As the two endpoints of e_j are not in the same blue tree now, they were certainly not in the same blue tree when the edge e_j was examined, and hence e_j should have been colored blue, a contradiction.

Next suppose, again for the sake of contradiction, that after all edges were examined, the set of blue edges is not a spanning tree. Let S be the set of vertices in one of the trees in the blue forest. As the graph is connected, the cut $(S, \bar{S})$ is non-empty and all its edges are uncolored. Let e_i be an edge of minimum weight in this cut. Then, e_i should have been colored blue when it was examined, a contradiction. □

We next consider the implementation of Kruskal's algorithm. The edge weights can be sorted in $O(m \log n)$ time. (If the edge weights are small integers, or if the word RAM model of computation is assumed, then a faster sorting time can be obtained.)

That leaves us with the problem of determining, for every edge e_i, whether its two endpoints belong to the same blue tree or not. This can be easily done using a data structure for maintains a collection of disjoint sets that supports the following three operations:

> *makeset*(x): Create a new set containing the single element x, previously in no set.
>
> *find*(x): Return an identifier of the set currently containing element x.
>
> *union*(x, y): Form a new set that is the union of the two sets currently containing elements x and y, destroying the two old sets.

Using such a union-find data structure, Kruskal's algorithm can be easily implemented as shown in Figure 1.

A simple implementation of a union-find data structure that uses two simple heuristics, union by rank, and path compression, was shown by Tarjan [Tar75] to require only $O(\alpha(m, n))$ amortized time per operation, where $\alpha(m, n)$ is an extremely slowly growing functional inverse of Ackermann's function. In particular $\alpha(m, n) \ll \log n$. Unfortunately, the running time of the mandane sorting stage of Kruskal's algorithm dominates the running time of the more interesting second stage, and the total running time of Kruskal's algorithm, when the edge weights cannot be quickly sorted, is $O(m \log n)$.

```
Function Kruskal(G = (V, E, w))
    Sort the edges so that w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m)
    foreach v ∈ V do
        makeset(v)
    T ← ∅
    for i ← 1 to m do
        (u, v) ← e_i
        if find(u) ≠ find(v) then
            T ← T ∪ {e_i}
            union(u, v)
```

Figure 1: Kruskal's algorithm

4.2 Prims's algorithm

Kruskal's algorithm starts with a blue forest in which each vertex forms a tree. In each iteration, two blue trees are joined by a blue edge and merge. When the blue forest is a tree, it is a minimum spanning tree. Prim's algorithm [Pri57], on the other hand, grows a single blue tree. In each iteration one new vertex is added to the tree. When all vertices are added to the tree, the tree is, of course, a minimum spanning tree.

Prim's algorithm starts with some vertex $s \in V$. In each iteration it finds a lightest edge connecting a vertex in the blue tree to a vertex not in the blue tree and adds it to the tree. (In other words, it applies the blue rule to the cut defined by the blue tree.)

Prim's algorithm can be efficiently implemented using a *priority queue*. Such an implementation is given in Figure 2. The priority-queue P maintained by the algorithm contains all vertices not yet added to the tree. The key $d[v]$ of a vertex v in P is the weight of the lightest edge joining v to the tree. If no edge connecting v to the tree was found yet, then $d[v] = \infty$. If $d[v] < \infty$, then $(p[v], v)$ is an edge of weight $d[v]$.

In each iteration the algorithm extracts a vertex u with minimum key from P. It adds the edge $(p[u], u)$ to the tree. It then examines all the edges incident to u. For each such edge (u, v) such that v is in P it checks whether $w[u, v] < d[v]$. If so, a lighter edge connecting v to the tree was found, $d[v]$ and $p[v]$ are updated, and the key of v in P is decreased to $d[v]$.

Theorem 4.2 *Prim's algorithm finds a minimum spanning tree. It running time is* $O(m+n \log n)$.

Proof: Correctness follows immediately from the blue rule. To obtain a running time of $O(m + n \log n)$ we use a priority-queue data structure that supports `decrease-key` operations in $O(1)$ amortized time, and all other operations in $O(\log n)$ amortized time, such as Fibonacci heaps [FT87]. □

```
Function Prim(G = (V, E, w), s)
    P ← priority-queue()
    foreach v ∈ V \ {s} do
        p[v] ← null
        d[v] ← ∞
        insert(P, v, d[v])
    d[s] ← 0
    insert(Q, s, d[s])
    T ← ∅
    while P ≠ ∅ do
        u ← extract-min(P)
        T ← T ∪ {(p[u], u)}
        foreach (u, v) ∈ E do
            if v ∈ P and w[u, v] < d[v] then
                p[v] ← u
                d[v] ← w[u, v]
                decrease-key(P, v, d[v])
```

Figure 2: Prim's algorithm

4.3 Borůvka's algorithm

In this section, and several subsequent sections, we assume that all edge weights are distinct. This is not a serious restriction as any comparison-based algorithm for finding minimum spanning trees that works under this assumption can be easily converted into an algorithm that allows ties by consistently breaking ties, if they occur. (See Exercise 2.) It is also not difficult to show that when all edge weights are distinct, the minimum spanning forest is unique. (See Exercise 1.)

Borůvka's algorithm [Bor26] is also easy to describe:

> **Borůvka's algorithm:** Repeat the following until the set of blue edges forms a tree: For each blue tree, in parallel, select the lightest edge leaving it and color it blue.

Borůvka's algorithm is in fact a *parallel* algorithm. (We are only interested here, however, in sequential implementations of it.) Borůvka's algorithm applies the blue rule simultaneously to several cuts. This needs to be justified, as the blue and red rules, as presented in Section 3, were meant to be used sequentially. Actually, Borůvka's algorithm is correct only if all edge weights are distinct, which is why we made this assumption at the beginning of the section.

Theorem 4.3 *If all edge weights are distinct, then Borůvka's algorithm finds a minimum spanning tree.*

Lemma 4.4 *Each iteration of Borůvka's algorithm reduces the number of blue trees by a factor of at least 2.*

Theorem 4.5 *Borůvka's algorithm can be implemented to run in $O(m \log n)$ time.*

In the next section we consider the implementation of MST algorithms using *contraction*. This leads, in particular, to two additional implementation of Borůvka's algorithm, one running in $O(n^2)$ time, and another running in $O(m \log(n^2/m))$.

5 Finding Minimum spanning trees using contractions

We begin with a formal definition of *contraction*.

Definition 5.1 (Contracting an edge) *Let $G = (V, E)$ be an undirected graph and let $e = (u, v) \in E$. The graph G/e is obtained from G by merging the two endpoints of e into a single vertex. The vertex set of G/e is $V \setminus \{u, v\} \cup \{uv\}$, where uv is the vertex obtained by merging u and v. The edge set of G/e is $E \setminus \{e\}$. Edges that touched u and v are now considered to touch the new vertex uv.*

Definition 5.2 (Contracting a set of edges) *Let $G = (V, E)$ be an undirected graph and let $B \subseteq E$. The graph G/B is obtained from G by contracting all the edges of B. (It is not difficult to check that the order of contractions does not matter.) Alternatively, G/B is a graph whose vertex set are the connected components of the subgraph (V, B). The edges of G/B are edges that connect different connected components of (V, B).*

Contraction is an important operation that would be used extensively throughout the course. There are two approaches to implementing graph algorithms using contractions. The first is to carry out the contractions *explicitly*. In an explicit representation of a contracted graph, the adjacency lists of vertices that were merged together are catenated, and endpoint of edges are updated. This can be expensive in general. An alternative approach is to maintain a contracted graph *implicitly*. This usually involves the use of a union-find data structure. In fact, the implementation of Kruskal's algorithm given in Section 4.1 may be viewed as an implementation using implicit contractions.

The greedy approach of Section 3 and the contraction framework can be easily combined to obtain the following theorem:

Theorem 5.3 *Let $G = (V, E, w)$ be a weighted undirected graph and let $B \subseteq E$ be the set of edges colored blue and let $R \subseteq E$ be the set of edges colored red using a sequence of applications of the red and cut rules. Let T' be a minimum spanning tree of $(G - R)/B$. Then, $B \cup T'$ is a minimum spanning tree of G.*

We mention, for the last time, the slight abuse of notation used when referring to the set $B \cup T'$ above. Here T' is used both as a set of edges, forming a minimum spanning tree, in $(G - R)/B$, and also as the set of edges in the original graph.

6 Minimum spanning tree verification

In this short section we consider the easier problem of checking whether a given spanning tree T is a minimum spanning tree of a weighted graph $G = (V, E, w)$. We begin with the following useful definition.

Definition 6.1 ($w_T(e)$) *Let $G = (V, E, w)$ be a weighted undirected graph and let T be a spanning tree of G. For an edge $e = (u, v) \in E$ let $w_T(e)$ be the weight of the heaviest edge on the unique path in T connecting u and v. (Note that if $e \in T$, then $w_T(e) = w(e)$.)*

The following lemma follows easily from the cycle rule:

Lemma 6.2 *A spanning tree T of $G = (V, E, w)$ is a minimum spanning tree if and only if for every $e \notin T$ we have $w(e) \geq w_T(e)$.*

Komlós [Kom85], Dixon, Rauch and Tarjan [DRT92], King [Kin97], and Hagerup [Hag09] proved the following theorem:

Theorem 6.3 *There is a deterministic linear time algorithm that given a weighted undirected graph $G = (V, E, w)$ and a spanning tree T of G computes $w_T(e)$ for every $e \in E$.*

Corollary 6.4 *There is a deterministic linear time algorithm that given a weighted undirected graph $G = (V, E)$ and a spanning tree G checks whether T is a minimum spanning tree of G.*

7 A randomized linear time algorithm

In this section we describe a *randomized* minimum spanning tree algorithm of Karger, Klein and Tarjan [KKT95] that has an expected *linear* running time. The algorithm uses the linear time minimum spanning tree verification algorithm mentioned in the previous section. We assume again in this section that all edge weights are distinct.

The randomized algorithm is recursive. The graphs on which the recursive calls are applied may not be connected, even if the original graph is. We therefore describe the algorithm as an algorithm for finding a *minimum spanning forest* (MSF) of a not necessarily connected graph. A spanning forest of a graph $G = (V, E)$ is a subgraph composed of a spanning tree of each connected component of G.

A minimum spanning forest is a spanning forest of minimum total weight. The greedy framework of Section 3 can clearly be used for finding a minimum spanning forest.

A *subforest* of $G = (V, E)$ is a subgraph of G which is a forest, i.e., contains no cycle. A subforest is not required to be spanning. Let $G = (V, E, w)$ be a weighted undirected graph and let F be a subforest of G. Extending the notation of the previous section, we define $w_F(e)$, for $e \in E$, to be the weight of the heaviest edge on the unique path in F between e's endpoints, if the endpoints of e are in the same tree of F, and $w_F(e) = +\infty$, otherwise.

Definition 7.1 (F-heavy and F-light edges) *Let $G = (V, E, w)$ be a weighted undirected graph and let F be a subforest of G. An edge $e \in E$ is said to be* F-heavy *if and only if $w(e) > w_F(e)$, and F-light otherwise. (In particular, if e is in F or connects two different trees of F, then e is F-light.)*

Lemma 7.2 *Let $G = (V, E, w)$ be a weighted undirected graph and let F be an arbitrary subforest of G. Then, there the minimum spanning tree of G does not include any F-heavy edges.*

Proof: All F-heavy edges can be colored red using the red rule. □

The algorithm of Karger, Klein and Tarjan [KKT95] relies on the following *sampling lemma*.

Lemma 7.3 *Let $G = (V, E, w)$ be a weighted undirected graph and let $0 < p < 1$. Let $G' = (V, E', w)$ be a random subgraph of G obtained by choosing every edge, independently, with probability p. Let F be a minimum spanning forest of G'. Then, the* expected *number of edges of G that are F-light is at most $\frac{n}{p}$, where $n = |V|$.*

An interesting thing to note in the statement of the lemma is that the bound on the expected number of F-light edges depends only on n, the number of vertices, and not on m, the number of edges in the original graph.

Proof: Algorithm `SamplingLemmaProof` of Figure 3 generates a random subset of edges $E' \subseteq E$ to which each edge is added independently with probability p. It uses Kruskal's algorithm to construct a minimum spanning forest F of the subgraph $G' = (V, E')$. In the process it also constructs the set $L \subseteq E$ of edges of $G = (V, E)$ that are F-light. Note that algorithm `SamplingLemmaProof` is only used in the proof of the Lemma. We are not interested here in efficient implementations of it.

Algorithm `SamplingLemmaProof` starts by sorting the edges of G in increasing order of weight. It then simultaneously chooses the subset E', constructs the minimum spanning forest of $G' = (V, E')$, and constructs the set L of F-light edges. Initially, E', F and L are empty. As in Kruskal's algorithm, the algorithm examines the edges in increasing order of weight. Each edge e_i is added to E' with probability p. If e_i connects two different trees of F, then e_i is an F-light edge and is added to L. Note that this is true whether or not e_i was added to E'. At the time of its examination, e_i connects two different trees in F. The weight of e is greater than the weight of all edges currently in F. If the two trees e_i is currently connecting would not merge into a single tree in the final minimum

```
Function SamplingLemmaProof(G = (V, E, w))
    Sort the edges so that w(e_1) ≤ w(e_2) < ··· < w(e_m)
    E', F, L ← φ
    for i ← 1 to m do
        With prob. p: E' ← E' ∪ {e_i}
        if e_i connects two different trees of F then
            L ← L ∪ {e_i}
            if e_i ∈ E' then F ← F ∪ {e_i}
```

Figure 3: Algorithm for generating a random subgraph, its minimum spanning forest F, and the set of original edges that are F-light. Used in the proof of Lemma 7.3

spanning forest F, then e_i would of course be an F-light edge. If these two trees merge as the result of adding e_i to E' and hence to F, then e_i would be a forest edge, and thus F-light. Finally, if e_i is not added to F but two trees eventually merge, then the edge joining them would be heavier than e_i, thus e_i would again be an F-light edge. Also note that if the two endpoints of e_i are in the same tree of F when e_i is examined, then e_i is F-heavy, as all edges currently in F are lighter than e_i.

As `SamplingLemmaProof` is randomized, $|L|$ and $|F|$ are random variables. Whenever an edge e is added to L, it is also added to F with probability p. Thus $\mathbb{E}[|F|] = p\mathbb{E}[|L|]$. However, as F is a spanning forest of $G' = (V, E')$, we get that $|F| \leq n - 1$ and clearly $\mathbb{E}[|F|] \leq n - 1$. Thus $\mathbb{E}[|L|] \leq \frac{n-1}{p} \leq \frac{n}{p}$, as claimed. □

For an alternative proof, due to Chan [Cha98], of a version of the sampling lemma in which the random edge set E' contains exactly $p|E|$ edges, see Exercise 9.

We are now ready to describe the algorithm of Karger, Klein and Tarjan [KKT95]. The algorithm, described in Figure 4 works as follows. If the edge set E of the input graph is empty, then the minimum spanning forest is also empty. Otherwise, the algorithm starts by performing two steps of Borůvka algorithm. We assume here that these steps contract the graph and remove isolated vertices. We let F' and F'' be the edge sets chosen for inclusion in the minimum spanning forest during these two steps, and let G'' be the graph returned. The algorithm now chooses a random subgraph $G_1 = (V, E_1)$ of G'' to which each edge is added, independently, with probability 1/2. A first recursive call is then made on the random subgraph G_1 to find the minimum spanning forest F_1 of G_1. Next, a linear time algorithm is used to find the set of edges E_2 of G'' which are F_1-light. All other edges may be removed using the cycle rule. Finally a second recursive call is made on the subgraph $G_2 = (V, E_2)$ of G'' to find a minimum spanning forest F_2 of G_2. It then follows easily that $F' \cup F'' \cup F_2$ is the minimum spanning forest of the input graph $G = (V, E)$.

Theorem 7.4 *Algorithm* `RandMSF` *returns a minimum spanning forest of the input graph. Its*

Function RandMSF($G = (V, E, w)$)

if $E = \phi$ **then return** ϕ

$(G', F') \leftarrow$ `BoruvkaStep`(G)
$(G'', F'') \leftarrow$ `BoruvkaStep`(G)

$G_1 \leftarrow$ `RandSubgraph`$(G'', \frac{1}{2})$
$F_1 \leftarrow$ `RandMSF`(G_1)

$E_2 \leftarrow$ `LightEdges`(G'', F_1)
$G_2 \leftarrow (V[G''], E_2, w)$
$F_2 \leftarrow$ `RandMSF`(G_2)

return $F' \cup F'' \cup F_2$

Figure 4: A randomized linear time algorithm for finding minimum spanning forests.

expected running time is $O(m + n)$.

Proof: The correctness of the algorithm follows easily from the cut and cycle rules. All edges of F' and F'' chosen by the two Borůvka steps can be colored blue by the cut rule and hence belong to the minimum spanning forest. All F_1-heavy edges, removed from G_2 before the second recursive call, can be colored red by the cycle rule and hence do not belong to the minimum spanning forest. The rest of the proof follows by induction.

It remains to analyze the expected running time of the algorithm. Let $G = (V, E)$ be a (random) graph. We let $m = |E|$ and $n = |V|$. As G is random, m and n are random variables. We let $\bar{m} = \mathbb{E}[m]$ and $\bar{n} = \mathbb{E}[n]$. Let $G_1 = (V_1, E_1)$ and $G_2 = (E_2, V_2)$ be the two (random) graphs on which recursive calls are made when `RandMSF` is applied to G. We let $m_i = |E_i|$, $n_i = |V_i|$, and $\bar{m}_i = \mathbb{E}[m_i]$, $\bar{n}_i = \mathbb{E}[n_i]$, for $i = 1, 2$.

As G_1 and G_2 are both subgraphs of the graph G'' obtained by performing two Borůvka steps on G, we get that $n_1, n_2 \leq n/4$, and therefore $\bar{n}_1, \bar{n}_2 \leq \bar{n}/4$. As G_1 is obtained by choosing each edge of G'', independently, with probability $1/2$, we get that $\bar{m}_1 \leq \bar{m}/2$. Also note that we always have $m_1 < m$, as some edges were contracted by the two Borůvka steps. Finally by the sampling lemma (Lemma 7.3) we get that $\bar{m}_2 \leq (\bar{n}/4)/(1/2) = \bar{n}/2$.

Let $T(G)$ be the running time of the algorithm on a (random) graph G. Then

$$T(G) \;=\; a(m + n) + T(G_1) + T(G_2)\ ,$$

where $a > 1$ is a large enough constant so that $a(m + n)$ covers the linear cost of the two Borůvka steps and the call `LightEdges`(G'', F_1). We prove by induction on M and N that if G is a random graph with at most M edges and at most N edges, then

$$\mathbb{E}[T(G)] \;\leq\; 2a(\bar{m} + 2\bar{n})\ .$$

If the two Borůvka steps always find a minimum spanning forest of G, the claim is obvious. This forms the basis of the induction. Suppose now that the claim holds for all values smaller than M and N and let G be a (random) graph with at most M edges and N vertices. As the induction hypothesis can be applies to G_1 and G_2, we get that,

$$\begin{aligned}
\mathbb{E}[T(G)] &= a(\bar{m}+\bar{n}) + \mathbb{E}[T(G_1)] + \mathbb{E}[T(G_2)] \\
&\leq a(m+n) + 2a(\bar{m}_1 + 2\bar{n}_1] + 2a(\bar{m}_2 + 2\bar{n}_2] \\
&\leq a(m+n) + 2a(\tfrac{\bar{m}}{2} + 2\cdot\tfrac{\bar{n}}{4}) + 2a(\tfrac{\bar{n}}{2} + 2\cdot\tfrac{\bar{n}}{4}) \\
&= 2a(\bar{m}+2\bar{n}) \ .
\end{aligned}$$

□

Exercises

Exercise 1 Let $G = (V, E, w)$ be a connected weighted graph. (a) The graph G may have many minimum spanning trees. Show, however, that all these minimum spanning trees have the same (multi-)set of edge weights. (b) Show that if all the edge weights in G are distinct, then the minimum spanning tree is unique.

Exercise 2 Let ALG be an algorithm for computing minimum spanning trees that is guaranteed to work only if all edges weights in the input graph are distinct. Suppose that the only operation performed by ALG on edge weights are *comparisons*. Show how to convert ALG into an algorithm ALG′, with the same asymptotic running time, that works even if some of the edge weights are equal.

Exercise 3 (c) Show that a minimum spanning tree of G is also a spanning tree of G whose maximal edge weight is minimal.

Exercise 4 Describe a *deterministic* linear time algorithm for finding a spanning tree whose maximal weight is minimal. (Hint: Start by computing the median of the edge weights.)

Exercise 5 Describe a simple implementation of `Boruvka`'s algorithm that runs in $O(n^2)$ time, and another simple implementation that runs in $O(m \log n)$ time. Show next that these two implementations can be combined to yield an implementation whose complexity is $O(m \log \frac{n^2}{m})$ time.

Exercise 6 Show that Prim's algorithm, which runs in $O(m + n \log n)$ time, can be combined with `Boruvka`'s algorithm to yield an $O(m \log \log n)$ algorithm for finding a minimum spanning tree.

Exercise 7 Describe a *deterministic* linear time algorithm algorithm for finding a minimum spanning tree in a *planar* graph. (Hint: a contraction of a planar graph is planar graph. A planar graph on n vertices with no parallel edges has at most $3n - 6$ edges.)

Exercise 8 The randomized linear time algorithm of Karger, Klein and Tarjan for finding a minimum spanning tree uses sampling steps in which each edge of the graph is chosen, independently, with probability 1/2. What would be the expected running time of the algorithm if this sampling probability were changed to p, where $0 \leq p \leq 1$?

Exercise 9 In this exercise we obtain an alternative proof, due to Chan, of the following variant of the sampling lemma used to obtain the randomized linear MST algorithm:

> Let $G = (V, E)$ be a weighted graph with distinct edge weights. Let $G' = (V, R)$ be a random subgraph of G containing *exactly* r edges. Let F be a minimum spanning forest of G'. Then, the expected number of edges of G that are F-light is at most nm/r.

(a) Show that $e \in E$ is F-light if and only if $e \in MSF(R \cup \{e\})$, where $MSF(R \cup \{e\})$ is the minimum spanning forest of the subgraph $(V, R \cup \{e\})$.

(b) Show that if e is a random edge of G and R is a random subset of exactly r edges of G then $\mathbb{P}[e \in MSF(R \cup \{e\})] < n/r$. (Hint: note that e is a random element of $R \cup \{e\}$, a random set of size r or $r + 1$. How many edges from this set are in $MSF(R \cup \{e\})$?)

(c) Finish of the proof of the lemma.

References

[Bor26] O. Borůvka. O jistém problému minimálním. 1926.

[Cha98] T.M. Chan. Backward analysis of the Karger-Klein-Tarjan algorithm for minimum spanning trees. *Information Processing Letters*, 67:303–304, 1998.

[Cha00a] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[Cha00b] B. Chazelle. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM*, 47(6):1012–1027, 2000.

[CT76] D. Cheriton and R.E. Tarjan. Finding minimum spanning trees. *SIAM Journal on Computing*, 5(4):724–742, 1976.

[DRT92] B. Dixon, M. Rauch, and R.E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[FW94] M.L. Fredman and D.E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–551, 1994.

[GGST86] H.N. Gabow, Z. Galil, T.H. Spencer, and R.E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6:109–122, 1986.

[Hag09] T. Hagerup. An even simpler linear-time algorithm for verifying minimum spanning trees. In *Proc. of 35th WG*, 2009.

[Kin97] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–270, 1997.

[KKT95] D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42:321–328, 1995.

[Kom85] J. Komlós. Linear verification of spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[Kru56] J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[PR02] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[PR08] S. Pettie and V. Ramachandran. Randomized minimum spanning tree algorithms using exponentially fewer random bits. *ACM Transactions on Algorithms*, 4(1):1–27, 2008.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[Tar75] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[Tar83] R.E. Tarjan. *Data structures and network algorithms*. SIAM, 1983.

[Yao75] A.C.-C. Yao. An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees. *Information Processing Letters*, 4(1):21–23, 1975.