Minimum Spanning Tree Verification

Gregory Bint

October 16, 2013

Chapter 1

Minimum Spanning Trees

1.1 Minimum Spanning Trees

Let $G = (V, E)$ be an undirected connected graph with a cost function w mapping edges to positive real numbers. A spanning tree is an undirected tree connecting all vertices of G. The *cost* of a spanning tree is equal to the sum of the costs of the edges in the tree. A *minimum* spanning tree (MST) is a spanning tree whose cost is minimum over all possible spanning trees of G. It is easy to see that a graph may have many MSTs with the same cost (e.g., consider a cycle on 4 vertices where each edge has a cost of 1; deleting any edge results in a MST, each with a cost of 3).

As in the CLRS book[4], we will describe the two main algorithms for building MSTs, Kruskal's and Prim's. Both of these algorithms are greedy algorithms and are based on the following generic algorithm (Algorithm 1.1). The algorithm maintains a subset of edges A, which is a subset of some MST of G.

Algorithm 1.1: Generic-MST
Input: Graph G, cost function w
Output: A minimum spanning tree of G

```
1 A ← 0
2 while A ≠ MST do
3 |   find a safe edge {u, v} for A
4 |   A ← A ∪ {u, v}
5 end
6 return A
```

Intuitively this algorithm is straight-forward except for two pressing questions: What is a safe edge, and how do we find one? To answer these questions, we first need a few definitions.

Cut A cut $(S, V \setminus S)$ of $G = (V, E)$ is a partition of vertices of V.

Edge crossing a cut
An edge $(u, v) \in E$ crosses the $cut(S, V \setminus S)$ if one of its end point is in the set S and the other one in the set $(V \setminus S)$.

Cut respecting A
A cut $(S, V \setminus S)$ respects the set A if none of the edges of A crosses the cut.

Light edge An edge which crosses the cut and which has the minimum cost of all such edges.

Theorem 1.1. *Let A be a subset of the edges of E which is included in some MST, and let $(S, V \setminus S)$ be a cut which respects A. Let (u, v) be a light edge crossing the cut $(S, V \setminus S)$, then (u, v) is safe for A.*

Proof. Assume that T is a MST that includes A (similarly, you may think of A as being a subset of the edges of T, or being "the makings of" a MST). If T includes the edge (u, v) then (u, v) is safe for A. If T does not include (u, v), then we will show that there is another MST, T', that includes $A \cup \{(u, v)\}$, and this will prove that $\{(u, v)\}$ is safe for A. Since T is a spanning tree, there is a path, say $P_T(u, v)$, from the vertex u to vertex v in T. By inserting the edge (u, v) in T we create a cycle. Since u and v are on different sides of the cut, there is at least one edge $(x, y) \in P_T(u, v)$ that crosses the $cut(S, T \setminus S)$. Moreover $(x, y) \notin A$, since the cut respects A. But the cost of the edge (x, y) is at least the cost of the edge (u, v), since edge (u, v) is a light edge crossing the cut. Construct a new tree T' from T by deleting the edge (x, y) in T and inserting the edge (u, v). Observe that the cost of the tree T' is at most the cost of the tree T since the cost of (x, y) is at least the cost of (u, v). Moreover $A \cup \{(u, v)\} \subset T'$ and $(x, y) \notin A$, hence edge (u, v) is safe for A. □

The above theorem leads to the following corollary, where we fix a particular cut (i.e. the $cut(C, V \setminus C)$).

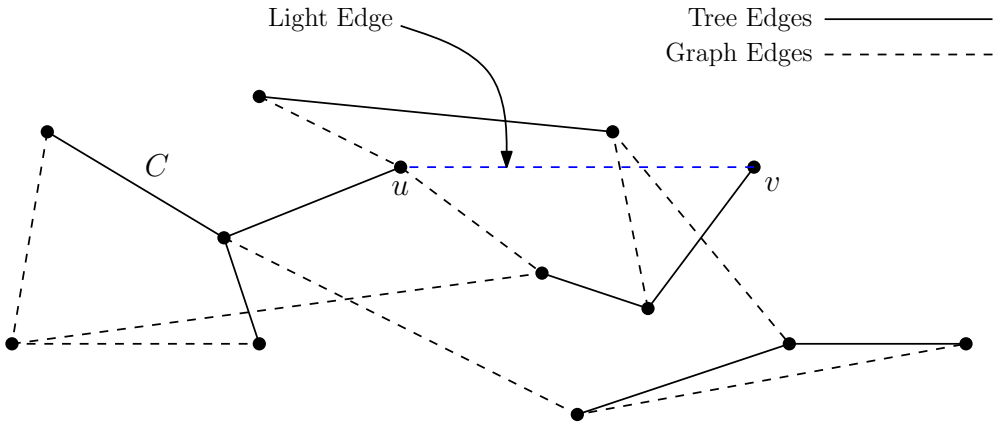

Figure 1.1: An example of Corollary 1.1. The edge (u, v) connects C to some other component of G_A and is a light edge; it is therefore safe to add to the MST.

Corollary 1.1. *Let $A \subset E$ be included in some MST. Consider the forest consisting of $G_A = (V, A)$, i.e., the graph with the same vertex set as G but restricted to the edges in A. Let $C = (V_C, E_C)$ be a connected component of G_A. Let (u, v) be a light edge connecting C to another connected component in G_A, then (u, v) is safe for A (See Figure 1.1).*

1.2 Kruskal's Algorithm for MST

Proposed by Kruskal in 1956, this algorithm follows directly from Corollary 1.1. Here are the main steps. To begin with the set A consists of only isolated vertices, and no edges (so, $|V|$ "connected" components in all).

1. Sort the edges of E in non-decreasing order with respect to their cost.
2. Examine the edges in order; if the edge joins two components then add that edge (a safe edge) to A.

To implement Step 2, we do the following. Let e_i be the edge under consideration, implying that all edges with a lesser cost than $e_i = (a, b)$ have already been considered. We need to check whether the endpoints a and b are within the same component or whether they join two different components. If the endpoints are within the same component, then we discard the edge e_i. Otherwise, since it is the next lightest edge overall, it must be the lightest edge between some pair of connected components, and so we know from Corollary 1.1 that it is safe to add to A. We will need to merge these two components to form a bigger component.

To accomplish all of this, we will need some data structure which supports the following operations:

- MAKE-SET(v) - create a new set containing only the vertex v.
- FIND(v) - Find the set which presently contains the vertex v.
- MERGE(V_x, V_y) - Merge the two sets V_x and V_y together such that FIND will work correctly for all vertices in merged set.

We can implement this data structure as follows. For each vertex we keep track of which component it lies in using a label associated with the vertex. Initially each vertex belongs to its own component, which is done with MAKE-SET. During the algorithm the components will be merged, and the labels of the vertices will be updated. Assume that we need to merge the two components V_a and V_b corresponding to the end points a and b of the edge $e_i = (a, b)$. We use FIND(a) and FIND(b) to get the sets V_a and V_b respectively. We then call MERGE which will relabel all of the vertices in one of the components to have the same labels as the vertices of the other. The component which we relabel will be the one which is smaller in size. Given such a data structure, we can implement Kruskal's algorithm as in Algorithm 1.2.

Let us analyze the complexity of Kruskal's algorithm. Sorting the edges takes $O(|E| \log |E|)$ time. The test for an edge, whether it joins two connected components or not, can be done in constant time. (In all $O(E)$ time for all edges.) What remains is to analyze the complexity of merging the components which can be bounded by the total complexity of relabeling the vertices. Consider a particular vertex v, and let us estimate the maximum number of times this will be relabeled. Notice that the vertex gets relabeled only if it is in a smaller component and its component is merged with a larger one. Hence after merging, the size of the component containing v becomes at least double. Since the maximum size of a component is $|V|$, this implies that v can be relabeled at most $\log_2 |V|$ times. Therefore, the total complexity of the Step 2 of the algorithm is $O(|E| + |V| \log |V|)$ time. These results are summarized in the following theorem.

Theorem 1.2 (Kruskal)**.** *A minimum (cost) spanning tree of an undirected connected graph $G = (V, E)$ can be computed in $O(|V| \log |V| + |E| \log |E|)$ time.*

Algorithm 1.2: Kruskal-MST
Input: Graph $G = (V, E)$, cost function w
Output: A minimum spanning tree of G

```
1  A ← ∅
2  foreach v ∈ V do
3  |  MakeSet(v)
4  end
5  sort the edges of E in non-decreasing order w.r.t. w
6  foreach e = {a, b} ∈ E, where e is taken in sorted order do
7  |  V_a ← Find(a)
8  |  V_b ← Find(b)
9  |  if V_a ≠ V_b then
10 |  |  A ← A ∪ {e}
11 |  |  Merge(V_a, V_b)
12 |  end
13 end
14 return A
```

1.3 Prim's MST algorithm

Prim's algorithm is very similar to Dijkstra's single source shortest path algorithm[5], and, in fact, their complexity analysis will be the same. Here the set A at any stage of the algorithm forms a tree, rather than a forest of connected components as in Kruskal's. Initially the set A consists of just one vertex. In each stage, a light edge is added to the tree connecting A to a vertex in $V \setminus A$.

The key to Prim's algorithm is in selecting that next light edge efficiently at each iteration. For each $v \in V \setminus A$, we keep track of the least cost edge which connects v to A, and the cost of this edge is used as the "key" value of v. These key values are then used to build a priority queue Q. See Figure 1.2 for an example of these sort of light edges.

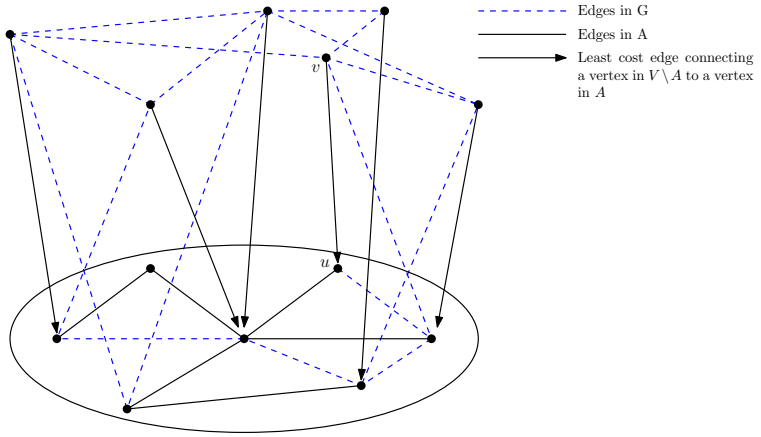

Figure 1.2: Example of Prim's Algorithm showing the set A (encircled) and the least cost edges associated with each vertex in $V \setminus A$.

In each step of the algorithm, the vertex v with the least priority is extracted out of Q. Suppose that corresponds to the edge $e = \{u, v\}$, where $u \in A$, then observe that e is a safe edge since it is the light edge for $cut(A, V \setminus A)$. We update $A := A \cup \{e\}$. Finally, after extracting v out of Q, we need to update Q. The details are explained in Algorithm 1.3.

Algorithm 1.3: PRIM-MST
Input: Graph $G = (V, E)$, cost function w, root vertex r
Output: A minimum spanning tree of G

```
1  foreach v ∈ V do
2  |  key(v) ← ∞
3  |  π(v) ← nil /* π keeps track of the parent of a vertex in the tree.      */
4  end
5  key(r) ← 0
6  Q ← V /* Priority queue consists of vertices with their key values        */
7  while Q ≠ ∅ do
8  |  u ← Extract-Min(Q)
9  |  foreach v adjacent to u do
10 |  |  if v ∈ Q and w(u, v) < key(v) then
11 |  |  |  π(v) ← u
12 |  |  |  key(v) ← w(u, v)
13 |  |  end
14 |  end
15 end
```

The vertices that are in the set A at any stage of the algorithm are the vertices in $V \setminus Q$, i.e., the ones that are not in Q. $kev(v)$ is the weight of the light edge $\{v, \pi(v)\}$ connecting v to some vertex in the MST A. Notice that the *key* value for any vertex starts at infinity, when it is not adjacent to A via any edge, and then keeps decreasing.

Let us analyze the complexity of the algorithm. The main steps are the priority queue operations, namely *decrease-key* and *extract-min*. We perform $|V|$ extract-min operations in all, one for each vertex. We also perform $O(|E|)$ decrease-key operations, one for each edge. The following table shows the complexity of these operations depending on the type of priority queue you choose. These complexities are per operation, although the complexities of Fibonacci Heaps are *amortized* (kind of an average over the worst possible scenario! - more on that later).

	Binary Heaps	Fibonacci Heaps
Extract-min	$O(\log n)$	$O(\log n)$
Decrease-key	$O(\log n)$	$O(1)$

1.4 Randomized Algorithms for Minimum Spanning Trees

Here we discuss some results related to randomized algorithms for computing minimum spanning trees. These results are based on Section 10.3 of Raghavan and Motwani's book on Randomized Algorithms [16] and T. Chan's simplified analysis from [3]. Assume that all edge weights are distinct and hence there is a unique MST in the given graph $G = (V, E)$. The randomized algorithm uses

a few concepts which are discussed in the following subsections followed by the actual algorithm itself in Section 1.4.3. The first concept is an algorithm due to Boruvka from 1926[1, 2] which helps us to reduce the number of vertices in the graph. The second is about heavy and light edges with respect to a spanning tree.

1.4.1 Boruvka's Algorithm

Observe that for any vertex $v \in V$, the edge, say $\{v, w\}$, with the minimum weight incident to that vertex will be included in the MST, as per Corollary 1.1. This leads to a simple way to compute the MST as given in Algorithm 1.4.

Algorithm 1.4: Boruvka-MST
Input: Graph G, cost function w (all costs distinct)
Output: A minimum spanning tree T of G

```
1 T ← ∅
  // Each iteration of this loop is called a Phase
2 while G contains more than a single vertex do
3 |   Mark the edges with the minimum weight incident to each vertex. Add these edges to T
4 |   Identify the connected components of the marked edges
5 |   Replace each connected component by a vertex
6 |   Eliminate all self loops. Eliminate all multiple edges between a pair of vertices, except
  |   the edge with the minimum weight
7 end
8 return T
```

A few observations about this algorithm:

- Let G' be the graph obtained from G after contracting the edges in a single phase of the algorithm. Then the MST of G is the union of the contracted edges from that phase and the edges in the MST of G'.
- In each contraction phase, the number of vertices in the graph is reduced by at least a half. Hence there will only be $O(\log |V|)$ phases in all.
- Each phase can be implemented in $O(|V| + |E|)$ time and so we obtain yet another MST finding algorithm. Total running time for this algorithm is $O(|E| \log |V|)$.

1.4.2 Heavy Edges

We've already seen the definition of a light edge. Now we examine edges which are not light.

Let F be any spanning tree of G (in particular, F may or may not be a minimum spanning tree). Consider any two vertices, say u and v, of G and there is a unique path $P(u, v)$ between them in F. Let $w_F(u, v)$ be the edge with the maximum weight along this path.

We say an edge $\{u, v\}$ is F-heavy if the weight of this edge is larger than the weight of each of the edges on the unique path between u and v in F. More formally, we define an edge $\{u, v\} \in E$ to be F-heavy if $w(u, v) > w_F(u, v)$, otherwise it is F-light. Observe that by this definition, all

edges of F are F-light (every edge of a path in F is at most as heavy as the heaviest edge in that path of F).

Lemma 1.1. *Let F be a spanning tree of G which is not necessarily minimum. If an edge of G is F-heavy then it does not lie in the Minimum Spanning Tree of G.*

Proof. It is left for you to prove it formally. The proof proceeds by contradiction. Assume that the edge $e = \{u, v\} \in E$ is F-heavy, that T is an MST of G, and that $e \in T$ (so, we are talking about two trees here, T, which is known to be an MST, and F, which may be one as well). Consider the edges on the path $P(u, v)$ in F. Add all these edges to T and remove e from T, to obtain a graph G' (that is, $G' = T \cup P(u, v) \setminus \{e\}$, but since it may have some cycles, we cannot call it a tree). G' is still connected. Remove edges from $P(u, v)$ one-by-one from G' until G' is once again a tree. Observe that this new tree is still a spanning tree, but it must have weight lower than that of T, contradicting the minimality of T.

□

It is very important to note that because of the way F-light is defined, an edge which is F-light may or may not be in the MST. For example, if we construct F such that the heaviest edge of E is in F, that edge will be counted as F-light, even though it may not be present in any minimum spanning tree.

1.4.3 Randomized Algorithm

Algorithm 1.5 gives the main steps of the randomized algorithm we have been discussing. The analysis is based on Timothy Chan's paper[3].

Algorithm 1.5: Randomized-MST
Input: Connected Graph $G = (V, E)$ with distinct edge weights
Output: A minimum spanning tree T

1. Execute 3 phases of Boruvka's algorithm (reduces number of vertices). Let the resulting graph be $G_1 = (V_1, E_1)$, where $|V_1| \leq |V|/8$ and $|E_1| \leq |E|$. Let C be the set of contracted edges. (Running time: $O(|V| + |E|)$)
2. Random Sampling: Choose each edge in E_1 with probability $p = 1/2$ to form the set E_2 and obtain the sampled graph $G_2 = (V_2 = V_1, E_2)$.
3. Compute Recursively the Minimum Spanning Tree of G_2, and let it be F_2. $(T(|V|/8, |E|/2))$
4. Verification: Compute the set of F_2-light edges in E_1, and let this set be E_3. ($O(|V_1| + |E_1|)$ time)
5. Final MST: Compute MST, F_3, of the graph $G_3 = (V_3 = V_1, E_3)$. $(T(|V|/8, |V|/4))$
6. **return** MST of G as $C \cup F_3$.

Theorem 1.3. *Algorithm 1.5 correctly computes the MST of G in $O(|V| + |E|)$ time.*

Proof. The correctness of the algorithm is straightforward. To estimate the complexity, the crux is in estimating the size of the set E_3, i.e., the size of the set of F_2-light edges in E_1. We will prove in the Sampling Lemma (Lemma 1.3) that for a random subset $R \subset E$, the expected number of edges that are light with respect to MST(R) is at most $(|E| \cdot |V_1|)/|R|$. In our case, the expected

value of $|R| = |E_1|/2 \leq |E|/2$, and hence the expected number of F_2-light edges will be at most $2|V_1| \leq |V|/4$. Hence the running time of this algorithm is given by the recurrence

$$T(|V|, |E|) = O(|V| + |E|) + T(|V|/8, |E|/2) + T(|V|/8, |V|/4),$$

which magically solves to $O(|V| + |E|)$.

□

Before we describe the Sampling Lemma here are some technicalities. Consider that we are sampling the edges of the graph $G = (V, E)$, and the sampled edges form the subgraph R.

We use the notation R for both the set of edges as well as the sampled graph. Since we are sampling the edges, it is possible that the sampled graph R of the graph G is not connected, and hence there will not be any minimum spanning tree. To ensure connectedness, we will fix any spanning tree T_0 of G, consisting of $|V| - 1$ edges and we will consider the minimum spanning tree of $R \cup T_0$, denoted as $MST(R)$.

Lemma 1.2 (Observation about light edges)**.** *An edge $e \in E$ is light with respect to $MST(R)$ if and only if $e \in MST(R \cup \{e\})$.*

Proof. If $e = (u, v)$ is light then there is some edge e' on the unique path between u and v in $MST(R)$ such that its weight, $w(e') = w_{MST(R)}(u, v)$, and hence e can be added to $MST(R)$ and e' can be removed to obtain MST of $R \cup \{e\}$. Therefore $e \in MST(R \cup \{e\})$.

Now suppose $e \in MST(R \cup \{e\})$. We need to show that e is light with respect to $MST(R)$. Since e is part of the MST, by definition it is light with respect to that MST. □

Lemma 1.3 (Sampling Lemma)**.** *For a graph $G = (V, E)$ and a random subset $R \subset E$, of edges, the expected number of edges that are light with respect to $MST(R)$ is at most $(|E| \cdot |V|)/|R|$.*

Proof. Pick a random edge $e \in E$ (this choice is independent of the edges in R). We will prove that e is light with respect to $MST(R)$ with probability at most $|V|/|R|$. From Lemma 1.2 we see that this is equivalent to finding the bound on the probability that $e \in MST(R \cup \{e\})$. Let $R' = R \cup \{e\}$. We will use a technique called *backward analysis*. First we analyze the probability on a fixed set R', and then we will show that the expression obtained is not dependent on the elements of R', but just the cardinality, and hence the probability holds unconditionally as well.

Instead of adding a random edge to R, we will think of deleting a random edge from R'. This is an easier proposition since we know the elements of R', having just fixed it. $MST(R')$ has $|V| - 1$ edges, and e is a random edge of R', hence the probability that e is an edge from $MST(R')$, given a fixed choice of R', is $(|V| - 1)/|R'| \leq |V|/|R|$. This bound is independent upon the choice of the set R', and holds unconditionally as well.

□

1.5 MST Verification

If I give you any tree F derived from a graph $G = (V, E)$, can you identify whether F is a minimum spanning tree of G?

A trivial method for determining this would be to run a known-correct algorithm such as Kruskal's or Prim's on G and compare the output to F, however there are two main drawbacks to this approach:

1. It is too slow

2. The MST may not be unique, making a direct comparison difficult.

What we would like is to be able to calculate this in linear time with respect to the graph. The following lemma has been shown to be very useful in this respect, and it is used by virtually every MST verification algorithm.

Lemma 1.4. *Let F be a spanning tree of G, then F is a minimum spanning tree of G if and only if every edge in $E \setminus F$ is F-heavy.*

Proof. Let $P(u, v)$ be the unique tree path between u and v in F, and let $w_F(u, v)$ be the weight of the heaviest edge along that path. $w(e)$ or $w(u, v)$ is the weight of the edge e having endpoints u and v.

We first show that if F is a MST, then every edge in $E \setminus F$ is F-heavy. Let e be any edge in $E \setminus F$ with endpoints u and v and assume that e is F-light. Note that $P(u, v) \cup \{e\}$ is a cycle. Let $e' = \{x, y\}$ be the edge corresponding to $w_F(u, v)$. Since e is F-light, $w(e') > w(e)$, meaning we could replace e' by e in F to obtain a lighter tree overall, contradicting the minimality of F.

For the other half of the proof, we show that if every edge in $E \setminus F$ is F-heavy, then F is a MST of G. Again, we proceed by contradiction. Suppose that F is not a MST of G, then we should be able to lower the weight of the tree by replacing some edges in F with those from $E \setminus F$. But, for every $einE \setminus F$ with endpoints u and v, we have that $w(u, v) > w_F(u, v)$, so exchanging e for any other edge in $P(u, v)$ will increase the weight of F.

□

Given the above lemma, a natural idea for an algorithm would be to try to classify every edge in the graph, and then check if each non-tree edge is in fact F-heavy. This turns out to be something which *is* possible: given a graph $G = (V, E)$ and a tree F, we can partition the edges of G in two sets, the set of heavy edges and the set of light edges, with respect to F in $O(|V| + |E|)$ time.

We will see that many of the algorithms for doing so fairly complex, although there has been recent progress in simplifying it somewhat.

1.5.1 Overview of verification algorithms

Here we look at a brief history of the literature on MST Verification. As hinted at above, every single one of the following methods uses Lemma 1.4 as its underpinning. This problem can also be restated as the following:

Problem 1.1 (The Tree Path Maxima Problem)**.** Let F be a spanning tree of G, then we want to identify the cost of the heaviest edge along each tree path $P(u, v)$.

Given an answer to Problem 1.1, we can perform a simple linear scan through the the edges of $G \setminus F$. For each edge $e \in G \setminus F$, we compare the cost of e with the tree path of its endpoints. If every edge e is heavier than its corresponding tree path maxima, then F is a MST of G. We look at this sort of translation of the problem in more detail in Section 1.5.6.

Here is a timeline of some results:

- In 1979, Tarjan introduced a method which uses path compression[21] of trees to achieve a near-linear time of $O(m\alpha(m, n))$ where α is the Inverse Ackermann[1] function.

- In 1984, Komlós's provided an algorithm[14, 15] which showed that only a linear number of *comparisons* of the edge costs would be sufficient to solve the problem, however the algorithm itself has significantly more than linear overhead.

- In 1992, Dixon *et al.*[6] combine methods from Tarjan's 1979 algorithm and Komlós's 1984 algorithm to produce the first linear time MST verification algorithm. The problem is divided into one large problem and several small problems, with the larger being attacked with path compression, and the smaller with a lookup scheme which is bounded in size by Komlós's algorithm.

- In 1994, Karger *et al.* present an algorithm for computing the MST of a graph in *expected* linear time.[10, 13] While not a verification algorithm in itself, its output could be useful to help verify another tree, or to take the place of the other tree altogether (e.g., why bother verifying a potential MST in linear time when you can just *create* one!)

- In 1995, King produced another linear time MST verification method[11, 12] which is a great deal simpler than that of Dixon *et al.*. In this method, Boruvka's algorithm is used to reduce a general tree down to one which can be handled entirely by the full branching tree base case of Komlós's algorithm, which is simpler than his algorithm for general trees.

- In 2010, Hagerup simplifies King's method[7] even further and provides an implementation in the *D* programming language. Like King's method, Hagerup continues to use Komlós's *full branching tree* case, but eschews complex edge encoding schemes in favour of a richer logical data type.

We will walk through parts of Komlós's algorithm, Dixon *et al.*'s algorithm, King's algorithm, and finally Hagerup's algorithm as we piece together the tools needed for a reasonably simple approach to solving this problem.

1.5.2 Komlós's Algorithm

In 1984, Komlós[14, 15] gives an algorithm of sorts which can solve Problem 1.1 in $O(n + m)$ *comparisons*. He does not provide an implementable algorithm, however, and there are other factors of overhead in his method which would drive up the actual cost of a straight implementation. Nevertheless, this method of breaking down the problem is built upon by later papers, notably Dixon *et al.* in 1992, and King, which we cover in Section 1.5.4.

Komlós begins by considering two special cases of spanning trees. In each case, we consider F to be a directed tree with edges oriented away from the root. Additionally, we shift the edge costs down to their lower endpoint vertices, as this simplifies the conceptual model.

The first case occurs when the tree is a path. For the path, we construct a *symmetric order heap*, H, which a tree with both the binary search property on the ordering determined by the path, and the (maximum) heap property determined by the vertex costs. The root of H represents the heaviest vertex, and the heaviest vertex of any path from u to v is found at $LCA(u, v)$. Determining

[1] `http://en.wikipedia.org/wiki/Ackermann_function#Inverse`

the LCA of two vertices in a tree can be accomplished in several ways and is covered elsewhere in these notes, but Komlós cites Harel[8] specifically.

The second case is somewhat more interesting and is concerned with processing *full branching trees*. Not to be confused with full *binary* trees, a full branching tree is defined as one where every leaf is at the same level, and every internal vertex has *at least* 2 children. Let F be our full branching tree with root r and all edges directed away from r.

We need to calculate the maximum cost edge of every path through F. Given a vertex y, let $A(y)$ be the set of all paths through F which contain y. That is $A(y) = \{P(x,z) | x \geqq y \geqq z\}$ where $u \geqq v$ denotes that u is a predecessor of v (or, u may equal v). Since F is directed away from the root, this means that u is at least as close to the root than v. Note that if F was not directed, then it might be possible for $x \geqq y \geqq z$ to hold even though $y \notin P(x,y)$. Given $A(y)$, let $A^*(y)$ be the set of all paths through y, but restricted to just the interval $[r,y]$; that is, just the subpath from the root down to y.

We process F one level at a time, starting from the root, finding the maximum weights of all paths in the sets $A^*(y)$. To do this, assume that we have calculated the maximum costs in all such paths up to level i, and that we are now trying to process some vertex y on level $i+1$. Let $\bar{y}$ be the parent of y. Since $\bar{y}$ resides on level i, we know the maximum cost of all paths in $A^*(\bar{y})$.

The key observation to make here is the following.

Property 1.1. *Consider two paths $P(x,\bar{y})$ and $P(x',\bar{y})$. If x is a predecessor of x' then the maximum cost in $P(x,\bar{y})$ is at least as large as the maximum cost in $P(x',\bar{y})$ (since, under these conditions, $P(x',\bar{y})$ is a subpath of $P(x,\bar{y})$).*

In $A^*(\bar{y})$, the shortest path is $P(\bar{y},\bar{y})$ while the longest is $P(r,\bar{y})$. By the above property, the maximum costs form a non-decreasing sequence with respect to the length of the path. That is, we can order the maximum costs by considering the path length. This observation about the ordering helps us while building $A^*(y)$, as we can use a binary search insertion of $f(y)$ to compare $f(y)$ against all path cost maximums in $A^*(\bar{y})$ simultaneously.

By now you should be asking "How are all these sets like $A(y)$ and $A^*(\bar{y})$ created, copied, and updated?" As far as Komlós's paper is concerned, the answer is "slowly". Essentially what Komlós shows us is that a linear number of *comparisons* are sufficient to determine maximum path costs, however finding those comparisons is left open.

The remainder of Komlós's paper details how these two primitive cases can be applied to any general tree, however this method is fairly complex, and as he states, results in too much overhead. No implementable algorithm is given in this paper.

1.5.3 Dixon *et al.*'s Technique

This technique has the distinction of being the first to achieve a linear running time, requiring $O(m)$ time on a graph with n vertices and m edges. The underlying process is fairly complex, however, and involves first preprocessing the graph into a suitable form.

The preprocessing itself is interesting as it shows a method of massaging a graph into a more attractive form for the problem at hand without affecting anything about the spanning tree that we wish to verify. The preprocessing involves the following steps.

For a given graph $G = (V,E)$ and spanning tree F, not necessarily minimum, we choose an arbitrary vertex r to be the root of F. Now consider any non-tree edge $\{v,w\}$ with cost $c(v,w)$ and lowest common ancestor u. If u is not one of v or w, then this implies that v and w are not *related*;

that is, the v is neither ancestor nor descendant of w. In such a case, we replace the edge $\{v, w\}$ by $\{v, u\}$ and $\{u, w\}$, each with cost $c(v, w)$. F is unchanged by this process and, more importantly, the maximum weight along $P(v, w)$ is also unchanged, which preserves the current minimality of F. There are several linear time algorithms for finding lowest common ancestors in a tree (e.g., Harel & Tarjan, 1984[9] or Schieber & Vishkin, 1988[18]).

Taken over the entire graph, this will at most double the number of non-tree edges. When completed, every non-tree edge in F is a backedge.

In the second stage of preprocessing, we will mark several vertices. We can imagine these marks as subdividing the tree into edge-disjoint subtrees where a marked vertex represents a "root", and any marked descendants are ignored.

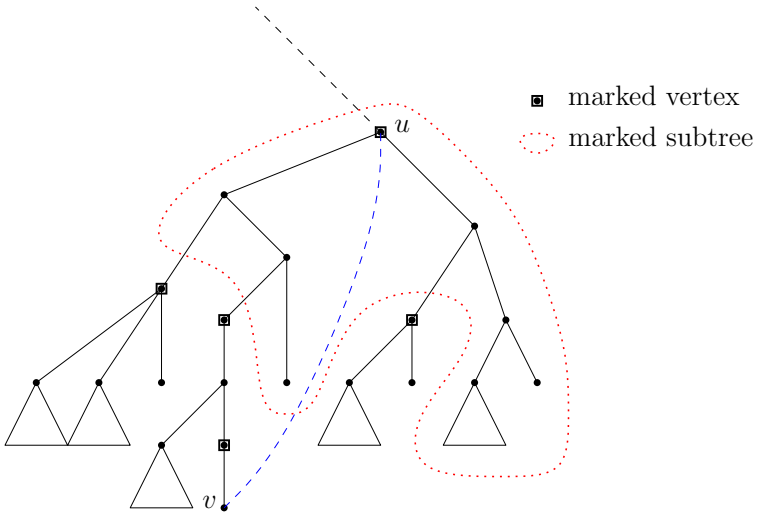

Figure 1.3: An example tree with marked vertices. The marked subtree defined by u is enclosed in red.

The choice of which vertices to mark is based on subtree size. Using a post-order traversal, for each vertex v we calculate $h = 1 + \sum\{s(w)|w \text{ is a child of } v\}$. Let g be a small integer, then if $h \leq g$ we assign $s(v) := h$, otherwise $s(v) := 1$ and v becomes marked. We will look more at the specific choice of g later. Note the following important properties resulting from this process:

1. The number of marked vertices, and hence the number of subtrees, is at most $(n-1)/g + 1$.
2. Considering any subtree, if its root (marked vertex) is deleted, along with incident edges, we get a collection of disjoint trees, each with size at most g. We call each of these a *microtree*.

Following that process, r is also marked, although it will probably not have Property 2.

A final phase of edge replacements will ensure that all backedges either span two vertices belonging to the same microtree, or span between microtrees and marked vertices only (i.e., the edge $\{u, v\}$ in Figure 1.3 will be replaced).

To accomplish this, we first build the tree F' whose vertex set consists of all of the marked vertices of F, and where, for two vertices s and t in F', s is the parent of t (i.e, there is a tree edge between them) if s is the first marked vertex that we encounter when walking from t to r. We call T' the *macrotree*.

We can now eliminate the "long" edges, like $\{u, v\}$, by doing the following. Let $p(v)$ be the nearest marked vertex to v which is a proper ancestor of v. Note that if v is marked then $p(v) \neq v$. We also assume that $p(r)$ is undefined (but it won't be needed anyway). We can calculate $p(v)$ for the entire tree using a depth-first search. For every non-tree edge $\{u, v\}$, assume w.l.o.g. that u is an ancestor of v and find $p(u)$ and $p(v)$. If $p(u) = p(v)$, then u and v are part of the same microtree (recall that the root of a microtree is *not* marked).

Otherwise, if $p(u) \neq p(v)$, then we know that there is at least one marked vertex between them. Let $r_1 = u$ if u is marked, or $r_1 = p(u)$ if u is not marked. Similarly, let $r_3 = v$ if v is marked, or $r_3 = p(v)$ if v is not marked. Let r_2 be the child of r_1 in F' (note: F', not F). We then replace $\{u, v\}$ by $\{u, r_2\}$, $\{r_2, r_3\}$, and $\{r_3, v\}$, skipping any edge that creates either a self-loop or which duplicates a tree edge. For edges which we did not skip, assign the cost $c(u, v)$. As in the first phase of edge replacements, assigning this cost preserves the current minimality of F.

With this preprocessing finished, we have now divided the problem into one large tree rooted at r with several microtrees around the periphery. The authors complete the process by using Tarjan's Path Compression on the large tree.

The microtrees are processed in a very different way. Essentially, the authors precalculate all possible minimum spanning trees on graphs containing at most g vertices. Leveraging Komlós's result, they show that for any such input, the corresponding decision tree for comparing edges and determining minimality is not too big. The choice of g is such that the total size of these precalculations is only $O(n)$, which places g in the neighbourhood of $O(\log \log n)$ [12].

1.5.4 King's Method

Presented by King[11, 12] in 1995, this method is not the first MST verification algorithm to achieve linear time (that falls to Dixon *et al.*[6]), however it is quite a bit simpler. King's method uses Boruvka's algorithm in a clever way to change any input tree into a full binary tree, which can then be entirely processed by the appropriate case presented by Komlós. This method requires linear time and space in the unit-cost RAM model with $\Theta(\log n)$ word size.

Boruvka Tree Property

The first step is to take our input tree F and convert it to a full binary tree. This is accomplished by running Boruvka's algorithm on the tree F (we usually would run Boruvka's on an entire graph, but not in this case). As Boruvka's runs on F, we can build a new tree B which represents the *execution* of the algorithm on F, rather than a modification of F itself.

Algorithm 1.6 details the construction of B. In the first step, a leaf is added to B for each vertex of F, so we already know that $|B| \geq |F|$. In fact, B will have at most twice as many vertices as F when we are finished. The algorithm proceeds by colouring the vertices and edges to represent subtrees within F, such that any vertices connected along a coloured (blue) path is considered part of the same subtree.

Refer to Figure 1.4 for an example of the algorithm's execution. Note the following important properties which ensure that B is a full branching tree.

1. In each step of Loop 1, an edge joins two blue trees into one.

2. In each phase of the while loop, every blue tree is combined by some edge with another blue tree. Thus, from every level of B, every vertex has a parent in the next level.

Algorithm 1.6: FullBranchingTree
Input: A spanning tree $F = (V, E)$ with distinct edge weights
Output: A full branching tree B satisfying Lemma 1.5

```
1  Initialize B as an empty tree
2  foreach vertex v of V do
3  |  Colour v blue, considering it as a singleton tree
4  |  Add the leaf f(v) to B
5  end
6  while there is more than one blue tree do
   |  // Loop ''1'', joins blue trees together
7  |  foreach blue tree a do
8  |  |  Select a minimum cost edge e incident to a and colour it blue
9  |  end
   |  // Loop ''2'', updates B
10 |  foreach new blue tree t do
11 |  |  Add f(t) to B
12 |  |  Let A be the set of trees joined into t in Loop 1
13 |  |  Add an edge {f(t), f(a)} for each a ∈ A
14 |  |  Set the cost of {f(t), f(a)} to that of edge selected by a in Loop 1 (i.e., e)
15 |  end
16 end
17 return B
```

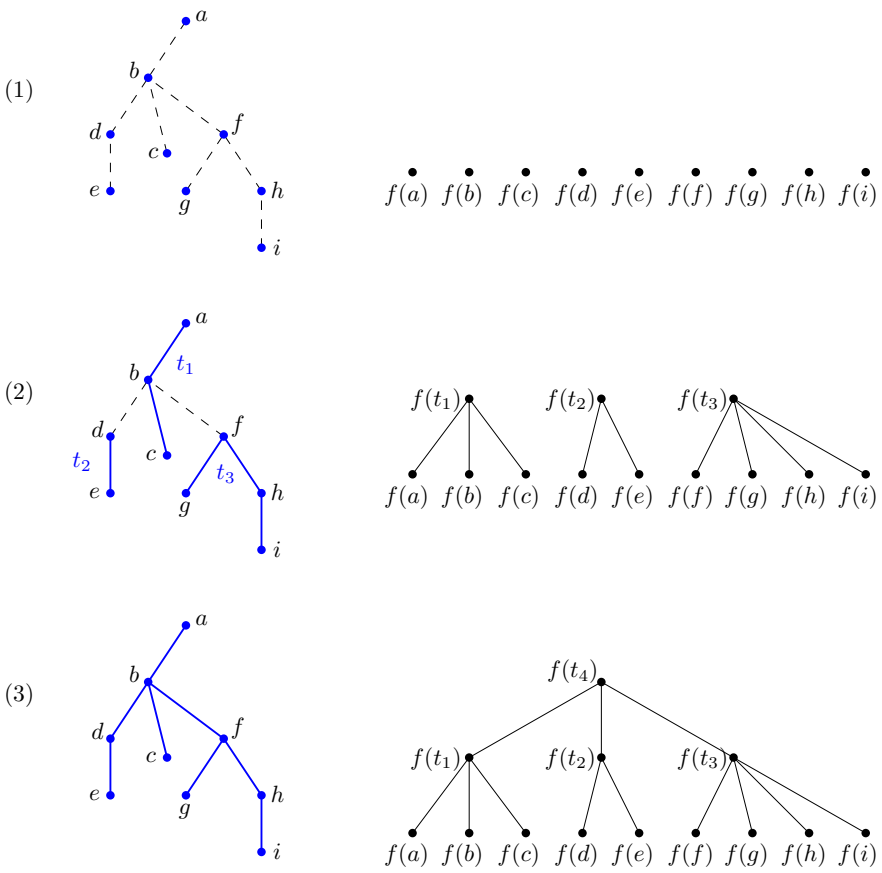

Figure 1.4: An example of the Full Binary Tree construction given by Algorithm 1.6. F is shown on the left and B on the right. Edge weights are not shown, so imagine that each tree chooses its minimum weight edge at each step.

For every v in F there is a vertex $f(v)$ in B, and by construction we also have that for every path $F(x, y)$ there is a path $B(f(x), f(y))$. However, to show that there is any meaningful correspondence between these paths beyond their existence, we need the following lemma, presented as Theorem 1 in King's paper.

Lemma 1.5. *Let F be any spanning tree and let B be the tree constructed by Algorithm 1.6. For any pair of vertices $x, y \in F$, the cost of the heaviest edge in $F(x, y)$ equals the cost of the heaviest edge in $B(f(x), f(y))$.*

Proof. Let the cost of an edge e be denoted by $w(e)$. For every edge $e \in B(f(x), f(y))$, we will show that there is an edge $e' \in F(x, y)$ such that $w(e') \geq w(e)$.

Suppose that $e = \{a, b\}$ such that a is the endpoint of e which is farthest from the root. As a is in B, $a = f(t)$ for some blue tree t, and t must contain either x or y, but not both. Similarly, $b = f(t')$ which is new blue tree consisting of $f(t)$ and others from the previous phase of the algorithm. Since $e \in B$, e was selected by t.

Let e' be the edge in $F(x, y)$ with exactly one endpoint in t. Since e' is adjacent to t, t would have considered e'. Since t ultimately chose e, it must be that $w(e') \geq w(e)$ since t chooses the

edge with minimum cost.

To finish the proof we also need to show the following: The cost of the heaviest edge in $F(x, y)$ is the cost of the heaviest edge in $B(f(x), f(y))$. Let e be the heaviest edge in $F(x, y)$ (for simplicity, assume that there is a unique such edge). If e is ever selected by a blue tree which contains either x or y, then $B(f(x), f(y))$ contains an edge with the same weight.

Otherwise, assume that e is selected by some other blue tree t' not containing x or y. We know that e is on the path from x to y in F, so t' contained at least one intermediate vertex on that path. But since F is a tree, if it contains an intermediate vertex of $F(x, y)$, it must be incident to at least two edges of $F(x, y)$. By our assumption, e is the heaviest edge on this path, so t' would have selected the other edge, giving a contradiction.

□

The intuition with the last part of the above proof is that, since e is the heaviest edge along $F(x, y)$, any blue tree which includes part of that path, but which does not yet include x or y always has another edge to select which brings it "closer" to x or y.

King's algorithm now continues with B rather than F, which maintains the path maximum cost property for each path in F, implying that if Lemma 1.4 holds for B it will also hold for F.

The remainder of King's paper shows a bit-wise labeling scheme from for the vertices and edges of B which exploits Property 1.1 of the full binary tree case presented by Komlós. We will now jump to Hagerup's method to conclude our verification method, which he wrote specifically to simplify away from this labeling scheme, but which otherwise picks up at exactly this point of the algorithm.

1.5.5 Hagerup's Method

Hagerup presents an algorithm for solving the Tree Path Maxima problem (TPM) rather than MST Verification, *per se*, but as we have mentioned, a solution to TPM implies a solution to MST Verification. A sketch of such a translation is given in the next section.

The input to Hagerup's method assumes that we are given a tree on n vertices and a list of pairs $(u_1, v_1), \ldots, (u_m, v_m)$ such that in each pair u_i is a proper ancestor of v_i. At most, this list would describe the endpoints of every root to leaf path in B, and every subpath of such a root to leaf path. Any subset is also permissible. In practice, we choose a subset equivalent to the non-tree edges of G, the graph containing the spanning tree F we are trying to verify.

The basic algorithm involves collecting several types of information about each vertex. For every vertex u in B, we store the depth $d(u)$, and, if u is not the root r, we let $w(u)$ represent the cost of the edge from u to its parent. For each u we also build the following set:

$$D_u = \{d(u_i) | u_i \text{ is a proper ancestor of } u \text{ and } v_i \text{ is a descendant of } u\}$$

Simply put, D_u stores the set of depths corresponding to proper ancestors of u such that u is in the subpath represented by some pair (u_i, v_i) from the input.

We would also like to create the set M_u for each u, which stores a subset of the ancestors of u indicated by D_u. The choice of which ones are stored again exploits Property 1.1.

Consider any two successive ancestors d and d' of u which are indicated by D_u such that d is closer to the root, and d' is closer to u. Then $d \in M_u$ if the path maximum cost of the path $d \to u$ is greater than that of $d' \to u$. Put another way, we store only those ancestors of u where there is an actual increase in path maximum cost between it and the previous (closer) ancestor.

This can still work out to be a lot of entries, however, and a lot of copying between vertices, which breaks linear time. Fortunately, Hagerup was able to find an alternate, yet equivalent set representation which does satisfy our needs, and our desired running time, using the *set infix* operator. The details of this operator and its equivalence to M_u take a few pages to discuss and can be found in his paper.

1.5.6 Putting it all together

One way of applying all of the tools we have seen so far to build a complete MST Verification algorithm is as follows.

Taking a graph $G = (V, E)$ with spanning tree F, let $U = E \setminus F$ be the set of non-tree edges. We use King's method of using Boruvka's method to convert F to the full branching tree B. Translate U onto B so that $\forall e = \{x, y\} \in U$ we create $e' = \{f(x), f(y)\}$ and call the resulting graph G'. We next apply Dixon *et al.*'s first preprocessing step to G' to replace all cross edges with back edges. Let U' be the set of non-tree edges in G' after all of this.

The set U' corresponds to the pairs (u_i, v_i) that we need to input into Hagerup's algorithm. After that algorithm has run, MST Verification is completed by examining every non-tree edge in G, translating it to the equivalent one or two edges in U', querying B, and determining whether the non-tree edge is costlier than the tree path maximum.

The extra steps required to find U, translate it to B, and then find U' all take time linear in the number of edges.

1.6 Bibliographic Notes

Kruskal's algorithm, presented in Section 1.2 makes use of a data structure known as Union-Find or Disjoint-Set. A near-linear time implementation was first described by Tarjan[20].

Boruvka's algorithm is quite old[1, 2], and not originally published in English. Nešetřil *et al.* published a translation from the original Czech in 2001 along with some comments.[17]

Komlós mentions that his method of using symmetric order heaps for processing paths is something of a well-known method by the time he covers it in his own paper. However, he was unable to find a reference to it in any other literature, which is why he took the time to write about it.

The way that King uses Boruvka's algorithm is first described by Tarjan in 1983[19].

1.7 Exercises

1.1. Let S=(V,T) be a minimum cost spanning tree, where $|V| = n + 1$. Let $c_1 \leq c_2 \leq ... \leq c_n$ be the costs of the edges in T. Let S' be an arbitrary spanning tree with edge costs $d_1 \leq d_2 \leq .... \leq d_n$. Show that $c_i \leq d_i$, for $1 \leq i \leq n$.

1.2. Assume all edges in a graph G have distinct cost. Show that the edge with the maximum cost in any cycle in G cannot be in the Minimum Spanning Tree of G. Can you use this to design an algorithm for computing MST of G by deletion of edges, and what will be its complexity?

1.3. Recall that Dijkstra's SSSP algorithm was for directed (or undirected) graphs where the weights of the edges are positive and we need to compute shortest paths from the source vertex to all other vertices in the graph. What happens when some of the edges have negative weights. Try to consider the cases where the algorithm will fail and where the algorithm will still work.

1.4. Design an efficient algorithm to find a spanning tree of a connected, (positive) weighted, undirected graph $G = (V, E)$, such that the weight of the maximum-weight edge in the spanning tree is minimized (Justify your answer).

1.5. Let $G = (V, E)$ be a weighted directed graph, where the weight of each edge is a positive integer and is bounded by a number X. Show how shortest paths from a given source vertex s to all vertices of G can be computed in $O(X|V| + |E|)$ time (Justify your answer).

1.6. Prove that if all edge weights are distinct then the minimum spanning tree of a simple undirected graph is unique.

1.7. Provide a formal proof of Lemma 1.1.

1.8. Suppose all edge weights are positive integers in the range $1..|V|$ in a connected graph $G = (V, E)$. Devise an algorithm for computing Minimum Spanning Tree of G whose running time is better than that of Kruskal's or Prim's algorithm.

1.9. Consider a connected graph $G = (V, E)$ where each edge has a non-zero weight. Furthermore assume that all edge weights are distinct. Show that for each vertex $v \in V$, the edge incident to v with minimum weight belongs to a Minimum Spanning Tree.

(Bonus Problem: Can you use this to devise an algorithm for MST - the above step identifies at least $|V|/2$ edges in MST - you can collapse these edges (by identifying the vertices and then recursively apply the same technique - the graph in the next step has at most half of the vertices that you started with - and so on!)

1.10. Prove that the distance values extracted from the priority queue over the entire execution of Dijkstra's single source shortest path algorithm, in a directed connected graph with positive edge weights, is a NON-Decreasing sequence. Where is this fact used in the correctness of the algorithm?

1.11. Can you devise a faster algorithm for computing single source shortest path distances when all edge weights are 1? (Think of an algorithm that runs in $O(|V| + |E|)$ time on a graph $G = (V, E)$.)

1.12. Execute Dijkstra's SSSP algorithm on the following graph on 7 vertices and 18 edges starting at the source vertex s. The edges and their weights are listed in the following (the entry $(xy, 10)$ means the edge directed from the vertex x to the vertex y with edge weight 10):

$(sb, 5)$, $(sa, 10)$, $(sf, 5)$, $(bf, 6)$, $(ba, 3)$, $(be, 5)$, $(bc, 5)$, $(fs, 2)$, $(fe, 4)$, $(ca, 3)$, $(ce, 2)$, $(cd, 5)$, $(df, 1)$, $(de, 1)$, $(ef, 1)$, $(ec, 1)$, $(af, 1)$, $(ae, 2)$.

1.13. Recall that Dijkstra's SSSP algorithm only computes distances from source vertex to all the vertices. What modifications we should make to the algorithm so that it reports the shortest paths as well (in fact the collection of all these paths can be represented in a directed tree rooted at the source vertex).

1.14. Suppose in place of computing shortest path distance from a vertex to every other vertex, we are interested in finding the shortest path distances between every pair of vertices. Then one way to do this is to run Dijkstra's algorithm $|V|$ times, where each vertex in the graph $G = (V, E)$ is considered as a source vertex once. Can you devise an algorithm that is asymptotically faster than just running Dijkstra's algorithm $O(|V|)$ times?

1.15. Which of the following algorithms result in a minimum spanning tree? Justify your answer. Assume that the graph $G = (V, E)$ is connected.

1. Sort the edges with respect to decreasing weight.
   Set $T := E$.

For each edge e taken in the order of decreasing weight do, if $T - \{e\}$ is connected, then discard e from T.
Set $MST(G) = T$.

2. Set $T := \emptyset$.
   For each edge e, taken in arbitrary order do, if $T \cup \{e\}$ has no cycles then
   $T := T \cup \{e\}$.
   Set $MST(G) = T$.

3. Set $T := \emptyset$.
   For each edge e, taken in arbitrary order do
   begin
   $T := T \cup \{e\}$.
   If T has a cycle c then let e' be a maximum weight edge on c.
   Set $T := T - \{e'\}$.
   end
   Set $MST(G) = T$.

1.16. A spanning tree T of a undirected (positively) weighted graph G is called a minimum bottleneck spanning tree (MBST) if the edge with the maximum cost is minimum among all possible spanning trees. Show that a MST is always a MBST. What about the converse?

1.17. Design a linear time algorithm to compute MBST. (Note that an edge with medium weight can be found in linear time. Consider the set of edges whose weight is smaller than the weight of the 'median edge'. What happens if this graph is connected? disconnected?

1.18. Consider an undirected (positively) weighted graph $G = (V, E)$ with a MST T and a shortest path $\pi(s, t)$ between two vertices $s, t \in V$. Will T still be an MST and $\pi(s, t)$ be a shortest path if
a) Weight of each edge is multiplied by a fixed constant $c > 0$.
b) Weight of each edge is incremented by a fixed constant $c > 0$.

Bibliography

[1] O. Borůvka. O jistém problému minimálním. *Práce mor. přírodověd. spol. v Brně III*, 3:37–58, 1926.

[2] O. Borůvka. Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí. *Elektrotechnickỳ obzor*, 15:153–154, 1926.

[3] T. M. Chan. Backwards analysis of the Karger-Klein-Tarjan algorithm for minimum spanning. *Inf. Process. Lett.*, 67(6):303–304, 1998.

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

[5] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

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

[7] T. Hagerup. An even simpler linear-time algorithm for verifying minimum spanning trees. In C. Paul and M. Habib, editors, *Graph-Theoretic Concepts in Computer Science*, volume 5911 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin Heidelberg, 2010.

[8] D. Harel. A linear time algorithm for the lowest common ancestors problem. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 308–319, 1980.

[9] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

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

[11] V. King. A simpler minimum spanning tree verification algorithm. In S. Akl, F. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 440–448. Springer Berlin Heidelberg, 1995.

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

[13] P. N. Klein and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, STOC '94, pages 9–15, New York, NY, USA, 1994. ACM.

[14] J. Komlos. Linear verification for spanning trees. In *Foundations of Computer Science, 1984. 25th Annual Symposium on*, pages 201–206, 1984.

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

[16] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[17] J. Nešetřil, E. Milková, and H. Nešetřilová. Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1–3):3 – 36, 2001. ¡ce:title¿Czech and Slovak 2¡/ce:title¿.

[18] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. In J. Reif, editor, *VLSI Algorithms and Architectures*, volume 319 of *Lecture Notes in Computer Science*, pages 111–123. Springer New York, 1988.

[19] R. Tarjan. *Data Structures and Network Algorithms*, volume 44, chapter 6. Minimum Spanning Trees, pages 71–83. SIAM, 1983.

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

[21] R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, Oct. 1979.