1 Minimum Spanning Trees

Suppose we are given a connected, undirected, weighted graph $G = (V, E)$, and an edge-weight function $w : e \to \mathbb{R}$ that returns the real-valued weight of an edge for each $e \in E$.

A *spanning tree* T is a set of edges $E' \subseteq E$, of size $|E'| = |V| - 1$, such that for every pair of vertices $u, v \in V$, there exists a path from u to v in T.

A *minimum spanning tree* (MST) is defined as a spanning tree T that minimizes the "cost" function

$$w(T) = \sum_{e \in T} w(e) \ .$$

That is, the "cost" of a solution is the sum of the weights of the edges included.

To simplify our analysis, let us assume that edge weights are distinct, that is, $\forall_{e,e' \in E}\ w(e) \neq w(e')$. This assumption is convenient because it guarantees that the minimum spanning tree of the graph will be unique.[1] (Can you prove this is true?)

If there are ties among some of the edge weights, then there may be more than one MST. As an extreme example, consider the case where all the edges have unit weight, i.e., $\forall_{e \in E}\ w(e) = 1$. In this case, not only is there no unique MST, but every spanning tree (including every SSSP) is a MST and the weight of every spanning tree is exactly $w(T) = |V| - 1$.

What good are MSTs? Generally, they are only useful if you need a subgraph T that connects all pairs within the graph G *and* you need that subgraph to have minimum weight.

For instance, suppose we have a set of points embedded in space, e.g., cities in a country, homes in a city, sea ports worldwide, etc., and we want to build a distribution "network" that allows fast transportation of some good, e.g., freight cargo, electricity over wires, plush toys to market, etc., between the different points. Every feasible solution to this question must contain a spanning tree on the vertices, as otherwise, there will be some pair of vertices for which there is no path between them.

[1] More generally, this implies that the function $w(T)$ has a single global minimum over the space of all trees T on G. It does not, however, say anything about the number of alternative trees that are very close in weight to the global minimum, or how similar these alternative trees are to the global optimum. Normally, the number and dissimilarity of these approximately-good solutions is irrelevant, but in applications where an algorithm is not guaranteed to always find the best solution in a reasonable amount of time, as with what we call NP-Hard problems, or when it is not clear that the minimum-weight solution is the best solution for other reasons, the structure of these alternatives can matter a lot.

If cost is no object, then the optimal solution is to connect every pair of points with an edge, i.e., to choose a complete graph on the n nodes. This solution provides the minimum cost route between any pair of points because every pair is connected by a single edge. But, this solution also has the highest cost of all solutions, because every possible edge is included.

On the other hand, if cost matters, even a little, then we can lower our cost by including fewer edges, at the expense of increasing the weight of the path between some pairs of nodes. That is, we will tradeoff slightly less efficient routing for a lower cost solution. A MST provides a minimum-cost solution to this problem that still allows routing between any pair of points. MSTs can also be used as approximate solutions to genuinely hard problems like the Traveling Salesman Problem (TSP), which is NP-Hard.[2]

SSSP tree $\neq$ MST. Note that a single-source shortest-path (SSSP) tree satisfies the requirements to be a spanning tree. That is, if the input graph $G = (V, E)$ is connected, and if T is a solution to the SSSP problem on G (rooted at vertex s), then for all pairs $u, v \in V$, there exists a path from u to v in T.

But, T is not necessarily a minimum spanning tree for G. Observe that there are at most $|V|$ SSSP trees—one for each $s \in V$—while there is at least one MST. They *can* be the same, but they don't have to be. The figure below illustrates this point.

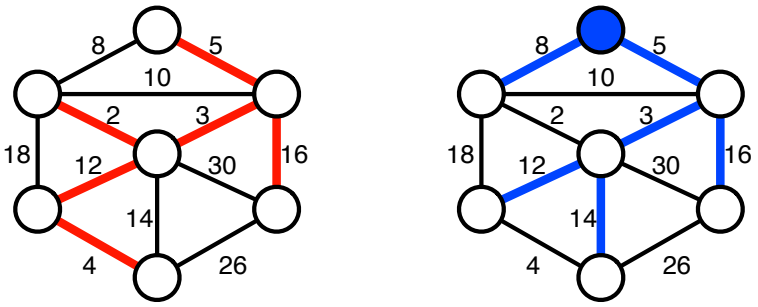


Figure 1: The left-hand figure shows a minimum spanning tree (MST) while the right-hand figure shows a single-source shortest path (SSSP) tree rooted at the blue node. In fact, for this graph, none of the SSSP trees (each rooted at a different vertex) equals the MST.

[2] As in the case of the Christofides algorithm for approximating TSP, first published in 1976. When the input graph represents vertices embedded in a metric space, meaning that distances respect the triangle inequality, Christofides' algorithm is guaranteed to return a tour within a factor of 3/2 of the optimal, and is still the best-known such *approximation algorithm* for this subclass of TSP problems.

1.1 A generic MST algorithm

As with the search-tree problem (BFS, DFS, etc.) and the single-source shortest path (SSSP) problem, different MST algorithms are in fact special cases of a more generic spanning tree algorithm.

Here is the basic structure that all of these algorithms share:

- At each step of the algorithm, we will maintain and update an acyclic subgraph F on the input graph G that we call an *intermediate spanning forest*.
- The forest F is a subgraph of the MST of G, and every component of F is a MST on that component's vertices.
- Initially, F has no edges and thus is a collection of V disconnected singleton components; when the algorithm halts, F is a single V-node tree, which is a MST of G.
- At each step of the algorithm, we add some edges to F (but to guarantee the halting condition, we must be careful about which edge we choose to add).

As the algorithm proceeds, it recognizes two special types of edges, as well as a third less-special type. Here are the definitions:

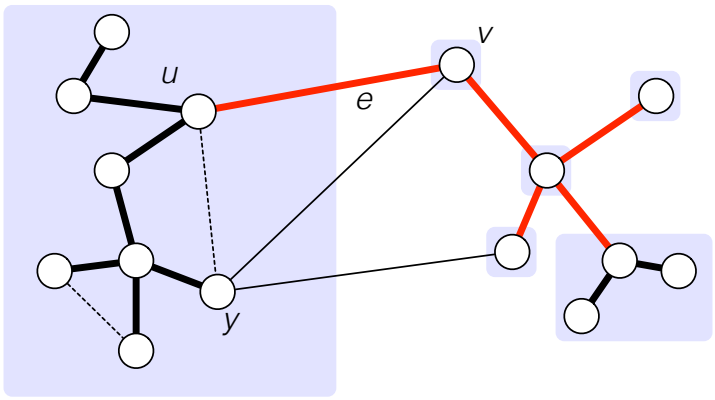
- Call an edge *useless* if it is not an edge in F but both of its endpoints are in the same component of F. Adding a useless edge to F would induce a cycle, and can thus never be part of the MST we are building, which is why we call them useless.
- In contrast, for each component of F, an edge is called *safe* if it is the minimum weight edge among all edges with exactly one endpoint in that component. Because we assumed that edge weights are distinct, a given component can only have one safe edge. Adding a safe edge to F increases the size of the spanning tree and maintains its minimum weight, which is why we call them safe. (If two components in F have the same safe edge, what happens next?)
- Edges that are neither safe nor useless are called *undecided*.

The figure below illustrates these definitions: (i) bold edges indicate edges that are both safe and that have already been added to F, and (ii) dashed lines indicated edges that have already been labeled as *useless*. All other edges are *undecided*: red ones are safe for some component in F, while regular-weight black ones are not.

Given these definitions, we may observe that if F is a MST, then every edge $e \in E$ is either an edge in F or a useless edge. Furthermore, if F is an MST, there are no safe edges and no undecided edges. (Do you see why?) This immediately implies our generic algorithm for building an MST:

While there are any safe edges, add one or more of them to F.

From these observations, we can write down pseudo-code for our generic algorithm:




```
Generic-MST(G,w) {
   F = {}                                  // initially F contains no edges
   while (F not a spanning tree) {         // are we done yet?
      find edge (u,v) that is safe for F   //    make a safe choice
      add (u,v) to F                       //    and grow F
   }
   return F                                // we are done
}
```

At each step of the algorithm, we identify a safe edge and add it to the intermediate spanning forest. When we add an edge to F, some undecided edges become safe while others become useless. (For example, in the figure above, adding e to F makes (y, v) useless.) We stop when F is a spanning tree, which is equivalent to stopping when there are no more safe edges.

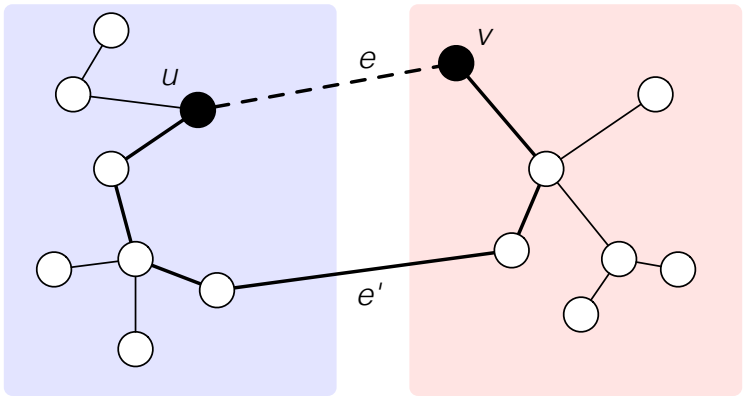
1.2 Correctness

We now prove that this algorithm is correct, by proving that every MST contains only safe edges. The figure below illustrates the idea of a "bad" component (on the left), in which the path from u to v on T' is given in bold and traverses e'.

Claim: A minimum spanning tree contains all safe edges and no useless edges.

Proof: Let T' be a minimum spanning tree. Suppose F has a "bad" component in it, whose safe edge $e = (u, v)$ does not end up in T'. Since T' is connected, it contains a unique path from u to v and therefore there is at least one edge e' on this path that has exactly one endpoint in the bad component. If we replace e' in T' with e, we get a new spanning tree T. By assumption, e is the bad component's safe edge, which implies that $w(e') > w(e)$ and that the new spanning tree T

has smaller weight than T'. However, this contradicts our assumption that T' is a MST; therefore, T' must, in fact, already contain every safe edge and e cannot be a safe edge. Furthermore, if we added any useless edge to F, we would introduce a cycle. □



1.3 Three flavors of MST algorithms

The minimum spanning tree problem is very old—older, in fact, than the term "computer science" (which was first used in 1959). In this section, we will cover three special cases of the generic MST algorithm defined above. The differences between the algorithms come from how we add safe edges to F.

1.3.1 Borůvka's algorithm

One of the earliest solutions to the MST problem is due to Borůvka (1926).[3] Unlike the more commonly known versions, this algorithm does not add safe edges one at a time. In Borůvka's version of `Generic-MST`, F is again a forest. The critical choice is to

add all current safe edges simultaneously and then recurse until there are no safe edges left.

Each recursion is a "phase" and during each phase the algorithm first elects an arbitrary "leader" for each component (it does not matter which vertex is a leader, just that each component has one) and then identifies the set of safe edges. There are several ways to choose the leaders; a simple one is to use the `Search-Forest` algorithm (the generalization of `Search-Tree` that builds a forest of search-trees on G), e.g., DFS.

[3]Depending on the literature, this algorithm is also called Choquet's algorithm, Florek-Łukaziewicz-Perkal-Stienhaus-Zubrzycki's algorithm or Sollin's algorithm, all of whom reinvented it after Borůvka's original 1926 paper.

Here is pseudo-code for Borůvka's algorithm (assume that $w(e) = +\infty$ if $e \notin E$):

```
Boruvka(G,w) {
   F = {V,{}}
   while F has more than one component
      choose component leaders via DFS/BFS   % how long does this take?
      Find-Safe-Edges(G,w)                   % find all the currently safe edges
      for each leader v' {                   % for each component
         add safe(v') to F                   %   add its safe edge to F
}}
```

(The pseudocode for `Find-Safe-Edges` is on the next page.) One nice thing about this algorithm is that it is naturally parallel, in the sense that the identification of safe edges can be done in parallel for each of the components.

```
Find-Safe-Edges(G,w) {
   for each leader v' {          % for each component,
      safe(v') = NULL            %   no safe edge
   }
   for each edge (u,v) in E {
      u' = leader(u)             % look up component name of u
      v' = leader(v)             % look up component name of v
      if u' not= v' {            % are u, v in same component
         if w(u,v) < w(safe(u')) { safe(u') = (u,v) } % update estimated safe edge
         if w(u,v) < w(safe(v')) { safe(v') = (u,v) } %
}}
```

Running time. Note that `Find-Safe-Edges` is essentially a brute-force search to identify which edges are safe for each component. Because it examines every edge, calling `Find-Safe-Edges` takes $O(E)$ time, if the edges are stored in an adjacency list format. (How long if we use an adjacency matrix?) Because we add one safe edge to F for each component, and every safe edge joins two components, each pass through the main loop reduces the number of components by at least half. (Do you see how it could be more than half?) Since there are initially V components, this yields $O(\log V)$ passes through the main loop, each of which takes $O(E)$ time. Thus, the running time of Borůvka's algorithm is $O(E \log V)$.

An example. The following figure shows an example of running Borůvka's algorithm on a small graph. Initially, every vertex is in a component by itself.

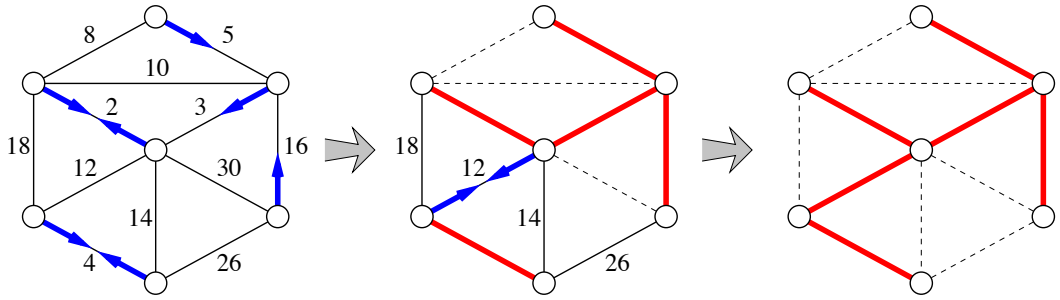


Figure 2: Borůvka's algorithm run on the example graph. The thick (red) edges are in F and the (blue) arrows indicate each component's safe edges. Useless edges are shown as dashed lines. (Figure from Jeff Erickson's lecture notes.)

1.3.2 Jarník's (Prim's) algorithm

The second oldest MST algorithm, and one that typically appears in many textbooks, is originally due to Jarník (1929), but usually bears Prim's name, who rediscovered it in 1957.[4] In Prim's algorithm, `Generic-MST` maintains F as a forest, composed of one tree, which we call T, containing S vertices and $V - S$ singleton components. The critical choice is to

find T's safe edge, add it to T, and recurse.

In this way, we grow the MST out from some initial vertex until it spans the graph (much like the way we solved the SSSP problem).

Initially, T contains only this arbitrarily selected vertex. The algorithm then grows T outward by repeatedly finding and adding T's safe edge. When the algorithm halts, T spans the graph. To implement Prim's algorithm, we simply need an efficient way to find T's current safe edge. This can be done using a min-heap of the edges adjacent to T, using the edge weights as the keys. When we pop the minimum-weight edge from the top of the heap, we need to check whether it is a useless edge (both end-points in T) or not. If not, then it is a safe edge and we both add it to T and add the edges attached to the new member of T to the heap.

An example. This figure shows running Prim's algorithm on the example graph.

Running time. Note that because Prim's algorithm grows the MST outward from a single vertex, it is also a version of the `Generic-SSSP` algorithm we saw in a previous lecture, albeit one that

[4] It was also discovered by Kruskal, Loberman and Weinberger, and by Dijkstra.

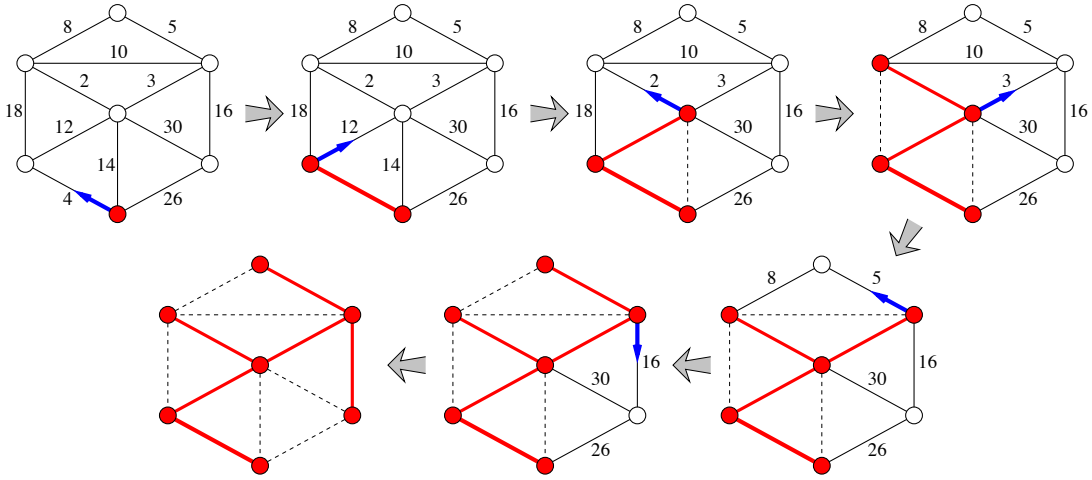


Figure 3: Prim's algorithm, from the bottom-most vertex. Red edges are in F, blue arrows indicate the safe edge, and dashed lines are useless edges. (Figure from Jeff Erickson's lecture notes.)

uses the min-heap as the "set" ADT (this makes it equivalent to Dijkstra's algorithm). As a result, the running time is $O(E \log E) = O(E \log V)$.

But note that Prim's algorithm only visits each vertex once. This means that we can speed up the running time if we are smarter about how we find safe edges. Instead of maintaining a min-heap on the edges attached to T, we can maintain a heap of the vertices themselves, where the key of a vertex v is the length of the minimum-weight edge between v and T (or ∞ if we haven't seen an edge yet). Now, when we add a new edge to T, we might need to decrease the keys of its neighbors.

In this version of Prim's algorithm, the running time is dominated by the heap operations, and in particular, the cost of inserting all V keys into the initial heap, the cost of getting the minimum V times, and the cost of modifying the keys. The first and second of these take $O(V)$ time, and at most, each edge induces a key modification, so there are $O(E)$ of those. If we use a standard binary heap, each operation costs $O(\log V)$, so the overall running time would be $O((V+E) \log V) = O(E \log V)$ for a connected graph. We can do slightly better if we use the Fibonacci heap, which yields a running time of $O(E + V \log V)$.

1.3.3 Kruskal's algorithm

Another popular (as in, it appears in many textbooks) MST algorithm is named for Kruskal (1956). In this version of `Generic-MST`, F is maintained as a forest. The critical choice is to

repeatedly add to F the smallest weight edge that is also safe.

Thus, we sort the edges by their weights and then (repeatedly) scan them in order; we add the first safe edge we find to F and then start scanning again.

Recall that a safe edge is one that has its endpoints in different components. To determine if an edge (u, v) is safe, we use the same approach as Borůvka, i.e., each component is assigned a "leader" and each vertex keeps track of which leader it is following. Unlike Borůvka's algorithm, however, we do not recompute leaders each time we add an edge; instead, we let the old leader follow the new leader, and we follow this leader chain until we find a leader who follows no one. In this way, each component has a unique "leader."[5]
Here is pseudo-code for Kruskal's algorithm:

```
Kruskal(G,w) {
   sort E by weight
   F = {}
   for each vertex v in V { Make-Set(v) }  // each vertex by itself
   for i = 1 to E {                        // scan the edges, in order
       (u,v) = ith lightest edge in E      //   the edge we consider next
       if Find(u) not= Find(v) {           //   is this edge safe?
          Union(u,v)                       //     then, merge their components
          add (u,v) to F
}}}
```

Running time. The running time of Kruskal's algorithm depends on the time it takes to do the `Find` and `Union` operations. If we use the Union-Find algorithm to implement a disjoint set data structure (see Chapter 21 in CLRS), these operations can be made very fast.

This algorithm performs $O(E)$ `Find` operations, one for each end-point of each edge in the graph, and $O(V)$ `Union` operations, one for each edge in the spanning tree. If we use the Union-Find algorithm, these take $O(\alpha(E, V))$ time, where $\alpha(.,.)$ is the inverse-Ackerman function. The main thing you need to know about this function is that it grows extremely slowly[6] and can, for most purposes be treated as a constant function. Thus, the overall running time is dominated not by the `Union` and `Find` operations, but by the time to sort the edges by their weight, which takes $O(E \log E) = O(E \log V)$ time.

[5] Formally, this is equivalent to using the Union-Find algorithm to keep track of the leader hierarchy; see Chapter 21 on Data Structures for Disjoint Sets.

[6] Slower, in fact, than any function you can express in standard algebraic notation. If this sounds crazy to you, and it should, you should take a look at Scott Aaronson's short and highly entertaining essay "Who Can Name The Bigger Number?" which you can find here: `http://www.scottaaronson.com/writings/bignumbers.html`

An example. On the next page is an example of Kruskal's algorithm running on the same graph as we saw in the previous two MST algorithms. (Note that Kruskal's algorithm finds exactly the same MST as the previous two algorithms find.)

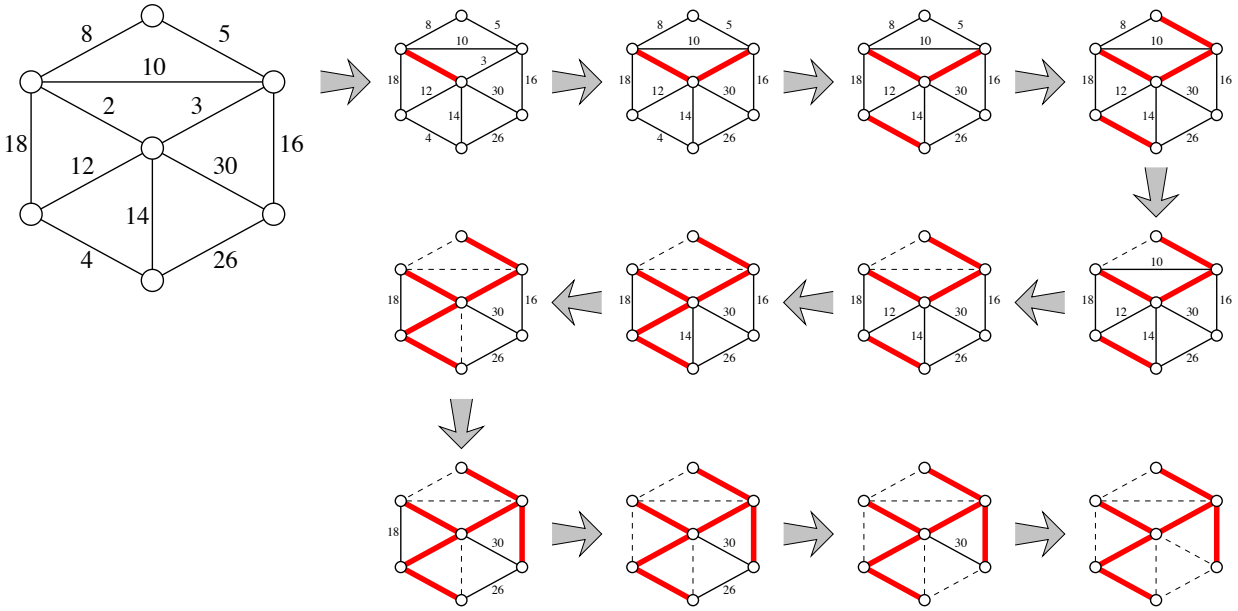


Figure 4: Kruskal's algorithm. Thick red edges are in F and dashed lines are useless edges. (Figure from Jeff Erickson's lecture notes.)

2 On your own

1. Read Chapter 23, Minimum Spanning Trees