Spanning trees

Kent Quanrud

February 24, 2021

Let $G = (V, E)$ be an undirected graph. For a set of edges $F \subseteq E$, and an edge $e \in E$, we say that ***F* spans *e*** if the endpoints of e are connected in F. We say that F is **spanning** if it spans every edge in E. We denote

$$\text{span}(F) = \{e \in E : F \text{ spans } e\}.$$

Observer that $\text{span}(\text{span}(F)) = \text{span}(F)$ for all F.

A **circuit** in F is any nonempty walk that starts and ends at the same vertex. A **forest** is an edge set with no circuits. A **tree** is a forest with one connected component.

Consider the following two problems.

1. Compute a spanning edge set F of minimum cardinality (or weight).
2. Compute a forest F of maximum cardinality (or weight).

We will develop algorithms that solve both of the above problems. As we do so, it is worth paying attention to the *style* of our investigation. We will make a sequence of small, incremental observations about the above definitions. As we get to understand the definitions well, the algorithmic problem will melt away. In fact we will end up with several different algorithms for the above problems, and (if done right) it will be easy to see why they all work.

1 Unweighted graphs

Throughout this section, let $G = (V, E)$ be an undirected graph with m edges and n vertices. In the following, we let κ denote the number of connected components in G.

Lemma 1.1. *Let $F \subset E$.*

1. *If F is spanning, then $|F| \geq n - \kappa$*
2. *If F is a forest, then $|F| \leq n - \kappa$*

Proof. Observe if F is spanning, then every connected component of G must be connected by F. For if not, and there is a connected component of C that is not a connected component of G, then we can find an edge leaving C spanning F.

Thus F has the same number of connected components as G. Now imagine starting with a graph over V with no edges, and adding the edges of F one by one. Initially there are n connected components, one for each vertex. Each edge from F can reduce the number of connected components by 1. If, by contradiction, we have $|F| < n - \kappa$, when there wouldn't be enough edges in F to reduce the number of connected components to the same number as G. So $|F| \geq n - \kappa$.

For the second claim, suppose F is a forest. Imagine starting with an empty graph over V and adding the edges of F one by one. Each edge must decrease the number of connected components by 1, since otherwise we are adding an edge already spanned by other edges in F, and introducing a cycle. On the other hand, the number of connected components by F cannot be less than the number of connected components in G. This implies that $\kappa \leq n - |F|$; rearranging gives the desired inequality. ∎

It follows that if F_1 is a spanning set of edges and F_2 is a forest, then $|F_2| \leq |F_1|$. Can the two cardinalities ever be equal?

Theorem 1. *Let $F \subseteq E$.*

1. *If F is a **minimal** spanning set of edges, then F is a forest.*

2. *If F is a **maximal** forest, then F is spanning.*

In both cases, F is a spanning forest with exactly $n - \kappa$ edges.

Proof. For the first claim, let $F \subseteq E$ be a minimal spanning edge set and suppose by contradiction that F is not a forest. In particular, F contains a cycle. Deleting an edge from a cycle in F does not effect connectivity. But then F is not minimal, a contradiction.

For the second claim, let $F \subseteq E$ be a maximal forest, and suppose by contradiction that F is not spanning. Any edge e not spanned by F can be added to F without introducing a cycle. But then F is not maximal, a contradiction. ∎

All minimal spanning edge have the same size, and therefore are all *minimum* size spanning sets. Likewise all maximal edge sets have the same size, and therefore are all *maximum* size forests. It is extremely convenient that minimal equals minimum and maximal equals minimum – it suggests a sort of discrete analogue of convexity and concavity. We also highlight they way that forests and spanning sets played off each other in the proof of Theorem 1. This is our first example of a more general phenomena called *duality*.

Theorem 1 also gives us two different algorithms for finding a spanning forest:

1. Starting from $F = E$, repeatedly remove edges $e \in F$ as long as $F - e$ remains spanning.

2. Starting from $F = \emptyset$, repeatedly add edges $e \in E \setminus F$ to F as long as $F + e$ remains a forest.

The first algorithm terminates with a minimal spanning edge set, which by Theorem 1 is a forest. The second algorithm terminates with a maximal forest, which by Theorem 1 is also spanning.

An important special case is when a graph is connected. In the case, the spanning forest F is a spanning *tree*.

Lemma 1.2. *Every connected graph has a spanning tree, with $n - 1$ edges.*

2 Distinct weights

Let $G = (V, E)$ be a connected and undirected graph, with edge weights given by $w : E \to \mathbb{R}$. We assume for simplicity that the edge weights are distinct. However the algorithms we develop will work for general weights as well, and we address this at the end of the section. It will also be able to see that the algorithms generalize to unconnected graphs as well, where we find a minimum weight spanning forest rather and not a tree.

Thus, consider the problem of finding the minimum weight spanning tree where we assume the graph is connected and the edge weights are distinct. Our algorithms will all be based on the following key lemma.

Lemma 2.1. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

$$w(e) < \max_{f \in F} w(f).$$

Proof.

(1) $\implies$ (2). Suppose every minimum weight spanning tree T contains e. Suppose by contradiction that (2) does not hold. Let $F \subseteq E$ be a set of edges spanning e where $w(e) > w(f)$ for all $f \in F$. In particular F contains a path P where $w(e) > w(f)$ for all $f \in P$.

Consider the forest $T - e$ obtained by removing e from T. Let $e = \{a, b\}$. $T - e$ has two connected components A and B, where A contains a and B contains b. As a path from $a \in A$ to $b \in B$, P must have an edge f crossing from A to B. This edge f is not spanned by $T - e$, so $T' \stackrel{\text{def}}{=} T - e + f$ is a forest. Since T' has $n - 1$ edges, it is also a spanning tree. It has weight

$$\bar{w}(T') = \bar{w}(T) - w(e) + w(f) \stackrel{\text{(a)}}{<} \bar{w}(T)$$

where (a) is because $w(f) < w(e)$ by assumption. But then T is not the minimum weight spanning tree, a contradiction.

(2) $\implies$ (1). Suppose (2) holds. Suppose by contradiction that T is a minimum weight spanning tree excluding e. Consider the unique path $P \subseteq T$ connecting the endpoints of e. Then P spans e, so P contains an edge f with $w(e) \leq w(f)$.

We claim that $T' = T - f + e$ is a spanning tree. We first observe that $T - f$ does not span e. Indeed, if $T - f$ did span e, then this implies a second path P' in T spanning e distinct from P, a contradiction. Thus $T - f$ is a forest that does not span e, with $n - 2$ edges. $T - f + e$ is then a forest with $n - 1$ edges. Any forest with $n - 1$ edges is a spanning tree.

Thus $T - f + e$. It has weight

$$\bar{w}(T') = \bar{w}(T) - w(f) + w(e) < \bar{w}(T),$$

which means that T is not the minimum weight spanning tree, a contradiction. ■

We note that the lemma holds even when weights are not distinct.

Now, consider the following abstract algorithm where in particular (2.A) is left unspecified and is not necessarily implementable.

generic-MST($G = (V, E), w : E \to \mathbb{R}$)

1. $T \leftarrow \emptyset$
2. until T is a spanning tree
   A. add an edge $e \in E \setminus \text{span}(T)$ satisfying condition (2) in Lemma 2.1 to T
   // *Or throw an error if no such edge exists.*
3. return T

generic-MST is very conservative, only adding edges that we know must be in every minimum weight spanning tree. However it is not clear that these edges are sufficient to find the minimum spanning tree. Still, almost tautologically, we have the following.

Lemma 2.2. *If* generic-MST *terminates, then it returns the minimum spanning tree, which is unique.*

Proof. Indeed, it returns a spanning tree T where, by Lemma 2.1, every edge in T is in every spanning tree. ∎

It remains to show that generic-MST always terminates. In particular, we need to show that an edge satisfying step (2.A) always exists.

2.1 The greedy algorithm

Lemma 2.3. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e satisfies the equivalent conditions of Lemma 2.1.*

Proof. Let $F \subseteq E - e$ be any set of edges spanning e. Since F spans e and H does not, F is not a subset of $\text{span}(H)$. Any edge $f \in F \setminus \text{span}(H)$ must have $w(f) > w(e)$ by choice of e, as required. ∎

The key point to the above lemma is that the set H gives a **certificate** that e should be included the minimum weight spanning tree. Without such a certificate, it is hard to justify taking e. The lemma leads to perhaps our simplest implementation of generic-MST: repeated add the minimum weight edge that does not introduce a cycle, until we have a spanning tree. This called the **greedy algorithm** and is also known as Kruskal's algorithm.

greedy-MST($G = (V, E), w : E \to \mathbb{R}$)

/* *Also know as Kruskal's algorithm* */

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in increasing order of weight.
2. $T \leftarrow \emptyset$
3. for $i = 1, \ldots, n$

```
A. if T + e_i is a forest
   // e_i is the minimum weight edge not spanned by T
   1. T ← T + e_i
4. return T
```

Running time and the disjoint union data structure. Correctness follows immediately from `generic-MST`. Running time is another matter and we briefly review the algorithm to set the stage. The algorithm sorts the edges by weight, and repeatedly adds edges in order as long as they do not introduce a cycle. Sorting is easy. However, checking that we do not introduce a cycle in step (3.A) is not so obvious. Given an edge $e_i = (u, v)$, we need to quickly figure out if the endpoints u and v are in the same components with respect to the current forest. One could do so by running a searching algorithm from u, which leads to a $O(mn)$ running time.

To do better, imagine the connected components of T over the course of the algorithm. Initially, when $T = \emptyset$, every vertex is in its own singleton set. Thereafter, whenever we add an edge e_i to T, we also combine the connected components of the endpoints, replacing the two components with their *union*. Along the way, we are constant checking two vertices are already in the same component.

These operations are facilitated by the **disjoint union** data structure, which has the following interface.

1. `union`(u, v): Given two elements u and v, take the union of the their sets.
2. `same-set`(u, v): Returns whether or not x and y are in the same set.
3. `new-set`(x): Given a new element x, creates the singleton set $\{x\}$.

We plan to cover the disjoint union data structure later when we focus on data structures. For now we treat it as a black box with the following guarantees.

Theorem 2. *There is a data structure that can implement the* `union`, `same-set`, *and* `new-set` *operations listed above that, for any m operations over n total elements, takes $O(m + n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function.*

We refer to the wikipedia article for more on the inverse Ackerman function, suffice it to say that it is asymptotically smaller than $\log(n)$, or $\log(\log(n))$, or $\log(\log(\log(n)))$, or $\log(\log(\log(\log(n))))$...

```
greedy-MST(G = (V, E), w : E → ℝ)
/* Also know as Kruskal's algorithm */
1. Initialize a new disjoint-union data structure U
2. call U.new-set(v) for all v ∈ V
3. Sort and index E = {e_1, ..., e_n} in increasing order of weight.
4. T ← ∅
```

5. `for` $i = 1, \ldots, n$

   A. `let` $e_i = \{u, v\}$

   B. `unless same-set`(u, v)

      1. $T \leftarrow T + e_i$

      2. U`.union`(u, v)

6. `return` T

Now we put everything together.

Theorem 3. *The greedy algorithm computes the MST in $O(m \log n)$ time.*

Proof. We have already addressed the running time above in Lemma 2.2. For the running time, we observe that we make $O(m)$ calls to the `disjoint-union` data structure to maintain disjoint sets over n elements. The total running time from these operations is $O(m + n\alpha(n))$. The remaining operations take $O(m \log n)$ time, where the bottleneck is from sorting. ■

Note that the $O(m + n\alpha(n))$ bound for the `disjoint-union` data structure was overkill, since sorting was already a bottleneck.

2.2 A parallel algorithm

For a set of vertices $S \subset V$, the **cut** induced by S, denoted $\partial(S)$, is the set of edges with exactly one endpoint in S:

$$\partial(S) \stackrel{\text{def}}{=} \{\{u, v\} \in E : u \in S,\ v \notin S\}$$

The name "cut" comes from the fact that removing $\partial(S)$ from G cuts off S from the rest of the graph.

Lemma 2.4. *Let $S \subset V$ be any set of vertices. Then the minimum weight edge in the cut induced by S satisfies condition (2) of Lemma 2.1.*

Proof. Let $H = E - \partial(S)$. Then $E \setminus \operatorname{span}(H) = \partial(H)$. The claim now follows from Lemma 2.3. ■

This inspires the following instantiation of `generic-MST` with a parallel character: each round, for each connected component S of the current tree T, identify the smallest weight edge in $\partial(S)$. Add all of these edges to T.

`parallel-MST`$(G = (V, E), w : E \to \mathbb{R})$

/* *More popularly known as Borůvka's algorithm.* */

1. $T \leftarrow \emptyset$

2. `while` $|V| > 1$

   A. `for each component` S `of` T`, identify the smallest weight edge in` $\partial(S)$`.`

B. add all of these edges to T.

3. return T

Theorem 4. `parallel-MST` *computes the minimum spanning tree, and can be implemented to run in* $O(m\log(n))$.

Proof. Correctness follows from Lemma 2.2 and Lemma 2.4. For the running time, we first observe that there are at most $O(\log(n))$ iterations of the outer loop. This is because each iteration reduces the number of components by at least a factor of 2.

Next we claim that each iteration can be implemented in $O(m)$ time. To build some intuition, we point out that this is very easy in the first round: each vertex scans its list of incident edges to identify the smallest weight edge incident to that vertex. In subsequent iterations, some additional effort is required to do this on a per-component basis, but it is not too difficult either. By running BFS or DFS, we identify the connected component of each vertex, labeling the vertex with some identifier for the component. (e.g., a unique integer.) We then run through the edges and relabel the endpoints by the corresponding endpoints. Then, similar to the first iteration, it is easy to find the smallest edge cut by each component. ∎

Benefits beyond running time. Borůvka's algorithm is clearly very simple, and although it does not give the fastest theoretical running time for MST (as we will see), it is reported to work very well in practice[1]. Another important property of Borůvka's algorithm it is that it is inherently *parallel*. The algorithm takes $O(\log n)$ time on most distributed/parallel models of computation.

2.3 The search algorithm

Our final algorithm, like the previous one, also certifies its decisions by only taking the minimum weight edge of cuts induced by components of the spanning tree. Rather than adding many edges in parallel, this algorithm focuses on one component (which is initially one vertex) and keep adding the minimum weight edge to add the next component. Finding the next edge can be implemented very efficiently with a Fibonacci heap.

search-MST($G=(V,E), w: E\to\mathbb{R}$)

// Better known as Prim's algorithm

1. $T \leftarrow \emptyset$, $S \leftarrow \{v\}$ for an arbitrary vertex v

// We maintain the invariant that S is the connected component of v in T.

2. while T is not a spanning tree

   A. let e be the minimum weight edge in $\partial(S)$

   B. $T \leftarrow T+e$, $S \leftarrow S\cup e$

[1]We also point out that the theoretically fastest algorithms [1, 4], not covered here, are based on Borůvka's algorithm.

// The above steps can be made very efficient by using a Fibonacci heap.

3. `return` T

Theorem 5. `search-MST` *returns the MST, and can be implemented to run in* $O(m + n \log n)$ *time.*

Proof. We maintain a Fibonacci heap [3] over $V \setminus S$. For each vertex $v \in V \setminus S$, we use the minimum weight of any edge from v to S as the priority for v. This may be $+\infty$ if (early on) there is no such edge, and can be revised as more vertices are added to S. Wit this setup, `remove-min` returns the next vertex to add to S.

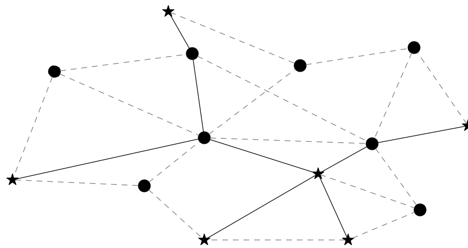
We call `decrease-key` once for every edge, and `remove-min` once for every vertex. This gives an overall running time of $O(m + n \log n)$. ■

2.4 When weights are not distinct

Above we assumed the edge weights are distinct which simplified the analysis substantially. In particular, when edge weights are distinct, the minimum spanning tree is unique.

We can drop this assumption without redoing the analysis, by the following though experiment. Consider any of the concrete algorithms above. The algorithm - taking minimum weight edges in different ways - still mostly make sense, although the choice of edge may not be unique. The one caveat is Borůvka's algorithm, but as long as we break ties consistently, it is sensible. That said, consider any fixed run of one of these algorithms, which produces a tree T. As a thought experiment, we can perturb the edge weights so that (a) the edge weights are unique, and (b) the algorithm would still returns the same set of edges. One such way is to add ϵi to the weight of an edge taken in the ith iteration, and ϵn to the weight of any edge not taken, where $\epsilon > 0$ is sufficiently small such that this change only acts as a tiebreaker.

3 Steiner trees



We now turn to a natural and useful generalization of spanning trees called **Steiner tree**. Let $G = (V, E)$ be an undirected graph. Let $T \subset V$ be a subset of vertices called **terminals**. A **Steiner tree** over T is a tree $F \subseteq E$ in which U is connected. For example, a spanning tree is a Steiner tree for $T = V$. Above is a picture of a Steiner tree where terminals are indicated by $\star$'s, and excluded edges are drawn dashed.

Given positive edge weights $w : E \to \mathbb{R}$, the **minimum weight Steiner tree** problem is to find the Steiner tree whose edges have minimum total weights. Minimum weight Steiner tree is a very

important problem in network design – interpreting the edge weights as costs, the minimum weight Steiner tree is the minimum cost network that connects a fixed set of points. Similarly it is very useful in automated VLSI design.

Theorem 6. *The Steiner tree problem is NP-Hard, even in unweighted graphs.*

Proof. We describe a reduction from 3-SAT. Let $f(x_1, \ldots, x_n)$ be a 3CNF with m clauses $C_1, \ldots, C_m$.

1. For each variable x_j, a terminal vertex.
2. For each clause C_i, a terminal vertex.
3. For each assignment such as $x_j = \texttt{true}$ or $x_j = \texttt{false}$, a (non-terminal) vertex.

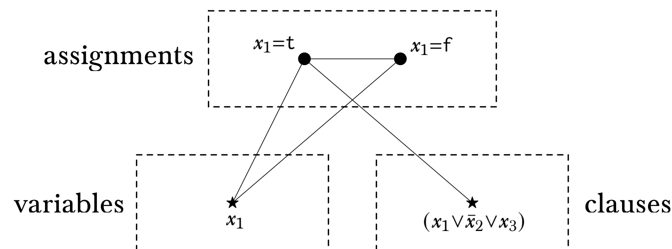
This creates a total of $3n + m + 1$ vertices. Next we create the edges.

4. An edge between every variable and its two assignments.
5. An edge between every clause and every satisfying assignment.
6. An edge between every pair of assignments.

This creates

$$2n + 3m + \binom{2n}{2} = O\left(m + n^2\right)$$

edges. A high-level diagram of the construction is given below.



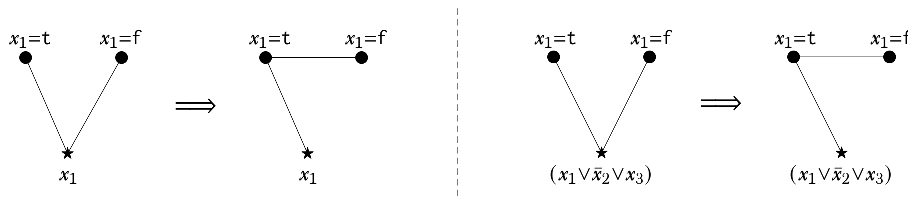
The idea behind the proof is as follows. Ultimately, we need to connect the variables and clauses, via assignments. Let us say that a Steiner tree "takes an assignment" if it the edge between the corresponding edge and assignment is included in the tree. The terminals corresponding to the terminals requires the Steiner tree to take at least one edge incident to each variable. The terminals corresponding to each clauses requires us to take at least one edge incident to every clause. We can also assume that every minimum Steiner tree has exactly one edge to each variable and to each clause, because when there is more than one, we can replace one of them with an edge between the corresponding assignments. This implies that every Steiner tree contains at least $m + n + n - 1 = m + 2n - 1$ edges – n for the variables, m for the clauses, and then (at least) $n - 1$ to connect the assignments of the variables to one another. A satisfying assignment for f maps to a Steiner tree with $m + 2n - 1$ edges by connecting the clauses and variables to the corresponding assignments with $m + n$, and then connecting the n assignments we are using arbitrarily with $n - 1$

edges. Conversely, any Steiner tree with $m + 2n - 1$ edges can be converted to one using exactly n assignments, one per variable, that gives a satisfying solution for f.

We now proceed through the proof more carefully. We claim that f is satisfiable iff there is a Steiner tree of size $m + 2n - 1$.

Suppose we have a satisfying assignment for x. We take the corresponding n edges between variables and assignments. For each clause, we select an edge corresponding to the single-variable assignment that satisfied that clause. Then we add any spanning tree over the assignment vertices, which adds $n - 1$ edges. This gives a Steiner tree of size $m + 2n - 1$.

Now, consider any Steiner tree T in the graph. We claim that if $|T| \leq m + 2n - 1$, then f is satisfiable. We first claim that we can modify T so that each variable is incident to exactly one edge, and each clause is incident to exactly one edge, in T. Indeed, suppose T has both edges incident to a variable x_i. We can delete one of those edges and replace it with the edge between the two assignments $x_i = \texttt{true}$ and $x_i = \texttt{false}$, and T remains a Steiner tree. Likewise if a clause is two edges incident to two different satisfying vertex assignments, we can delete one of these edges and replace it with an edge between the assignments. See the diagrams below.



Thus, for any Steiner tree T, we may assume that each variable and each clause is incident to exactly one edge. Consider just these $m + n$ edges, and the assignments that are at the opposite endpoints. Suppose there are k of them. None of these assignments are connected to each other by the $m + n$ edges, and is at least one for each variable x_j. Thus we have $k \geq n$ connected components, and the rest of T must have at least $k - 1 \geq n - 1$ edges to connect them. That is $|T| \geq m + n + k - 1 \geq m + 2n - 1$. We have $|T| = m + 2n - 1$ iff $k = n$, iff we are using one assignment for every variable. These assignments give a satisfying assignment for the SAT formula. ■

4 Exercises

Please see [2, Chapter 7] for further exercises.

References

[1] Bernard Chazelle. "A minimum spanning tree algorithm with Inverse-Ackermann type complexity". In: *J. ACM* 47.6 (2000), pp. 1028–1047. Preliminary version in FOCS, 1997.

[2] Jeff Erickson. *Algorithms*. 2019. URL: http://jeffe.cs.illinois.edu/teaching/algorithms/.

[3] Michael L. Fredman and Robert Endre Tarjan. “Fibonacci heaps and their uses in improved network optimization algorithms”. In: *J. ACM* 34.3 (1987), pp. 596–615. URL: https://www.cl.cam.ac.uk/teaching/1112/AlgorithII/1987-FredmanTar-fibonacci.pdf. Preliminary version in FOCS, 1984.

[4] David R. Karger, Philip N. Klein, and Robert Endre Tarjan. “A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees”. In: *J. ACM* 42.2 (1995), pp. 321–328.