CS 341 - Algorithms, Winter 2023. University of Waterloo

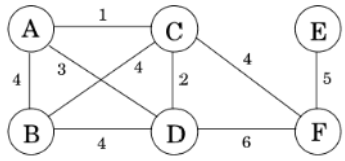
Lecture 10: Minimum Spanning Trees

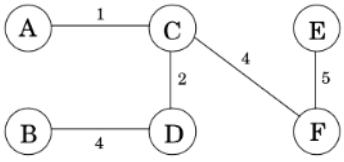
We design and analyze greedy algorithms for the classical problem of finding a minimum spanning tree. This is a good example where we first observe some mathematical property of the problem, then use it to come up with (different) algorithms. then implement them efficiently using data structures.

Minimum Spanning Trees

Input: An undirected graph $G=(V,E)$ with a cost $c_e>0$ on each edge $e \in E$.

Output: A minimum-cost spanning tree T, where the cost of T is defined as $\sum_{e \in T} c_e$.





[DPV, chapter 5]

This is a basic network optimization problem where the goal is to find a cheapest way to build a subgraph that connects all the vertices of a graph.

When the cost on the edges are positive, any optimal solution must be a spanning tree because of the following simple property.

Property Removing an edge of a cycle in a connected graph cannot disconnect the graph.

Cheapest Edge

The most tempting greedy strategy is to choose an edge of minimum cost.

Could it go wrong? The following claim shows that it can't go wrong.

The proof is by a simple exchange argument.

Claim 1 There is a minimum spanning tree that contains a cheapest edge.

Proof Let $e=uv$ be a cheapest edge, and T be a minimum spanning tree. If $e \in T$, then we are done. So suppose $e \notin T$. Consider $T+e$.

Since there is a (unique) path connecting u and v in T, there is a (unique) cycle containing e in $T+e$. Let $f \neq e$ be an edge in this cycle, then $T+e-f$ is connected (by the property) and a spanning tree. As e is a cheapest edge, $\text{cost}(T+e-f) \leq \text{cost}(T)$ and so $T+e-f$ is also a minimum spanning tree, and $T+e-f$ contains e, thus proving the claim. □

Actually, this simple claim alone already leads to a polynomial time algorithm for finding a MST. The idea is to find a cheapest edge $e=uv$, then "contract" its two endpoints to obtain a graph with one fewer vertices, and recursively find a MST T' using the same procedure (i.e. repeatedly find a cheapest edge and contract it), and return $T'+e$ as a MST in the original graph.

We leave it to the reader to prove the correctness of this algorithm formally.

The reason that we don't do this is that directly implementing this idea doesn't give as fast an algorithm. We will see that the Kruskal's algorithm that we will present later is essentially the same algorithm.

Cheapest Edge on a Vertex

The above claim is not as easy to use because it only concerns the cheapest edges of the whole graph. Let's generalize the idea so that it can be applied more broadly to other edges.

Claim 2 For each vertex v, there is a minimum spanning tree containing a cheapest edge incident on v.

We won't prove this claim as we will prove a more general claim and also the proof is similar to Claim 1. This claim leads to a fast MST algorithm, called the Borůvka's algorithm, the first MST algorithm according to wikipedia.

Assume the edge costs are distinct. The idea is to add the cheapest edge incident to each vertex to the (partial) solution.

This will form a forest F where each vertex is of degree at least one, and hence at least $n/2$ edges. Then, we can "contract" each component in this forest F into a single vertex, recursively applying the same procedure to find a MST T' in the contracted graph, and return $T' \cup F$ as a MST.

This algorithm can be implemented in $O(m \log n)$ time, as each iteration can be implemented in $O(m+n)$ time to find and contract the forest, and there are at most $O(\log n)$ iterations as each iteration we decrease the graph size by at least half.

We leave the correctness proof and the time complexity analysis to the reader, as the correctness proof also follows from a more general claim that we will prove later.

Cheapest Edge of a Cut

We prove a generalization of Claim 2 that is even more flexible to use.

A subset of vertices $\phi \neq S \subset V$ defines a bipartition of the vertex set $(S, V-S)$.

We say the set of edges with one endpoint in S and one endpoint in $V-S$ the cut of S.

denoted by $\delta(S) := \{ uv \in E \mid u \in S, v \notin S \}$.

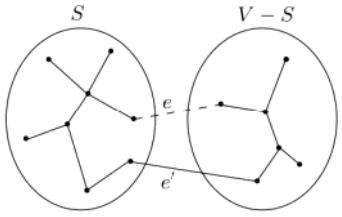
Claim 3 For every subset $\emptyset \neq S \subset V$, there is a minimum spanning tree containing a cheapest edge in $\delta(S)$.

Note that Claim 2 is a special case where $S = \{v\}$ is a singleton.

We also won't prove Claim 3, but a slightly more general version that is more useful in analysing algorithms.

The Cut Property Suppose edges in $F \subseteq E$ are part of a MST of $G = (V, E)$. For any $\emptyset \neq S \subset V$ with $\delta(S) \cap F = \emptyset$ (i.e. no edges in F are in the cut of S), let e be a cheapest edge in $\delta(S)$, $F + e$ is part of some MST.

Proof The proof is by a simple exchange argument.



Let T be a MST that contains the edges in F.

If $e \in T$, then we are done. So assume $e \notin T$. Consider $T + e$.

There is a unique path P connecting the two endpoints of e in T, one endpoint in S and the other endpoint in $V - S$.

Therefore, there must exist an edge f in P with one endpoint in S and one endpoint in $V - S$.

That is, $f \in \delta(S)$. Since e is a cheapest edge in $\delta(S)$, it holds that $\text{cost}(e) \leq \text{cost}(f)$.

This implies that $T' = T + e - f$ is also a MST in G.

Finally, note that $F + e \subseteq T'$ as $F \subseteq T$ and $f \notin F$ because $f \in \delta(S)$ but $F \cap \delta(S) = \emptyset$. □

Algorithms

With the cut property, we can prove that several algorithms work to find a minimum spanning tree.

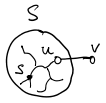
Prim's algorithm

The idea of Prim's algorithm is to start from an arbitrary vertex s and to grow the component containing s one vertex at a time.

$F := \emptyset$, $S = \{s\}$.

while $S \neq V$ do

let $e = uv$ be a cheapest edge in $\delta(S)$ with $u \in S$ and $v \notin S$.



$F \leftarrow F + \{e\}$, $S \leftarrow S + \{v\}$.

return F

The correctness of Prim's algorithm follows from repeatedly applying the cut property.

We will discuss how to implement Prim's algorithm efficiently in the next section.

Kruskal's algorithm

The idea of Kruskal's algorithm is to consider the edges from cheapest to most expensive, and add an edge to the solution as long as it doesn't create a cycle.

$F := \emptyset$.

Sort the edges in non-decreasing cost so that $cost(e_1) \leq cost(e_2) \leq \cdots \leq cost(e_m)$.

for $1 \leq i \leq m$ do

    if $F + e_i$ does not have a cycle, then $F \leftarrow F + e_i$.

return F

Note that the correctness of Kruskal's algorithm also follows from repeatedly applying the cut property, because if $F + e_i$ does not create a cycle, then e_i is a cheapest edge leaving a component of F, and so by the cut property $F + e_i$ is also contained in some MST.

We will discuss how to implement Kruskal's algorithm efficiently in the next section.

Remark: The correctness of Kruskal's algorithm also follows from the "cycle property", which says that a most expensive edge in a cycle can be removed from the graph and the remaining graph still has a MST. See (4.20) in [kT] or wikipedia for a proof.

Borůvka's algorithm

We can also prove the correctness of Borůvka's algorithm by repeatedly applying the cut property. We leave it as an optional exercise.

Reverse-delete algorithm

The idea is to keep removing a heaviest edge as long as the remaining graph is still connected. The correctness of this algorithm can be established by repeatedly applying the cycle property. We also leave it as an optional exercise.

Implementations

Finally, we show how to implement the algorithms efficiently using appropriate data structures.

Prim's algorithm

Note that Prim's algorithm has exactly the same structure as Dijkstra's algorithm, as both involve

adding a vertex $v \notin S$ "closest" to S and growing $S \leftarrow S + \{v\}$.

In Dijkstra's algorithm, for each $v \notin S$, we defined $dist[v] = \min_{w \in S} \{ dist[w] + l_{wv} \}$ and add the vertex $v \notin S$ with smallest $dist[v]$ into S.

In Prim's algorithm, for each $v \notin S$, we can define $connect[v] := \min_{w \in S} \{ c_{wv} \}$ and add the vertex $v \notin S$ with smallest $connect[v]$ into S.

```
connect(v) = ∞ for every v ∈ V , connect(s) = 0 .
F := φ , S := {s} , parent[s] = s
Q = make-priority-queue (V)          // with key value of v being connect[v]

While Q is not empty do
    u = delete-min (Q)
    for each neighbor v of u
        if cost(uv) < connect(v)
            connect(v) = cost(uv)
            decrease-key (Q, v)
            parent(v) = u
    S ← S + {u} , F ← F + (u, parent[u]).
```

Time complexity: This is the same as Dijkstra's algorithm, with running time $O((m+n)\log n)$ as each each priority queue operation can be done in $O(\log n)$ time.

kruskal's algorithm

For kruskal's algorithm, the main step that we need to speed up is to check whether the two endpoints of an edge uv belong to the same component in the forest F or not.

There is an interesting data structure called "disjoint sets" that supports the following operations:

- makeset (x) : create a singleton set containing just x.
- find(x) : return which set that x belongs to.
- union (x,y) : merge the sets containing x and y.

All of these operations can be done in $O(\log n)$ time when there are at most n elements.

With this disjoint-sets data structures, we can implement kruskal's algorithm as follows.

```
F := φ
```

makeset(v) for each $v \in V$.

Sort the edges by cost so that $\text{cost}(e_1) \le \text{cost}(e_2) \le \cdots \le \text{cost}(e_m)$

for $1 \le i \le m$ do

  let $e_i = uv$

  if find(u) $\neq$ find(v),

    then $F \leftarrow F + \{uv\}$, union(u,v)

return F

Time complexity: As each operation of the data structure can be done in $O(\log n)$ time, it is clear that the total time complexity is $O((m+n)\log n)$.

The details of the disjoint-set data structure can be found in [DPV 5.1] and [KT 4.4].

Borůvka's algorithm

This can be implemented in $O((m+n)\log n)$ time without any non-trivial data structures.

We leave it as an optional exercise. (I wonder why this algorithm is not discussed more in books...)

References: [DPV 5.1], [KT 4.4].