NOI.PH 2023 Training Week 18

Minimum-cost spanning trees

Cisco Ortega

August 31, 2023

Contents

1 Spanning Trees

1.1 Trees and Spanning Subgraphs

Let's recall what a tree is, to graph theorists.

Definition 1.1. A tree is an undirected graph in which there exists a *unique* path connecting any two pairs of vertices.

Let's also recall some useful properties of trees.

Claim 1.2. An undirected graph is a tree if and only if:

- They are connected.
- They are acyclic.
- If they have n vertices, then they have $n - 1$ edges.

Proof. We'll show that a tree has these three properties. The other direction is left as an exercise (but it's pretty similar to this argument).

Connectedness directly follows from the definition.

Acyclicness can be proved by contrapositive—if there were a cycle in the graph, then there would exist a pair of vertices with more than one path between them ("go left" or "go right" in the cycle).

The last property is slightly more nuanced to prove, but the form of the statement nudges us towards induction. First, note that a graph with 1 vertex and 0 edges counts as a tree.

All trees have a vertex whose degree is 1 (so-called "leaf nodes"): Pick an arbitrary vertex u and run a DFS from it. Because the tree is acyclic, when one branch of the DFS terminates, it must be at a vertex with no neighbors except the one you just came from.

But if you remove that leaf and the edge connecting it to the tree, then the remaining subgraph (with $n - 1$ vertices) must be a tree, which (by inductive hypothesis) has $n - 2$ edges. Add the leaf back in, and we see that the original tree has n vertices and $n - 1$ edges. $\square$

Exercise 1.3. Prove the other direction: That an undirected graph with these three properties is a tree.

Hint: You only need the connected and acyclic properties.

In fact, in some settings, these properties may be used as the *definition* of a tree. Interestingly, only two of the three properties are needed in order to know that a graph is a tree.

Exercise 1.4. Show that an undirected graph with *any two* of the properties in Claim 1.2 implies that it necessarily satisfies the third one.

Exercise 1.5. For each property in Claim 1.2, construct an undirected graph which has that property but **not** the other two.

The important intuition to take away about trees is that they are quite sparse. In fact, they are the *sparsest* that a graph can be while still being connected.

For example, let's consider a scenario such as the following problem.

Problem 1.6. A country has n cities, labeled 1 to n. There are m roads in this country, the ith of which bidirectionally connects the cities u_i and v_i. Initially, it is possible to start at any city and travel to any other city, using only the roads.

In order to cut down on costs, the national government wants to choose some subset of the roads such that:

- We keep **only these roads**, and close down all other roads
- It should still be possible to start at any city and travel to any other city, using only the *remaining* roads.
- Among all such solutions, the total number of remaining roads is minimized.

Intuitively, if we want to preserve the "connectedness" of a graph, then only some of the edges matter—some edges can be thrown away while still keeping the graph connected. And in fact, if we take a look at any minimum-sized subset of edges that keeps the graph connected, then we'll find that these edges form a tree!

Definition 1.7. Given an undirected graph G, a **spanning tree** is any subgraph of G that includes all of G's vertices and is also a tree.

Claim 1.8. A graph is connected if and only if it has a spanning tree.

Proof. If a graph is not connected, then no subgraph of it can be connected, which is a necessary condition of a tree.

If a graph *is* connected, then we can constructively create a spanning tree for it.

Begin by throwing away all of G's edges. Then, reintroduce the edges one by one. If an edge would connect two vertices in separate connected components, then insert it; otherwise, throw away this edge.

This subgraph is connected because the original graph was connected, and our algorithm only threw away an edge (u, v) if there was already a path from u to v; thus the connectivity of the subgraph is the same as that of the original graph. This subgraph is also acyclic, by construction. Therefore, this subgraph is a tree. □

Note that we said "*a*" spanning tree of a graph. In general, graphs are not guaranteed to have a unique spanning tree. For a simple counterexample, consider a cycle graph—remove any of its edges will yield a spanning tree.

Exercise 1.9. Construct an undirected graph with n vertices and $\mathcal{O}(n)$ edges such that the number of spanning trees it has is exponential in n.

Remark 1.10. *Counting* the number of spanning trees of an arbitrary undirected graph can actually be done in polynomial time using something called Kirchhoff's matrix tree theorem[a]. It says that the number of spanning trees can be counted by finding the determinant of a special matrix related to the graph. There are straightforward $\mathcal{O}(n^3)$ algorithms for finding the determinant of a matrix, and if you accept that such methods exist, then the algorithm described by the matrix tree theorem is not too hard to implement.

That said, we won't be proving *why* it works here, as it is wildly out of scope. You do not need to force yourself to memorize this, since it is unlikely to appear in most competitions. It's just a neat fun fact!

[a] Yes, the same Kirchhoff as in "Kirchhoff's circuit laws".

The spanning tree gives us a sparse connected subgraph, and it's not too hard to show that we can't get any sparser.

Claim 1.11. Suppose a graph has n vertices and m edges. If $m < n - 1$, then the graph cannot be connected.

Proof. Start with the empty graph, which consists of n connected components (each an isolated vertex).

Now, introduce the edges one by one. Each edge either unites two connected components (lowering the total number of CCs by 1), or connects two nodes already in the same connected component (keeping the number of CCs the same).

In any case, if $m < n - 1$, then there are not enough edges to bring the number of connected components down to 1, therefore the graph cannot be connected. □

We now have the standard machinery that allows us to answer Problem 1.6.

Solution. (to Problem 1.6). Since the graph is connected, it has a spanning tree. Take any of them (such as by using the algorithm described above).

This spanning tree uses $n - 1$ roads. As proved earlier, no connected subgraph exists with $n - 2$ roads or fewer, therefore the spanning tree is an optimal solution to this problem.

1.2 DFS and BFS trees

Before proceeding, we'll briefly go over two other special kinds of spanning trees, called the DFS and BFS trees. I just want you to be exposed to these ideas for the sake of familiarity, and also so that you will be prepared for more advanced algorithms that do leverage these properties. It's also good practice for playing with graphs and thinking about spanning trees!

First, note that any graph traversal (such as DFS and BFS) induces a spanning tree on a connected graph.

Claim 1.12. Suppose we have a connected undirected graph. Consider the subgraph of it that is generated in the following manner:

- Traverse the graph (either via BFS or DFS)
- Connect each node u (besides the source) to its parent (i.e. when node u is visited for the first time, take the edge connecting it to the node you just came from).

This subgraph is a tree.

Proof. The proof is straightforward.

- The subgraph is connected because all nodes have a path to the source node.
- The subgraph has all n nodes, and $n - 1$ edges (every non-source node takes the edge to its parent).
- By Exercise 1.4 it also satisfies the third property in Claim 1.2, therefore, the subgraph is a tree.

$\square$

Definition 1.13. The *DFS tree* and *BFS tree* are the spanning trees induced by the algorithm in Claim 1.12 (for DFS and BFS, respectively). They are *rooted* trees, with their root being the starting vertex of the graph traversal.

In short, this just means that if you need *any* spanning tree, then it will already suffice to simply run DFS or BFS.

Here's a special property of the DFS tree in particular.

Claim 1.14. In a DFS tree, all edges of G not included the DFS tree connect a node with one of its ancestors in the tree.

Remark 1.15. This property of the DFS tree is greatly useful in developing bridge-finding algorithms, such as the one described in this blog.

The BFS also has a few special properties which may be helpful when problem solving.

Claim 1.16. In a BFS tree, suppose an edge connects vertices u and v. Let the depth of a vertex be the number of edges between it and the root. Then, the depth of u and the depth of v differ by at most 1.

As a corollary, BFS trees (unlike DFS trees) *have no* node-to-ancestor edges among the unchosen edges.

Claim 1.17. Suppose you label each node $1, 2, 3, \ldots, n$ in the order that you encounter them during the BFS traversal. Then, each *level* of the tree (i.e. nodes of the same depth) consists of a contiguous interval of label numbers.

Exercise 1.18. Prove Claim 1.14 to Claim 1.17. These follow from the properties of the DFS/BFS algorithm itself.

As a final aside, we note that since Dijkstra's algorithm is just a fancy BFS, it too induces a spanning tree on a connected undirected graph with non-negative edge weights. This tree is typically referred to as a *shortest-path tree.*

I won't dwell on it further here, since that would be too much of a tangent into Dijkstra's algorithm, which isn't the point of this module. But perhaps having the "tree-ness" of Dijkstra's pointed out will inspire you when solving shortest-paths problems in the future!

2 Minimum-Cost Spanning Tree

2.1 Finding an MCST

Now, let's move on to *weighted* undirected graphs.

Problem 2.1. An archipelago consists of n islands, labeled 1 to n. The local government is interested in opening up ferry services between certain pairs of islands. Initially, no ferries are running. There are m candidate ferry routes, the ith of which would bidirectionally connect the islands u_i and v_i, and would cost w_i pesos to operate. Here, each w_i is a positive integer.

Which ferries should the government select, such that:

- It should be possible to travel from any island to any other island just by using the ferry service
- Among all such solutions (if they exist), the total cost of the chosen ferry services is minimized.

In general graph theory terms, we want to construct a graph that is *connected* at the minimum possible cost. And since a *tree* is the sparsest possible graph that is still connected, that's what we should aim to construct.

Claim 2.2. If the task in Problem 2.1 is possible, then any optimal solution must be a tree.

Proof. Suppose we construct a connected graph that is not a tree—then, it has a cycle in it somewhere. But we can remove an edge from that cycle while still keeping the graph connected, and since all edge weights are positive, this would yield a solution with an even lower cost. □

It turns out that this problem can be solved with a greedy algorithm.

Solution. Model this as a graph problem. Let the islands be vertices, and let the ferry services be potential edges. Initially, the graph is empty.

Sort these edges in non-decreasing order of their edge weight. Then, for each edge (in order), do the following:

- Suppose this edge connects vertices u and v
- If u and v currently belong to separate connected components, then insert this edge into the graph.
- If u and v already belong to the same connected component, then ignore this

edge.

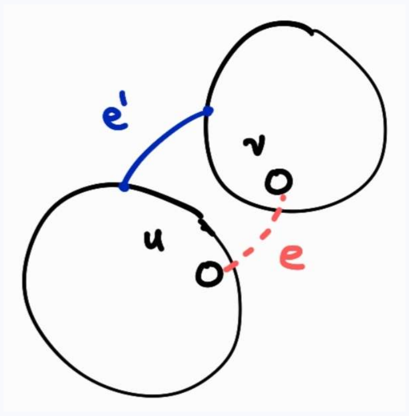
A union-find data structure is required in order to be able to quickly test for connectivity while allowing for edge insertions.

If the entire graph is still not connected after all the edges have been processed, then the task is impossible.

We'll argue that our greedy algorithm is correct by showing it never makes a sub-optimal choice.

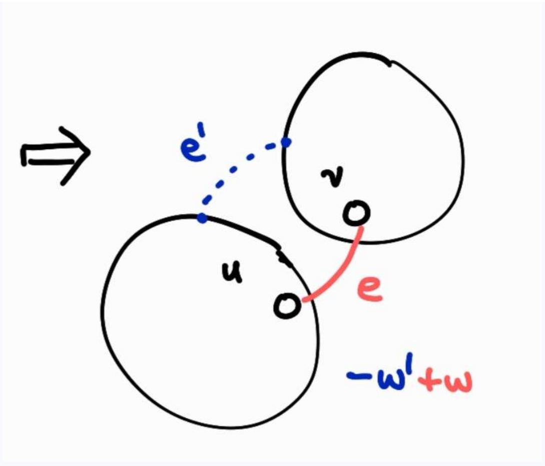
Let's suppose there is an edge e between vertices u and v whose weight is w, and that these vertices currently belong to separate connected components. Our algorithm tells us to include this edge in our solution.

Let's consider a hypothetical solution that opted to *not* just take this edge right now. Well, we want the graph to be connected, so we're going to have to eventually end up (indirectly) connecting u and v together at some point. Let's say that in this solution, the connected components containing u and v are connected later on by a later edge e' with weight w'.



But actually, I can think of a way to improve the alternate solution:

- Disconnect e'
- Include e instead



You can verify that this modified solution still produces a connected graph. But because e comes before e' in the list, we have that $w \leq w'$. So taking e instead of e' cannot hurt the solution (either the cost stays the same, or is improved). Therefore, why not just take e in the first place?

Notice the similarity to our algorithm from earlier which constructively proved that spanning trees always exist in a connected graph: just insert the edges one-by-one, and don't make any cycles. The only modification is that we made sure to process the edges in non-decreasing order of edge weight so that the cheaper edges are prioritized. That seems quite intuitive to me!

In fact, consider this slightly "different" problem. Instead of deciding which edges to include, how about deciding which edges to *keep*?

Problem 2.3. A country has n cities, labeled 1 to n. There are m roads in this country, the ith of which bidirectionally connects the cities u_i and v_i and costs w_i million pesos per year to maintain. Here, each w_i is a positive integer. Initially, it is possible to start at any city and travel to any other city, using only the roads.

In order to cut down on costs, the national government wants to choose some subset of the roads such that:

- We keep **only these roads**, and close down all other roads
- It should still be possible to start at any city and travel to any other city, using only the *remaining* roads.
- Among all such solutions, the total cost of all remaining roads is minimized.

Well, actually, it should be pretty clear that this problem is basically the same as Problem 2.1.

Solution. Let's destroy all the roads and then decide which ones we want to restore. Framed this way, we can then just apply our solution to Problem 2.1.

We have a graph, and want to keep a subset of its edges such that the graph would still be connected. Now it's extra clear that what we've been doing is creating *spanning trees*, and we've been trying to do so at minimum cost.

Definition 2.4. Given a weighted undirected graph, a **minimum cost spanning tree** (MCST) is any spanning tree such that the total cost of all its edges is minimal.

Finding an MCST can be done in $\mathcal{O}(E \lg E)$ using the solution to Problem 2.1. This algorithm is classically known as **Kruskal's algorithm**.

Note that we say "a" minimum cost spanning tree, because (just like with spanning trees) a graph can have multiple MCSTs. Once again, consider a cycle graph where all edges have the same weight—removing any one of its edges results in an MCST.

Exercise 2.5. Construct an undirected weighted graph with n vertices such that the number of MCSTs it has is exponential in terms of n.

Exercise 2.6. Using Kruskal's algorithm, argue that the *multiset* of edge weights is the same across all MCSTs of a graph.

Exercise 2.7. Consider Problem 2.3, but with a slight modification. The local councils give a subset of the edges $S \subseteq E$ and request that *all roads in this subset must be kept.* Modify the solution to accommodate this change.

2.2 Properties of MCSTs

Consider the following problem.

Problem 2.8. A wasteland region has n outposts, labeled 1 to n. There are m trails, the ith of which bidirectionally connects the outposts u_i and v_i, and requires w_i liters of fuel in order to traverse by car. Here, each w_i is a positive integer. Traveling outside of these trails is not allowed, but a car with a sufficiently large fuel tank can go from any outpost to any other outpost, using only the trails. You may refill your fuel tank back to full at each outpost.

You are given Q queries, each of which contains two outposts a_i and b_i. For each one: If I wanted to start at outpost a_i and travel to outpost b_i, how big does my fuel tank have to be, at minimum, so that I can complete the journey?

Here, $2 \leq n \leq 2 \times 10^5$ and $1 \leq m \leq 5 \times 10^5$ and $1 \leq Q \leq 500$.

Let's phrase our solution entirely in terms of general graph theory terms:

- Suppose we examine some path from a_i to b_i
- Then, for us to be able to use this path, the size of our fuel tank must be at least as big as the **maximum** w_i along this path. Let's call this the **max-weight** of the path.
- Therefore, the problem is reduced to: Among all such paths from a_i to b_i, find the one whose max-weight is minimal.

We now introduce a powerful property of minimum-cost spanning trees that will help us solve this problem.

Claim 2.9. Let G be an undirected weighted graph, and let T be any MCST of G. Let the *minimum bottleneck* between two vertices u and v in G be the minimum possible max-weight across all paths from u to v.

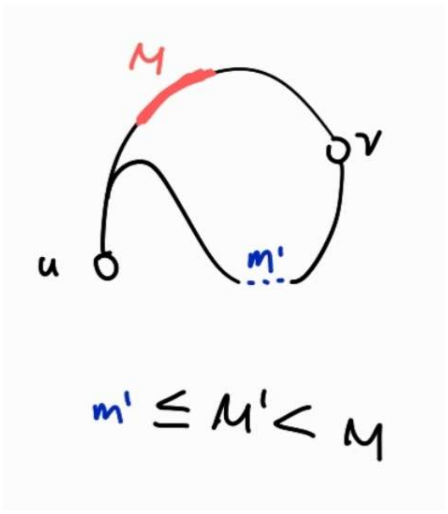
Then, for any two vertices u and v, their minimum bottleneck in G **is the same as** their minimum bottleneck in T.

This property is incredibly useful because **there is a unique path between any two vertices in a tree**. From an exponentially-sized search space, we now only need to find the max-weight of *this one path*!

Proof. We're going to prove this via contradiction.

Consider any two vertices u and v. Let M be the max-weight of the path from u to v in the minimum-cost spanning tree T. Now, *suppose* that there exists a path in G whose max-weight is some value M' such that $M' < M$.

Now, this path must contain an edge that isn't in T; after all, *there's only one path* from u to v in T, and we know its max-weight isn't M'. Let's suppose the weight of that not-in-T edge is m'.



But by definition of max-weight, we must have that $m' \leq M'$, since M' is the max-weight of the path that m' is on.

This is a problem, because then it allows us to improve the cost of T:

- Disconnect any edge in path from u to v in T, whose weight is M
- Insert that not-in-T edge whose weight is m'.

But we had said that T is a *minimum-cost* spanning tree! Since we have run into a contradiction, we conclude that no such path with max-weight $< M$ exists in G.

Therefore, the path in T always provides the minimum bottleneck. $\square$

We clarify that there can be many possible paths from u to v that attain this minimum bottleneck, and also that not all of them are on every single MCST of the graph. However, the MCST of the graph always gives *one of the* optimal solutions.

Now we have the tools to answer the original problem.

Solution. (to Problem 2.8) Model this as a graph problem (outposts are vertices and trails are edges) and take any MCST of the graph.

Then, each query can be answered in $\mathcal{O}(n)$ by performing a DFS in the MCST.

Remark 2.10. Problem 2.8 can actually be answered in $\mathcal{O}(\lg n)$ per query by further pre-processing the tree that we get and building additional data structures on top of it. An outline of this algorithm is:

- Find the *lowest common ancestor* (LCA) of a_i and b_i in the tree
- Decompose the path query into two node-to-ancestor max-weight queries, which can be solved via *binary jumping*

We will not go into further details in this tutorial, and will defer that for when LCAs are covered in the future.

Exercise 2.11. A computer network has n servers, labeled 1 to n. There are m cables, the ith of which bidirectionally connects the servers u_i and v_i and has a *bandwidth* of w_i. Here, each w_i is a positive integer. Messages can be sent through those cables, and can be forwarded along from server to server; however, each message also has a *size*, and it is not possible to send a message through a cable that has a lower bandwidth than the message's size. It is guaranteed that a sufficiently small message can start at any server and ultimate arrive at any other server.

You are given Q queries, each of which contains two servers a_i and b_i. For each one: What is the maximum possible size of a message that starts at server a_i and should ultimately arrive at server b_i?

Here, $2 \le n \le 2 \times 10^5$ and $1 \le m \le 5 \times 10^5$ and $1 \le Q \le 500$.

Hint: If you examine the proof of correctness for Kruskal's algorithm, it would actually produce a minimum-cost spanning tree of *any* weighted undirected graph. This is unlike Dijkstra's algorithm, which has a certain important restriction...

Bonus: (Revisit this later) Solve the problem for if there are 2×10^5 queries.