Graph Algorithms (Spring 2025)

3. Minimum spanning trees (and matroids)

Teacher: Arnaud Casteigts *Assistant: Matteo De Francesco*

In this class, we discuss some properties of spanning trees and spanning forests. Then we present two well-known algorithms computing minimum spanning trees in a weighted graph. These algorithms are surprisingly simple, but the reason why they work is less simple. Thus, we take some time to prove that they indeed compute an optimal solution. Then, we discuss what makes the MST problem efficiently solvable, namely, its Matroid structure.

Keywords: Subgraphs, Spanning forests, Spanning trees, Minimum spanning trees (MST), Kruskal, Prim, Greedy algorithms, Matroid.

3.1 Spanning tree

Let $G = (V, E)$ be a graph. A *subgraph* of $G' \subseteq G$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. A subgraph such that $V' = V$ is a *spanning subgraph*. If a spanning subgraph is a tree, then it is a *spanning tree* of G.

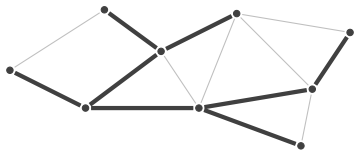

Figure 1: A spanning tree of a graph.

Spanning trees have many applications. In particular, they are used in routing protocols over the Internet. As we already saw, spanning trees can be built by starting a traversal (DFS or BFS) from an arbitrary node of G, and recording which node "discovered" which node first, using a `parent` list. In this case, the initial node is the *root* of the tree.

All spanning trees have the same size, they are made of $n - 1$ edges (one parent for each node, except for the root).

3.1.1 Spanning forest

A spanning tree is a particular case of *spanning forest*, which is just a spanning subgraph without cycles. For example, the graph $G' = (V', \emptyset)$ is a spanning forest, although it is not very interesting. This graph has n connected components, i.e. each node is a separate component in G'. Figure 2 shows a spanning forest made of 4 trees, one of which contains only one node.

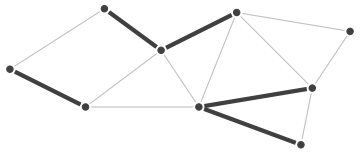

Figure 2: A spanning forest made of 4 trees.

More generally, the number of components (or trees) in a spanning forest depends exactly on its number of edges.

Lemma 3.1. *A spanning forest with k edges has exactly $n - k$ components (trees).*

Proof. (By induction.) If $k = 0$, each node is a separate component, so there are n components and the statement holds. Now, suppose the statement holds for some k. What happens if we add an edge without creating a cycle? This edge must connect two separate components, thus the number of components goes down by one and the statement holds. □

3.2 Minimum spanning tree

If the graph is weighted, the problem is more interesting. Here, we want to find a spanning tree whose edges have minimum sum of weights over all the possible spanning trees, called a *minimum spanning tree* (or MST). This notion is different from a tree of shortest paths. Indeed, Dijkstra's algorithm produces a spanning tree whose paths are optimal for a specific node, but there is no guarantee that the sum of weights is *globally* minimum. This can be seen by a simple example:

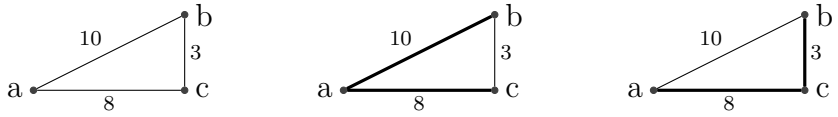

Figure 3: Left: A graph G. Middle: A tree of shortest paths from node a. Right: A minimum spanning tree whose cost is lower.

In this particular case, the MST is also a tree of shortest path from node c, but this is accidental. In general, the MST does not correspond to a tree of shortest paths. It should be seen as a global tradeoff.

3.2.1 Algorithms

There are two well-known greedy algorithms for this problem: Kruskal's algorithm and Prim's algorithm. Let $G = (V, E, w)$ be the input graph, supposed to be connected, for example like the one in Figure 4.

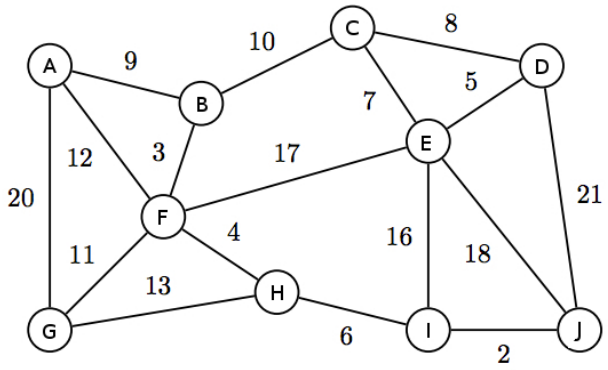

Figure 4: Example of an instance to the MST problem.

Kruskal's Algorithm

1. Initialize the solution to $G' = (V, \emptyset)$,
2. Add to G' the smallest edge that does not create a cycle,
3. Repeat step 2 until G' has $n - 1$ edges.

In each intermediate step, the solution consists of a spanning forest. The trees in this forest merge gradually into fewer trees, until the forest is a single tree.

To implement this algorithm, we can start by sorting the edges in ascending order, which has a time complexity of $O(m \log m) = O(m \log n)$ (why both are the same?). Then, we only need to make a pass over this list. The main difficulty is to detect whether the addition of a certain edge creates a cycle. This can be done by maintaining an updated list of the connected components and checking that the new edge has its endpoints in *different* components. Technically, this can be done using a `Union-find` data structure, at the cost of $O(m \log n)$ operations again. Thus Kruskal's algorithm has time complexity $O(m \log n)$.

Prim's Algorithm

1. Pick an arbitrary starting node s and initialize the solution to $G' = (\{s\}, \emptyset)$,
2. Find the smallest edge uv such that only u is in G'. Add uv and v to G',
3. Repeat steps 2 and 3 until all vertices are marked.

In intermediate steps, the solution may *not* be a spanning forest. However, it is always a tree, which is also nice. This algorithm is easier to implement, because we do not need to detect cycles. Its complexity is slightly better: the best implementations run in time $O(m + n \log n)$, using `Fibonacci heaps` to find the smallest edge at each step.

3.2.2 Why does it work?

The above algorithms are *greedy*: they select the best elements in each step without worrying about the final picture. It is not obvious that these "local" choices should lead to a "global" optimum. Yet, they do! Why? Before diving into this, let's take a closer look at Kruskal's algorithm and prove that it does find the optimum.

Correctness of Kruskal's algorithm.
Let G be the input graph and let $G' \subseteq G$ be the computed solution. We need to show two things about G': (1) it is a spanning tree; and (2) it has minimum possible weight.

(1) Spanning tree: The set of nodes of G' is initialized with V and never changes, thus G' is a spanning subgraph of G. To be a tree, it must (a) be a forest (i.e. a graph without cycles) and (b) be connected. Property (a) is straightforward, as the edges are added only if they do not create cycles. As for (b), the algorithm stops when G' has $n-1$ edges, which by Lemma 3.1 implies that it is made of a single component, and thus a tree.

(2) Minimum sum of weights: By contradiction, suppose that G' is not minimum. Let $e_1, e_2, \cdots, e_{n-1}$ be the sequence of edges selected by the algorithm, and e_i be the first of these such that $e_1, \cdots, e_{i-1}$ belongs to an MST G'' but $e_1, \cdots, e_i$ does not (i.e., selecting e_i was the first mistake of the algorithm). The edge e_i must form a cycle with some edges of G''. If any of these edges is larger than e_i, then we can replace it by e_i, contradicting the optimality of G'', thus their weight must be lower or equal to the weight of e_i. In addition, at least one of them does not belong to G' (since G' has no cycle), call this edge f. Since $w(f) \leq w(e_i)$ we have either $w(f) < w(e_i)$ or $w(f) = w(e_i)$. In the first case, f would have been selected by the algorithm instead of e_i (a contradiction), and in the other case, $G'' - f + e_i$ is still an optimal solution, contradicting the fact that $e_1, \cdots, e_i$ does not belong to an MST.

Matroids
The proof highlighted above is actually a special case of a more abstract argument, involving a structure called a *matroid*. The literature on matroids is vast and its terminology may clash with graph theory. Let us keep it simple.

A *set system* is a pair $(E, \mathcal{I})$, where E is a set of elements called the *ground* set and $\mathcal{I}$ is a set of subsets of E that satisfy some user-defined condition. A *matroid* is a special case of set system that satisfies three additional conditions:

1. $\emptyset \in \mathcal{I}$
2. (Heredity.) If $A \in \mathcal{I}$ and $B \subseteq A$, then $B \in \mathcal{I}$
3. (Augmentation property.) If $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| > |B|$, then there exists an element $e \in A \setminus B$ such that $B \cup \{e\} \in \mathcal{I}$. In other words, one can find an element of A that is transferable to B without breaking the property for B.

This framework is very general; depending on the problem, we may want E to represent numbers, nodes, edges, *etc.* In the context of the MST problem, we will consider that E is the set of edges of our graph (so this E is the same as E, great!), and we will define $\mathcal{I}$ as the set of all subsets of E that do *not* contain cycles, i.e., all forests of G. Let us check whether our set system for MST is a matroid:

1. The empty subset is a valid forest (it has no cycle).
2. (Heredity.) If we remove edges from a forest, we still have a forest.
3. (Augmentation property.) Let A and B be two forests such that $|A| > |B|$. Then A has fewer connected components than B (Lemma 3.1), so there exists a component C of A that contains nodes from two or more components of B. Along any path in C from a node in one component of B to a node of another component, there must be an edge with endpoints in two components, and this edge may be added to B without creating a cycle.

Why all these efforts? Because the following greedy algorithm always finds the optimum solution in a matroid. Let $(E, \mathcal{I})$ be a matroid, let $w : E \to \mathbb{R}^+$ be a weight function.

Greedy algorithm:

1. Initialize the solution S to $\emptyset$
2. Find the smallest (or largest, if we want to maximize) element e such that $S + e$ is still in $\mathcal{I}$ and add it to S.
3. Repeat step 2 until no further element can be added.

We can see that this algorithm is exactly the algorithm of Kruskal in the case of the MST. However, it is more general. The interesting thing is that, once you suspect that a problem has a matroid structure, the only technical task is to prove the augmentation property. Then you get for free a proof that the greedy algorithm finds the optimal solution.

What about Prim's algorithm? It turns out that Prim algorithm is also greedy, although its solutions do not form a matroid. They however satisfy the axioms of a more general structure called *greedoid*. There exist many types of set systems, with various algorithmic advantages, matroids being just the most famous of them.