Graph Algorithms (Spring 2026)

3. Minimum spanning trees (and matroids)

Teacher: Arnaud Casteigts *Assistant: Himika Das*

In this class, we define the concept of minimum spanning tree (MST). Then, we present two well-known algorithms computing an MST (Kruskal's algorithm and Prim's algorithm). These algorithms are surprisingly simple, yet, they indeed compute an optimal solution. We prove this for one of them. Then, we discuss what makes the MST problem efficiently solvable by a greedy algorithm, 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*.

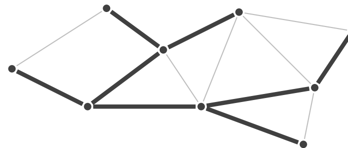

Figure 1: A spanning tree of a graph.

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 adding an edge whenever one of the two corresponding nodes "discovered" the other for the first time.

All spanning trees have the same size, they are made of $n - 1$ edges: one discoverer for each node, except for the initial node (we could also say one "parent" for each node except the root, if we see the tree as being rooted in the initial node).

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, each one consisting of a single node. Figure 2 shows a spanning forest made of 4 trees of sizes $3, 3, 2$, and 1, respectively.

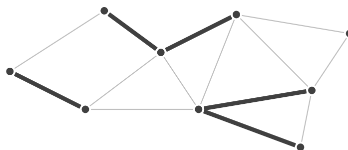

Figure 2: A spanning forest made of 4 trees.

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 (given by Dijkstra's algorithm), as illustrated by a simple example:

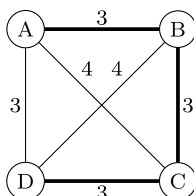

Figure 3: A graph where the MST does not correspond to any tree of shortest paths.

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.

Kruskal's Algorithm

1. Initialize the solution to $G' = (V, \emptyset)$,

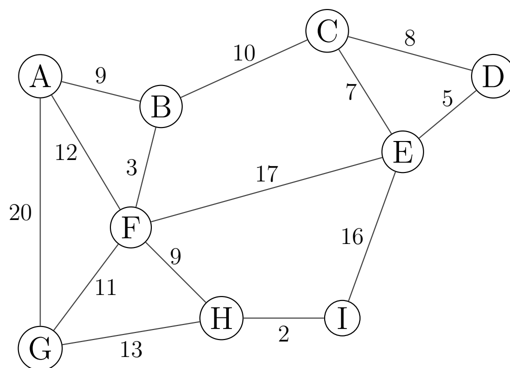

Figure 4: Example of an instance to the MST problem.

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 (non-spanning) 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 $T \subseteq G$ be the tree constructed by Kruskal's algorithm by selecting a sequence of edges $e_1, e_2, \cdots, e_{n-1}$ in this order.

Theorem 3.2. *The total weight of T is minimum.*

Proof. By contradiction, suppose that T is not minimum. Then there exists an edge e_i that was the "first wrong choice" of the algorithm, in the sense that the edges selected before, namely $e_1, \cdots, e_{i-1}$, could still be extended to an optimal tree T', but $e_1, \cdots, e_i$ cannot. Since T' is a spanning tree that does not contain e_i, the graph $T' + e_i$ must have a cycle involving e_i. Consider this cycle. If one of the other edges in the cycle has weight greater than e_i, then we can replace that edge with e_i, which contradicts the optimality of T'; therefore each of these edges has weight less than or equal to e_i. Moreover, at least one of them, say f, does not belong to T (since T has no cycle). Now, 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 (contradicting the rules of Kruskal's algorithm), and in the other case $T' - f + e_i$ is an optimal solution, contradicting the claim that choosing e_i was a mistake. □

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, we'll just say a few words.

A *set system* is a pair $(E, \mathcal{I})$, where E is a set called the *ground* set and $\mathcal{I}$ is a set of subsets of E that satisfy some desired condition. A *matroid* is a set system that satisfies two additional conditions:

1. (Heredity.) If $A \in \mathcal{I}$ and $B \subseteq A$, then $B \in \mathcal{I}$
2. (Exchange.) 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 violating the desired condition.

This framework is very general; depending on the problem, E may represent numbers, nodes, edges, vector bases, *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 our E, great!).

Lemma 3.3. *For any graph $G = (V, E)$, the set system $(E, \mathcal{I})$ where $\mathcal{I}$ is the set of (spanning) forests of G forms a matroid.*

Proof. We just need to check that our set system satisfies both conditions:

1. (Heredity.) For all forest $F \in \mathcal{I}$, any subset of F is also a forest. (Removing an edge from a forest cannot create a cycle.)
2. (Exchange.) Let $A \in \mathcal{I}$ and $B \in \mathcal{I}$ be two forests such that $|A| > |B|$. Then A has fewer connected components than B (Lemma 3.1), so there exists a component c in A that contains nodes belonging to distinct components of B. As these nodes are connected, there must be at least one edge in c whose endpoints are in distinct components of B, so this edge can be added B without creating a cycle.

□

Why all these efforts? Because there exists a general theorem that says that, in matroids, the greedy algorithm always finds a solution of maximum size and optimal cost. More precisely, let $(E, \mathcal{I})$ be a matroid, let $w : E \rightarrow \mathbb{R}^+$ be a weight function, then the following algorithm always finds the optimum.

General 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.

Observe that Kruskal is just a special case of this algorithm. So, we could have avoided writing the proof of Theorem 3.2 if we already knew that forests were matroid and that the greedy algorithm is always optimal in matroids. Now, you know it!

What about Prim's algorithm? It turns out that Prim algorithm is also greedy, although its partial solutions do not form a matroid (removing an edge from a tree does not always give a tree). They however satisfy the axioms of a more general structure called *greedoid*. Indeed, there exist many types of set systems, with various algorithmic properties, matroids being just the most famous of them.