1 Minimum Spanning Trees

Let $G = (V, E)$ be a undirected connected graph having $|V| = n$ nodes, $|E| = m$ edges, and edge weights $c : E \to \mathbb{R}^+$. A spanning tree of G is a connected, cycle free subgraph $T = (V, E')$ of G. The weight of a spanning tree equals the sum of the weights of all its edges, i.e. $\sum_{e \in E'} c(e)$. A spanning tree T is a *minimum spanning tree*, if there is no other spanning tree of G having a smaller weight than T. A graph can have several different minimum spanning trees.

The following problem that can be solved by computing a minimum spanning tree: Given a set of hubs, we want to build a connected communication network between the hubs. We can build wires between certain pairs of hubs. The cost of a wire depends on the two end hubs of the wire. The problem to build a set of wires such that each hub can communicate with each other hub using the wires and such that the cost to build these wires is minimal is equivalent to the problem to compute a minimum spanning tree. Regarding this, the node set V represents the set of hubs, E the set of possible wires, and $c(\{v, w\})$ the costs to build a wire between the hubs v and w.

The algorithms for computing a minimum spanning tree that we are considering begin with an empty edge set $E' = \emptyset$ and add one edge after another to E' until E' equals the edge set of a minimum spanning tree. At each point in time, the current set E' partitions the nodes of G into a set of subtrees. At the beginning, i.e. $E' = \emptyset$, each of these subtrees consists of one single node. The addition of an edge $e = (u, v)$ to E' joins the two subtrees containing u and v. In order to ensure that no cycles are introduced, only edges that connect two different subtrees can be added. At the end, i.e. after the addition of $n-1$ edges, only one subtree remains. The rule which determines the edge to be added must be chosen in such a way that the resulting spanning tree has minimum weight.

Lemma 1 *Let $E' \subset E$ such that there is a minimum spanning tree T of G containing all edges of E'. Let T' be one of the subtrees induced by E'. Let e be an edge having minimal weight of all the edges connecting a node of T' with a node of $G \setminus T'$. Then, there is a minimum spanning tree of G containing the edges $E' \cup \{e\}$.*

Proof: As defined in the lemma, let e denote the currently added edge, E' the current set of edges before the addition of e, T' the current subtree such that e is incident to a node of T' and a node of $G \setminus T'$, and T a minimum spanning tree of G which contains E'. We must show that there is a minimum spanning tree of G containing $E' \cup \{e\}$.

If e is contained in T, then T is the minimum spanning tree of interest and we are finished. Therefore, let's assume that e is not contained in T. Then, the addition of e to T creates a cycle k which contains nodes of T' as well as nodes of $G \setminus T'$. Hence, k has to contain another edge $e' \neq e$ which connects a node of T' with a node of $G \setminus T'$, and therefore leads out of T', see figure 1.

Since e has minimal weight over all these edges, $c(e') \geq c(e)$ holds. If we now remove the edge e' from T and add the edge e, then we obtain a spanning tree T'' whose weight is not larger than the weight of T. Since T is a minimum spanning tree, the weight of T'' equals the weight of T. Moreover, T'' contains $E' \cup \{e\}$ implying that T'' is the minimum spanning tree having the properties mentioned in the lemma. □

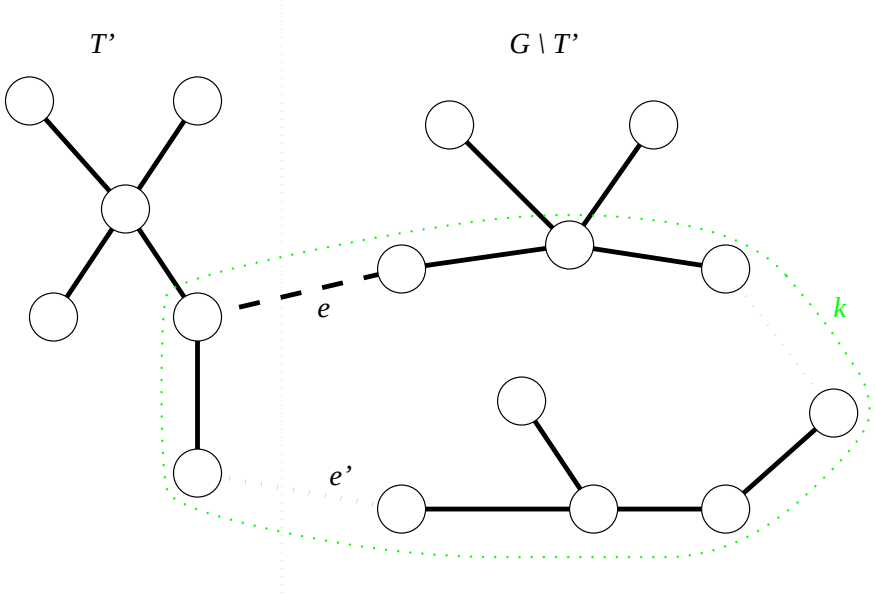

Abbildung 1: Sketch for the proof of Lemma 1

This lemma tells us that we can compute a minimum spanning tree in the following way: We start with an empty edge set E' (which is always a subset of the edge set of a minimum spanning tree), and at each step we add an edge e to E' which leads out of one of the current subtrees and has minimal weight with respect to all such edges.

1.1 Algorithm of Kruskal

The algorithm of Kruskal considers the edges of the graph in ascending order regarding their edge weights. (If two edges have the same weight, they can be ordered arbitrarily). If the currently considered edge e connects two nodes which are part of the same subtree, then this edge is discarded and not added to E'. If, on the other hand, e connects two different subtrees, then e is added to E' and, by doing this, both subtrees are merged into one single subtree.

Since the algorithm only adds edges e to E' which have minimal weight regarding the edges which are still candidates for minimum spanning tree edges, the condition of Lemma 1 is satisfied, and the algorithm finishs with a minimum spanning tree.

For an efficient implementation of the algorithm of Kruskal, there are two crucial points regarding the running time: first, we have to think how we can iterate over the edges in ascending order regarding the edge weights; second, it is not clear how we can efficiently find out if the end nodes of the current edge e is part of the same subtree (or not).

To solve the first problem, there are two methods: We can simply sort the edges in time $O(m \log m) = O(m \log n)$ regarding their weights at the beginning of the algorithm, or we use a priority queue. A priority queue is a data structure Q which offers at least the following operations in an efficient manner:

- INSERT(e,p): insert element e with priority p into Q

- DecreasePriority(e,p): decrease the priority of an element e of Q to p (e has to have at least priority p at that moment to execute this operation).
- DeleteMin(): return an element of Q having the lowest priority of all the elements of Q and delete this element of Q.

If priority queues are implemented using Fibonacci heaps, then the (amortized) running time of Insert and DecreasePriority is $O(1)$ and the running time of DeleteMin is $O(\log n)$, if Q contains n elements at that moment. LEDA provides such priority queues as `p_queue` or as `node_pq` (for the nodes of a graph).

At the beginning of the algorithm of Kruskal, every edge can be added to the priority queue having its weight as its priority in time $O(m)$. The operation DeleteMin is then used to find the next candidate edge. Each DeleteMin needs $O(\log m) = O(\log n)$ time. If the minimum spanning tree is computed after processing ℓ edges, the time needed for all priority queue opeations is in $O(m + \ell \log n)$. In the worst case this equals $O(m \log n)$. The worst case is attained when the edge with the largest weight is part of any minimum spanning tree which implies that each edge has to be processed. However, the implementation with priority queues can be advantageous if fewer edges must be processed.

The second problem was to find out if the two end nodes of an edge are part of the same subtree or not. This is a so called union/find problem: we want to dynamically manage the partition of a set into disjunct subsets such that only the operations Find ("which subset contains a given element?") and Union ("union two subsets") will be executed. For this task there exist very efficient methods which, given a set of n elements, are able to execute $k \geq n$ Find and Union operations in time $O(k\alpha(k, n))$ where α is the inverse Ackermann function which grows extremely slowly. In the context of our application we have to execute at most $n - 1$ Union and at most $2m$ Find operations. Hence, the total running time is in $O(m\alpha(m, n))$. LEDA provides Union/Find data structures as `partition` and `node_partition` (specialized for partitions of the node set of a graph).

Since $\alpha(m, n)$ grows much slower than $\log m$ and $\log n$, respectively, the total running time of the algorithm of Kruskal having the described implementation is $O(m \log n)$.

1.2 Algorithm of Prim

The algorithm of Prim chooses an arbitrary node of G as a start node. At each step of the algorithm, always the subtree is considered that contains the start node. The algorithm adds and edge e to E' which leads out of the considered subtree and which has minimal weight of all edges leading out of this tree. By doing this, the tree containing the start node grows by one node at each step, and is a spanning tree after $n - 1$ steps.

It is easy to see that this edge choosing scheme satisfies the prerequisits of Lemma 1. Therefore, the algorithm of Prim returns a minimum spanning tree.

When this algorithm is implemented, we have to take into consideration how we can efficiently find the edge to be added to E'. This edge must have minimal weight over all edges connecting a node of the current subtree T with a node of $G \setminus T$. Here, too, a priority queue is a useful tool: we save all nodes of $G \setminus T$ which are incident to an edge leading out of T in our priority queue Q. Let v be such a node of $G \setminus T$ and let e_v be an

edge which connects a node of T with v and has minimal weight of all the edged from T to v. Then, the priority of v should equal $c(e_v)$. In addition, we memorize that e_v is the cheapest edge from T to v. At the beginning, we initialize Q by putting all neighbors v of the start node s into Q such the priority of v equals the weight of the edge $\{s, v\}$. Furthermore, we memorize each of these edges $\{s, v\}$ as our provisional cheapest edge to reach v from s.

If we want to choose the next edge to be added to E', we simply execute the DeleteMin operation of the priority queue; this operation returns the node v of $G \setminus T$ which can be reached from T over an edge e_v having minimal weight of all the edges leading from T to $G \setminus T$. We add e_v to E' and have to update some data; since v is now contained in T, there can be a neighbor w of v which is not contained in T, and previously either wasn't reachable from T at all, or only using an edge with a larger weight than $c(\{v, w\})$. In the former case, we add w with priority $c(\{v, w\})$ to Q; in the latter case, we decrease the priority of w to $c(\{v, w\})$ using the DecreasePriority operation. In both cases, we memorize that the cheapest edge from T to w is the edge $\{v, w\}$.

The algorithm of Prim computes a minimum spanning tree in $n - 1$ steps where, at each step, a new node v for our spanning tree is chosen by using the DeleteMin operation, and then each neighbor of v which is not already part of T is either inserted into the priority by a Insert operation, or is updated regarding its priority with the DecreasePriority operation, if necessary. Without taking the operations of the priority queue into consideration, the running time is in $O(n + m) = O(m)$. Since the amortized running time of each Insert- and DecreasePriority operation is in $O(1)$, and the amortized running time of DeleteMin is in $O(\log n)$, and $n - 1$ Insert operations, $n - 1$ DeleteMin operations, and $O(m)$ DecreasePriority operations are executed, the total running time is in $O(m + n \log n)$.