

## Prim's minimum spanning tree algorithm

**High-level algorithm.** Input: connected graph  $G$ , non-negative edge weights  $w_e$ .

Output: a tree connecting all vertices of  $G$  and of minimum weight.

Initialization: Let  $S = \{s\}$ , some arbitrary vertex of  $G$ , and  $T = \emptyset$ .

For  $i = 1, 2, \dots, n - 1$ :

Assume that vertex set  $S$  is already defined.

For each  $v \notin S$ , let  $d_v = \min\{w_{\{u,v\}}, u \in S, \{u,v\} \in E\}$  and  $e_v$  be the corresponding edge.

Let  $v \notin S$  be such that  $d_v$  is minimum.

Add  $v$  to  $S$  and  $e_v$  to the tree.

Output the tree.

**Proof of correctness.** It is easy to see that the output  $T$  is a tree (exercise). The rest of the proof is by contradiction. Assume that it is not of minimum cost, and let  $T^*$  be a minimum spanning tree (if there are several minimum spanning trees, pick one such that the first  $k$  edges picked by the algorithm are also in  $T^*$ , and  $k$  is as large as possible).

Consider the edges in the order in which they are added to  $T$  by the algorithm, let  $e$  be the first one which is not in  $T^*$ , and let  $S$  be the set just before  $e$  is added:  $e = \{u, v\}$  with  $u \in S$  and  $v \notin S$ . In  $T^* \cup \{e\}$ , edge  $e$  closes a cycle  $C$ , so there must exist some other edge  $e^* = \{v^*, u^*\}$  on the cycle with  $u^* \in S$  and  $v^* \notin S$ . Why did the algorithm choose  $e$  and not  $e^*$  at that point? Because  $w_e \leq w_{e^*}$ . But observe that  $T' = T^* - \{e^*\} + \{e\}$  is still a spanning tree (any two vertices which in  $T^*$  were connected by a path going through  $e^*$  are still connected, replacing  $e^*$  by going around the cycle  $C$ ). Since  $T^*$  is of minimum cost, it has cost less than  $T'$ , and so  $w_e \geq w_{e^*}$ . Thus  $w_e = w_{e^*}$ . But then,  $T'$  is also a minimum spanning tree, and has one more edge in common with the algorithm's choices: this contradicts the definition of  $T^*$ .

**Implementation.** Instead of recomputing all the  $d_v$ 's at each iteration, we maintain, for each  $v$ , a quantity  $d(v)$  representing the length of the shortest of all the edges from  $S$  to  $v$ , and the corresponding neighbor  $\pi(v)$ . We update those whenever  $S$  changes.

Initialization: Let  $S = \{s\}$  and  $d_s = 0$ .

For each neighbor  $v$  of  $s$ , let  $d_v = w_{\{s,v\}}$ . For the other vertices,  $d_v = \infty$ .

For  $i = 1, 2, \dots, n - 1$ :

Let  $v \notin S$  be such that  $d_v$  is minimum.

Add  $v$  to  $S$  and  $\{v, \pi(v)\}$  to  $T$ .

Update  $d_z$  for every  $z \notin S$  adjacent to  $v$ : if  $w_{\{v,z\}} < d_z$  then  $d_z \leftarrow w_{\{v,z\}}$  and  $\pi(z) \leftarrow v$ .

Output  $T$ .

**Data structure.** To be efficient, the algorithm needs a good data structure to find the minimum and update values, i.e. to manipulate the set  $\{d_v : v \notin S\}$ . The operations are: find the minimum (to find  $v$ ), extract the minimum (when we add  $v$  to  $S$ ), decrease some values (when we update  $d_z$ ). Extract-min and Decrease-Key are operations supported by priority queues.

A heap implementation of priority queues does both operations in time  $O(\log n)$ . The number of extract-min operations is at most  $n - 1$  (once per vertex which we add to  $S$ ) and the number of decrease-key operations is at most once per edge (when we add one of its endpoints to  $S$ ), i.e. at most  $m$  total. This gives overall complexity  $O((m + n) \log n)$ .

Note that we do need, when we add  $v$  to  $S$ , to access its neighbors (via the adjacency list representation of the graph), and for each neighbor  $z$ , to access the corresponding heap element, so there may be a bit of bookkeeping involved in the actual program and a little bit of additional thought would be needed to transform an elegant algorithm into an elegant piece of code. You should be able to figure this out by yourselves...