

## Dijkstra's shortest path algorithm

**High-level algorithm.** Input: graph  $G$ , non-negative edge weights  $w_e$ , and two vertices  $s$  and  $t$ .  
Output: the length of a shortest path from  $s$  to  $t$  in  $G$ .

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

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

Assume that vertex set  $S$  and  $\{d_u\}_{u \in S}$  are already defined.

Let  $v \notin V$  be a vertex adjacent to some  $u \in S$  and such that  $d_u + w_{\{u,v\}}$  is minimum.

Add  $v$  to  $S$  and let  $d_v = d_u + w_{\{u,v\}}$ .

Output  $d_t$ .

**Proof of correctness.** We will prove by induction that at every iteration through the “for” loop,  $S$  is the set of the  $|S|$  vertices closest to  $s$ , and for every  $u \in S$ ,  $d_u$  is the distance from  $s$  to  $u$ .

Base case: Since all edge weights are non-negative,  $s$  is certainly the vertex closest to  $s$  and its distance to  $s$  is 0.

General case: Assume that when we enter the loop for  $i$ ,  $S$  consists of the  $i$  vertices closest to  $s$  and that their distances to  $s$  are given by  $\{d_u\}$ . The algorithm looks at all quantities of the form  $d_u + w_{\{u,v\}}$ , for every edge between  $S$  and  $V \setminus S$ , and takes the minimum in order to decide which vertex  $v$  to add to  $S$ .

First, note that each such quantity  $d_u + w_{\{u,v\}}$  corresponds to the length of some path from  $s$  to  $v \notin S$ : indeed,  $d_u$  is the distance from  $s$  to  $u$ , so it is the length of some shortest path  $p$  from  $s$  to  $u$ , and so  $d_u + w_{\{u,v\}}$  is the length of the path  $(p, \{u, v\})$  from  $s$  to  $v$ .

For a fixed  $v \notin S$ , when the algorithm looks at all quantities  $d_u + w_{\{u,v\}}$  for every adjacent  $u \in S$  and takes the minimum, it is computing the length of the shortest path from  $s$  to  $v$  which is constrained to end with an edge from  $S$  to  $v$ .

Is this the length of the actual shortest path from  $s$  to  $v$ ? No, not for every  $v \in V \setminus S$ . But for vertex  $v^*$ , the vertex in  $V \setminus S$  which is closest to  $s$ , yes, this is the length of the shortest path from  $s$  to  $v^*$ : indeed, let  $p^*$  be a shortest path from  $s$  to  $v^*$  in  $G$ . On  $p^*$ , all internal vertices are closer to  $s$  than  $v^*$  is, so by definition of  $v^*$  they must all be in  $S$ . So, for  $v^*$  the algorithm computes the actual shortest path length; for other vertices of  $V \setminus S$ , the algorithm computes the length of some path from  $s$  to  $v$ : taking the minimum, we'll get  $v^*$ , and  $d_{v^*}$  is the correct distance from  $s$  to  $v^*$ .

**Implementation.** We maintain, for  $v$ , a quantity  $d(v)$  representing the length of the shortest path from  $s$  to  $v$  constrained to end with an edge from  $S$ . 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  $S$

Update  $d_z$  for every  $z \notin S$  adjacent to  $v$ : if  $d_v + w_{\{v,z\}} < d_z$  then  $d_z \leftarrow d_v + w_{\{v,z\}}$ .

Output  $d_t$ .

**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. Ask Phil Klein for details.

**Enriched implementation: shortest path tree.** Dijkstra's algorithm does more than just compute the distance from  $s$  to  $t$ : it computes the distance from  $s$  to everything else in the graph. And it does more than just compute distances: implicitly, it also computes, not just the lengths of the shortest paths, but also the actual shortest paths themselves. We can make that explicit by storing as we go along in the computation, for each vertex  $v$ , in some  $\pi(v)$ , the name the vertex  $u$  which is the last vertex before  $v$  in the shortest path from  $s$  to  $v$  constrained to end with an edge from  $S$ .

The set of edges  $\{v, \pi(v)\}_{v \neq s}$  defines a tree, known as the shortest path tree.