

1 Shortest Paths (Section 11.4)

1.1 Problem Statement

Given a weighted graph $G = (V, E)$, where $w : E \rightarrow \mathcal{R}$, and a “source” vertex s such that $s \in V$, we want to find the shortest paths from s to any other vertex in G . Note that “shortest” really means “lightest”, i.e., paths that have the smallest total weight (the sum of the weight for the edges on the path).

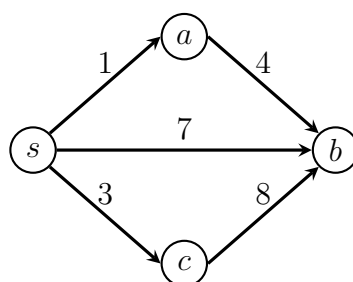


Figure 1: A Directed and Weighted Graph

1.2 Dijkstra's Algorithm

Idea: Suppose that we already have some subset S of vertices such that for every vertex v in S , we know that shortest paths from s to v . Consider the vertices outside of S . In particular, consider the first vertices (say a , b , and c) outside of S . What would be the shortest way to get to a from s ? It might seem like simply adding a to S directly is the best thing to do, but it does not quite work: for example, it might be a shorter path to a going through c .

What we do know is that the shortest path from s to a is no-longer than the path we have found from vertices in S . This allows us to keep track of an upper bound on the length of the shortest paths from s to every vertex outside of S , but we don't know for sure if that upper bound is the exact value or not. So how do we get around this?

Is there at least one vertex for which we know for sure we've found a shortest path? If we pick a vertex w outside of S that has the smallest upper bound, we must have in fact a shortest path from s to w because if there was a shorter way, we would have picked it first.

This gives the following idea for an algorithm: for each vertex v in G , we will keep track of $d[v]$, an upper bound on the weight of a shortest path from s to v . Initially we set $S = \{s\}$, $d[s] = 0$, and $d[v] = \infty$ for every $v \neq s$. Then, we repeatedly pick a w outside of S such that $d[w]$ is smallest, add it to S , and update $d[v]$ for every v that is adjacent to w , until all the vertices are in S . If we also keep track of the predecessor for each vertex as we do this, we can reconstruct the paths from s to every other vertex easily.

```

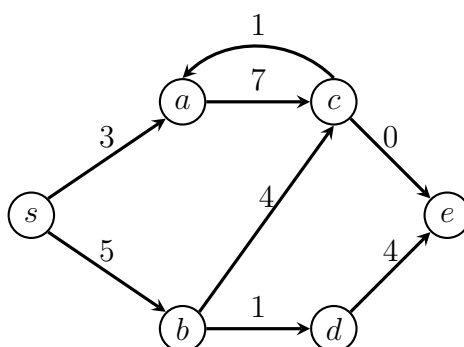
DIJKSTRA( $G=(V,E)$ ,  $w: E \rightarrow R$ ,  $s \in V$ )
  foreach  $v$  in  $V$  do
     $d[v] = \text{inf}$  // shortest distance found to  $v$ 
     $\text{pred}[v] = \text{nil}$  // predecessor of  $v$ : endpoint of the last edge in a shortest
                        // path to  $v$ )
  endfor

 $S = \{\}$  // set of vertices whose shortest path is known.
 $d[S] = 0$ 
while( $v$  is not empty) do
  for  $u$  in  $V$  such that  $d[u]$  is minimal
     $S = S \cup \{u\}$ ,  $V = V - \{u\}$ 
    for each vertex  $v$  adjacent to  $u$ 
      if ( $v$  in not in  $S$  and  $d[v] > d[u] + w(u,v)$ )
         $d[v] = d[u] + w(u,v)$ 
         $\text{pred}[v] = u$ 
      end if
    end for
  end while
END

```

Time Complexity: Consider the case in which we maintain the set V as a linear array. For such an implementation, each operation to find the minimal in V takes time $O(V)$, and there are $|V|$ such operations, for a total of $O(V^2)$. Meanwhile, during the course of the algorithm, each edge is examined exactly once, in the for loop. Since the total number of edges is $|E|$, there are a total of $|E|$ iterations of this for loop, with each iteration taking $O(1)$ time. The running time of the entire algorithm is thus $O(V^2 + E) = O(V^2)$.

1.3 Examples

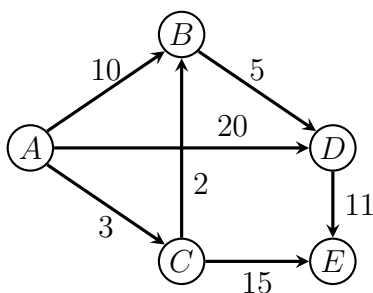


Example 1. Tracing the algorithm on this graph produces the following values for S , $d[\]$, and $\text{pred}[\]$ after each iteration of the while-loop (for every iteration, we indicate the values of $d[\]$ and $\text{pred}[\]$ only for the vertices that remain outside of S).

	d[]						pred[]					
S	s	a	b	c	d	e	s	a	b	c	d	e
{}	0	∞	∞	∞	∞	∞	-	-	-	-	-	-
{s}		3	5	∞	∞	∞		s	s	-	-	-
{s, a}			5	10	∞	∞			s	a	-	-
{s, a, b}				9	6	∞				b	b	-
{s, a, b, d}				9		10				b		d
{s, a, b, d, c}						9						c
{s, a, b, d, c, e}												

which gives the following final values (from which we can easily find shortest paths and their lengths).

	d[]						pred[]					
S	s	a	b	c	d	e	s	a	b	c	d	e
{s, a, b, d, c, e}	0	3	5	9	6	9	-	s	s	b	b	c



Example 2 (Textbook Figure 11.16). Tracing the algorithm on this graph produces the following values

	d[]					pred[]				
S	A	B	C	D	E	A	B	C	D	E
{}	0	∞	∞	∞	∞	-	-	-	-	-
{A}		10	3	20	∞		A	A	A	-
{A, C}		5		20	18		C		A	C
{A, C, B}				10	18				B	C
{A, C, B, D}					18					C
{A, C, B, D, E}										

which gives the following final values (from which we can easily find shortest paths and their lengths).

	d[]					pred[]				
S	A	B	C	D	E	A	B	C	D	E
{A, C, B, D, E}	0	5	3	10	18	-	C	A	B	C

2 Minimum-Cost Spanning Trees (Section 11.5)

Assume in this section G is an undirected graph. The degree of a vertex v is the number of edges touching v . G is connected if between every pair of distinct vertices there is a path. A tree is a connected acyclic graph. A spanning tree of a connected graph G is a subset $T \subseteq E$ of the edges such that (V, T) is a tree.

If a connected G has a cycle, then there is more than one spanning tree for G , and in general G may have exponentially many spanning trees, but each spanning tree has the same number of edges (the proof is by induction on n):

Lemma 3. *Every tree with n vertices has exactly $n - 1$ edges.*

We are interested in finding a minimum cost spanning tree for a given connected graph G , assuming that each edge e is assigned a cost $c(e)$. (Assume for now that the cost $c(e)$ is a nonnegative real number.) In this case, the cost $c(T)$ is defined to be the sum of the costs of the edges in T . We say that T is a minimum cost spanning tree (or an optimal spanning tree) for G if T is a spanning tree for G , and given any spanning tree T' for G , $c(T) \leq c(T')$.

Given a connected graph $G = (V, E)$ with n vertices and m edges, e_1, e_2, \dots, e_m , where $c(e_i) = \text{"cost of edge } e_i\text{"}$, we want to find a minimum cost spanning tree. It turns out that in this case, an obvious greedy algorithm (Kruskal's algorithm) always works. Kruskal's algorithm is the following: first, sort the edges in increasing (or rather nondecreasing) order of costs, so that $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$; then, starting with an initially empty tree T , go through the edges one at a time, putting an edge in T if it will not cause a cycle, but throwing the edge out if it would cause a cycle.

Kruskal(G)

Sort the edges so that : $c(e_1) \leq c(e_2) < \dots < c(e_m)$

$T = \{\}$

for $i = 1$ to m

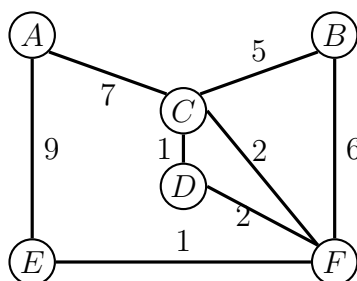
if $T \cup \{e_i\}$ has no cycle (*)

$T = T \cup \{e_i\}$

end if

end for

end



Example 4 (Textbook Example 11.4). First sort the edges in nondecreasing order.

$CD(1), EF(1), CF(2), DF(2), BC(5), BF(6), AC(7), AE(9)$

and the MST according to Kruskal is: CD, EF, CF, BC, AC .