Chapter 56

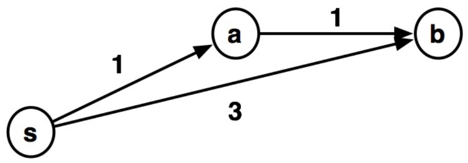
Dijkstra's Algorithm

Dijkstra's algorithm solves the single source shortest path problem with non-negative edge weights, i.e., the SSSP$^+$ problem. It uses priority-first search. It is a sequential algorithm.

1 Dijkstra's Property

In Chapter 53, we saw how BFS can be used to solve the single-source shortest path problem on graphs without edge weights, or, equivalently, where all edges have weight 1. BFS, however, does not work on general weighted graphs, because it ignores edge weights (there can be a shortest path with more edges).

Example 56.1. Consider the following directed graph with 3 vertices.



In this graph, a BFS visits b and a on the same round, marking the shortest path to both as directly from s. However the path to b via a is shorter. Since BFS never visits b again, it will not find the actual shortest path.

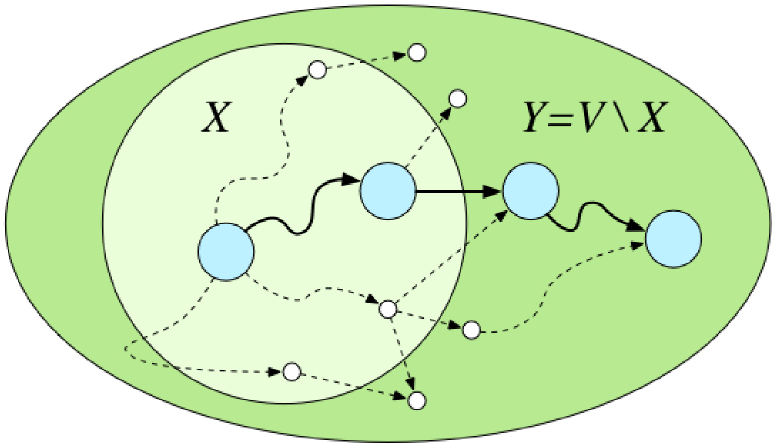
Brute Force. Let's start by noting that since no edges have negative weights, there cannot be a negative-weight cycle. One can therefore never make a path shorter by visiting a vertex twice—i.e., a path that cycles back to a vertex cannot have less weight than the path that ends at the first visit to the vertex. When searching for a shortest path, we thus have to consider only the simple paths, i.e., paths with no repeated vertices.

Based on this observation we can use a brute-force algorithm for the SSSP$^+$ problem, that, for each vertex, considers all simple paths between the source and a destination and selects the shortest such path. Unfortunately there can be a large number of paths between any pair of vertices (exponential in the number of vertices), so any algorithm that tries to look at all paths is not likely to scale beyond very small instances.

Applying the Sub-Paths Property. Let's try to reduce the work. The brute-force algorithm does redundant work since it does not take advantage of the sub-paths property, which allows us to build shortest paths from smaller shortest paths.

Suppose that we have an oracle that tells us the shortest paths from s to some subset of the vertices $X \subset V$ with $s \in X$. Also let's define Y to be the vertices not in X, i.e.,

$$Y = V \setminus X.$$



Consider now this question: can we efficiently determine the shortest path to any one of the vertices in Y? If we could do this, then we would have an algorithm to add new vertices to X repeatedly, until we are done.

To see if this can be done, let's define the ***frontier*** as the set of vertices that are neighbors of X but not in X, i.e. $N^+(X) \setminus X$ (as in graph search). Observe that any path that leaves X must go through a frontier vertex on the way out. Therefore for every $v \in Y$ the shortest path from s must start in X, since $s \in X$, and then leave X via a vertex in the frontier.

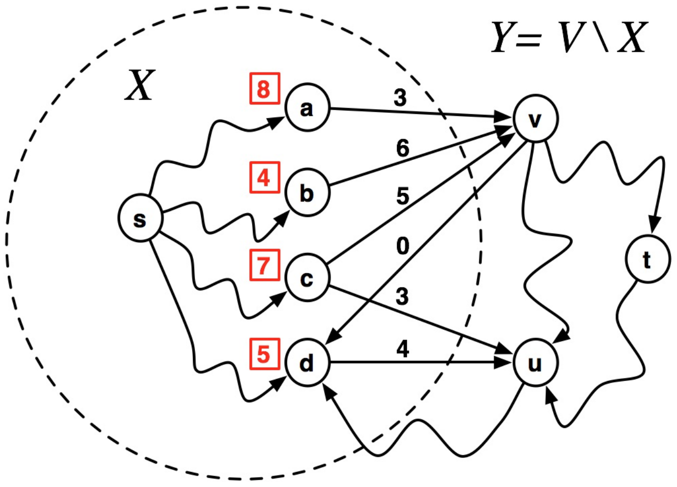
Now consider for each vertex v in the frontier F, the shortest path length that consists of a path through X and then an edge to v, i.e., $p(v) = \min_{x \in X}(\delta_G(s,x) + w(x,v))$. Among these consider the vertex v with overall shortest path length, $\min_{v \in F} p(v)$. Observe that no other vertex in Y can be closer to the source than v because

- all paths to Y must go through the frontier when exiting X, and
- edge weights are non-negative so a sub-path cannot be longer than the full path.

Therefore for non-negative edge weights the path that minimizes $\delta_G(s,x) + w(x,v)$, over all $x \in X$ and $v \in F$, is a minimum length path that goes from s to Y, i.e., $\min_{y \in Y} \delta_G(s,y)$. There can be ties. We thus have answered the question that we set out to answer: we can indeed determine the shortest path to one more vertex.

Lemma 56.1 establishes this intuition more formally.

Example 56.2. In the following graph suppose that we have found the shortest paths from the source s to all the vertices in X (marked by numbers next to the vertices). The overall shortest path to any vertex on the frontier via X and one more edge, is the path to vertex u via vertex d, with lenght $5 + 4 = 9$. If edge weights are non-negative there cannot be any shorter way to get to u, whatever the path length from v to u is, therefore we know that $\delta(s,u) = 9$. Note that if edges can be negative, then it could be shorter to go through v since the path length from v to u could be negative.



Lemma 56.1 (Dijkstra's Property)**.** Consider a (directed) weighted graph $G = (V,E,w)$, and a source vertex $s \in V$. Consider any partitioning of the vertices V into X and $Y = V \setminus X$ with $s \in X$, and let

$$p(v) = \min_{x \in X}(\delta_G(s,x) + w(x,v))$$

then $\min_{y \in Y} p(y) = \min_{y \in Y} \delta_G(s,y)$.

Note (The Lemma Explained in plain English). The overall shortest-path weight from s via a vertex in X directly to a neighbor in Y (in the frontier) is as short as any path from s to any vertex in Y.

Proof. Consider a vertex $v_m \in Y$ such that $\delta_G(s,v_m) = \min_{v \in Y} \delta_G(s,v)$, and a shortest path from s to v_m in G. The path must go through an edge from a vertex $v_x \in X$ to a vertex v_t in Y. Since there are no negative weight edges, and the path to v_t is a sub-path of the path to v_m, $\delta_G(s,v_t)$ cannot be any greater than $\delta_G(s,v_m)$ so it must be equal. We therefore have

$$\min_{y \in Y} p(y) \leq \delta_G(s,v_t) = \delta_G(s,v_m) = \min_{y \in Y} \delta_G(s,y),$$

but the leftmost term cannot be less than the rightmost, so they must be equal. □

The reader might have noticed that the terminology that we used in explaining Dijkstra's algorithm closely relates to that of graph search. More specifically, recall that priority-first search is a graph search, where each round visits the frontier vertex with the highest priority. If, as usual, we denote the visited set by X, we can define the priority for a vertex v in the frontier, $p(v)$, as the weight of the shortest-path consisting of a path to $x \in X$ and an additional edge from x to v, as in Lemma 56.1. We can thus define Dijkstra's algorithm in terms of graph search.

Algorithm 56.1 (Dijkstra's Algorithm)**.** For a weighted graph $G = (V, E, w)$ and a source s, Dijkstra's algorithm is priority-first search on G that

- starts at s with $d(s) = 0$,
- uses priority $p(v) = \min\limits_{x \in X}(d(x) + w(x, v))$ (to be minimized), and
- sets $d(v) = p(v)$ when v is visited.

When finished, returns $d(v)$.

Note. Dijkstra's algorithm visits vertices in non-decreasing shortest-path weight since on each round it visits unvisited vertices that have the minimum shortest-path weight from s.

Theorem 56.2 (Correctness of Dijkstra's Algorithm)**.** Dijkstra's algorithm returns $d(v) = \delta_G(s, v)$ for v reachable from s.

Proof. We show that for each step in the algorithm, for all $x \in X$ (the visited set), $d(x) = \delta_G(s, x)$. This is true at the start since $X = \{s\}$ and $d(s) = 0$. On each step the search adds vertices v that minimizes $P(v) = \min_{x \in X}(d(x) + w(x, v))$. By our assumption we have that $d(x) = \delta_G(s, x)$ for $x \in X$. By Lemma 56.1, $p(v) = \delta_G(s, v)$, giving $d(v) = \delta_G(s, v)$ for the newly added vertices, maintaining the invariant. As with all priority-first searches, it will eventually visit all reachable v. □

2 Dijkstra's Algorithm with Priority Queues

As described so far, Dijkstra's algorithm does not specify how to calculate or maintain the priorities. One way is to calculate all priorities of the frontier on each round of the search. This is effectively how Dijkstra originally described the algorithm. However, it is more efficient to maintain the priorities with a priority queue. We describe and analyze an implementation of Dijkstra's algorithm using priority queues.

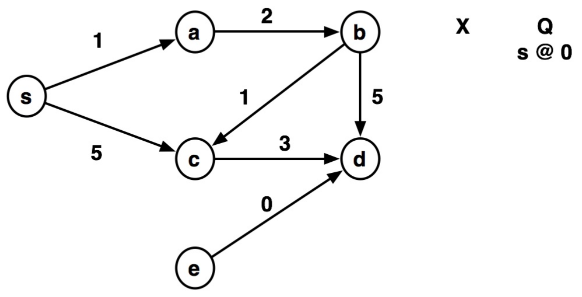
Algorithm 56.2 (Dijkstra's Algorithm using Priority Queues)**.** An implementation of Dijkstra's algorithm that uses a priority queue to maintain $p(v)$ is shown below. The priority queue PQ supports *deleteMin* and *insert* operations.

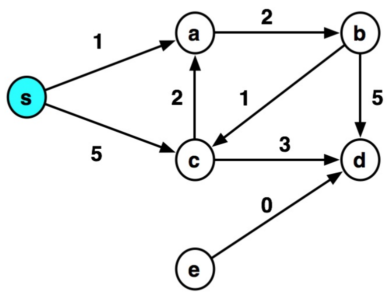
```
$dijkstraPQ\ \ G\ s =$
let
  $dijkstra\ \ X\ Q =$            (* $X$ maps vertices to distances. *)
    case $\underline{PQ.deleteMin}\ \ Q$ of
      (None, _) $\Rightarrow X$            (* Queue empty, finished. *)
    | (Some $(d, v), Q'$) $\Rightarrow$
        if $\underline{(v, \_) \in X}$ then $dijkstra\ \ X\ Q'$       (* Already visited, skip. *)
        else
          let
            $X' = \underline{X \cup \{(v, d)\}}$            (* Set final distance of $v$ to $d$. *)
            $relax\ \ (Q, (u, w)) = \underline{PQ.insert}\ (d + w, u)\ Q$
            $Q'' = iterate\ \ relax\ \ Q'\ (\underline{N_G^+(v)})$    (* Add neighbors to $Q$. *)
        in $dijkstra\ \ X'\ Q''$ end
  $Q_0 = \underline{PQ.insert}\ \ (0, s)\ PQ.empty$       (* Initial $Q$ with source. *)
in $dijkstra\ \ \{\}\ Q_0$ end
```

We assume $N_G^+(v)$ returns a pair $(u, w(v, u))$ for each neighbor u.

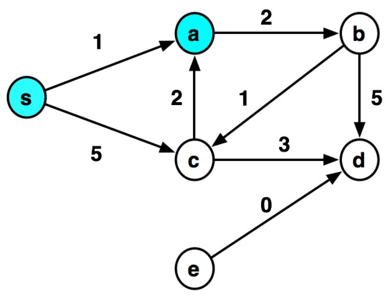
Remark. This algorithm only visits one vertex at a time even if there are multiple vertices with equal distance. It would also be correct to visit all vertices with equal minimum distance in parallel. In fact, BFS can be thought of as the special case of Dijkstra in which the whole frontier has equal distance.

Example 56.3. An example run of Dijkstra's algorithm. Note that after visiting s, a, and b, the queue Q contains two distances for c corresponding to the two paths from s to c discovered thus far. The algorithm takes the shortest distance and adds it to X. A similar situation arises when c is visited, but this time for d. Note that when c is visited, an additional distance for a is added to the priority queue even though it is already visited. Redundant entries for both are removed next before visiting d. The vertex e is never visited as it is unreachable from s. Finally, notice that the distances in X never decrease.

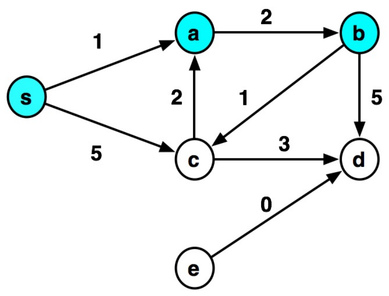




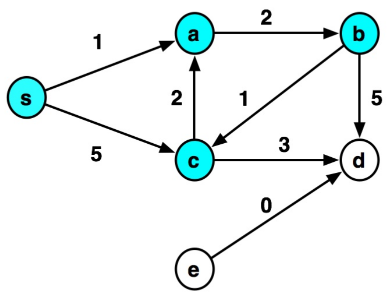
X	Q
s @ 0	a @ 1
	c @ 5



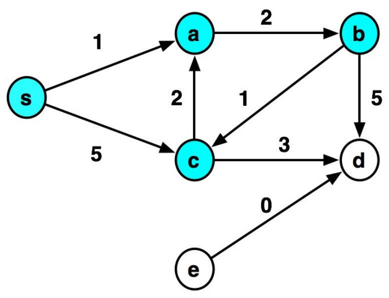
X	Q
s @ 0	b @ 3
a @ 1	c @ 5



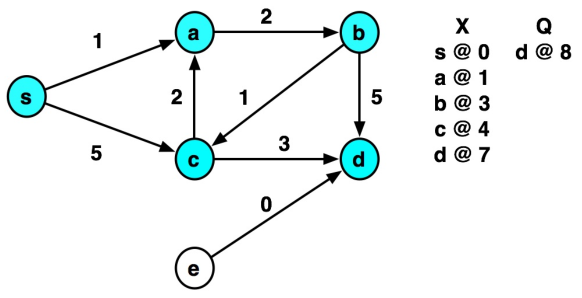
X	Q
s @ 0	c @ 4
a @ 1	c @ 5
b @ 3	d @ 8

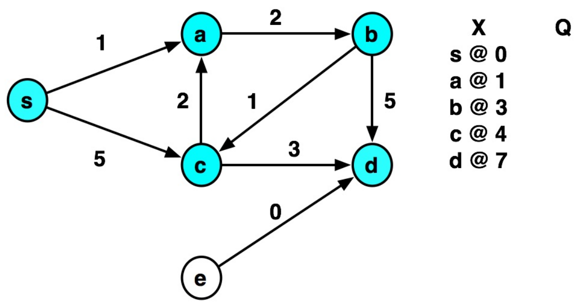


X	Q
s @ 0	c @ 5
a @ 1	a @ 6
b @ 3	d @ 7
c @ 4	d @ 8



X	Q
s @ 0	d @ 7
a @ 1	d @ 8
b @ 3	
c @ 4	





The algorithm maintains the visited set X as a table mapping each visited vertex u to $d(u) = \delta_G(s,u)$. It also maintains a priority queue Q that keeps the frontier prioritized based on the shortest distance from s directly from vertices in X. On each round, the algorithm selects the vertex x with least distance d in the priority queue and, if it has not already been visited, visits it by adding $(x \mapsto d)$ to the table of visited vertices, and then adds all its neighbors v to Q with the priorities $d(x) + w(x,v)$ (i.e. the distance to v through x).

Note that a neighbor might already be in Q since it could have been added by another of its in-neighbors. Q can therefore contain duplicate entries for a vertex with different priorities, but what is important is that the minimum distance will always be pulled out first. Line 7 checks to see whether a vertex pulled from the priority queue has already been visited and discards it if it has. This algorithm is just a concrete implementation of the previously described Dijkstra's algorithm.

Remark. There are a couple other variants on Dijkstra's algorithm using priority queues.

One variant checks whether u is already in X inside the $relax$ function, and if so does not inserts it into the priority queue. This does not affect the asymptotic work bounds, but might give some improvement in practice.

Another variant decreases the priority of the neighbors instead of adding duplicates to the priority queue. This requires a more powerful priority queue that supports a $decreaseKey$ function.

3 Cost Analysis of Dijkstra's Algorithm

Data Structures. To analyze the work and span of priority-queue based implementation of Dijkstra's algorithm shown above, let's first consider the priority queue ADT's that we use. For the priority queue, we assume $PQ.insert$ and $PQ.deleteMin$ have $O(\lg n)$ work and span. We can represent the mapping from visited vertices to their distances as a table, and input graph as an adjacency table.

Analysis. To analyze the work, we calculate the work for each significant operation and sum them up to find the total work. Each vertex is only visited once and hence each out edge is only visited once. The table below summarizes the costs of the operations, along with the number of calls made to each. We use, as usual, $n = |V|$ and $m = |E|$.

Operation	Line	Number of calls	PQ	Tree Table
$PQ.deleteMin$	4	m	$O(\lg m)$	–
$PQ.insert$	11	m	$O(\lg m)$	–
find	7	m	–	$O(\lg n)$
insert	10	n	–	$O(\lg n)$
$N_G^+(v)$	12	n	–	$O(\lg n)$
Total	–	$O(m \lg n)$	$O(m \lg n)$	$O(m \lg n)$

The work is therefore

$$W(n, m) = O(m \lg n).$$

Since the algorithm is sequential, the span is the same as the work.

Decrease Key Operation. We can improve the work of the algorithm by using a priority queues that supports a $decreaseKey$ operation. This leads to $O(m + n \log n)$ work across all priority queue operations.

Exercise 56.1. Give a version of Dijkstra's algorithm that uses the $decreaseKey$ operation.

Remark. For enumerated graphs the cost of the tree tables could be improved by using adjacency sequences for the graph, and ephemeral or single-threaded sequences for the distance table, which are used linearly. This does not improve the overall asymptotic work of Dijkstra's algorithm because the cost of the priority queue operations at $O(m \log n)$ dominate.

In the $decreaseKey$ version of the algorithm, the cost of priority queue operations are $O(m + n \log n)$ In this case, using sequences for graphs still does not help, because of the $O(n \log n)$ term dominates. But a more efficient structure for the mapping of vertices to distances could help.