Chapter 16

Shortest Paths

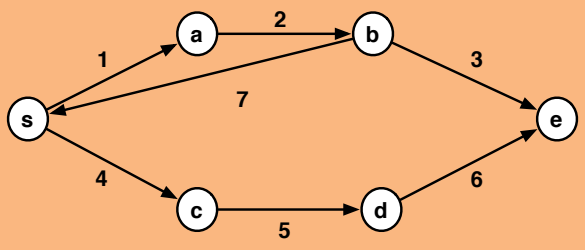
Given a graph where edges are labaled with weights (or distances) and a source vertex, what is the shortest path between the source and some other vertex? Problems requiring us to answer such queries are broadly known as *shortest-paths problems*. Shortest-paths problem come in several flavors. For example, the *single-source shoretest path* problem requires finding the shortest paths between a given source and all other vertices; the *single-pair* shortest path problem requires finding the shortest path between given a source and a given destination vertex; the *all-pairs shortest path problem* requires finding the shortest paths between all pairs of vertices.

In this chapter, we consider the single-source shortest-paths problem and discuss two algorithms for this problem: Dijkstra's and Bellman-Ford's algorithms. Dijkstra's algorithm is more efficient but it is mostly sequential and it works only for graphs where edge weights are non-negative. Bellman-Ford's algorithm is a good parallel algorithm and works for all graphs but requires significantly more work.

16.1 Shortest Weighted Paths

Consider a weighted graph $G = (V, E, w)$, $w : E \to \mathbb{R}$. The graph can either be directed or undirected. For convenience we define $w(u, v) = \infty$ if $(u, v) \notin E$. We define the ***weight of a path*** as the sum of the weights of the edges along that path.

Example 16.1. *In the following graph the weight of the path* $\langle s, a, b, e \rangle$ *is* 6*. The weight of the path* $\langle s, a, b, s \rangle$ *is* 10*.*



For a weighted graph $G(V, E, w)$ a shortest weighted path from vertex u to vertex v is a path from u to v with minimum weight. There might be multiple paths with equal weight, and if so they are all shortest weighted paths from u to v. We use $\delta_G(u, v)$ to indicate the weight of a shortest path from u to v.

Question 16.2. *What happens if we change the weight of the edge* (b, s) *from* 7 *to* -7*?*

If we allow for negative weight edges then it is possible to create shortest paths of infinite length (in edges) and whose weight is $-\infty$. In Example 16.1 if we change the weight of the edge (b, s) to -7 the shortest path between s and e has weight $-\infty$ since a path can keep going around the cycle $\langle s, a, b, s \rangle$ reducing its weight by 4 each time around. Recall that a path allows repeated vertices—-a simple path does not. For this reason, when computing shortest paths we will need to be careful about cycles of negative weight. As we will see, even if there are no negative weight cycles, negative edge weights make finding shortest paths more difficult. We will therefore first consider the problem of finding shortest paths when there are no negative edge weights. As briefly mentioned before, in this chapter, we are interested in finding single-source shortest paths, defined as follows.

Problem 16.3 (Single-Source Shortest Paths (SSSP))**.** *Given a weighted graph* $G = (V, E, w)$ *and a source vertex* s*, the* single-source shortest path (SSSP) problem *is to find a shortest weighted path from* s *to every other vertex in* V*.*

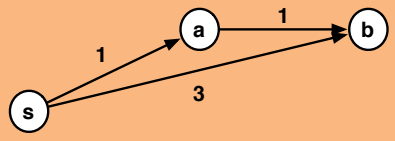
Although there can be many equal weight shortest paths between two vertices, the problem only requires finding one. Also sometimes we only care about the weight $\delta(u, v)$ of the shortest path and not the path itself.

In Chapter 14 we saw how Breadth-First Search (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.

Question 16.4. *Can we use BFS to solve the single-source shortest path problem on weighted graphs?*

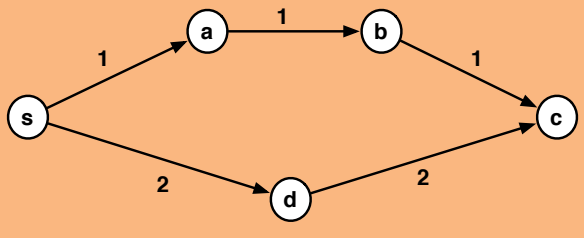
Ignoring weight and using BFS, unfortunately, does not work on weighted graphs.

Example 16.5. *To see why BFS does not work, consider the following directed graph with* 3 *vertices:*



BFS first visits b *and then* a*. When it visits* b*, it assigns it an incorrect weight of* 3*. Since BFS never visit* b *again, it will not find the shortest path going trough* a*, which happens to be shorter.*

Example 16.6. *Another graph where BFS fails to find the shortest paths correctly*



The reason why BFS works on unweighted graphs is quite interesting and helpful for understanding other shortest path algorithms. The key idea behind BFS is to visit vertices in order of their distance from the source, visiting closest vertices first, and the next closest, and so on. More specifically, for each frontier F_i (at round i), BFS has the correct distance from the source to each vertex in the frontier. It then can determine the correct distance for unencountered neighbors that are distance one further away (on the next frontier). We will use a similar idea for weighted paths.

16.2 Dijkstra's Algorithm

Consider a variant of the SSSP problem, where all the weights on the edges are non-negative (i.e. $w : E \to \mathbb{R}^*$). We refer to this as the SSSP$^+$ problem. Dijkstra's algorithm solves efficiently the SSSP$^+$ problem. Dijkstra's algorithm is important because it is efficient and also because it is a very elegant example of an efficient greedy algorithm that guarantees optimality (of solutions) for a nontrivial problem.

In this section, we are going to (re-)discover this algorithms by taking advantage of properties of graphs and shortest paths. Before going further we note that since no edges have negative weights, there cannot be a negative weight cycle. Therefore one can never make a path shorter by visiting a vertex twice—i.e. a path that cycles back to a vertex cannot have lesser 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.

Question 16.7. *Give a brute-force algorithm for finding the shortest path?*

Let us start with a brute force algorithm for the SSSP$^+$ problem, that, for each vertex, considers all simple paths between the source and the vertex and selects the shortest such path.

Question 16.8. *How many simple paths can there be between two vertices in a graph?*

Unfortunately there can be an exponential number of paths between any pair of vertices, so any algorithms that tries to look at all paths is not likely to scale beyond very small instances. We therefore have to try to reduce the work. Toward this end we note that the sub-paths of a shortest path must also be shortest paths between their end vertices, and look at how this can help. This ***sub-paths property*** is a key property of shortest paths. We are going to use this property both now to derive Dijkstra's algorithm, and also again in the next section to derive Bellman-Ford's algorithm for the SSSP problem on graphs that do allow negative edge weights.

Example 16.9. *If a shortest path from Pittsburgh to San Francisco goes through Chicago, then that shortest path includes the shortest path from Pittsburgh to Chicago.*

To see how sub-paths property can be helpful, consider the graph in Example 16.12 and suppose that an oracle has told us the shortest paths to all vertices except for the vertex v. We want to find the shortest path to v.

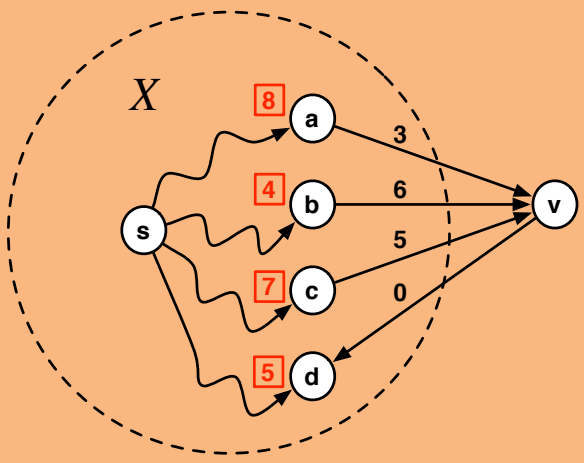
Question 16.10. *Can you find the shortest path to v?*

By inspecting the graph, we know that the shortest path to v goes through either one of a,b, or c. Furthermore, by sub-paths property, we know that the shortest path to v consists of the shortest path to one of a,b, or c, and the edge to v. Thus, all we have to do is to find the u among the in-neighbors of v that minimizes the weight of the path to v, i.e. $\delta_G(s,u)$ plus the additional edge weight to get to v.

Question 16.11. *Could the shortest-path be going through v? If so, then are we still guaranteed to find a shortest path?*

Note that we have decided to find only the shortest paths that are simple, which cannot go through v itself.

Example 16.12. *In the following graph G, suppose that we have found the shortest paths from the source s to all the other vertices except for v; each vertex is labeled with its distance to s. The weight of the shortest path to v is $\min \delta_G(s,a)+3, \delta_G(s,b)+6, \delta_G(s,c)+5$. The shortest path goes through the vertex (a, b, or c) that minimizes the weight, which in this case is vertex b.*



Let's now consider the case where the oracle gave us the shortest path to all but two of the vertices u, v.

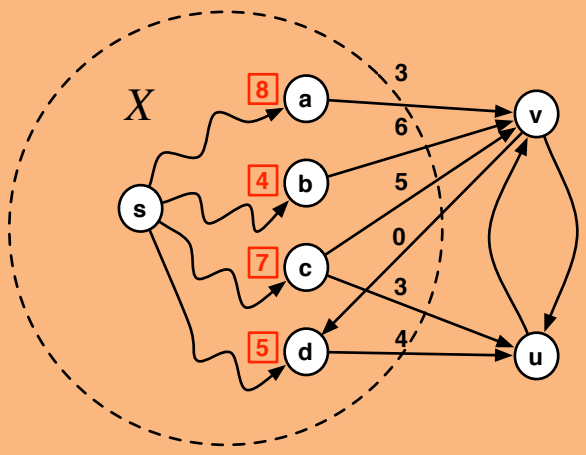
Question 16.13. *Can we find the shortest path to any one of u or v?*

Let X denote the set of vertices for which we know the shortest paths. We may think of this set as the visited set as in graph search. Let's calculate for u (and v) the shortest path that goes through a neighbor in X and then goes to u (v). Consider now the shortest of these two distances and assume without loss of generality that it is the path to u.

Question 16.14. *Could the shortest path to u be this path?*

Since source is in X, we know that the shortest paths to u cross out from X via some edge that goes to either u or v. Since we know that the weight of such a path to u is the smaller, and since all edge weight are non-negative, there cannot be a shorter path that goes to u and than comes back to v again. We thus conclude that the shortest path to v is the path that we have found by considering all incoming edges from X. Example 16.15 illustrates this.

Example 16.15. *In the following graph suppose that we have found the shortest paths from the source s to all the vertices in X. The shortest paths are indicated by labels next to the vertices. The shortest path from the source s to any vertex in Y is the path to vertex d with weight 5 followed by the edge (d, v) with weight 4 and hence total weight 9. If edge weights are non-negative there cannot be any shorter way to get to v, whatever $w(u, v)$ is, therefore we know that $\delta(s, v) = 9$.*



.

Question 16.16. *Can you see how we can generalize this idea?*

Let's try to generalize the argument and consider the case where the oracle tells us the shortest paths from s to 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. $V \setminus X$. The question is whether we can efficiently determine the shortest path to any one of the vertices in Y.

Question 16.17. *What would be the point of doing that?*

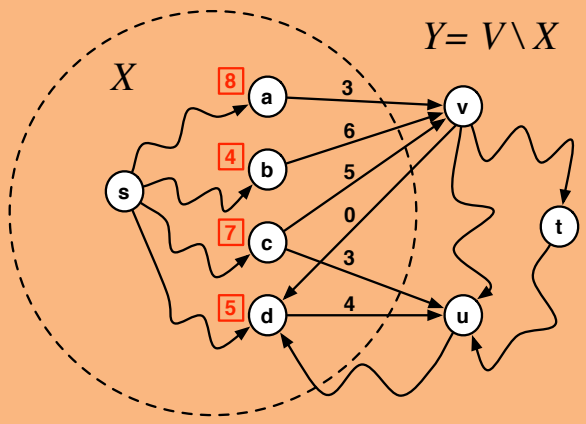
If we could do this, then we would have a crank to repeatedly add new vertices to X until we are done. As in graph search, we call the set of vertices that are neighbors of X but not in X, i.e. $N^+(X) \setminus X$, the frontier.

It should be clear that any path that leaves X has to 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.

Question 16.18. *Can you use this property to identify a vertex $v \in Y$ that is no farther from the source than any other vertex in Y?*

Consider the vertex $v \in Y$ that has an edge to some already visited vertex $x \in X$ and that minimizes the sum $\delta sx + w(x, v)$. Since all paths to Y must go through the frontier when exiting X, and since edges are non-negative, a sub-path cannot be longer than the full path. Thus, no other vertex in Y can be closer to the source than x. See Example 16.19. Furthermore, since all other vertices in Y are farther than v and since all edges are non-negative, the shortest path for v is $\delta sx + w(x, v)$.

Example 16.19. *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 shortest path from the source s to any vertex in Y is the path to vertex d with weight 5 followed by the edge (d, v) with weight 4 and hence total weight 9. If edge weights are non-negative there cannot be any shorter way to get to v, whatever $w(u, v)$ is, therefore we know that $\delta(s, v) = 9$.*



.

This intuition is stated and proved in Lemma 16.20. The Lemma tells us that once we know the shortest paths to a set X we can add more vertices to X with their shortest paths. This gives us a crank that can churn out at least one new vertex on each round. Dijkstra's algorithm simply does exactly this: it turns the crank until all vertices are visited.

Question 16.21. *You might have noticed that the terminology that we used in explaining Dijkstra's algorithm closely relates to that of graph search. Does this algorithm remind you of a particular graph search that we discussed?*

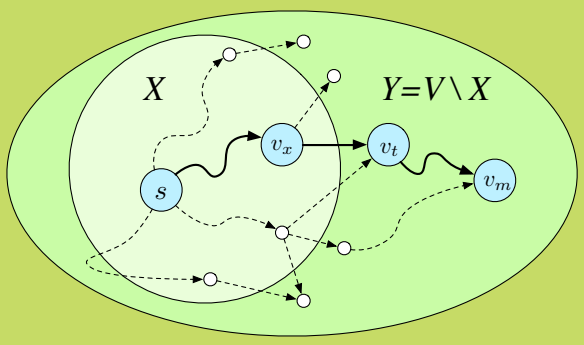
Recall that priority-first search is a graph search in which on each step we visit the frontier vertices with the highest "priority". In our case the visited set is X, and the priority is defined in terms of $p(v)$, the shortest-path weight consisting of a path to $x \in X$ with an additional edge from x to v. In other words, this algorithm is actually an instance of priority-first search.

We are now ready to define precisely Dijkstra's algorithm.

Lemma 16.20 (Dijkstra's Property)**.** *Consider a (directed) weighted graph $G = (V, E, w)$, $w : E \to \mathbb{R}^*$, 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*

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

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



In 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 a any vertex in Y.

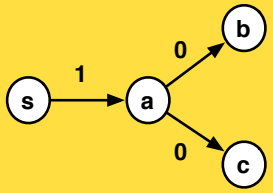
Proof. Consider a vertex $v_m \in Y$ such that $\delta_G(s, v_m) = \min_{v \in Y} \delta_G(s, v)$, and a shortest path from s to v_m in G. The path must go through an edge from a vertex $v_x \in X$ to a vertex v_t in Y (see the figure). Since there are no negative weight edges, and the path to v_t is 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. □

Implication: This gives us a way to easily find $\delta_G(s, v)$ for at least one vertex $v \in Y$. In particular for all $v \in Y$ where $p(v) = \min_{y \in Y} p(y)$, it must be the case that $p(v) = \delta_G(s, v)$ since there cannot be a shorter path. Also we need only consider the frontier vertices in calculating $p(v)$, since for others in Y, $p(y)$ is infinite.

Algorithm 16.22 (Dijkstra's Algorithm)**.** *For a weighted graph $G = (V, E, w)$ and a source s, Dijkstra's algorithm is priority-first search on G starting at s with $d(s) = 0$, using priority $p(v) = \min_{x \in X}(d(x) + w(x, v))$ (to be minimized), and setting $d(v) = p(v)$ when v is visited.*

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

Remark 16.23. *It may be tempting to think that Dijkstra's algorithm visits vertices strictly in increasing order of shortest-path weight from the source, visiting vertices with equal shortest-path weight on the same round. This is not true. To see this consider the example below and convince yourself that it does not contradict our reasoning.*



Lemma 16.24. *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 16.20, $p(v) = \delta_G(s, v)$, 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. □

16.2.1 Implementing Dijkstra's Algorithm with a Priority Queue

We now discuss how to implement this abstract algorithm efficiently using a priority queue to maintain $P(v)$. We use a priority queue that supports `deleteMin` and `insert`. The priority-queue based algorithm is given in Algorithm 16.25. This variant of the algorithm only adds one vertex at the time even if there are multiple vertices with equal distance that could be added in parallel. It can be made parallel by generalizing priority queues, but we leave this as an exercise.

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 (Line 9 in the code) and, if it hasn't already been visited, visits it by adding $(x \mapsto d)$ to the table of visited vertices (Line 16), and then adds all its neighbors v to Q along with the priority $d(x) + w(x, v)$ (i.e. the distance to v through x) (Lines 17 and 18). 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, but what is important is that the minimum distance will always be pulled out first. Line 12 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.

We note that there are a couple other variants on Dijkstra's algorithm using Priority Queues.

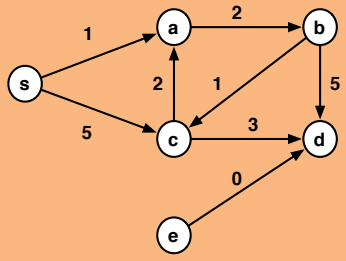
Algorithm 16.25 (Dijkstra's Algorithm using Priority Queues)**.**

```
1  function dijkstraPQ(G, s) =
2  let
3
4    % requires:
5        ∀(x ↦ d) ∈ X, d = δ_G(s, x)
6        {(d, y) ∈ Q | y ∈ V \ X} = {(d + w(x, y), y) : (x ↦ d) ∈ X, y ∈ N⁺(x) \ X}
7    % returns: {x ↦ δ_G(s, x) : x ∈ R_G(s)}
8    function dijkstra (X, Q) =
9      case [PQ.deleteMin](Q) of
10       (⊥, _) ⇒ X
11     | ((d, v), Q′) ⇒
12         if [(v, _) ∈ X] then
13           dijkstra (X, Q′)
14         else
15           let
16             X′ = [X ∪ {(v, d)}]
17             function relax (Q, (u, w)) = [PQ.insert](d + w, u) Q
18             Q″ = [iter] relax Q′ [N_G(v)]
19           in dijkstra (X′, Q″) end
20 in
21   dijkstra ({}, [PQ.insert](0, s) {})
22 end
```

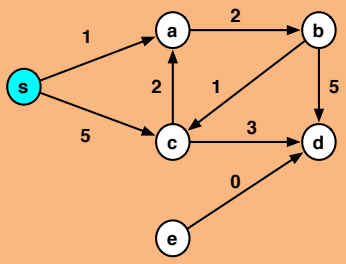
Firstly we could check inside the `relax` function whether u is already in X and if so not insert it into the Priority Queue. This does not affect the asymptotic work bounds but probably would give some improvement in practice. Another variant is to decrease the priority of the neighbors instead of adding duplicates to the priority queue. This requires a more powerful priority queue that supports a `decreaseKey` function.

Cost of Dijkstra's Algorithm. We now consider the work of the Priority Queue version of Dijkstra's algorithm, whose code is shown in Algorithm 16.25. Let's firs consider the ADT's that we will use along with their cost. For the priority queue, we assume `PQ.insert` and `PQ.deleteMin` have $O(\log n)$ work and span. To represent the graph, we can either use a table or an array, mapping vertices to their neighbors along with the weight of the edge in between. For the purposes of Dijkstra's algorithm, it suffices to use the tree based costs for tables. Note that since we don't update the graph, we don't need a single threaded array. To represent the table of distances to visited vertices, we can use a table, an array sequence, or a single threaded array sequences.

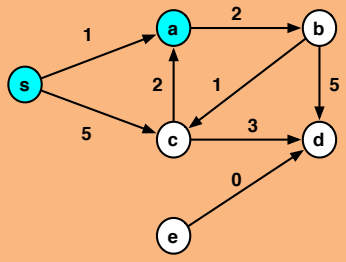
Example 16.26. *An example run of Dijkstra's algorithm. Note that after visiting s, a, and b, the queue Q contains two distances for c corresponding to the two paths from s to c discovered thus far. The algorithm takes the shortest distance and adds it to X. A similar situation arises when c is visited, 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.*



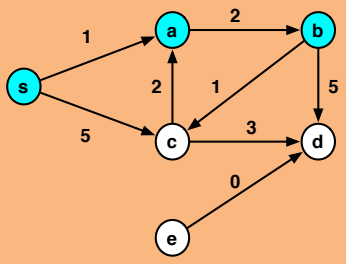
X: Q: s @ 0



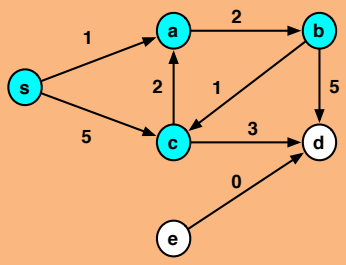
X: s @ 0 Q: a @ 1, c @ 5



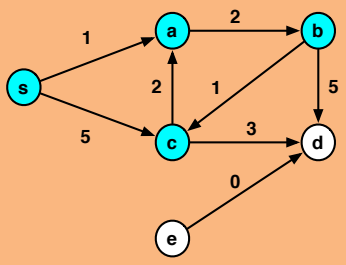
X: s @ 0, a @ 1 Q: b @ 2, c @ 5



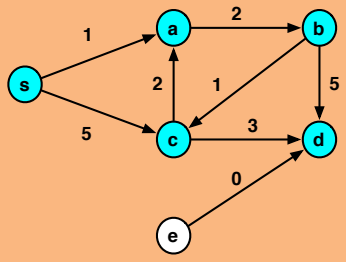
X: s @ 0, a @ 1, b @ 3 Q: c @ 4, c @ 5, d @ 8



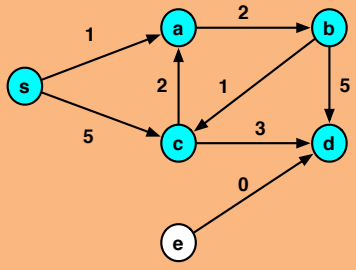
X: s @ 0, a @ 1, b @ 2, c @ 4 Q: c @ 5, a @ 6, d @ 7, d @ 8



X: s @ 0, a @ 1, b @ 2, c @ 4 Q: d @ 7, d @ 8



X: s @ 0, a @ 1, b @ 2, c @ 4, d @ 7 Q: d @ 8



X: s @ 0, a @ 1, b @ 2, c @ 4, d @ 7 Q:

Operation	Line	# of calls	PQ	Tree Table	Array	ST Array
`deleteMin`	Line 9	$O(m)$	$O(\log m)$	-	-	-
`insert`	Line 17	$O(m)$	$O(\log m)$	-	-	-
Priority Q total			$O(m \log m)$	-	-	-
`find`	Line 12	$O(m)$	-	$O(\log n)$	$O(1)$	$O(1)$
`insert`	Line 16	$O(n)$	-	$O(\log n)$	$O(n)$	$O(1)$
Distances total			-	$O(m \log n)$	$O(n^2)$	$O(m)$
$N_G(v)$	Line 18	$O(n)$	-	$O(\log n)$	$O(1)$	-
`iter`	Line 18	$O(m)$	-	$O(1)$	$O(1)$	-
Graph access total			-	$O(m + n \log n)$	$O(m)$	-

Figure 16.1: The costs for the important steps in the algorithm `dijkstraPQ`.

To analyze the work, we calculate the work for each different kind of operation and sum them up to find the total work. Figure 16.1 summarizes the costs of the operations, along with the number of calls made to each operation.

The algorithm in includes a box around each operation on the graph G, the set of visited vertices X, or the priority queue `PQ`. The `PQ.insert` in Line 21 is called only once, so we can safely ignore it. Of the remaining operations, The `iter` and $N_G(v)$ on Line 18 are on the graph, Lines 12 and 16 are on the table of visited vertices X, and Lines 9 and 17 are on the priority queue Q.

We can calculate the total number of calls to each operation by noting that the body of the let starting on Line 15 is only run once for each vertex. Thus, Line 16 and $N_G(v)$ on Line 18 are only called $O(n)$ times. All other operations are performed once for each edge. The total work for Dijkstra's algorithm using a tree table is therefore $O(m \log m + m \log n + m + n \log n)$. Since $m \leq n^2$, the total work is $O(m \log n)$.

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

Based on the table one should note that when using either tree tables or single threaded sequences, the cost is no more than the cost of the priority queue operations. Therefore there is no asymptotic advantage of using one over the other; there might, however, be differences in constant factors. One should also note that using regular purely functional arrays is not a good idea, because the cost is then dominated by the insertions and the algorithm runs in $\Theta(n^2)$ work.

16.3 The Bellman Ford Algorithm

We now turn to solving the single source shortest path problem in the general case where we allow negative weights in the graph. One might ask how negative weights make sense. When considering distances on a map for example, negative weights do not make sense (at least without time travel), but many other problems reduce to shortest paths, and in these reductions

negative weights do show up.

Before proceeding we note that if there is a negative weight cycle (the sum of weights on the cycle is negative) reachable from the source, then there cannot be a solution to the single-source shortest path problem, as discussed earlier. In such a case, we would like the algorithm to indicate that such a cycle exists and terminate.

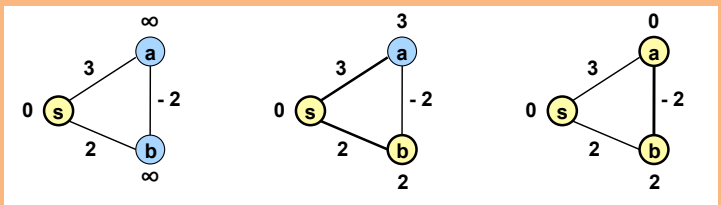
Exercise 16.27. *Consider the following* currency exchange *problem: given the a set currencies, a set of exchange rates between them, and a source currency s, find for each other currency v the best sequence of exchanges to get from s to v. Hint: how can you convert multiplication to addition.*

Exercise 16.28. *In your solution to the previous exercise can you get negative weight cycles? If so, what does this mean?*

Question 16.29. *Why can't we use Dijkstra's algorithm to compute shortest path when there are negative edges?*

Recall that in our development of Dijkstra's algorithm we assumed non-negative edge weights. This both allowed us to only consider simple paths (with no cycles) but more importantly played a critical role in arguing the correctness of Dijkstra's property. More specifically, Dijkstra's algorithm is based on the assumption that the shortest path to the vertex v in the frontier that is closest to the set of visited vertices, whose distances have been determined, can be determined by considering just the incoming edges of v. With negative edge weights, this is not true anymore, because there can be a shorter path that ventures out of the frontier and then comes back to v.

Example 16.30. *To see where Dijkstra's property fails with negative edge weights consider the following example.*



Dijkstra's algorithm would visit b then a and leave b with a distance of 2 instead of the correct distance 1. The problem is that when Dijkstra visits b, it fails to consider the possibility of there being a shorter path from a to b (which is impossible with non-negative edge weights).

Question 16.31. *How can we find shortest paths on a graph with negative weights?*

Question 16.32. *Recall that for Dijkstra's algorithm, we started with the brute-force algorithm and realized a key property of shortest paths. Do you recall the property?*

A property we can still take advantage of, however, is that the sub-paths of a shortest paths themselves are shortest. Dijkstra's algorithm took advantage of this property by building longer paths from shorter ones, i.e., by building shortest paths in non-decreasing order. With negative edge weights this does not work anymore, because paths can get shorter as we add edges.

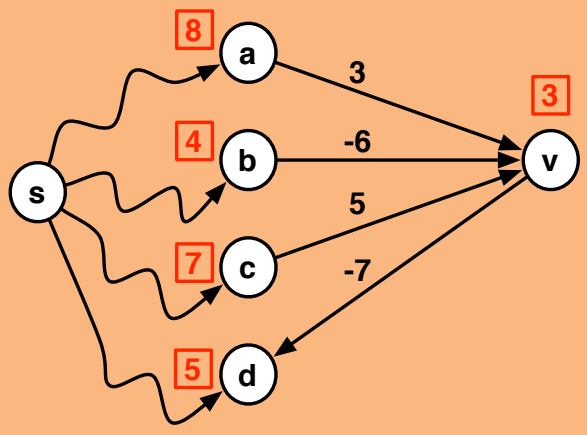
But there is another way to use the same property: building paths that contain more and more edges. To see how, suppose that you have found the shortest paths from source to all vertices with k or fewer edges.

Question 16.33. *How can you update the shortest paths for $k+1$ edges? That is you want to find the shortest paths that contain $k+1$ or fewer edges.*

We can compute the shortest paths with $k+1$ or fewer edges by extending all paths by one edge if this is beneficial. More specifically, all we need to do is consider each vertex v and all its incoming edges and pick the shortest path to that vertex that arrives at a neighbor u using k or fewer edges and then takes the edge from u to v.

To make this more precise, let's define $\delta_G^k(s,t)$ as the shortest weighted path from s to t using at most k edges.

Example 16.34. *In the following graph G, suppose that we have found the shortest paths from the source s to vertices using k or fewer edges; each vertex u is labeled with its k-distance to s, written $\delta_G^k(s,u)$. The weight of the shortest path to v using $k+1$ or fewer edges is*
is $\min \delta_G(s,v), \min \delta_G(s,a)+3, \delta_G(s,b)+6, \delta_G(s,c)+5$ *The shortest path with at most $k+1$ edges has weight -2 and goes through vertex b.*



Based on this idea, we can construct paths with more and more edges. We start by determining $\delta_G^0(s,v)$ for all $v \in V$. Since no vertex other than the source is reachable with a path of length 0, $\delta_G^0(s,v) = \infty$ for all vertices other than the source. Then we determine $\delta_G^1(s,v)$ for all $v \in V$, and iteratively, $\delta_G^{k+1}(s,v)$ based on all $\delta_G^k(s,v)$ for $k = 2, \ldots$. To calculate the updated distances with at most $k+1$ edges the algorithm, we can use the shortest path with k edges to each of its in-neighbors and then adds in the weight of the one additional edge. More precisely, for each vertex v,

$$\delta^{k+1}(v) = \min(\delta^k(v), \min_{x \in N^-(v)} (\delta^k(x) + w(x,v))).$$

Remember that $N^-(v)$ indicates the in-neighbors of vertex v.

Question 16.35. *Can the distances for a vertex stays the same?*

Since the new distance for a vertex is calculated in terms of the distances from most recent iteration, if the distances for all vertices remain unchanged, then they will continue remaining unchanged after one more iteration. It is thus unnecessary to continue iterating after the distances converge.

Question 16.36. *When should we stop?*

We can thus stop when the distances converge.

Algorithm 16.39 (Bellman Ford)**.**

```
1  function BellmanFord(G = (V, E), s) =
2  let
3      % requires: ∀v ∈ V, D_v = δ^k_G(s, v)
4      function BF(D, k) =
5      let
6          D' = {v ↦ min(D[v], min_{u∈N⁻_G(v)}(D[u] + w(u, v))) : v ∈ V}
7      in
8          if (k = |V|) then ⊥
9          else if (all{D[v] = D'[v] : v ∈ V}) then D
10         else BF(D', k + 1)
11     end
12     D = {v ↦ if v = s then 0 else ∞ : v ∈ V}
13 in BF(D, 0) end
```

Question 16.37. *Is convergence guaranteed?*

Convergence, however is not guaranteed. For example, if we have a negative cycle reachable from the source then, the distances will keep getting smaller and smaller at each iteration, which is expected.

Question 16.38. *Can we detect such a situation?*

We can detect negative cycles by noticing that the distances do not converge even after $|V|$ iterations. This is because, a simple (acyclic) path in a graph can include at most $|V|$ edges and, in the absence of negative-weight cycles, there always exist an acyclic shortest path.

Algorithm 16.39 defines the Bellman Ford algorithm based on these ideas. In Line 8 the algorithm returns $\perp$ (undefined) if there is a negative weight cycle reachable from s. In particular since no simple path can be longer than $|V|$, if the distance is still changing after $|V|$ rounds, then there must be a negative weight cycle that was entered by the search. An illustration of the algorithm over several steps is shown in Figure 16.2.

Theorem 16.40. *Given a directed weighted graph $G = (V, E, w)$, $w : E \to R$, and a source s, the* `BellmanFord` *algorithm returns either*

1. *$\delta_G(s, v)$ for all vertices reachable from s, or*
2. *$\perp$ if there is a negative weight-cycle in the graph reachable from s.*

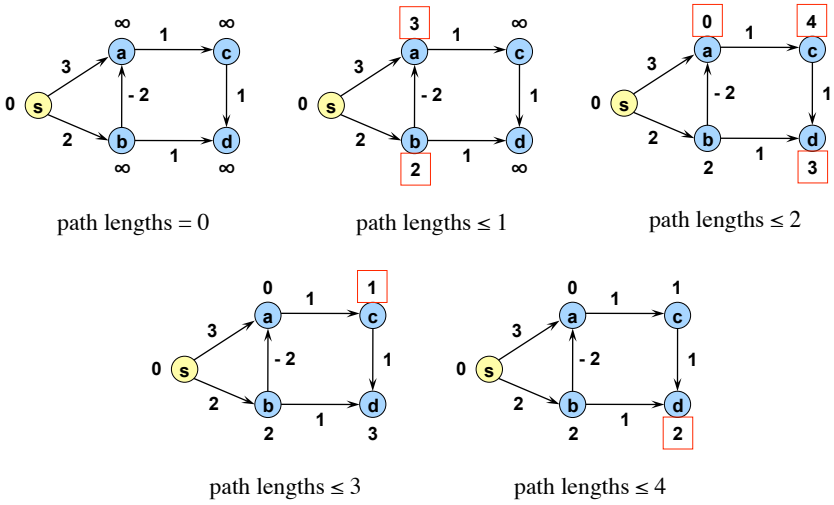


Figure 16.2: Steps of the Bellman Ford algorithm. The numbers with squares indicate what changed on each step.

Proof. By induction on the number of edges k in a path. The base case is correct since $D_s = 0$. For all $v \in V \setminus s$, on each step a shortest (s, v) path with up to $k + 1$ edges must consist of a shortest (s, u) path of up to k edges followed by a single edge (u, v). Therefore if we take the minimum of these we get the overall shortest path with up to $k+1$ edges. For the source the self edge will maintain $D_s = 0$. The algorithm can only proceed to n rounds if there is a reachable negative-weight cycle. Otherwise a shortest path to every v is simple and can consist of at most n vertices and hence $n - 1$ edges. □

Cost of Bellman-Ford. To analyze the cost of Bellman-Ford we consider two different representations for graphs, one using tables and the other using sequences.

For a table-based representation of the graph, we use a table mapping each vertex to a table of neighbors along with their real-values weights. We represent the distances D as a table mapping vertices to their distances. Let's consider the cost of one call to BF, not including the recursive calls. The only nontrivial computations are on Lines 6 and 9. Line 6 consists of a tabulate over the vertices. As the cost specification for tables indicate, to calculate, the work we take the sum of the work for each vertex, and for the span we take the maximum of the spans, and add $O(\log n)$. Now consider what the algorithm does for each vertex. First, it has to find the neighbors in the graph (using a `find G v`). This requires $O(\log |V|)$ work and span. Then it involves a map over the neighbors. Each instance of this map requires a find in the distance table to get $D[u]$ and an addition of the weight. The find takes $O(\log |V|)$ work and span. Finally there is a reduce that takes $O(1 + |N_G(v)|)$ work and $O(\log |N_G(v)|)$ span. Using

$n = |V|$ and $m = |E|$, the work is

$$\begin{aligned} W &= O\left(\sum_{v \in V}\left(\log n + |N_G(v)| + \sum_{u \in N_G(v)}(1 + \log n)\right)\right) \\ &= O\left((n+m)\log n\right). \end{aligned}$$

Similarly, span is

$$\begin{aligned} S &= O\left(\max_{v \in V}\left(\log n + \log|N_G(v)| + \max_{u \in N(v)}(1 + \log n)\right)\right) \\ &= O(\log n). \end{aligned}$$

The work and span of Line 9 is simpler to analyze since it only involves a tabulate and a reduction: it requires $O(n \log n)$ work and $O(\log n)$ span.

Since the number of calls to BF is bounded by n, as discussed earlier. These calls are done sequentially so we can multiply the work and span for each call by the number of calls giving:

$$\begin{aligned} W(n,m) &= O(nm \log n) \\ S(n,m) &= O(n \log n) \end{aligned}$$

Let's now consider the cost with a sequence representation of graphs. If we assume the vertices are the integers $\{0, 1, \ldots, |V| - 1\}$ then we can use sequences to represent the graph. Instead of using a `find` for a table, which requires $O(\log n)$ work, we can use `nth` (indexing) requiring only $O(1)$ work. This improvement in costs can be applied for looking up in the graph to find the neighbors of a vertex, and looking up in the distance table to find the current distance. By using the improved costs we get:

$$\begin{aligned} W &= O\left(\sum_{v \in V}\left(1 + |N_G(v)| + \sum_{u \in N_G(v)} 1\right)\right) \\ &= O(m) \\ S &= O\left(\max_{v \in V}\left(1 + \log|N_G(v)| + \max_{u \in N(v)} 1\right)\right) \\ &= O(\log n) \end{aligned}$$

and hence the overall complexity for *`BellmanFord`* with array sequences is:

$$\begin{aligned} W(n,m) &= O(nm) \\ S(n,m) &= O(n \log n) \end{aligned}$$

By using array sequences we have reduced the work by a $O(\log n)$ factor.

16.4 Problems

Exercise 16.41. *We mentioned that we are sometimes only interested in the weight of a shortest path, rather than the path itself. For a weighted graph $G = (V, E, w)$, assume you are given the distances $\delta_G(s, v)$ for all vertices $v \in V$. For a particular vertex u, describe how you could determine the vertices P in a shortest path from s to u in $O(p)$ work, where $p = \sum_{v \in P} d^-(v)$.*

Exercise 16.42. *Prove that the sub-paths property holds for any graph, also in the presence of negative weights.*

Exercise 16.43. *Argue that if all edges in a graph have weight 1 then Dijkstra's algorithm as described visits exactly the same set of vertices in each round as BFS does.*

.