Chapter 12

Shortest Paths

In this chapter we will cover problems involving finding the shortest path between vertices in a graph with weights (lengths) on the edges. One obvious application is in finding the shortest route from one address to another, however shortest paths have many other application. We will look at a version of the problem assuming we are finding distances from a single source to all other vertices in the graph, and two variants depending on whether we allow negative edge weights or not. We first introduce weighted graphs.

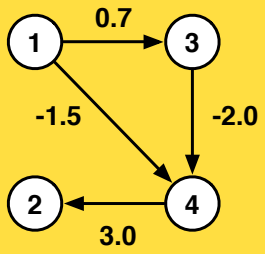
12.1 Weighted Graphs and Their Representation

Many applications of graphs require associating weights or other values with the edges of a graph. Such graphs can be formally defined as follows.

Definition 12.1 (Weighted and Edge-Labeled Graphs)**.** *A* weighted graph *or an* edge-labeled graph *is a triple $G = (E, V, w)$ where $w \colon E \to L$ is a function mapping edges or directed edges to their values, and L is the set of possible values.*

In a graph, if the data associated with the edges are real numbers, we often use the term "weight" to refer to the values, and use the term "weighted graph" to refer to the graph. In the general case, we use the terms "edge label" or "edge values" to refer to the values. Weights or other values on edges could represent many things, such as a distance, or a capacity, or the strength of a relationship.

Example 12.2. *An example directed weighted graph.*



Chapter 8 described three different representations of graphs suitable for parallel algorithms: edge sets, adjacency tables, and adjacency sequences. We can extend each of these representations to support edge values by separately representing the function from edges to values using a table (mapping)—the table maps each edge (or arc) to its value. This representation allows looking up the edge value of an edge $e = (u, v)$ by using a table lookup. We call this an *edge table*.

Example 12.3. *For the weighted graph in Example 12.2, the edge table is:*

$$W = \{(1,3) \mapsto 0.7,\ (1,4) \mapsto -1.5,\ (3,4) \mapsto -2.0,\ (4,2) \mapsto 3.0\}$$

A nice property of an edge table is that it works uniformly with all representations of the structure of a graph, and is clean since it separates the edge values from the structural information. However keeping the edge table separately creates redundancy, wasting space and possibly requiring extra work to access the edge values. The redundancy can be avoided by storing the edge values directly with the edge information. For edge sets we have effectively already done this with an edge table, which maps each edge to its value. We can extend adjacency tables by replacing each set of neighbors, with a mapping from each neighbor to the weight on the edge to that neighbor. We can extend an adjacency sequences by creating a sequence of neighbor-value pairs for each out edge of a vertex. This is illustrated in the following example.

Example 12.4. *For the weighted graph in Example 12.2, the adjacency table representation is*

$$G = \{1 \mapsto \{3 \mapsto 0.7, 4 \mapsto -1.5\}, 3 \mapsto \{4 \mapsto -2.0\}, 4 \mapsto \{2 \mapsto 3.0\}\},$$

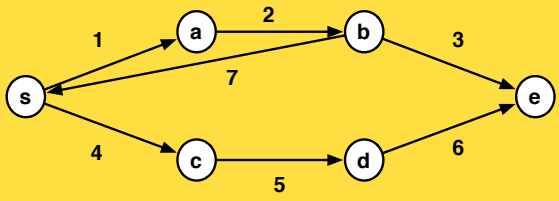
and the adjacency sequence representation is (assuming sequences are 1 based):

$$G = \langle \langle (3, 0.7),\ (4, -1.5) \rangle, \langle\ \rangle, \langle (4, -2.0) \rangle, \langle (2, 3.0) \rangle \rangle.$$

12.2 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 12.5. *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.

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 12.5 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 path when there are no negative edge weights.

Computing shortest paths is important in many practical applications. In fact, there are several variants of this problem such as the "single source" and the "multiple source" versions, both with or without negative weights. In this chapter we are interested in the single source version.

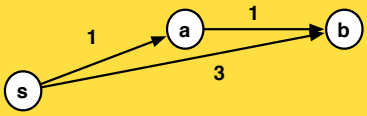
Problem 12.6 (Single-Source Shortest Paths (SSSP))**.** *Given a weighted graph* $G = (V, E, w)$ *and a source vertex* s*, the* single-source shortest path (SSSP) problem *is to find a shortest weighted path from* s *to every other vertex in* V*.*

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.

Exercise 12.7. *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 $p = \sum_{v \in P} d^-(v)$.*

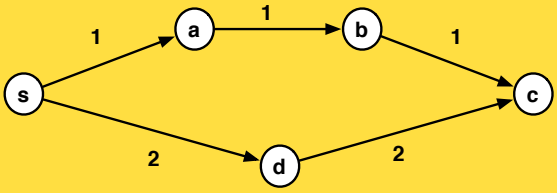
In Chapter 10 we saw how Breadth-First Search (BFS) can be used to solve the single-source shortest path problem on graphs without edge weights, or, equivalently, where all edges have weight 1. BFS, unfortunately, does not work on weighted graphs.

Example 12.8. *To see why BFS does not work, consider the following directed graph with 3 vertices:*



In this example, 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 12.9. *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, 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.

12.3 Dijkstra's Algorithm

Dijkstra's algorithm solves the SSSP problem when all the weights on the edges are non-negative (i.e. $w : E \to \mathbb{R}^*$). We will refer to this variant of SSSP as the SSSP$^+$ problem. Dijkstra's is

an important algorithm both because it is an efficient algorithm for an important problem, but also because it is a very elegant example of an efficient greedy algorithm that generates optimal solutions on a nontrivial task.

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. This means we only need to consider simple paths when searching for a 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. 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.

Example 12.10. *If a shortest path from Pittsburgh to San Francisco goes through Chicago, then that shortest path includes the shortest path from Pittsburgh to Chicago.*

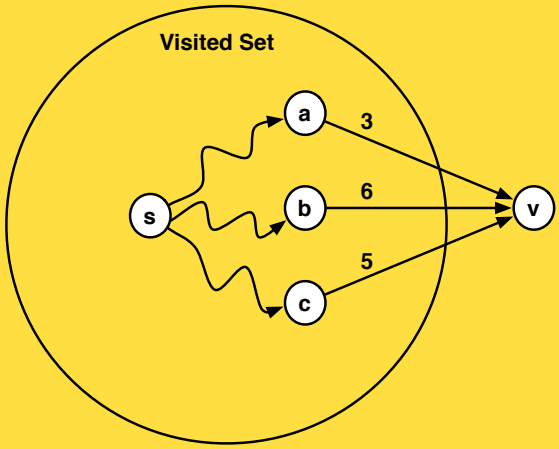
Exercise 12.11. *Prove that the sub-paths property holds for any graph, also in the presence of negative weights.*

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. To see how this property can be helpful, suppose an oracle has told you the shortest paths to all vertices except for one vertex, v. We can now find the shortest path to v, because we know the shortest subpath from the source to the vertex u immediately before v. All we need to do is find which u among the in-neighbors of v minimizes the weight of the path to u, i.e. $\delta_G(s, u)$ plus the additional edge weight to get to v. See Example 12.12. We note that this argument relies on the fact that a shortest path must be simple so cannot go through v itself.

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

Example 12.12. *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. 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.*



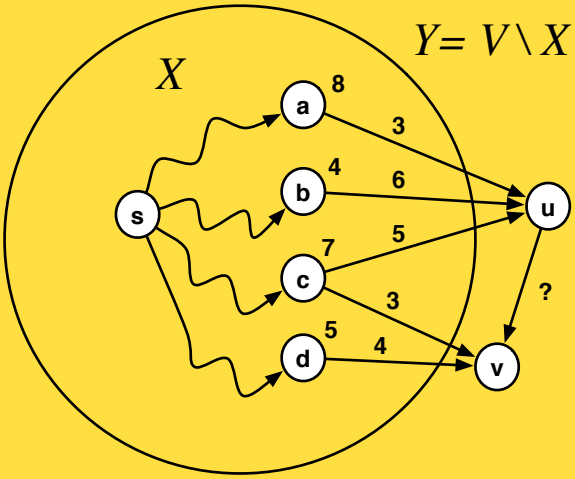
leave X via a vertex in the frontier. Basically the intuition now is that at least one path that goes from X directly to a vertex on the frontier, is an overall shortest path to any vertex in Y. This is because all paths to Y must go through the frontier when exiting X, and since edges are non-negative, a subpath cannot be longer than the full path. See Example 12.13.

This intuition is formalized in Lemma 12.14 along with a proof. 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. Note that this approach is actually an instance of priority-first search. 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. This gives us the following rather concise definition of Dijkstra's algorithm.

Algorithm 12.15 (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.*

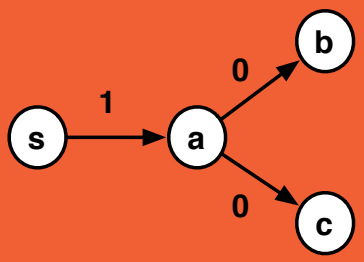
Note that Dijkstra's algorithm will visit vertices in non-decreasing shortest-path weight since on each step it visits unvisited vertices that have the minimum shortest-path weight from s.

Example 12.13. *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 any vertex in X directly to a 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 $\delta(s, v) = 9$.*



.

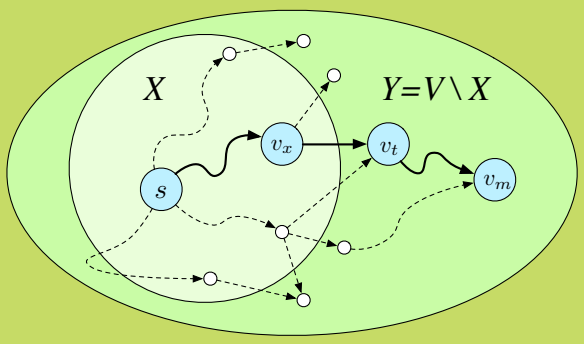
Remark 12.16. *It may be tempting to think that Dijkstra's algorithm visits vertices strictly in increasing order of shortest-path weight from the source, visiting vertices with equal shortest-path weight on the same step. This is not true. To see this consider the example below and convince yourself that it does not contradict our reasoning.*



Lemma 12.14 (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 subpath 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.

Lemma 12.17. *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 12.14, $p(v) = \delta_G(s, v)$, 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. □

Algorithm 12.19 (Dijkstra's Algorithm using Priority Queues)**.**

```
1  function dijkstraPQ(G, s) =
2  let
3    % requires:
4      ∀_{(x↦d)∈X}  d = δ_G(s,x)
5      {(d,y) ∈ Q | y ∈ V \ X} = {(d + w(x,y), y) : (x ↦ d) ∈ X, y ∈ N⁺(x) \ X}
6    % returns:  {x ↦ δ_G(s,x) : x ∈ R_G(s)}
7    function dijkstra' (X, Q) =
8      case PQ.deleteMin(Q) of
9        (NONE,_) ⇒ X
10     | (SOME(d,v), Q') ⇒
11         if (v,_) ∈ X then
12           dijkstra' (X, Q')
13         else
14           let
15             val X' = X ∪ {(v,d)}
16             function relax (Q, (u, w)) =
17               PQ.insert(d + w, u) Q
18             val Q'' = iter relax Q' N_G(v)
19           in dijkstra'(X', Q'') end
20 in
21   dijkstra'({}, PQ.insert(0, s) {})
22 end
```

Exercise 12.18. *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.*

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 12.19. 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 8 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 15), 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 10 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. 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.

Costs of Dijkstra's Algorithm. We now consider the work of the Priority Queue version of Dijkstra's algorithm. The algorithm is sequential so the span is the same as the work. We analyze the work by counting up the number of operations. The algorithm in Figure 12.19 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 ignore it. Of the remaining operations, The `iter` and $N_G(v)$ on line 18 are on the graph, lines 10 and 15 are on the table of visited vertices X, and lines 8 and 17 are on the priority queue Q. For the priority queue operations, we assume `PQ.insert` and `PQ.deleteMin` have $O(\log n)$ work and span. For the graph, we can either use a tree-based table or an array to access the neighbors[1] There is no need for single threaded array since we are not updating the graph. For the table of distances to visited vertices we can use a tree table, an array sequence, or a single threaded array sequences. Figure 12.1 summarizes the costs of the operations, along with the number of calls made to each operation.

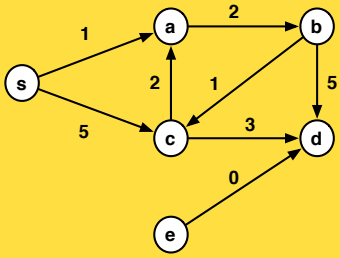
We can calculate the total number of calls to each operation by noting that the body of the let starting on line 14 is only run once for each vertex. This means that line 15 and $N_G(v)$ on line 18 are only called $O(n)$ times. Everything else is done once for every edge.

Based on the table one should note that when using either tree tables or single threaded arrays 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, although there might be a constant factor speedup. One should also note that using regular purely functional arrays is not a good idea since then the cost is dominated by the insertions and the algorithm runs in $\Theta(n^2)$ work.

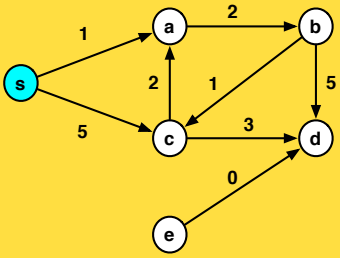
The total work for Dijkstra's algorithm using a tree table $O(m \log m + m \log n + m + n \log n) = O(m \log n)$ since $m \leq n^2$.

[1] We could also use a hash table, but we have not yet discussed them.

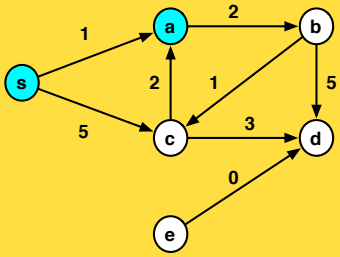
Example 12.20. *An example run of Dijkstra's algorithm. Note that after visiting s, a, and b, the queue Q contains two distances for c corresponding to the two paths from s to c discovered thus far. The algorithm takes the shortest distance and adds it to X. A similar situation arises when c is visited, 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.*



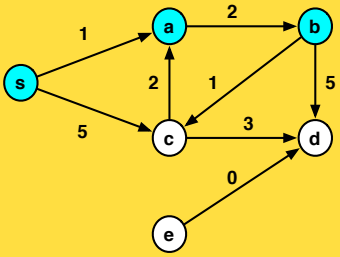
X	Q
	s @ 0



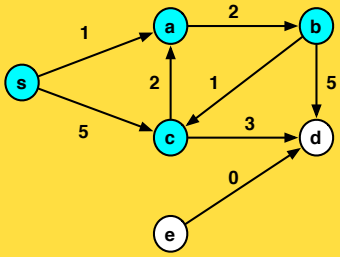
X	Q
s @ 0	a @ 1
	c @ 5



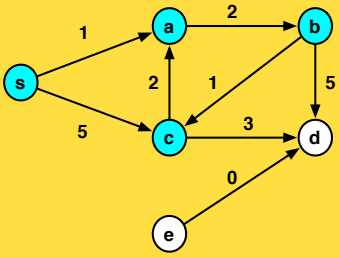
X	Q
s @ 0	b @ 2
a @ 1	c @ 5



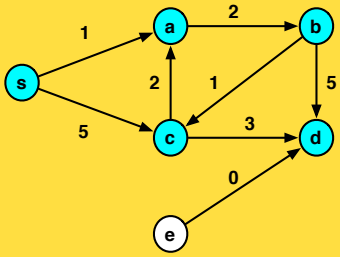
X	Q
s @ 0	c @ 4
a @ 1	c @ 5
b @ 3	d @ 8



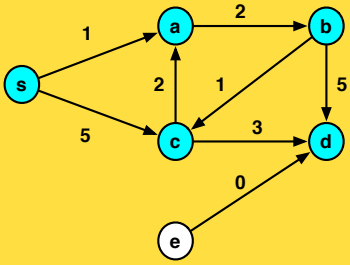
X	Q
s @ 0	c @ 5
a @ 1	a @ 6
b @ 2	d @ 7
c @ 4	d @ 8



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



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



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

Operation	Line	# of calls	PQ	Tree Table	Array	ST Array
`deleteMin`	8	$O(m)$	$O(\log m)$	-	-	-
`insert`	17	$O(m)$	$O(\log m)$	-	-	-
Priority Q total			$O(m \log m)$	-	-	-
`find`	10	$O(m)$	-	$O(\log n)$	$O(1)$	$O(1)$
`insert`	15	$O(n)$	-	$O(\log n)$	$O(n)$	$O(1)$
Distances total			-	$O(m \log n)$	$O(n^2)$	$O(m)$
$N_G(v)$	18	$O(n)$	-	$O(\log n)$	$O(1)$	-
`iter`	18	$O(m)$	-	$O(1)$	$O(1)$	-
Graph access total			-	$O(m + n \log n)$	$O(m)$	-

Figure 12.1: The costs for the important steps in the algorithm `dijkstraPQ`.

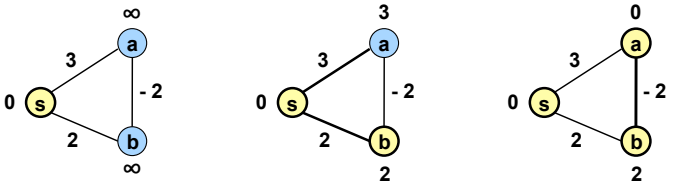
12.4 The Bellman Ford Algorithm

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. If talking about distances on a map, they probably do not (at least without time travel), but various other problems reduce to shortest paths, and in these reductions negative weights 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, as discussed earlier. In the case that a negative weight cycle can be reached from the source vertex, we would like solutions to the SSSP problem to return some indicator that such a cycle exists and terminate.

Exercise 12.21. *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 12.22. *In your solution to the previous exercise can you get negative weight cycles? If so, what does this mean?*

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. To see where Dijkstra's property fails with negative edge weights consider the following very simple 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 the overall shortest path directly from the visited set to the frontier is not necessarily the shortest path to that vertex. There can be a shorter path that first steps further away (to a in the example), and then reduces the length with the negative edge weight. A property we can still take advantage of, however, is that the subpaths of a shortest paths themselves are shortest. Using this property, we can build shortest paths in a slightly different way: by bounding the number of edges on a path.

If we have computed the shortest path with ℓ or less edges from s to all vertices, we can computer the shortest paths for $\ell + 1$ edges by considering extending all paths by one edge. To do this all we need to do is consider all incoming edges of each vertex. This is the idea behind the Bellman-Ford algorithm.

We define the following

$\delta^l_G(s,t)$ = the shortest weighted path from s to t using at most l edges.

We start by determining $\delta^0_G(s,v)$ for all $v \in V$. Since no vertex other than the source is reachable with a path of length 0, $\delta^0_G(s,v) = \infty$ for all vertices other than the source. Then perhaps we can use this to determine $\delta^1_G(s,v)$ for all $v \in V$. In general we want to determine $\delta^{k+1}_G(s,v)$ based on all $\delta^k_G(s,v)$. The question is how do we calculate this. It turns out to be easy since to determine the shortest path with at most $k+1$ edges to a vertex v all that is needed is the shortest path with k edges to each of its in-neighbors and then to add in the weight of the one additional edge. This gives us

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

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

Algorithm 12.23 defines the Bellman Ford algorithm based on this idea. 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 12.2.

Theorem 12.24. *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.*

Algorithm 12.23 (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          val 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     val D = {v ↦ if v = s then 0 else ∞ : v ∈ V}
13 in BF(D, 0) end
```

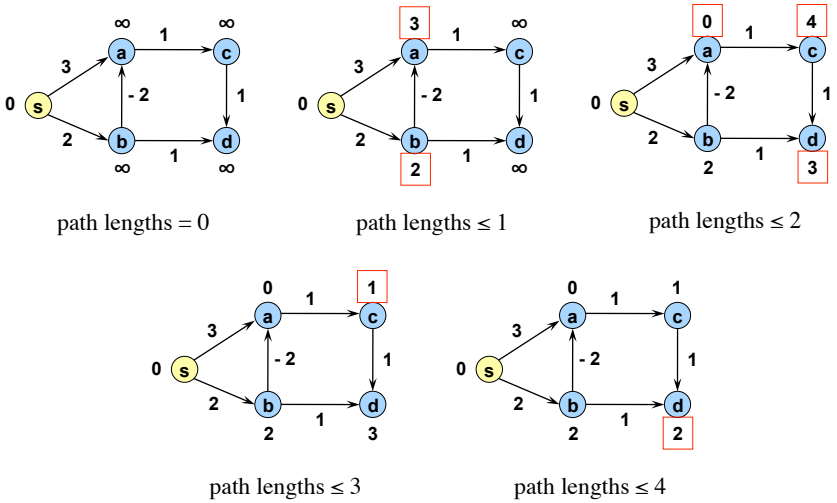


Figure 12.2: Steps of the Bellman Ford algorithm. The numbers with squares indicate what changed on each step.

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. We now analyze the cost of the algorithm. First we assume the graph is represented as a table of tables, as we suggested for the implementation of Dijkstra. We then consider representing it as a sequence of sequences.

For a table of tables we assume the graph G is represented as a $(\mathbb{R}$ `vtxTable`$)$ `vtxTable`, where `vtxTable` maps vertices to values. The $\mathbb{R}$ are the real valued weights on the edges. We assume the distances D are represented as a $\mathbb{R}$ `vtxTable`. 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 calculated 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 overall work and span are therefore

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

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.

Now 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:

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

Cost of Bellman Ford using Sequences If we assume the vertices are the integers $\{0, 1, \ldots, |V|-1\}$ then we can use array sequences to implement a `vtxTable`. Instead of using a `find`, which requires $O(\log n)$ work, we can use `nth` 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.