Lecture 12 and 13 — Shortest Weighted Paths

Parallel and Sequential Data Structures and Algorithms, 15-210 (Qatar-Spring 2014)

Lectured by Kemal Oflazer — 9 and 11 March 2014

What was covered in this lecture:

- Weighted Graphs
- Priority First Search
- Weighted Shortest Paths
- Dijkstra's Algorithm
- Bellman Ford Algorithm

1 Weighted Graph Representation

It is often necessary to associate weights or other values with the edges of a graph. Such a "weighted" or "edge-labeled" graph can be defined as a triple $G = (E, V, w)$ where $w : E \to \texttt{eVal}$ is a function mapping edges or directed edges to their values, and `eVal` is the set (type) of possible values. For a weighted graph `eVal` would typically be the real numbers, but for edge-labeled graphs they could be any type.

There are a few ways to represent a weighted or edge-labeled graph. The first representation translates directly from representing the function w. In particular we can use a table that maps each edge (a pair of vertex identifiers) to its value. This would have type

$$\texttt{eVal edgeTable}$$

That is, the keys of the table are edges and the values are of type `eVal`. For example, for a weighted graph we might have:

$$W = \{(0,1) \mapsto 0.7,\ (1,2) \mapsto -2.0,\ (0,2) \mapsto 1.5\}$$

This representation would allow us to look up the weight of an edge e using: $w(e) = \texttt{find}\ W\ e$.

Another way to associate values with edges is to use a structure similar to the adjacency set representation for unweighted graphs and associate a value with each out edge of a vertex. In particular, instead of associating with each vertex a set of out-neighbors, we can associate each vertex with a table that maps each out-neighbor to its value. It would have type:

$$(\texttt{eVal vertexTable})\ \texttt{vertexTable}\ .$$

The graph above would then be represented as:

$$G = \{0 \mapsto \{1 \mapsto 0.7,\ 2 \mapsto 1.5\},\ 1 \mapsto \{2 \mapsto -2.0\},\ 2 \mapsto \{\}\}\ .$$

We will mostly be using this second representation.

†Lecture notes by Umut Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, with additional edits by Kemal Oflazer

2 Priority First Search

Generalizing BFS and DFS, *priority first search* (also called best-first search) visits the vertices in some priority order. This priority order can either be static and decided ahead of time, or can be generated on the fly by the algorithm. Recall that a generic graph search maintains two sets: the set of visited vertices X, and the set of frontier vertices $F = N(X) \setminus X$ (in DFS the frontier is maintained implicitly on the recursion stack). The frontier vertices are the unvisited vertices we know about by having visited their in-neighbors. On each step we visit one or more vertices on the frontier, move them from F to X and add their unvisited out-neighbors to F. A priority first search can thus be seen as follows:

```
1  fun pfs(X, F) =
2  if (F = ∅) then X
3  else let
4     val M = highest priority vertices in F
5     val X' = X ∪ M
6     val F' = (F ∪ N(M)) \ X'
7  in pfs(X', F') end
```

This would typically start with $X = \emptyset$ and the frontier F containing a single source.

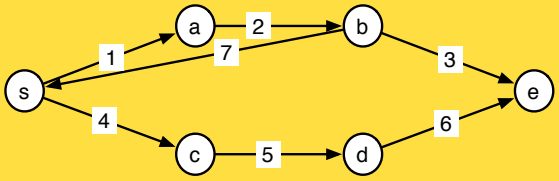
One can imagine using such a scheme, for example, to explore the web in a way that prioritizes the more interesting or relevant pages. The idea is to keep a ranked list of unvisited outgoing links based on how interesting the tag on the link appears to be. Then, when choosing what page to visit next, follow the link with the highest rank. When visiting the page, scratch it off the list and add all its unvisited outgoing links to the list with ranks.

Several famous graph algorithms are instances of priority first search. For example, Dijkstra's algorithm for finding single-source shortest paths (SSSP), discussed next and Prim's algorithm for finding Minimum Spanning Trees (MST). Priority First Search is a greedy technique since it greedily selects among the choices in front of it (the frontier) based on some cost function (the priorities) and never backs up. Algorithms based on PFS are hence often referred to as greedy algorithms.

2.1 Shortest Weighted Paths

The single-source shortest path (SSSP) problem is to find the shortest (weighted) path from a source vertex s to every other vertex in the graph. Consider a weighted graph $G = (V, E, w)$, $w : E \to \mathbb{R}$. The graph can be either directed or undirected. For convenience we define $w(u, v) = \infty$ if $(u, v) \notin E$. The *weight of a path* is the sum of the weights of the edges along that path.

Example 2.1. *In this graph the weight of the path from* $\langle s, a, b, e\rangle$ *is* 6*. The length of the path* s, a, b *is* 3*.*



Question 2.2. *What is the shortest path from* s *to* e*?*

The path $\langle s, a, b, e\rangle$ with weight 6 is the shortest path between.

Question 2.3. *What happens if we change the weight* 7 *to* -7*, negative value.*

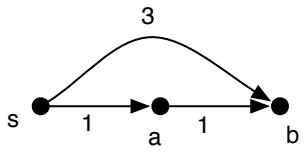
With this change the shortest path between s and e has lenght $-\infty$. For this reason, when computing shortest paths we will need to be careful about cycles of negative weight. As we will see, negative edge weight in general can be more difficult to deal with.

Shortest-path problems are important in many practical applications and come in many flavors. We often distinguish between different versions, such as "single source", "multi source", with or without negative edge weights.

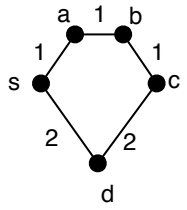
Definition 2.4 (The Single-Source Shortest Path (SSSP) **Problem**)**.** Given a weighted graph $G = (V, E, w)$ and a source vertex s, the *single-source shortest path (SSSP) problem* is to find the shortest weighted path from s to every other vertex in V.

For a weighted graph G we will use $\delta_G(u, v)$ to indicate the weight of the shortest path from u to v. Dijkstra's algorithm solves the SSSP problem when all the weights on the edges are non-negative (i.e. $w : E \to \mathbb{R}_+ \cup \{0\}$). It is a very important algorithm both because shortest paths have many applications, and also because it is a very elegant example of an efficient greedy algorithm that generates optimal solutions on a nontrivial task.

Before describing Dijkstra's algorithm, we would like to understand why we need a new algorithm for the weighted case. Why, in particular, doesn't BFS work on weighted graphs? Consider the following directed graphs on 3 nodes:



In this example, BFS would visit b then a. This means when we visit b, we assign it an incorrect weight of 3. Since BFS never visit it again, we'll never update it to the correct value. Note that we cannot afford to visit it again as it will be much more expensive. This problem still persists even if the graph is geometric (and satisfies the triangle inequality), as the following example shows:



But why does BFS work in unweighted case? The key idea is that it works outwards from the source. For each frontier F_i, it has the correct unweighted distance from 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).

Let's consider using a similar approach when the graphs has non-negative edge weights. Starting from the source vertex s, for which vertex can we safely say we know its shortest path from s? The vertex v that is the closest neighbor of s. There could not be a shorter path to v, since such a path would have to go through one of the neighbors that is further away from s and that path cannot get shorter because none of the edge weights are negative. More generally, if we know the shortest path distances for a set of vertices, how can we determine the shortest path to another vertex? This is the question that Dijkstra's algorithm answers.

Remark 2.5. One way to think of BFS algorithm, especially when reasoning about shortest paths, is to think of a "gadget" where each vertex is a bucket and each edge is a pipe that connects the buckets. Imagine now pumping water into the "source" bucket and watch the gadget flooded with water. The way water floods the system corresponds to how the BFS algorithm discovers the paths and vertices. The flood waters reach the shortest path, because they discover all paths simultaneously arriving at each place at the earliest possible time.

3 Dijkstra's Algorithm for SSSP

Let's suppose that we are given a weighted graph and a source and we wish to find the shortest paths from the source to each vertex in the graph.

Question 3.1. *What would be a brute force algorithm to do this?*

A brute force algorithm would consider all paths between the source and a vertex and pick the shortest one. There are exponentially many such paths unfortunately.

Question 3.2. *Can you think of a way to improve the brute-force algorithm? Why is it so inefficient.*

The brute force algorithm does a lot of redundant work because it does not use the information it gains from finding the shortest paths of vertices. The key property of shortest paths is that their subpaths are shortest. For example, if the shortest path from Pittsburgh to San Francisco goes through

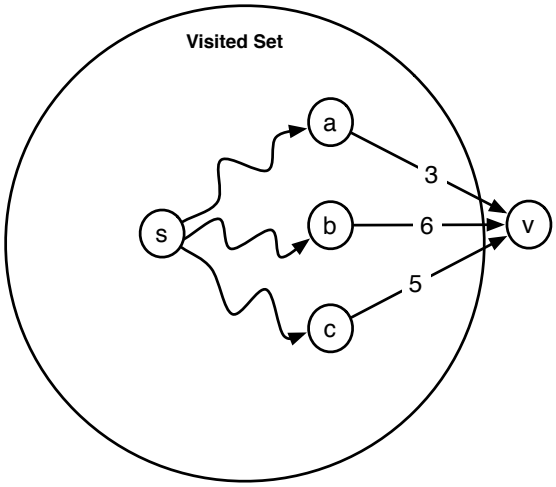
Chicago, the subpath from Pittsburgh to Chicago is the shortest path,between Pittsburgh and Chicago.

Question 3.3. *Bulding on this intuition, suppose that we have found the shortest paths from the source to all vertices except for one v. Can you find the shortest path to v?*

Yes, all we have to do is to consider all the incoming edges of v and pick the shortest path from each source to v. For example, in the following graph G, we would pick

$$\min(\delta_G(s,a)+3, \delta_G(s,b)+6, \delta_G(s,c)+5)$$

. Here $\delta_G(x,y)$ denotes the shortest path from x to y.



Question 3.4. *There is something wrong in fact with this argument, can you see what is missing?*

This reasoning breaks if the shortest path to v contains itself. In particular, suppose that v had a negative edge to a with weight -2, the shortest path to a now goes through a.

We were expecting to encounter this problem, so let's assume now that we are dealing with non-negative edge weights only.

Great, we have found a way to compute the shortest path to a vertex by using shortest paths to other vertices. A natural inclinations would now be to simple repeat this process by taking out vertices one by one. Or inversely, computing the shortest paths to some subset of the vertices and then by extending our set by adding another vertex, by then adding another and so on. In other words we are picking an ordering on the vertices and adding them to our visited set one by one.

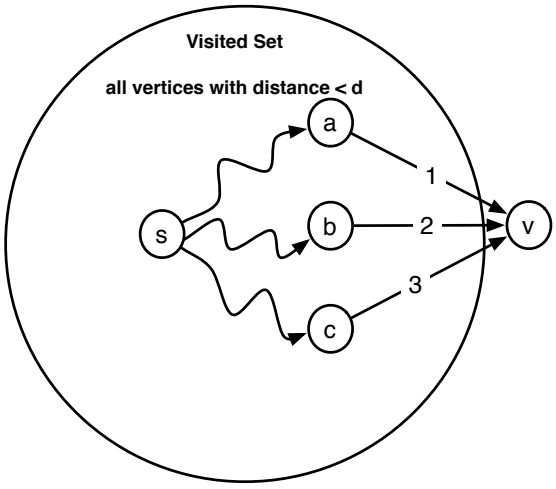
Question 3.5. *Would this algorithm work?*

It could but we would have to get really lucky. To see when it breaks, suppose that the shortest path to San Francisco goes through Chicago. If we consider San Francisco before Chicago, we will not be able to find the shortest path.

Question 3.6. *Can you pick an ordering on vertices so that all the shortest paths would be computed correctly by this algorithm?*

Suppose that we know the distance–that is, the length of the shortest path—of each vertex to our source. We can then visit the vertices in distance order.

We can build on this intuition to discover an algorithm. Let's convince ourselves that it works. Suppose that we have found the distance for all vertices up to some distance d and visited them. We are now considering the a vertex that has distance d as shown in the figure below.



Question 3.7. *How can we find the shortest path to v?*

Again, assuming non-negative edge weights all we have to do is to consider all incoming edges and compute the shortest path as

$$\min \delta_G(s,a)+1, \delta_G(s,b)+2, \delta_G(s,c)+3.$$

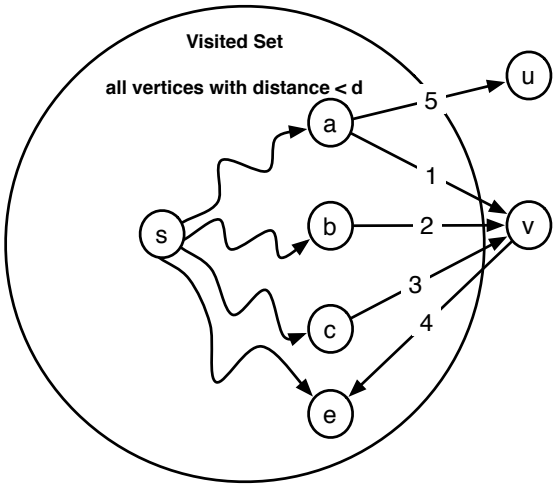
Since v cannot be on the shortest path to itself, we are done.

We are now really close to an algorithm, so all we have to do is consider the vertices in increasing order from the source.

Question 3.8. *But we don't know the order. Can you think of a way of figuring out the order?*

We can figure the order out as we go, because we only need to determine the next vertex to visit. Since that vertex is the one with the shortest distance and we know how to compute its distance by

considering its incoming edges, we can find it. The graph below shows an example where there are two vertices u and v. Suppose we compute their distances by considering their incoming edges.



Question 3.9. *Suppose that v is closer to source that u then would we calculate v's distance correctly?*

Since v is the next closest, by our reasoning above, we would compute v's distance correctly.

Question 3.10. *Can we compute u's distance to be smaller?*

Furthermore, u's distance cannot be smaller. Suppose that we did, then we would have path to u that is smaller than that of v, contradicting our assumption than v was closer. Note also that the shortest distance to e, can NOT be through v, since e has already been visited and has a shortest path from s.

We are now almost done.

Question 3.11. *Can you see now how we can compute the shortest distances from a source to all the other vertices?*

The idea is to maintain a visited set of vertices whose distances have already been computed correctly. We then consider the vertices in the frontier and calculate their distance by considering their incoming edges. We then extend the frontier by visiting the closest vertex. As usual we start our visited set with source at distance zero.

This is it. This is Dijkstra's algorithm.

Question 3.12. *Is this algorithm a graph search algorithm?*

Note that the algorithm walks paths be maintaining a visited set and a frontier, so it is a graph search algorithm.

Definition 3.13. Given a weighted graph $G = (V, E, w)$ and a source s, Dijkstra's algorithm is priority first search on G starting at s using priority $P(v) = \min_{v \in V}(d(v) + w(v,t))$ (to be minimized) and setting $d(v) = P(v)$ when v is visited and $d(s) = 0$.

Lemma 3.14. *When Dijkstra's algorithm returns, $d(v) = \delta_G(s,v)$ for all reachable v.*

Proof. We prove that $d(x) = \delta_G(s,x)$ for $x \in X$ by induction on the size of X. The base case is true since $d(s) = 0$. Assuming it is true for size i, then the algorithm adds the vertex v that minimizes $P(v) = \delta_{G,X}(s,v)$. By Lemma 4.1 $\delta_{G,X}(s, fv) = \delta_G(s,v)$ giving $d(v) = \delta_G(s,v)$ for $i+1$. □

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 algorithm is as follows:

```
1  fun dijkstra(G,s) =
2  let
3    fun dijkstra'(X, Q) =
4      case [PQ.deleteMin](Q) of
5          (NONE, _) ⇒ X
6        | (SOME(d,v), Q') ⇒
7          if ([(v, _) ∈ X]) then dijkstra'(X, Q')
8          else let
9              X' = [X ∪ {(v,d)}]
10           fun relax (Q,(u,w)) = [PQ.insert](d+w,u) Q
11             Q'' = [iter] relax Q' [N_G(v)]
12         in dijkstra'(X', Q'') end
13 in
14   dijkstra'({}, [PQ.insert] (0,s) {})
15 end
```

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 through vertices in X. On each round, the algorithm selects the vertex x with least distance d in the priority queue (Line 4 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 9), 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 10 and 11). 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 7 checks to see whether a vertex pulled from the priority queue

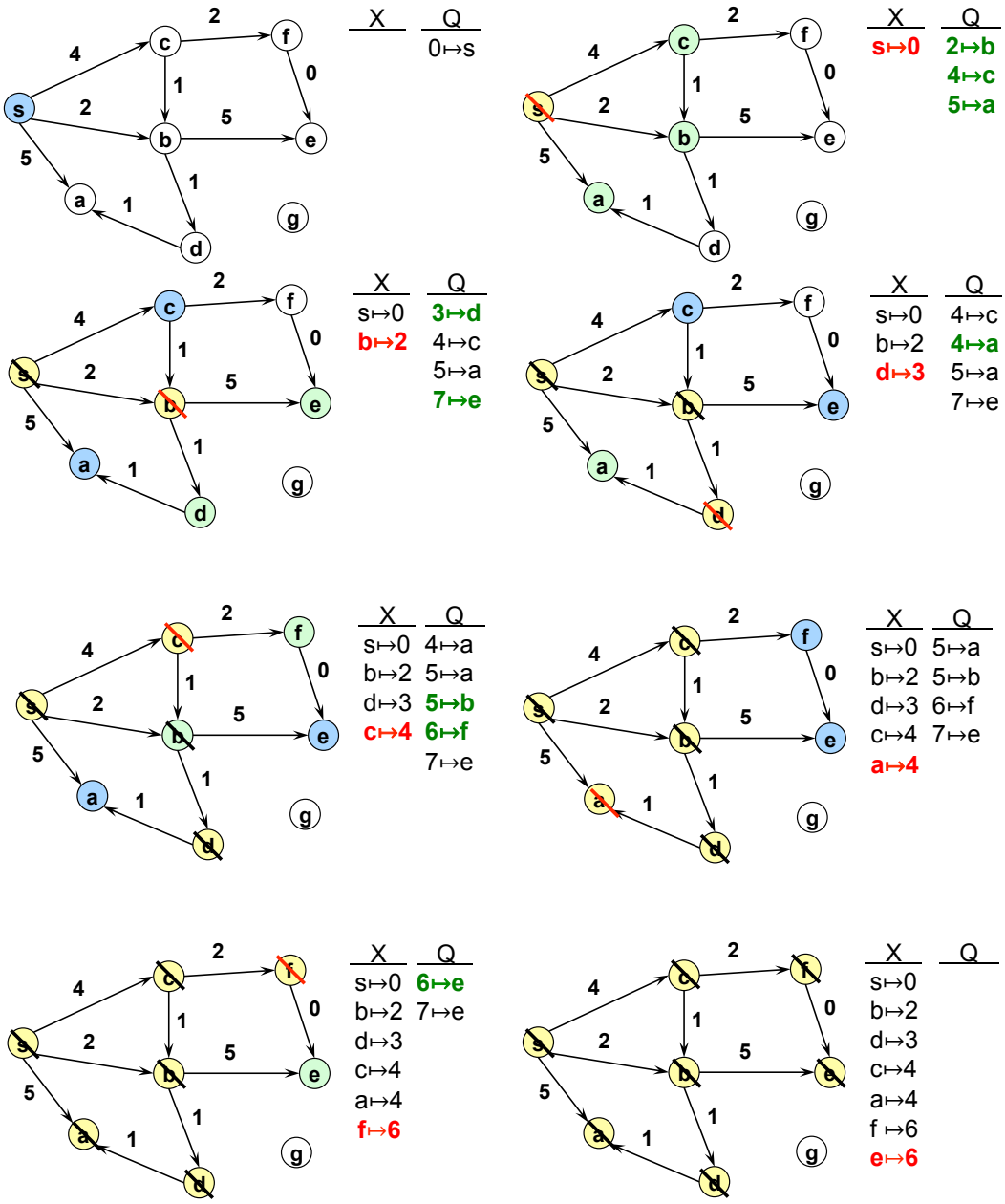


Figure 1: An example of Dijkstra's algorithm initialization and the end of each round. Notice that after visiting s, b, and d there are two distances in Q for a corresponding to the two paths from s to a; only the shortest distance will be added to X. In the next step, both f and b are added to Q, even though b is already visited. After visiting a, the redundant entries for a and b are discarded before visiting f. Finally, e is visited and the remaining element in Q is discarded. The vertex g is never visited as it is unreachable from s. If the priority queue supports a function that can return all minimum value keys, then the rounds for visiting c and a can be done in parallel, since they have the same shortest distance. Finally, notice that the distances in X never decrease.

has already been visited and discard it if it has. This algorithm is just a concrete implementation of the previously described Dijkstra's algorithm.

Remark 3.15. As with the BFS algorithm, we can think of Dijkstra's algorithm as flooding the graph from the source. The only difference is that we have to think of the weights as the time for the water to traverse a pipe (edge).

Dijkstra variants. A variant of Dijkstra's algorithm 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. Another variant is to instead of visiting just the closest vertex, to visit all the equally closest vertices on each round, as suggested by the more abstract algorithm. This variant has the advantage that the vertices can be visited in parallel. If all edges have unit weight, for example, this variant does the same as parallel BFS, visiting one level at a time. The amount of parallelism, however, depends on how many vertices have equal distance. This variant requires that the priority queue supports a function that returns all minimum valued keys.

Costs of Dijkstra's Algorithm. We now consider the cost of Dijkstra's algorithm by counting up the number of operations. In the code we have put a box around each operation. The `PQ.insert` in Line 14 is called only once, so we can ignore it. Of the remaining operations, The `iter` and $N_G(v)$ on line 11 are on the graph, Lines 7 and 9 are on the table of visited vertices, and Lines 4 and 10 are on the priority queue. For the priority queue operations, we have only discussed one cost model which, for a queue of size n, requires $O(\log n)$ work and span for each of `PQ.insert` and `PQ.deleteMin`. We have no need for a `meld` operation here. 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. The following table summarizes the costs of the operations, along with the number of calls made to each operation. There is no parallelism in the algorithm so we only need to consider the sequential execution of the calls.

Operation	Line	# of calls	PQ	Tree Table	Array	ST Array
`deleteMin`	4	$O(m)$	$O(\log m)$	-	-	-
`insert`	10	$O(m)$	$O(\log m)$	-	-	-
Priority Q total			$O(m \log m)$	-	-	-
`find`	7	$O(m)$	-	$O(\log n)$	$O(1)$	$O(1)$
`insert`	9	$O(n)$	-	$O(\log n)$	$O(n)$	$O(1)$
Distances total			-	$O(m \log n)$	$O(n^2)$	$O(m)$
$N_G(v)$	11	$O(n)$	-	$O(\log n)$	$O(1)$	-
`iter`	11	$O(m)$	-	$O(1)$	$O(1)$	-
Graph access total			-	$O(m + n \log n)$	$O(m)$	-

We can calculate the total number of calls to each operation by noting that the body of the let starting on Line 8 is only run once for each vertex. This means that Line 9 and $N_G(v)$ on Line 11 are only called $O(n)$ times. Everything else is done once for every edge.

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

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 that is not insignificant. 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 $m \leq n^2$.

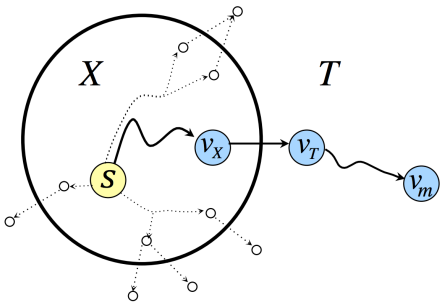
4 A more rigorous presentation of Dijkstra's algorithm

In order to re-invent Dijkstra's algorithm, we followed a path that simplified some properties of Dijkstra's algorithm. In this section, we present a more precise analysis of the algorithm.

The key property used by Dijkstra's algorithm is that for a set of vertices $X \subset V$ that include s and the rest of the vertices ($T = V \setminus X$), the closest vertex in T from s based on paths that only go through X is also an overall closest vertex in T. This property will allow us to select the next closest vertex by only considering the vertices we have already visited. Defining $\delta_{G,X}(s,v)$ as the shortest path length from s to v in G that only goes through vertices in X (except for v), the property can be stated more formally and proved as follows:

Lemma 4.1 (Dijkstra's Property)**.** *Consider a (directed) weighted graph $G = (V,E)$, $w : E \to \mathbb{R}_+ \cup \{0\}$, and a source vertex $s \in V$. For any partitioning of the vertices V into X and $T = V \setminus X$ with $s \in X$,*

$$\min_{t \in T} \delta_{G,X}(s,t) = \min_{t \in T} \delta_G(s,t)\,.$$



Proof. Consider a vertex $v_m \in T$ such that $\delta_G(s,v_m) = \min_{t \in T} \delta_G(s,t)$, and a shortest path from s to v_m in G. The path must cross from X to T at some point using some edge (v_X, v_T). Since subpaths of shortest paths are shortest paths, the subpath from s to v_T is the shortest path to v_T and, since edges are non-negative, distances don't decrease along the path implying $\delta_G(s,v_T) \leq \delta_G(s,v_m)$ (it could be that $v_T = v_m$). Furthermore since the shortest path to v_T only goes through X, $\delta_{G,X}(s,v_T) = \delta_G(s,v_T)$. Together we have:

$$\min_{t \in T} \delta_{G,X}(s,t) \leq \delta_{G,X}(s,v_T) = \delta_G(s,v_T) \leq \delta_G(s,v_m) = \min_{t \in T} \delta_G(s,t)\,.$$

Also $\min_{t \in T} \delta_{G,X}(s,t) \geq \min_{t \in T} \delta_G(s,t)$ since we are only restricting paths by requiring them to go through X. Together we have the equality. □

This lemma implies that if we have already visited a set of vertices X we can find a new shortest path to a vertex in $T = V \setminus X$ by just considering the path lengths through X to a neighbor of X. In particular we want to pick a vertex t that minimizes $\delta_{G,X}(s,t)$. This suggests an algorithm for shortest paths based on priority first search using the priority $P(v) = \delta_{G,X}(s,u)$. Also note that $\delta_{G,X}(u) = \min_{v \in V}(\delta_G(v) + w(v,t))$. Indeed this gives us Dijkstra's algorithm, at least in the abstract:

5 The Bellman Ford Algorithm

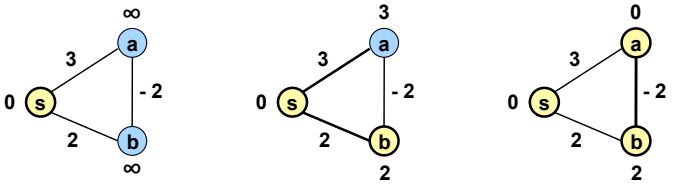
We now turn to solving the single source shortest path problem in the general case where we allow negative weights in the graph. One might ask how negative weights make sense. If talking about distances on a map, they probably do not, but various other problems reduce to shortest paths, and in these 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. This is because every time we go around the cycle we get a shorter path, so to find a shortest path we would just go around forever. 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 1. *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 2. *In your solution to the previous exercise can you get negative weight cycles? If so, what does this mean?*

Question 5.1. *Why can't we use Dijkstra's algorithm to compute shortest path when there are negative edges?*

Recall that in our re-construction of Dijkstra's algorithm, we made the assumption of positive edge weights in order to circumvent a problem that we encountered. But this does not constitute a proof. 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 when we consider the closest vertex v not in the visited set, its shortest path may is through only the visited set and then extended by one edge out of the visited set to v.

Question 5.2. *How can we find shortest paths on a graph with negative weights?*

Question 5.3. *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?*

The property 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 counting hops or the number of edges on the path.

Question 5.4. *Suppose that you have computed the shortest path with ℓ or less hops from s to all vertices. Can you come up with an algorithm to update the shortest paths for $\ell + 1$ hops?*

We can simply use our usual trick of considering the incoming edges and updating the shortest path. This is the idea behind the Bellman-Ford algorithm.

We define the following

$$\delta_G^l(s,t) = \text{the shortest weighted path from } s \text{ to } t \text{ using at most } l \text{ edges.}$$

We can start by determining $\delta_G^0(s,v)$ for all $v \in V$, which is infinite for all vertices except s itself, for which it is 0. Then perhaps we can use this to determine $\delta_G^1(s,v)$ for all $v \in V$. In general we want to determine $\delta_G^{k+1}(s,v)$ based on all $\delta_G^k(s,v)$. 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.

Here is the Bellman Ford algorithm based on this idea. It assumes we have added a zero weight self loop on the source vertex.

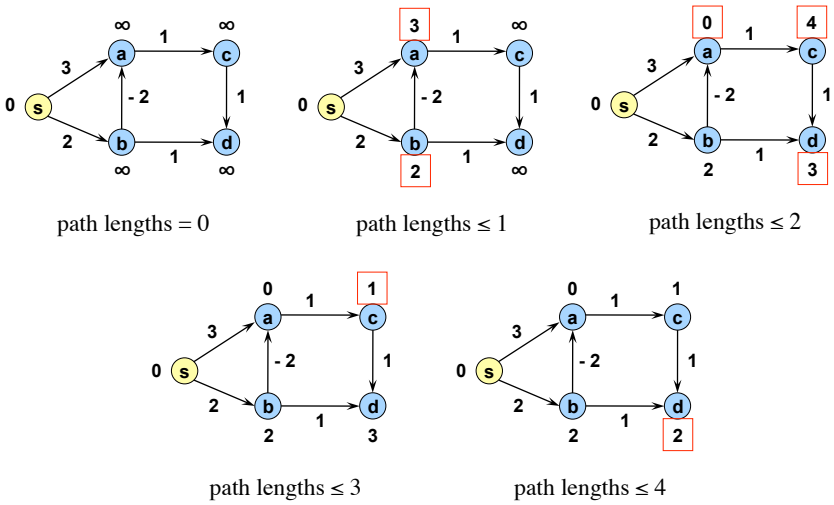


Figure 2: Steps of the Bellman Ford algorithm. The numbers with red squares indicate what changed on each step.

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

In Line 9 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 2.

Theorem 5.5. *Given a directed weighted graph $G = (V, E, w)$, $w : E \to R$, and a source s, the* `BellmanFord` *algorithm returns the shortest path length from s to every vertex or indicates that there is a negative weight cycle in G reachable from s.*

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 7 and 10. Line 7 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 10 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.

6 SML code

Here we present the SML code for Dijkstra.

```
functor TableDijkstra(Table : TABLE) =
struct
  structure PQ = Default.RealPQ
  type vertex = Table.key
  type 'a table = 'a Table.table
  type weight = real
  type 'a pq = 'a PQ.pq
  type graph = (weight table) table

  (* Out neighbors of vertex v in graph G *)
  fun N(G : graph, v : vertex) =
    case (Table.find G v) of
      NONE => Table.empty()
    | SOME(ngh) => ngh

  fun Dijkstra(u : vertex, G : graph) =
    let
      val insert = Table.insert (fn (x,_) => x)

      fun Dijkstra'(Distances : weight table,
                    Q         : vertex pq) =
        case (PQ.deleteMin(Q)) of
            (NONE, _)       => Distances
          | (SOME(d, v), Q) =>
            case (Table.find Distances v) of
```

```
              (* if distance already set, then skip vertex *)
              SOME(_) => Dijkstra'(Distances, Q)

            | NONE =>
              let
                val Distances' = insert (v, d) Distances
                fun relax (Q,(u,l)) = PQ.insert (d+l, u) Q

                (* relax all the out edges *)
                val Q' = Table.iter relax Q (N(G,v))
              in
                Dijkstra'(Distances', Q')
              end
    in
      Dijkstra'(Table.empty(), PQ.singleton (0.0, u))
    end
end
```