Lecture 11 — Shortest Weighted Paths II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Margaret Reid-Miller — 21 February 2012

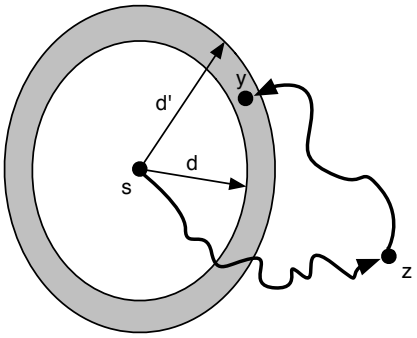
- Continuation of Dijkstra's algorithm
- Bellman-Ford's Algorithm, which allows negative weights

1 Dijkstra's Algorithm for SSSP

The crux of Dijkstra's algorithm is the following lemma, which suggests prioritiy values to use and guarantees that such a priority setting will lead to the weighted shortest paths.

Lemma 1.1. *Consider a (directed) weighted graph $G = (V, E)$, $w : E \to \mathbb{R}_+ \cup \{0\}$ with no negative edge weights, a source vertex s and an arbitrary distance value $d \in \mathbb{R}^+ \cup \{0\}$. Let $X = X' \cup X''$, where $X' = \{v \in V : \delta(s, v) < d\}$ be the set of vertices that are less than d from s, $X'' \subseteq \{v \in V : \delta(s, v) = d\}$ be the set of vertices that are exactly d from s. Also, let $d' = \min\{\delta(s, u) : u \in V \setminus X\}$ be the nearest distance greater than or equal to d. Then, if $V \setminus X \neq \emptyset$, there must exist a vertex u such that $\delta(s, u) = d'$ and a shortest path to u that only goes through vertices in X.*

Proof. Let $Y = \{v \in V : \delta(s, v) = d'\}$ be all vertices at distance exactly d'. Note that the set Y is nonempty by definition of d' and since $V \setminus X \neq \emptyset$.



Pick any $y \in Y$. We'll assume for a contradiction that that all shortest paths to y go through some vertex in $Z = V \setminus (X \cup Y)$ (i.e., outside of both X and Y). But for all $z \in Z$, $d(s, z) > d'$. Thus, it must be the case that $d(s, y) \geq d(s, z) > d'$ because all edge weights are non-negative. This is a contradiction. Therefore, there exists a shortest path from s to y that uses only the vertices in $X \cup Y$. Since $s \in X$ and the path ends at $y \in Y$, it must contain an edge $v \in X$ and $u \in Y$. The first such edge has the property that a shortest path to u only uses X's vertices, which proves the lemma. □

This suggests an algorithm that by knowing X, derives d' and one such vertex u. Indeed, X is the set of explored vertices, and we can derive d' and a vertex u attaining it by computing

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

$\min\{d(s,x) + w(xu) : x \in X, u \in N_G(x)\}$. Notice that the vertices we're taking the minimum over is simply $N_G(X)$. In the pseudocode shown below Dijkstra's algorithm basically operates on three data structures: (1) a structure for the graph itself, (2) a dictionary to maintain the shortest path distance to each vertex that has already been visited, and (3) a priority queue to hold the upper bound distances of vertices that are neighbors of the visited vertices. The priority queue makes finding the minimum distance in $N_G(X)$ fast.

This version of Dijkstra's algorithm differes somewhat from another version that is sometimes used. First, the `relax` function is often implemented as a `decreaseKey` operation. In our algorithm, we simply add in a new value in the priority queue. Although this causes the priority queue to contain more entries, it doesn't affect the asympotic complexity and obviates the need to have the `decreaseKey` operation, which can be tricky to support in many priority queue implementations.

Second, since we keep multiple distances for a vertex, we have to make sure that only the shortest-path distance is registered in our answer. We can show inductively through the lemma we proved already that the first time we see a vertex v (i.e., when `deleteMin` returns that vertex) gives the shortest path to v. Therefore, all subsequence occurences of this particular vertex can be ignored. This is easy to support because we keep the shortest-path distances in a dictionary which has fast lookup.

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

The `PQ.insert` in Line 14 is called only once, so we can ignore it. Of the remaining operations, Lines 10 and 11 are on the graph, Lines 7 and 12 are on the table of visited vertices, and Lines 4 and 9 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 $O(\log n)$ 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

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

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`	9	$O(m)$	$O(\log m)$	-	-	-
Priority Q total			$O(m \log m)$	-	-	-
`find`	7	$O(m)$	-	$O(\log n)$	$O(1)$	$O(1)$
`insert`	12	$O(n)$	-	$O(\log n)$	$O(n)$	$O(1)$
Distances total			-	$O(m \log n)$	$O(n^2)$	$O(m)$
$N_G(v)$	10	$O(n)$	-	$O(\log n)$	$O(1)$	-
`iterate`	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 Lines 10 and 12 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 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 $m \leq n^2$.

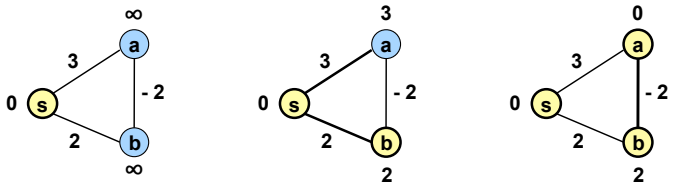
2 The Bellman Ford Algorithm

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), 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?*

So why is it that Dijkstra's algorithm does not work with negative edges? What is it in the proof of correctness that fails? Consider the following very simple example:

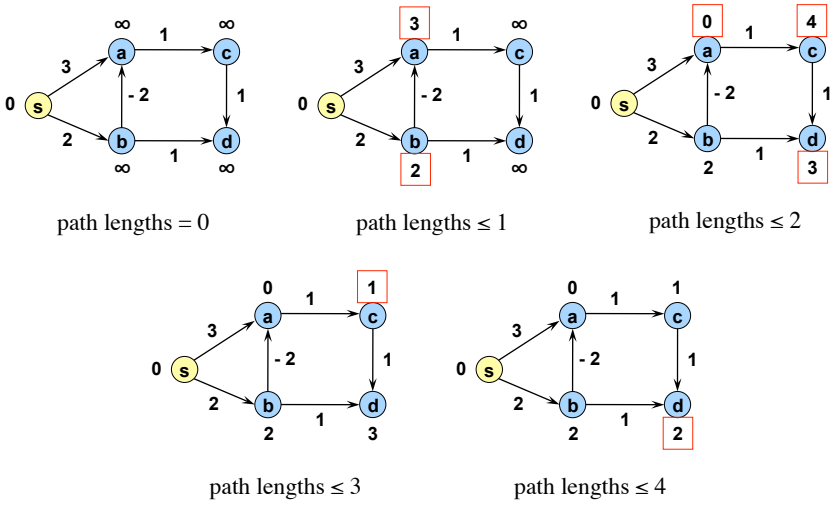


Dijkstra's algorithm would visit b then a and leave b with a distance of 2 instead of the correct -1. The problem is that it is no longer the case that if we consider the closest vertex v not in the visited set, then its shortest path is through only the visited set and then extended by one edge out of the visited set to v.

So how can we find shortest paths on a graph with negative weights? As with most algorithms, we should think of some inductive hypothesis. In Dijkstra, the hypothesis was that if we have found the i nearest neighbors, then we can add one more to find the $i+1$ nearest neighbors. Unfortunately, as discussed, this does not work with negative weights, at least not in a simple way.

What other things can we try inductively? There are not too many choices. We could think about adding the vertices one by one in an arbitrary order. Perhaps we could show that if we have solved the problem for i vertices then we can add one more along with its edges and fix up the graph cheaply to get a solution for $i+1$ vertices. Unfortunately, this does not seem to work. Similarly, doing induction on the number of edges does not seem to work. You should think through these ideas and figure out why they don't work.

How about induction on the unweighted path length (from now on we will refer to path length as the number of edges in the path, and path weight as the sum of the weights on the edges in the path)? In particular the idea based on induction is that, given the shortest weighted path of length at most i (i.e. involving at most i edges) from s to all vertices, then we can figure out the shortest weighted path of length at most $i+1$ from s to all vertices. It turns out that this idea does pan out, unlike the others. Here is an example:



Here is an outline of a proof that this idea works by induction. This proof also leads to an algorithm. We use the convention that a vertex that is not reachable with a path length i has distance infinity (∞) and set the initial distance to all vertices to ∞ except the source s which has distance 0. For the base case, on step zero no vertices except for the source are reachable with path length 0, and the distance to all such vertices is ∞. The distance to the source is zero. For the inductive case we note that any path of length $i+1$ has to go through a path of length i plus one additional edge. Therefore, we can figure out the shortest length $i+1$ path to v by considering all the in-neighbors $u \in N_G^-(v)$ and taking the minimum of $d(u) + w(u,v)$.

Here is the Bellman Ford algorithm based on this idea. The notation $\delta_G^i(s,v)$ indicates the shortest path from s to v in G that uses at most i edges.

```
1  % implements: the SSSP problem
2  fun BellmanFord(G = (V, E), s) =
3  let
4    % requires: all{D_v = δ^i_G(s, v) : v ∈ V}
5    fun BF(D, i) =
6      let
7        val D' = {v ↦ min_{u∈N_G^-(v)}(D_u + w(u, v)) : v ∈ V}
8      in
9        if (i = |V|) then ⊥
10       else if (all{D_v = D'_v : v ∈ V}) then D
11       else BF(D', i + 1)
12     end
13   val 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. 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.

We can 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.

Cost of Bellman Ford using a Tables Here we assume the graph G is represented as a ($\mathbb{R}$ `vTable`) `vTable`, where `vTable` maps vertices to values. The $\mathbb{R}$ are the real valued weights on the edges. We assume the distances D are represented as a $\mathbb{R}$ `vTable`. Lets consider the cost of one call to BF, not including the recursive calls. The only non trivial computations are on lines 7 and 10. Line 7 consists of a tabulate over the vertices. As the table indicates, 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 take 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 `vTable`. 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 Bellman Ford 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.

3 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
```