Lecture 13 — Shortest Weighted Paths II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Umut Acar — February 26

What was covered in this lecture:

- Continue Dijkstra's algorithm from last class
- Bellman Ford algorithm for graphs with negative weight edges – analysis of costs was covered in the next class.

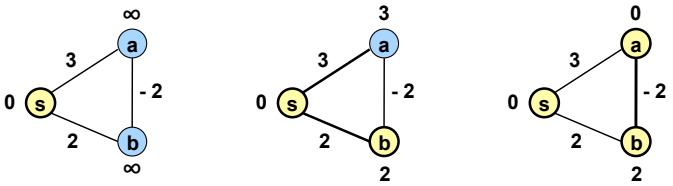
1 The Bellman Ford Algorithm

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?*

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

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

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 the number of edges in a path (i.e. the unweighted path length)? For this purpose we define the following

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

We can start by determining $\delta^0_G(s,v)$ 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 $\delta^1_G(s,v)$ 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 length of the one additional edge. This gives us

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

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.

```
1  % implements: the SSSP problem
2  function BellmanFord(G = (V,E), s) =
3  let
4    % requires: all{D_v = δ^k_G(s,v) : v ∈ V}
5    function BF(D,k) =
6      let
7          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
```

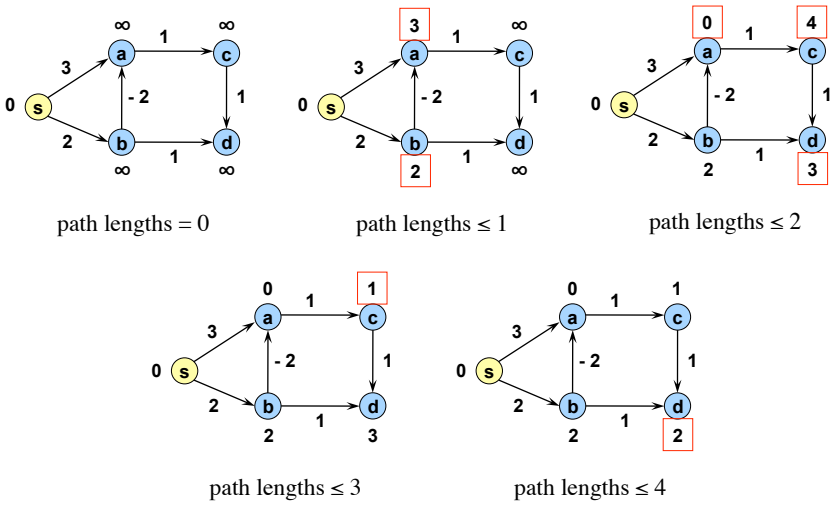


Figure 1: Steps of the Bellman Ford algorithm. The numbers with red squares indicate what changed on each step.

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 1.

Theorem 1.1. *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.