CS31 (Algorithms), Spring 2020 : Lecture 13

Date:

Topic: Graph Algorithms 4: Detecting Negative Cost Cycles, and Shortest Paths when there aren't any

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors.
Please discuss in Piazza/email errors to deeparnab@dartmouth.edu

---

1 Discussion

In the last lecture we saw that BFS finds shortest length paths from a given vertex s to every other vertex v in the graph. This takes $O(m+n)$ time. We also saw another algorithm, DIJKSTRA's algorithm, which finds the shortest cost/weight path from a given vertex s to every other vertex v, even when every edge has a cost $c(u,v)$ *as long as* the costs are ***non-negative***.

But in some applications, some edges can indeed have negative costs. Or to put it differently, sometimes certain edges can have "profits". Can you think of a situation? (Hint: the "Ferryman" problem in PSet 0?) And then DIJKSTRA's algorithm doesn't work. Can we think of an algorithm in this case?

Before we do so, let us recall what both the algorithms BFS and DIJKSTRA (and even the generalization of BFS, the weighted WBFS) did. These algorithms kept updating the distance labels of every vertex, where these labels were an *upper bound* on the shortest length/cost paths from s to that vertex. The reason for this was that if we took the reverse of $(v, \mathsf{parent}[v], \mathsf{parent}[\mathsf{parent}[v]], \ldots, s)$, this was a *candidate* path from s to v of length/cost $\mathsf{dist}[v]$. But, wait, why is that a *path*? Why is there nothing repeated? Why does the sequence of "parents" end at s?

The reasons are worth revisiting. In BFS, the reason was that every edge $(\mathsf{parent}[u], u)$ was added when $\mathsf{parent}[u]$ left the Q and u entered the Q. And in BFS, these vertices never returned in the Q, and thus no "cycles" are created. In DIJKSTRA, the reason is similar: every edge $(\mathsf{parent}[u], u)$ is from a vertex in R (the visited set) to outside R, and thus no "cycles" are created. But there is another reason. We know that BFS and DIJKSTRA returns shortest length/cost paths: can a *walk* have a shorter length/cost than a path? After all, doesn't shortcutting give a shorter path? Does it? Even with negative cost edges? And this is the big insight. *If there is a cycle in a graph so that the sum of costs of edges on that cycle is* ***negative****, then the shortest cost walk is $-\infty$: we will keep going round and round this cycle forever.* And indeed, these are the instances when the algorithm WBFS never terminates. And if there is *no negative cost cycle*, then the shortcutting always gives a shorter path, and thus the shortest walk will indeed be a path.

After that long prelude, we can finally say what we are going to see today. We are going to see an algorithm which finds shortest path/walk when there are *no negative cost cycles*, and indeed will *detect* when the graph has a negative cost cycle.

2 Directed Graphs without Negative Cost Cycles

We are going to take a circuitous route to the solution. (After all, we are talking about cycles. Ha!). Instead of solving the shortest cost *path* problem in G, we solve a slightly different problem: that of *length bounded shortest cost walks*.

LENGTH BOUNDED SHORTEST WALK
Input: Directed graph $G = (V, E)$ with costs $c(e)$ on edges which could be negative; a source vertex s; a parameter k.
Output: Minimum Cost *walk* w from s to v of length $\leq k$, for all $v \in V$.

The reason of doing so stems from the discussion above: if G has no negative cost cycles, these shortest cost walks are in fact shortest cost paths. The following lemma formalizes this.

Lemma 1. If G has no negative cost cycles, then the minimum cost walk from s to v of length $\leq n-1$ is the shortest path from s to v.

Proof. Clearly a path from s to v is a walk of length $\leq n-1$, so what we really need to prove is the reverse direction. That is, given a walk of length w from s to v of length $\leq n-1$, there is a path of the same or less cost. But note that we proved in graph-basics.pdf that any walk contains a subset of edges which is a path. If we recall that proof, we see that the edges left out constitutes a bunch of circuits. Since no circuit is of negative cost, we get that the cost of the path is at most the cost of the walk. □

You may wonder "Why are walks easier to handle than paths?". The answer will be clear when we use a favorite technique to solve this problem : dynamic programming. Now that I whispered dynamic programming in your ears, let us follow the principle of looking at an optimum solution, and then use it to look for solutionettes, and then figuring out the smaller subinstances.

Consider the minimum cost walk $w = (s, e_0, v_1, e_1, \ldots, v_{\ell-1}, e_{\ell-1}, v)$ from s to v of length $\ell \leq k$. Can we break this solution into pieces? Well, what can we say about the walk $w' = (s, e_0, \ldots, v_{\ell-1})$? We claim that this is the minimum cost walk from s to $v_{\ell-1}$ of length at most $k-1$. Why? Suppose not, and there was another walk w'' of strictly smaller cost. Then adding the edge $e_{\ell-1}$ to w'' would give a length $\leq k$ walk from s to v of strictly smaller cost than w. Furthermore, if we had the smallest cost length $\leq k-1$ length walks from s to x for all vertices x which have an edge to v, we should be able to find the smallest cost length $\leq k$ walk from s to v. Time to write the DP methodically now.

- ***Definition.*** We define $\mathsf{dist}[v, i]$ for all $v \in V$ and $0 \leq i \leq k$ as the cost of the minimum cost walk from s to v of length $\leq i$. We are interested in $\mathsf{dist}[v, k]$ for all $v \in V$.
- ***Base Case.*** $\mathsf{dist}[s, 0] = 0$; $\mathsf{dist}[v, 0] = \infty$ for all $v \neq s$.
- ***Recurrence.*** The recurrence is this

$$\forall v, i > 0, \quad \mathsf{dist}[v, i] = \min\left(\mathsf{dist}[v, i-1], \min_{u:(u,v)\in E} (c(u, v) + \mathsf{dist}[u, i-1])\right) \tag{1}$$

- ***Proof.*** Let's elaborate on the English explanation above. Let $\mathsf{Cand}(v, i)$ denote the set of walks from s to v of length at most i. Thus,

$$\mathsf{dist}[v, i] = \min_{w\in\mathsf{Cand}(v,i)} c(w)$$

  where $c(w) = \sum_{e\in w} c(e)$.

  ($\leq$) Clearly, $\mathsf{Cand}(v, i-1) \subseteq \mathsf{Cand}(v, i)$ and so $d[v, i] \leq d[v, i-1]$. Now fix any u with $(u, v) \in E$, and let w' be the walk in $\mathsf{Cand}(u, i-1)$ of cost $\mathsf{dist}[u, i-1]$. Then $w' \circ (u, (u, v), v)$ is a walk in $\mathsf{Cand}(v, i)$ of cost $\mathsf{dist}[u, i-1] + c(u, v)$. This takes care of this case.

Remark: *Do you see how the proof above **fails** if we said paths instead of walks? It is important that you do. w' could already have v...*

($\geq$) On the other hand, fix $w \in \mathsf{Cand}(v,i)$ of length $\leq i$ and cost $\mathsf{dist}[v,i]$. If the walk is not of length i, then $w \in \mathsf{Cand}(v,i-1)$ and thus $\mathsf{dist}[v,i-1] \leq \mathsf{dist}[v,i]$. Otherwise, the walk is of length $i \geq 1$, and so let u be the second-to-last vertex in this walk. But then the subset w' of the walk w ending at u is a walk in $\mathsf{Cand}(u,i-1)$. Since $c(w') = c(w) - c(u,v)$, we get $\mathsf{dist}[u,i-1] \leq \mathsf{dist}[v,i] - c(u,v)$.

- ***Pseudocode.***

```
1:  procedure LWSB(G, s, k):
2:      ▷ Returns dist[v, i] for every v and 0 ≤ i ≤ k.
3:      ▷ Also returns parent[v, i] for every v and 0 ≤ i ≤ k.
4:      dist[s, 0] ← 0; dist[v, 0] ← ∞ for all v ≠ s.  ▷ Base Case
5:      parent[s, 0] ← ⊥; parent[v, 0] ← ⊥ for all v ≠ s.  ▷ Base Case
6:      for i = 1 to k do:
7:          for v ∈ V do:
8:              dist[v, i] ← min(dist[v, i − 1], min_{u:(u,v)∈E}(dist[u, i − 1] + c(u, v))).
9:              ▷ Takes deg⁻(v) time
10:             ▷ We abuse notation and say ∞ + c = ∞ for any finite c.
11:             If u is the vertex that minimizes, parent[v, i] ← u.
12:     ▷ At this point dist[v, k] has the minimum cost walk of length at most k
13:     ▷ As before, the dist[v, k]'s can be used to recover the paths as well, but the parents
        can also be used.
```

- ***Running Time and Space.*** The space required is $O(nk)$. The running time is $O(k(n+m))$.

Lemma 1 states that if we are promised that G has no negative cost cycle, then $\mathrm{LBSW}(G,s,n-1)$ will indeed return $\mathsf{dist}[v,n-1]$ for all vertices which are the shortest path lengths. However, this does raise the following question: how would we ***know*** whether or not G has a negative cost cycle? Is there an algorithm that can detect it?

It so happens that the previous algorithm is powerful enough to do this as well. Consider running LBSW on G,s,k with $k = n$. Then note that if G has no negative cost cycle C reachable from s, then $\mathsf{dist}[v,n-1] = \mathsf{dist}[v,n]$ for all v; the minimum cost walk of length at most $n-1$ is the shortest path which is also the minimum cost walk of length at most n. The following lemma shows that if G has a negative cost cycle C reachable from s, then the above ***cannot be true*** for all v. And therefore, just checking this for all v would help us detect a negative cost cycle.

Lemma 2. Suppose there is a negative cost cycle C in G and suppose C has a vertex reachable from s. Consider running LBSW on G,s,k with $k = n$. Then, there exists a vertex $v \in C$ such that $\mathsf{dist}[v,n] < \mathsf{dist}[v,n-1]$.

Proof. Let $C = (x_1,\ldots,x_k,x_1)$ be the negative cost cycle reachable for s. Since (x_i,x_{i+1}) is an edge for all $1 \leq i \leq k$ (with the abuse $k+1=1$), using (1) we get

$$\mathsf{dist}[x_{i+1},n] \leq \mathsf{dist}[x_i,n-1] + c(x_i,x_{i+1}), \quad \forall 1 \leq i \leq k$$

Adding these up, we get

$$\sum_{v \in C} \mathsf{dist}[v, n] \leq \sum_{v \in C} \mathsf{dist}[v, n-1] + c(C) < \sum_{v \in C} \mathsf{dist}[v, n-1]$$

since $c(C) < 0$. This means $\mathsf{dist}[v, n] < \mathsf{dist}[v, n-1]$ for some $v \in C$. □

Therefore, given any graph G and a source s, if we run LBSW on G, s with $k = n$, then either some $\mathsf{dist}[v, n] < \mathsf{dist}[v, n-1]$ for some v implying a negative cost cycle reachable from s, or $\mathsf{dist}[v, n-1]$ contains the cost of the shortest paths from s to v. This algorithm is also called the BELLMAN-FORD algorithm.

```
1:  procedure BELLMAN-FORD(G, s):
2:      Run LBSW(G, s, n) to obtain dist[v, i] and parent[v, i] for all v and 0 ≤ i ≤ n.
3:      if there exists v with dist[v, n] < dist[v, n − 1] then:
4:          ▷ There is a negative cost cycle containing v.
5:          ▷ The following code uses the parent pointers to recover it.
6:          x = v; j = n
7:          while x ≠ v or j = n do:
8:              Add (parent[x, j], x) to C.
9:              x = parent[x, j].
10:             j = j − 1.
11:         return C
12:     else:
13:         ▷ dist[v, n]'s are indeed shortest path lengths, and construct the shortest path tree
14:         Add all vertices with dist[v, n] < ∞ to T.
15:         for v ∈ T do:
16:             Add (parent[v, n], v) to T.
17:         return T.
```

Theorem 1. If there is a negative cost cycle reachable from s, BELLMAN-FORD returns one, or it returns a shortest path tree T to all vertices reachable from s.

2.1 A Space Saver and a Better Presentation

Our presentation of the BELLMAN-FORD algorithm above is arguably a bit circuitous than what you would see in the textbooks. But it is actually the *same* algorithm. Let's try to see now how to get to these "textbook" presentations.

First thing we notice is that Line 8 of LBSW can be done "edge-by-edge" instead of "vertex-by-vertex". That is, we could replace the for-loop (Line 7 in LBSW) as follows:

$$\text{For all edges}(u, v) \in E : \mathsf{dist}[v, i] = \min\left(\mathsf{dist}[v, i-1], \mathsf{dist}[u, i-1] + c(u, v)\right))$$

That is, instead of minimizing over all out-edges in Line 8 of LBSW, we can stagger the minimization.

The second observation is that since the BELLMAN-FORD algorithm is only interested in the values of $k = n - 1$ and $k = n$, we really don't need to take care of the parameter i in $\mathsf{dist}[v, i]$ for all the other i's. Of

course, the way the DP builds up, it needs the smaller i's to get to the final $n-1$ and n. But note that the DP *can also forget* (we sort of saw this in FIBONACCI: instead of the array, one could just keep the "last two" values). That is, we just store $\mathsf{dist}[v]$ for each vertex, where the ith for-loop is supposed to be thought of as $\mathsf{dist}[v,i]$. With these two observations, we get the following algorithm (which is almost identical to the ones given in the textbooks).

```
1:  procedure BELLMAN-FORD-TXTBK(G, s):
2:      dist[s] = 0 for all j; dist[v] = ∞ for all v ≠ s.  ▷ Base Case
3:      for i = 1 to n − 1 do:
4:          for (v, u) ∈ E do:
5:              dist[v] = min(dist[u] + c(u, v), dist[v]).
6:              ▷ We abuse notation and say ∞ + c = ∞ for any finite c.
7:              If dist[v] modified, set parent[v] ← u.
8:      ▷ At this point, dist[v] is really dist[v, n]. We do a final check for negative cost cycles.
9:      for (u, v) ∈ E do:
10:         if dist[v] > dist[u] + c(u, v) then:
11:             return NEGATIVE CYCLE
```

Remark: *We have been talking about directed graphs so far. What about the same problem in* ***undirected graphs****? More precisely, can we detect if an undirected graph has a negative cost cycle? Turns out this is tricky. Why? The reason is that a cycle in an undirected graph has to be of length* 3 *or more. That is,* $\{(u,v),(v,u)\}$ *is not a cycle in an undirected graph. However, in a directed graph a pair of antiparallel edges is indeed a cycle. So, how will you go about finding whether an undirected graph has a negative cost cycle?*

Remark: *Ok, what about shortest* ***paths*** *whether there are negative cost edges* and *negative cost cycles? Paths are not allowed to repeat vertices, so this problem is indeed well-defined. How will we solve this one? Short answer: no one knows, and many people* ***believe*** *that no good algorithm exists! To understand what this "belief" means, wait for the last lecture of the course.*