CS 4349 Lecture—October 25rd, 2017

Main topics for #lecture include #SSSP.

Prelude

- Homework 7 due today.
- Homework 8 due Wednesday, November 1st.

Shortest Paths

- Last time, we were discussing algorithms for the single source shortest path problem.
- Given a weighted *directed* graph G = (V, E, w) and vertex s, find paths to all other vertices that minimize the sum of weights along each path. You'll end up computing a shortest path tree.
- Remember, you may have negative weights! Algorithms for this problem generally fail if you have a negative weight cycle, because a shortest walk would just go around the cycle over and over again forever.
- There was just the one algorithm.
- dist(v) is the length of a tentative shortest s to v path, or infinity if we haven't found one yet.
- pred(v) is the predecessor of v in the tentative shortest s to v path, or Null if we haven't found one yet.
- Call an edge u ➡ v *tense* if dist(u) + w(u ➡ v) < dist(v).
- We want to *relax* tense edges to represent our newly found shorter path.

```
RELAX(u→v):
    dist(v) ← dist(u) + w(u→v)
    pred(v) ← u
```

- The only SSSP algorithm repeatedly finds a tense edge and relaxes it.
- We do this by putting possible predecessors of tense edges in a bag.

```
INITSSSP(s):
    dist(s) ← 0
    pred(s) ← NULL
    for all vertices v ≠ s
        dist(v) ← ∞
        pred(v) ← NULL
```

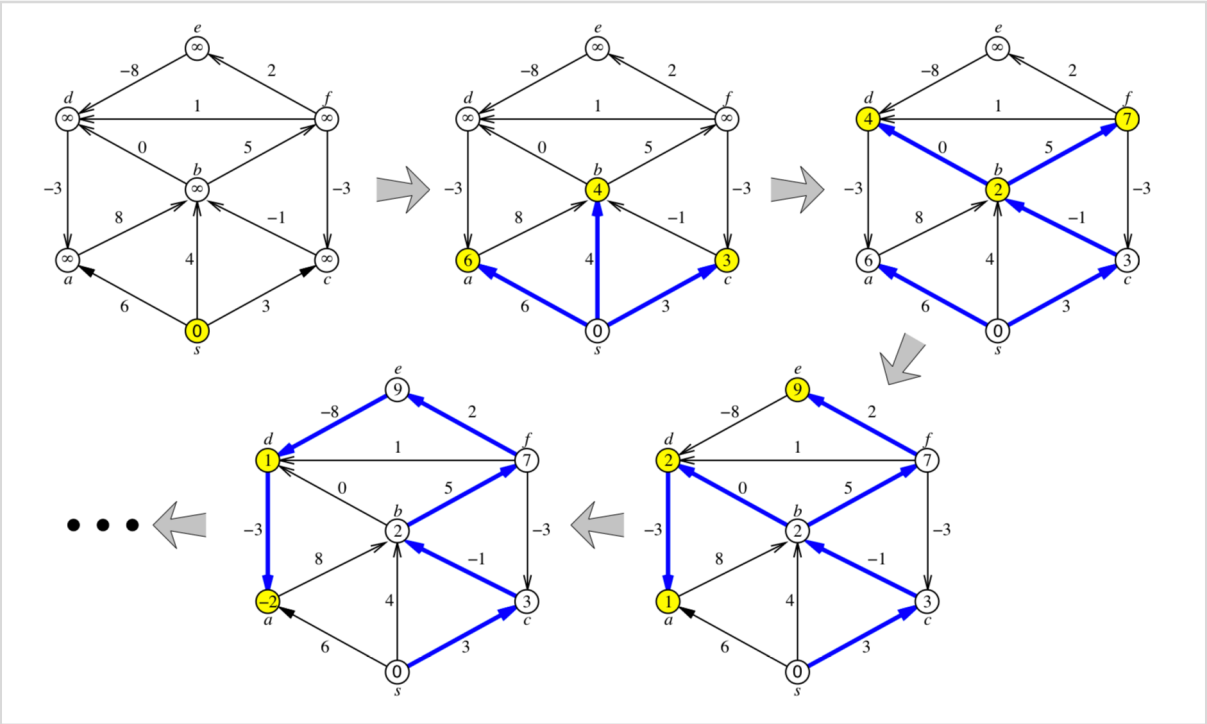
```
GENERICSSSP(s):
    INITSSSP(s)
    put s in the bag
    while the bag is not empty
        take u from the bag
        for all edges u→v
            if u→v is tense
                RELAX(u→v)
                put v in the bag
```

- The data structure we use for the bag determines the algorithm we're running.
- Stacks are just bad. The algorithm may take Theta(2^V) time.
- Priority queues keyed by distance are great if edges have non-negative weight. This was

Dijkstra's algorithm, and it ran in O(E + V log V) time if we used Fibonacci heaps. It even runs fine in practice if you have a few negative weight edges.

Shimbel's Algorithm

- Now let's try a normal first-in first-out queue. This algorithm was found by Shimbel in 1954, and then by some others including Bellman in 1958.
- The relaxation scheme we're using is by Ford, so it's often called Bellman-Ford.
- This algorithm is efficient even if there are negative weight edges, and it can be used to detect negative weight cycles.
- But how do we analyze it?
- Let's break the algorithm into phases by putting a *token* into the queue with the vertices.
- At the beginning of a phase, we'll add the token. The phase ends when we pull the token back out. The algorithm doesn't actually do anything with the token, it just puts it back in for the next phase.
- In the 0th phase, s is the only vertex removed from the queue. The algorithm ends when the token is the only member of the queue.
- Here's each phase of the algorithm:



- Here we removed from the queue s **token** a b c **token** d f b **token** a e d **token** d a **token** **token**
- Main Lemma: After i phases of the algorithm, dist(v) is *at most* the length of the shortest walk from s to v consisting of *at most* i edges.
  - Proof via induction on i.
- A simple path has at most V-1 edges, so the algorithm either halts after V phases or there is a negative cycle.

- Each vertex is removed at most once per phase, so each edge is scanned at most once per phase. Each phase takes O(E) time and the total time is O(V E).
- But now that we understand what's going on, why don't we simplify the algorithm? We just need to make sure that in the ith phase, we relax any incoming edges to a vertex whose shortest path uses i edges.

```
ShimbelSSSP(s)
  InitSSSP(s)
  repeat V times:
      for every edge u→v
          if u→v is tense
              Relax(u→v)
  for every edge u→v
      if u→v is tense
          return "Negative cycle!"
```

- We know that shortest paths are computed by the end of that for loop as long as there are no negative length cycles. But, there is always a tense edge if there is a negative length cycle, so the last pass through the edges will discover if there is a negative length cycle.
- Now the running time is obvious.
- That statement about paths of i edges still holds, so this new way of writing it is correct.

Shimbel via Dynamic Programming

- Let's talk about Shimbel's algorithms bit longer. We'll use it as an excuse to review a concept from earlier in the semester.
- We had that claim that Shimbel computes all shortest paths of at most i vertices by the end of the ith phase. What if we made wrote an algorithm with that goal in mind?
- Let dist_i(v) denote the length of the shortest path consisting of at most i edges.
- If there are no negative cycles, then dist_{V-1}(t) is the shortest path distance to t.
- It turns out dist_i(v) has a simple recursive definition!
- dist_i(v) =
  - 0 if i = 0 and v = s
  - infinity if i = 0 and v ≠ s
  - min{ dist_{i-1}(v), min_{u ➔ v \in E} (dist_{i-1}(v) + w(u ➔ v)} otherwise
- Basically, the shortest path with at most i edges is either the shortest path with at most i - 1 edges, or there's some ith edge which we append to the end of a shortest path with at most i -1 edges.
- We can use dynamic programming to turn this into an iterative algorithm.

```
SHIMBELDP(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i − 1,v]
          for every edge u→v
              if dist[i,v] > dist[i − 1,u] + w(u→v)
                  dist[i,v] ← dist[i − 1,u] + w(u→v)
```

- OK, but we can simplify this a bit.
- First, it doesn't matter what order we look at those edges since we'll get to every edge coming in v anyway, so let's push that last for loop to the left.
- Second, we're immediately setting dist[i,v] to be dist[i-1,v] before relaxing any edges. So we may as well use that (or something even smaller if we happen to find it first) while looping over the edges.

```
SHIMBELDP2(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i − 1,v]
      for every edge u→v
          if dist[i,v] > dist[i,u] + w(u→v)
              dist[i,v] ← dist[i,u] + w(u→v)
```

- OK, but now we don't really need dist[i-1] since that first inner for loop is just carrying the value forward to dist[i] anyway. Why don't we just use a single dist variable per vertex?

```
SHIMBELDP3(s)
  dist[s] ← 0
  for every vertex v ≠ s
      dist[v] ← ∞
  for i ← 1 to V − 1
      for every edge u→v
          if dist[v] > dist[u] + w(u→v)
              dist[v] ← dist[u] + w(u→v)
```

- Oh! It's just Shimbel's algorithm again. All we're missing is maintaining the predecessors, but that's easy to add back in as a last line in that inner for loop.

Shortest Paths in DAGs

- One last thing today, because I want to start fresh with a completely different problem on Monday.
- Suppose G is a directed acyclic graph. So there aren't any negative weight cycles, because there are no cycles at all!

- It turns out the queue is a great choice for DAGs, but you have to fill it with vertices before you do any relaxations.
- We should add vertices in topological order!

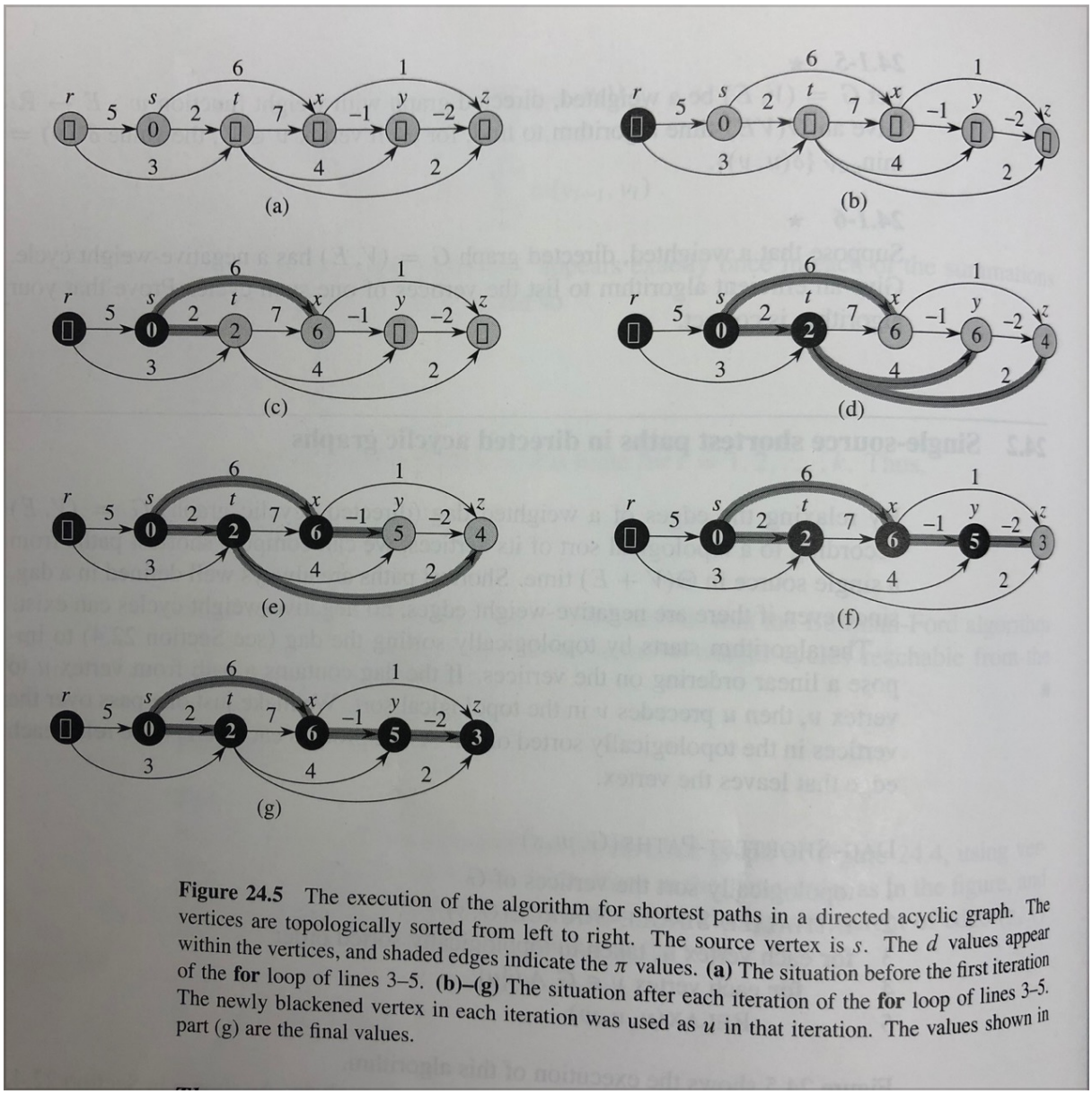


Figure 24.5 The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values appear within the vertices, and shaded edges indicate the π values. **(a)** The situation before the first iteration of the **for** loop of lines 3–5. **(b)–(g)** The situation after each iteration of the **for** loop of lines 3–5. The newly blackened vertex in each iteration was used as u in that iteration. The values shown in part (g) are the final values.

- Since there are no backward edges, we will never add a vertex back into the queue.
- So like before, we can simplify the algorithm somewhat.
- DAGSSSP(s):
  - InitSSSP(s)
  - Topologically sort vertices
  - for every vertex u in topological order
    - for every edge u ➡ v
      - if u ➡ v is tense
        - Relax(u ➡ v)
- It takes O(V + E) time to do a topological sort, and each edge is checked and possibly relaxed once, so the whole algorithm takes O(V + E) time.
- We can even use this as a starting point for a problem that's usually very difficult in graphs,

computing *longest* paths.

- Let's say we want to compute longest paths from s in DAG G. We can take two approaches.
  1. Multiply all edge weights by -1. Longest paths with the original weights are now shortest paths. G is a DAG so we certainly didn't create negative cycles.
  2. Change the algorithm itself. Initialize all dist values to -infinity instead of infinity and use a > in the definition of tense instead of a <.