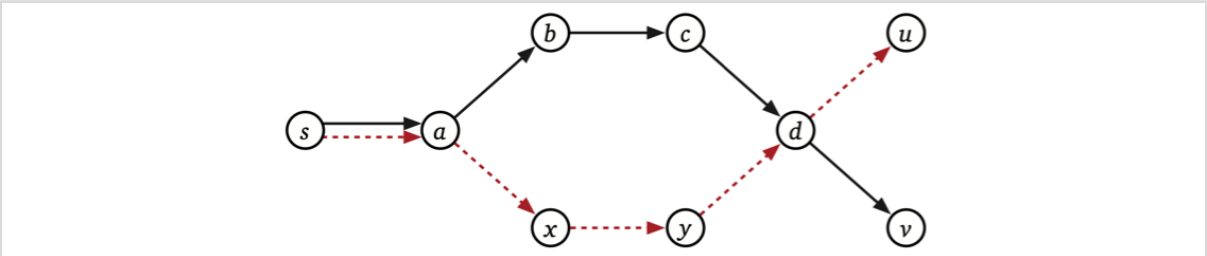
CS 6363.003.21S Lecture 15—March 30, 2021

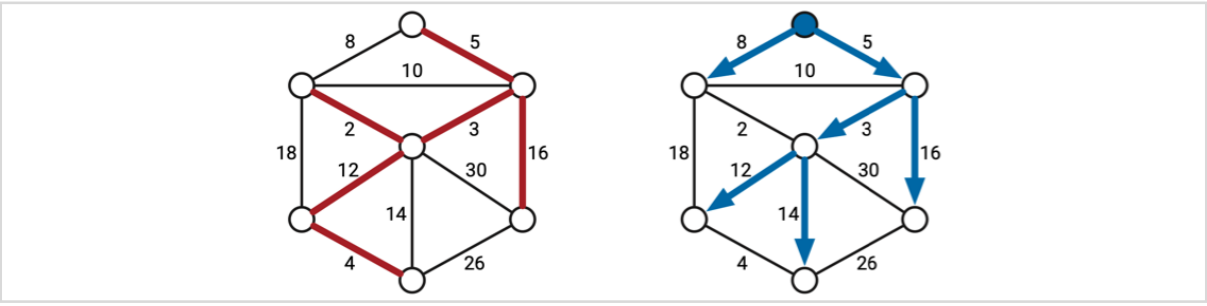
Main topics are #single_source_shortest_paths.

Single Source Shortest Paths

- For the next problem, let's say you're given a *directed* graph G = (V, E, w) where w : E ➔ R is another weight function. The shortest path between two vertices s and t is the s,t-path P minimizing w(P) = sum_{u ➔ v in P} w(u ➔ v). The minimum value is the *distance* from s to t.
- Most algorithms for shortest paths actually end up solving the more general *single source shortest paths* (SSSP) problem: find the shortest path from s to every vertex in G.
- A subpath of a shortest path is itself a shortest path, and we can always pick our shortest paths consistently so they form a spanning tree, rooted at s. Here, we can redirect the dotted path from a to d to go through the solid path from a to d instead so we indeed get that tree.



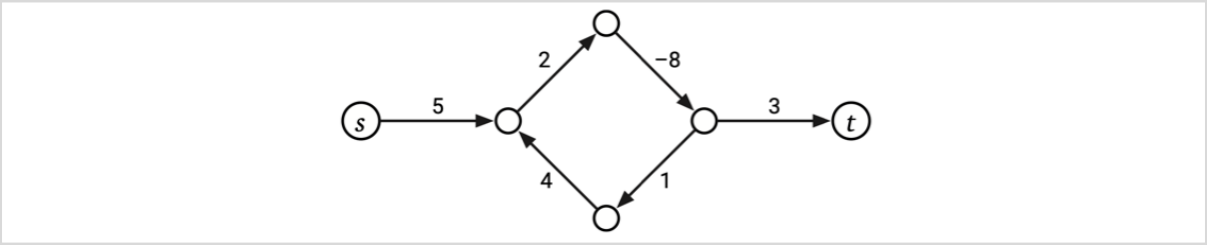
- We'll focus on computing a shortest path tree from s and the distances from s to every other vertex.
- Please please please don't confuse minimum spanning trees with shortest path trees. They're both optimal trees but minimum spanning trees are for undirected graphs while shortest path algorithms are best described for directed graphs and the trees themselves are directed away from a root. If edge weights are distinct, there is exactly one minimum spanning tree, but there is a different shortest path tree for every choice of source vertex s.
- If you want to do shortest paths in a non-negatively weighted undirected graph, replace every edge uv with a pair of edges u ➔ v and v ➔ u of the same weight. Even here, you may get different trees. In fact, every SSSP tree of an undirected graph may use a different set of edges from the minimum spanning tree. Below, we have a minimum spanning tree to the left and a SSSP tree to the right.



- Nothing about our problem definition forbids negative weights. If you think of positive

weights as a cost for following certain edges, negative weight would represent some benefit. Maybe you're trying to plan a trip and there's a few particularly pretty roads you'd like to drive down.

- However, we run into trouble if there's a directed cycle with negative total weight. The algorithms we're going to talk about today and Thursday are really working toward computing a shortest *walk* from s to each other vertex. If there are negative weight cycles, a "shortest" walk would go around and around an infinite number of times before reaching its destination. In other words, a shortest walk *does not exist*!



- If there are no negative weight cycles, though, we can compute shortest walks that just happen to be paths, because there is no reason to repeat a vertex if every cycle from that vertex to itself has non-negative weight.
- The trick of turning an undirected graph into a directed one only works if there are non-negative weights, because otherwise you'd create tiny negative weight cycles with two edges each. You *can* do shortest paths in undirected graphs with negative weights, but the algorithms for that rely on computing something called a *minimum cost matching*, something well beyond the scope for this course.

The Only SSSP Algorithm

- Like minimum spanning trees, there's really only one shortest path tree algorithm. It was independently discovered by Lester Ford, George Dantzig, and George Minty around the same time.
- The idea is that we'll keep an educated guess on the distance and shortest path to each vertex. The way I like to think about it is that we're pessimistically assuming the distance to each vertex is absolutely huge until prove otherwise. The proofs otherwise eventually become the incoming edges on the shortest path to each vertex.
- To that ends, we'll define two mutable variables for each vertex v:
  - dist(v) is our pessimistic guess on the distance to v. dist(v) will always be at least the actually distance to v. Initially, dist(s) ← 0 and dist(v) ← infinity for all v ≠ s.
  - pred(v) is the predecessor of v in a tentative shortest s to v walk, or the "proof" that our previous guess on the distance was too high. Initially, pred(v) ← Null for every vertex v.
- All variants of the shortest path algorithm begin by running the following initialization procedure. InitSSSP(s) takes the shortest path source s.

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

- Call an edge u ➔ v *tense* if dist(u) + w(u ➔ v) < dist(v).
- Edge u ➔ v being tense is our proof that dist(v) is too high.
- Therefore, we *relax* tense edge u ➔ v by lowering dist(v) just enough to remove the tension from u ➔ v.

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

- The only SSSP algorithm repeatedly finds some tense edge and relaxes it.

```
FORDSSSP(s):
  INITSSSP(s)
  while there is at least one tense edge
      RELAX any tense edge
```

- I'm going to claim that this algorithm does eventually terminate if there are no negative cycles. After it terminates, each value dist(v) is the shortest path distance from s to v, and following the sequence of pred pointers from any vertex v gives you the *reversal* of the shortest path to v. If dist(v) = infinity after the algorithm terminates, then v wasn't reachable from s to begin with.
- One surprising thing is that the algorithm never terminates if there is even one negative cycle reachable from s. Intuitively, no dist(v) is low enough for any v on that cycle, and we'll keep finding tense edges on the cycle and relaxing them forever. I'll give a more formal argument later.
- All of the shortest paths algorithms are just variations on how you pick which edge to relax in each iteration, and you pick your algorithm based on what kinds of edge weights or structures are present in your graph. The running time analyses for these algorithms are themselves variants on how to prove correctness of FordSSSP. I don't want to say the same things to you more often than I need to, so I'll skip the proof of correctness for generic FordSSSP.
- But I will prove one thing that's necessary to know for all of the variants: at all times, for any vertex v, then dist(v) is either infinity or the length of some walk from s to v.
  - We can use induction on the number of relaxations.
  - If the last change to to dist(v) was setting dist(v) ← dist(u) + w(u ➔ v) then dist(u) at that moment was the length of some s to u walk.
  - Just add u ➔ v to that walk to make an s to v walk of length dist(u) + w(u ➔ v) = dist(v).
- The rest of the argument for FordSSSP's correctness then follows this intuition: as long as

we haven't found all the shortest path distances, there exists a vertex v to which we can find a shorter walk by relaxing an edge into v.

- One more thing: As I present different variants of the shortest paths algorithm, I'm going to focus on running time analysis and proving the dist values are successfully set to the actual distances to each vertex. In all cases, you can use a similar induction proof to the one we just did to show that the *final* walk of length dist(v) into each vertex uses the pred edge of each vertex on the walk.
- Alright, let's finally get to some of those variants.

When Nothing Else is Appropriate: Bellman-Ford

- The variant we're going to start with actually works for *any* graph that doesn't have a negative cycle. I'd like to start with it, because I think its proof of correctness does a good job of demonstrating the main principles behind all variants of the generic algorithm.
- As is often this case, this algorithm was proposed by many people independently, but everybody calls it Bellman-Ford now.
- We attempt to relax all the edges and then recurse.

```
BellmanFord(s)
  InitSSSP(s)
  while there is at least one tense edge
      for every edge u→v
          if u→v is tense
              Relax(u→v)
```

- It's completely mystifying that this algorithm can be efficient. After all, we're not doing anything clever.
- But it turns out the analysis is actually more straightforward than most of the other variants of FordSSSP.
- Let dist_≤i(v) denote the length of the shortest *walk* in G from s to v with *at most* i edges. So dist≤0(s) = 0 and dist≤0(v) = infinity for all v ≠ s.
- Lemma: For every vertex v and non-negative integer i, after i iterations we have dist(v) ≤ dist≤i(v).
- Proof:
  - If i = 0, the lemma is trivially true.
  - Let W be a shortest walk from s to v with at most i edges. By definition, W has length dist≤i(v).
  - If W has no edges, it goes from s to s, meaning v = s and dist≤i(v) = 0. dist(s) ← 0 in InitSSSP and dist(s) never increases, so dist(s) ≤ 0.
  - Otherwise, let u → v be the last edge of W. After i - 1 iterations, dist(u) ≤ dist≤{i-1}(u).
  - In the ith iteration, we consider edge u → v. Either dist(v) ≤ dist(u) + w(u → v) already or

we set dist(v) ← dist(u) + w(u → v).
    - Either way, dist(v)
        - ≤ dist(u) + w(u → v)
        - ≤ dist{≤i-1}(u) + w(u → v)
        - = dist≤i(v).
    - And dist(v) can only decrease further by the time the loop ends.
- This lemma is true even if there are negative length cycles!
- Because dist(v) is always the length of *some* walk, it is always at least the shortest path distance.
- If there are no negative cycles, the shortest walk from s to any v has at most V - 1 edges, so dist(v) must be the true shortest path distance by the end of V - 1 iterations.
- Each iteration takes O(E) time, so the algorithm takes O(VE) time if there are no negative length cycles.
- But what if there is a negative cycle C reachable from s? As we discussed earlier, there is always a walk W to any vertex w on C where the length of W is less than dist(w). If we travel W from s, we'll have to hit some first vertex v (maybe w itself) where dist(v) is larger than the length of that prefix of W to v. The previous vertex u didn't have that problem, so dist(u) + w(u → v) < dist(v), implying u → v is tense.
- Luckily, knowing there is at least one tense edge means we can modify the algorithm slightly to "fail gracefully" if there exists a negative cycle reachable from s.

```
BellmanFord(s)
  InitSSSP(s)
  repeat V − 1 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!"
```

- This version runs in O(VE) time even if there are negative cycles.
- Next time, we'll discuss some faster variants of FordSSSP for when either the edge weights or the graph structure have some nice properties.