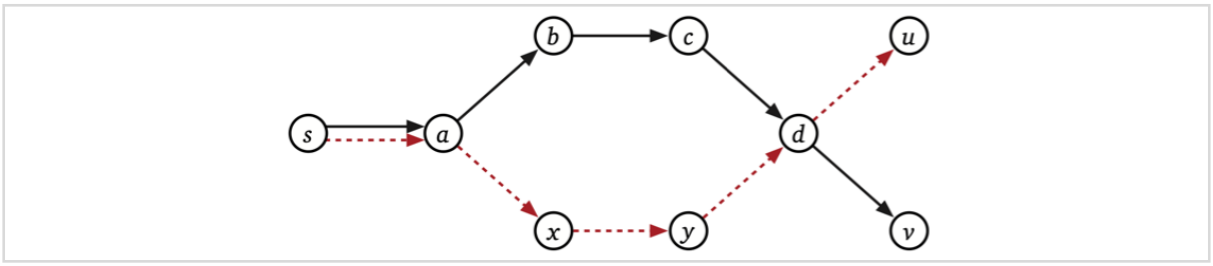
CS 4349.003.19F Lecture 17—October 23, 2019

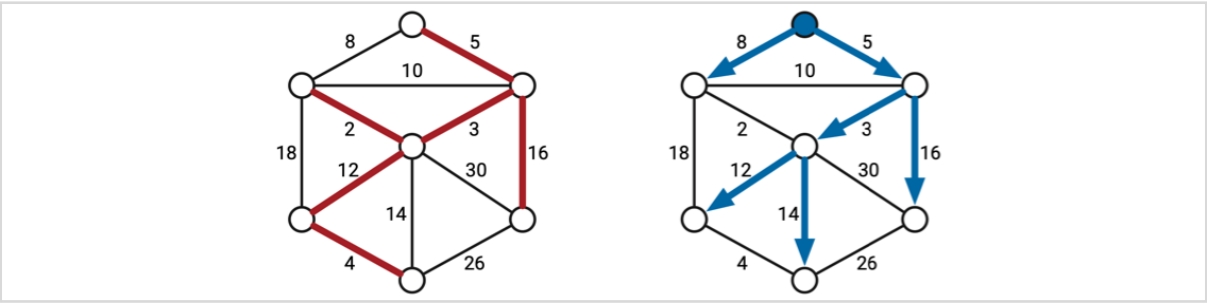
Main topics for #lecture include #single_source_shortest_paths and #breadth-first_search.

Single Source Shortest Paths

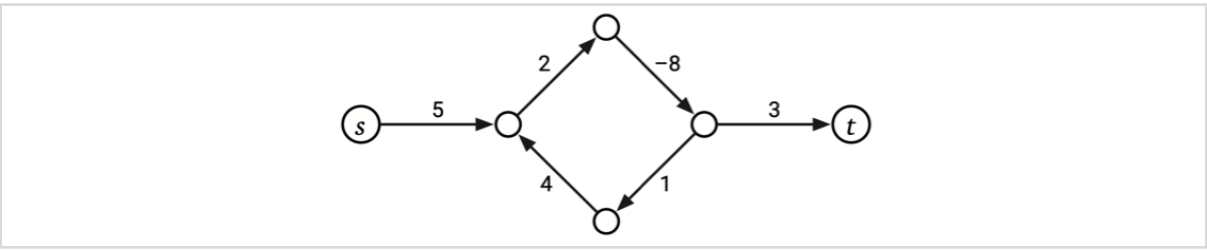
- For the next problem, let's say you're given a *directed* graph G = (V, E, w) where w : E ➔ R is another weight function. The shortest path between two vertices s and t is the s,t-path P minimizing w(P) = sum_{u ➔ v in P} w(u ➔ v). The minimum value is the *distance* from s to t.
- Most algorithms 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 the 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. In fact, every SSSP tree of an undirected graph may use a different set of edges from the minimum spanning tree.
- If you want to do shortest paths in a positively weighted undirected graph, replace every edge uv with a pair of edges u ➔ v and v ➔ u of the same weight. Again, the shortest path trees may all be different 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. What we're *really* going to compute today and Monday are shortest *walks* from s to every vertex. If there are negative weight cycles, a "shortest" walk would go around and around and around and infinite number of times before reaching its destination. In other words, the shortest walk is not well-defined.



- If there are no negative weight cycles, though, we can compute shortest walks and they'll be paths (no reason to repeat a vertex if coming back to it has non-negative cost).
- 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. You *can* do shortest paths in undirected graphs with negative weights, but the algorithms for that are well beyond the scope of this course.

The Only SSSP Algorithm

- Like minimum spanning trees, there's really only one shortest path tree algorithm 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.
- 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.
- At the beginning of the algorithm, we know dist(s) = 0 and pred(s) = Null. For every other vertex v ≠ s, we initially set dist(v) = infinity and pred(v) = Null, because we haven't found

any paths to those vertices yet!

```
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).
- If an edge is tense, then the distances are certainly not shortest path distances.
- 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 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 without proof that this algorithm does eventually terminate if there are no negative cycles. After it terminates, each value dist(v) is the shortest path distance to v from s, and the pred pointers define the shortest path tree. If dist(v) = infinity after the algorithm terminates, then v wasn't reachable from s to begin with.
- If there is even one negative cycle reachable from s, then this algorithm will never terminate. There will always be at least one tense edge on the cycle.
- Again, I won't directly prove those claims for the generic algorithm, because I haven't said anything about running time yet and the proof for the different efficient implementations would make anything I say about the generic algorithm redundant.
- But I will prove one thing: at all times, if dist(v) ≠ infinity, then dist(v) is 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.
  - We just added u ➡ v to that walk to make an s to v walk of length dist(u) + w(u ➡ v) = dist(v).
- So what are these different implements? The one you want to use depends on what kind of graph and edge weights you're given.

Unweighted Graphs: Breadth-First Search

- We'll start by considering what happens when all the edges have weight 1, so the length

of a path is just the number of edges.

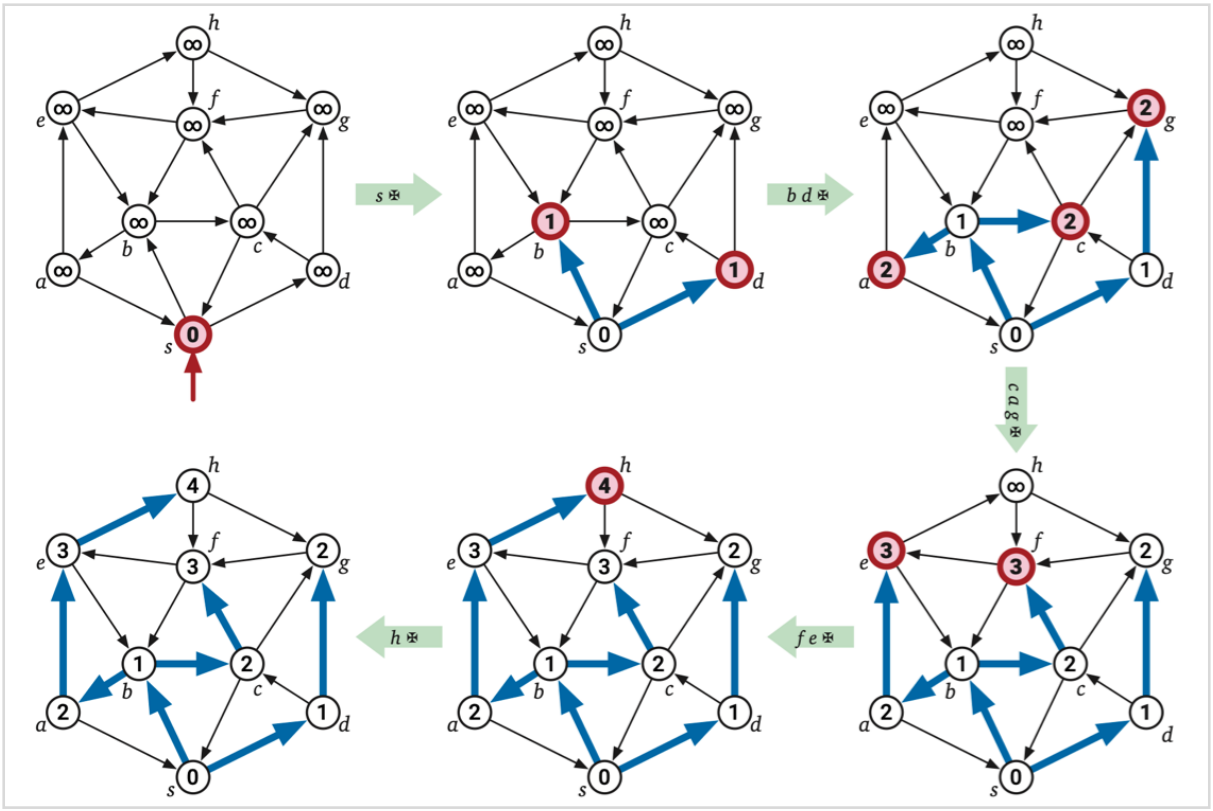
- Here, we want to use a breadth-first search (BFS). We can implement it using a whatever-first search with a queue as the bag.
- Today, though, I want to talk about a version that more closely resembles FordSSSP.
- We maintain a queue of vertices. Initially, the queue contains only s. We iteratively remove a vertex u from the queue and examine its outgoing edges. For each tense edge u ➔ v, we relax u ➔ v and push v into the queue.

```
BFS(s):
  InitSSSP(s)
  Push(s)
  while the queue is not empty
      u ← Pull( )
      for all edges u→v
          if dist(v) > dist(u) + 1          ⟨⟨if u→v is tense⟩⟩
              dist(v) ← dist(u) + 1
              pred(v) ← u                   ⟨⟨relax u→v⟩⟩
              Push(v)
```

- To analyze the algorithm, we need to break its execution into phases by introducing an imaginary *token* that gets pushed and pulled into he queue like the vertices. We push the token into the queue at the beginning of the algorithm. A phase ends when we pull the token out of the queue; we push the token back in at the start of the next phase.

```
BFSWithToken(s):
  InitSSSP(s)
  Push(s)
  Push(✻)                    ⟨⟨start the first phase⟩⟩
  while the queue contains at least one vertex
      u ← Pull( )
      if u = ✻
          Push(✻)            ⟨⟨start the next phase⟩⟩
      else
          for all edges u→v
              if dist(v) > dist(u) + 1      ⟨⟨if u→v is tense⟩⟩
                  dist(v) ← dist(u) + 1
                  pred(v) ← u               ⟨⟨relax u→v⟩⟩
                  Push(v)
```

- Again, the token is just to help with analysis. The actual order that we scan vertices and the distances we compute do not change. Here's an example run of the algorithm including what the queue looks like at the start of each phase.



- The phases help us keep track of changes to vertex distances.
- Lemma: For every i ≥ 0 and every vertex v, at the end of the ith phase, either dist(v) = infinity or dist(v) ≤ i. Also v is in the queue if and only if dist(v) = i.
- Proof:
  - At the end of phase 0, only s is in the queue and every other vertex v has dist(v) = infinity.
  - Suppose i > 0. At the start of the ith phase, the queue inductively contains every vertex u with dist(u) = i - 1 followed by the token.

$$\rightarrow \ \overline{\text{✠} \ \ i-1 \ \ i-1 \ \ \cdots \ \ i-1} \ \rightarrow$$

  - So, we'll remove all these vertices u from the queue before the phase ends.
  - For each, we look over each edge u ➔ v. For each tense u ➔ v, we set dist(v) ← dist(u) + 1 = i and put v in the queue. We won't pull any of these distance i vertices out until after the phase ends.

$$\rightarrow \ \overline{i \ \cdots \ i \ \text{✠} \ i-1 \ \cdots \ i-1} \ \rightarrow$$

  - And when the phase does end, we have added exactly the vertices v to which we assigned dist(v) = i.
- This lemma lets us analyze the running time.
- BFS assigns distance labels in non-decreasing order. However, dist(v) for any vertex never *increases*. Therefore,
  - each dist(v) is updated at most once
  - meaning each v is pushed into the queue at most once
  - meaning each u is pulled from the queue at most once

    - meaning each edge is checked for being tense at most once
- So BFS runs in O(V + E) time. But is it correct?
- Theorem: When BFS ends, dist(v) is the length of the shortest path from s to v, for every vertex v.
- Proof: Fix a vertex v and any path v_0 ➔ v_1 ➔ … ➔ v_ell where v_0 = s and v_ell = v. I'll prove dist(v_j) ≤ j for all j.
    - dist(v_0) = dist(s) = 0
    - For any j > 0, the induction hypothesis implies dist(v_{j - 1}) ≤ j - 1. Immediately after we pull v_{j - 1} from the queue, either dist(v_j) ≤ dist(v_{j - 1}) + 1 or we set dist(v_j) ← dist(v_{j - 1}) + 1.
    - Either way, dist(v_j) ≤ dist(v_{j - 1}) + 1 ≤ j.
    - I just argued that dist(v) = dist(v_ell) ≤ ell. This is true even if we started with the shortest path to v.
    - I already argued that dist(v) is the length of an actual walk from s to v, so it must be the shortest path distance.
- I'll spare you the details, but a similar induction proof shows dist(v) is the length of the particular path s ➔ ... ➔ pred(pred(v)) ➔ pred(v) ➔ v, meaning we really do get a shortest path tree using our pred values.

Directed Acyclic Graphs: Work in Topological Order

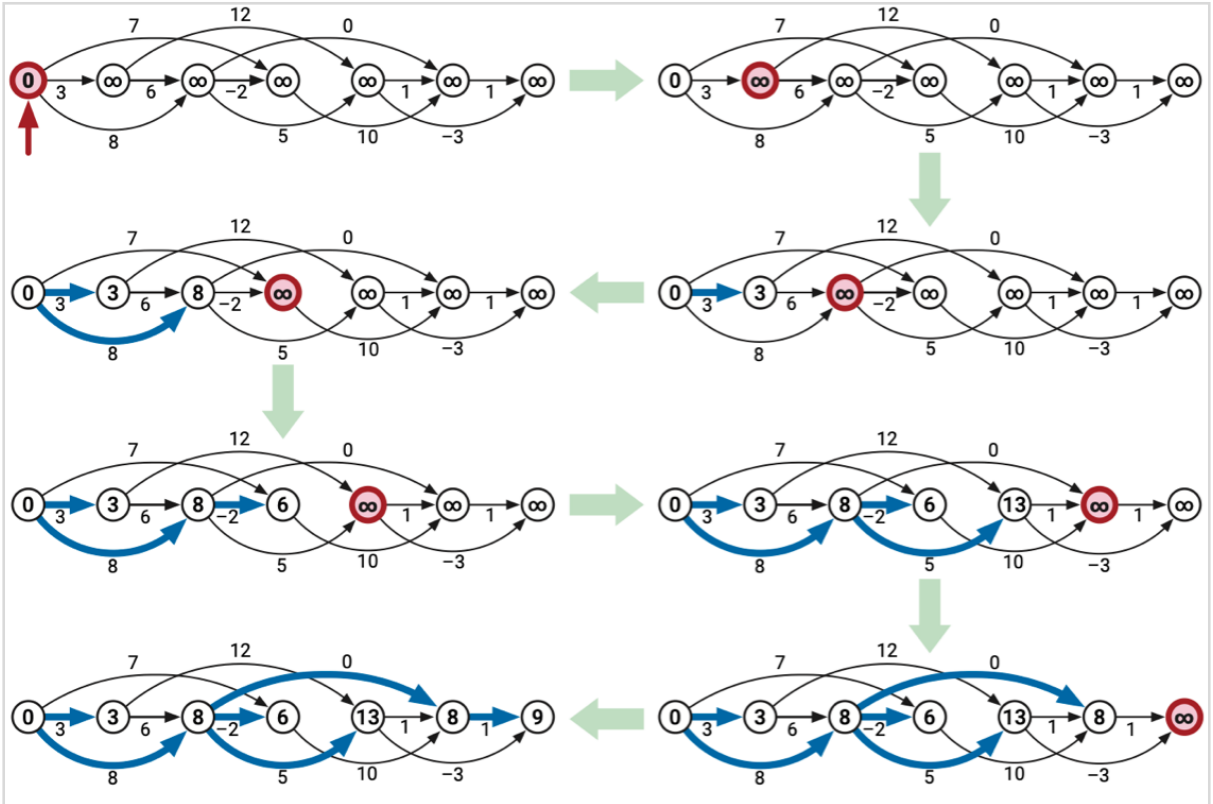
- If we somehow have time, let's consider an even easier case. Otherwise, we'll handle this case *next Wednesday*.
- Suppose we're given a directed acyclic graph with arbitrary edge weights.
- We don't have to worry about negative cycles at all with this case, because there aren't any cycles at all in a DAG!
- Surprisingly, the key to solving this case is to consider dynamic programming. It's back!
- Let dist(v) be the *actual* shortest path distance from s. If v = s, then dist(v) = 0. Otherwise, the shortest path contains some last edge u - > v. Therefore,
- dist(v) =
    - 0 if v = s
    - min_{u ➔ v} (dist(u) + w(u ➔ v)) otherwise
- This identity is true for *all* directed graphs, but we can't base an algorithm on it if there's a directed cycle. We'd just keep recursing backwards forever.
- But since we're working with a DAG, each recursive call visits an earlier vertex in topological order.
- So, we should just compute each dist value in topological order! Here's the normal looking dynamic programming algorithm based directly off the recurrence.

```
DAGSSSP(s):
  for all vertices v in topological order
      if v = s
          dist(v) ← 0
      else
          dist(v) ← ∞
          for all edges u→v
              if dist(v) > dist(u) + w(u→v)          ⟨⟨if u→v is tense⟩⟩
                  dist(v) ← dist(u) + w(u→v)          ⟨⟨relax u→v⟩⟩
```

- And here's an algorithm that uses our relax procedure to actual compute the pred values instead.

```
DAGSSSP(s):
  INITSSSP(s)
  for all vertices v in topological order
      for all edges u→v
          if u→v is tense
              RELAX(u→v)
```

- Here's an example run of the algorithm.



- In both variants of this algorithm, we end up looking at every vertex and edge exactly once after computing the topological order in O(V + E) time, so the algorithm runs in O(V + E) time total.
- Finally, you might not like the fact that we're looking at *incoming* edges of a vertex when we'd normally consider outgoing edges in any of our algorithms. In particular, breadth-first search and Dijkstra's algorithm as we'll see on Monday all consider outgoing edges of

a vertex u whenever we compute the true distance to u. We can do the same here:

```
PUSHDAGSSSP(s):
  INITSSSP(s)
  for all vertices u in topological order
    for all outgoing edges u→v
      if u→v is tense
        RELAX(u→v)
```

- By the time we process a vertex v, we have already checked every edge u ➜ v and so dist(v) will be correct.