CS 6363.003.21S Lecture 16—April 1, 2021

Main topics are #single_source_shortest_paths.

No (or few) Negative Edges: Dijkstra's Algorithm

- Let's continue talking about single source shortest paths today.
- Recall, the idea for all the algorithms 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 walk, or infinity if we haven't found one yet.
- pred(v) is the predecessor of v in the tentative shortest s to v walk, or Null if we haven't found one yet.
- We start by calling InitSSSP(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).
- We want to *relax* tense edges to represent our newly found shorter walk/proof that our old dist value was too large.

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

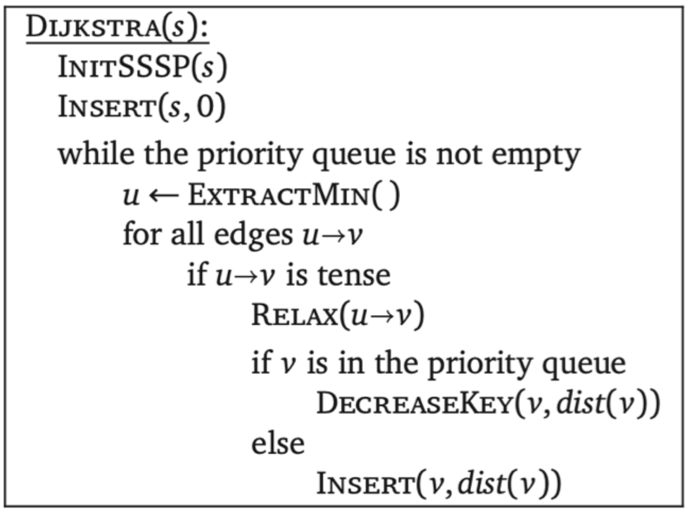
- The only SSSP algorithm repeatedly finds some tense edge and relaxes it.

```
FordSSSP(s):
    InitSSSP(s)
    while there is at least one tense edge
        Relax any tense edge
```

- Today, we'll discuss an algorithm that was independently discovered by many many many different researchers, but by now everybody has decided to call it Dijkstra's algorithm. This algorithm works well when edge weights are non-negative.
- It relies on two main observations:
  - 1) Edge u ➔ v can become tense only when u has its dist value decreased (or initiated as 0 if u = s).
  - 2) If edge weights are non-negative, dist(v) cannot be set lower than dist(u) after relaxing u ➔ v. So relaxing the tense edge u ➔ v with lowest dist(u) should not lead to a chain of relaxations that eventually decrease dist(u) again.
- To aid in finding these best tense edges, we'll keep a priority queue of possible *tail*

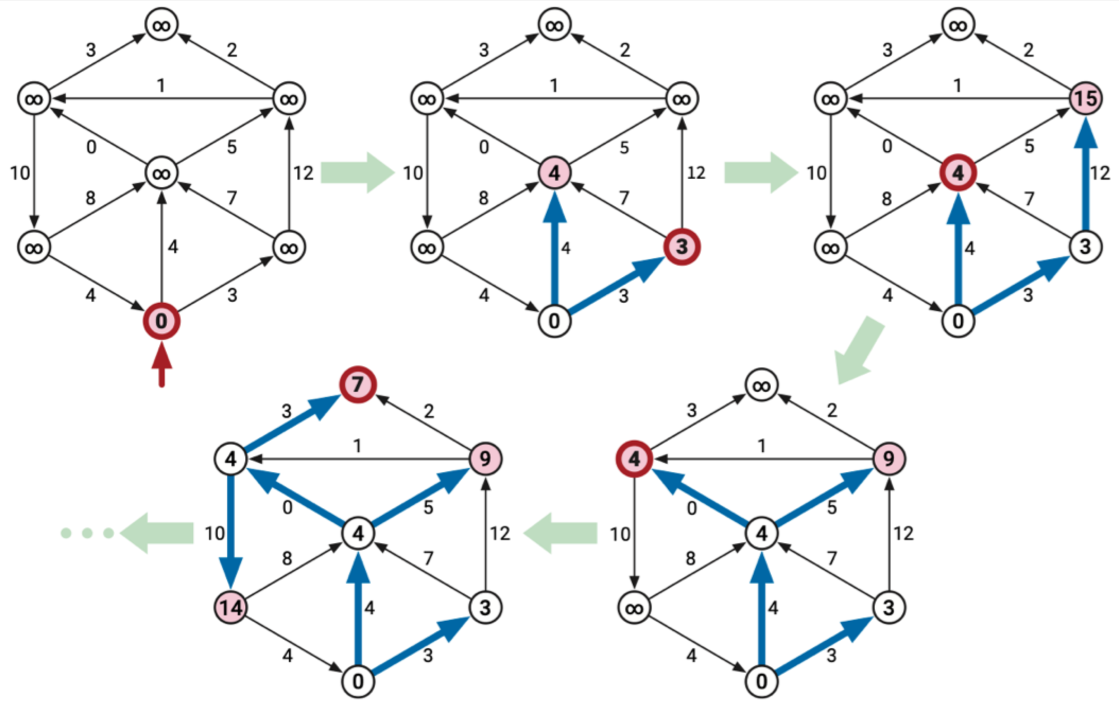
vertices with with the key of vertex u being dist(u). Whenever some dist(v) decreases, we'll either add v to the queue or lower its priority.

- To actually perform relaxations, we repeatedly extract the minimum distance vertex from the queue and relax all outgoing edges, updating the queue as described above whenever necessary.



```
DIJKSTRA($s$):
    INITSSSP($s$)
    INSERT($s, 0$)
    while the priority queue is not empty
        $u \leftarrow$ EXTRACTMIN( )
        for all edges $u \rightarrow v$
            if $u \rightarrow v$ is tense
                RELAX($u \rightarrow v$)
                if $v$ is in the priority queue
                    DECREASEKEY($v, dist(v)$)
                else
                    INSERT($v, dist(v)$)
```

- Now, this is an instance of Ford's general strategy, so I claim (without full proof) that it computes shortest paths as long as there are no negative *cycles* in the graph. Yes, negative edges alone are not enough to prevent the algorithm from working. I just can't claim anything good about the running time.
- Of course, something special happens if there are no negative edges at all.
- Intuitively, you could imagine a wavefront spreading out from s, passing over vertices in increasing order of their distance and never returning to a vertex that's already been passed over.



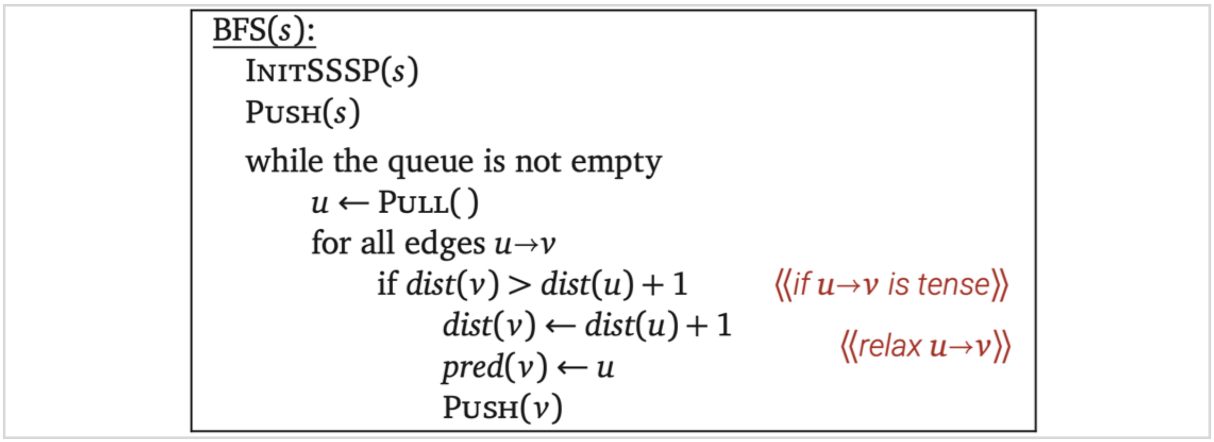
- Let's analyze this algorithm and prove correctness assuming no negative weight edges. Let u_i be the vertex returned by the ith ExtractMin call (so u_1 = s) and d_i be dist(u_i) just after the Extraction (so d_1 = 0). We cannot assume yet these vertices are distinct. For all we know u_i = u_j for some i < j.

- Lemma: For all i < j, we have d_j ≥ d_i. (Vertices are extracted in non-decreasing order of distance.)
    - Fix some i. We'll show d_{i+1} ≥ d_i.
    - If u_i → u_{i+1} is relaxed during the ith iteration, then afterward d_{i+1} = dist(u_{i+1}) = dist(u_i)+ w(u_i → u_{i+1}) ≥ dist(u_i) = d_i.
    - Otherwise, u_{i+1} is already in the priority queue when we extract u_i. But we Extracted u_i so d_{i+1} = dist(u_{i+1}) ≥ dist(u_i) = d_i.
- Lemma: Each vertex is extracted from the priority queue at most once.
    - Suppose v = u_i = u_j for some i < j.
    - The only way to put v in the queue after its first extraction would be to decrease dist(v) during a later relaxation.
    - But then d_j < d_i, contradicting the previous lemma.
- Lemma: When Dijkstra ends, for all vertices v, dist(v) is the length of the shortest path from s to v.
    - Let v_0 → v_1 → … → v_ell where v_0 = s and v_ell = v be the shortest path to v. Let L_j be the length of the subpath v_0 → … → v_j. We'll prove by induction on j that dist(v_j) ≤ L_j.
    - dist(v_0) = dist(s) = 0 = L_0.
    - Consider j > 0. By induction, at some point we Extract v_{j-1} from the queue. At that moment either dist(v_j) ≤ dist_{v_{j-1}) + w(v_{j-1} → v_j) already or we set dist(v_j) ← dist_{v_{j-1}) + w(v_{j-1} → v_j) by the end of v_{j-1}'s for loop. Either way, dist(v_j)
        - ≤ dist_{v_{j-1}) + w(v_{j-1} → v_j)
        - ≤ L_{j-1} + w(v_{j-1} → v_j) (by the IH)
        - = L_j.
    - In particular, dist(v) ≤ L_ell, the distance from s to v.
    - Again, dist(v) is at least the actual distance to v, because it is the length of some walk. We just argued dist(v) is at most the distance as well, so it is actually equal to the distance.
- If we use a standard binary heap for the priority queue, each operation takes O(log V) time. We perform at most one Insert and ExtractMin per vertex, and at most one DecreaseKey per edge, so O(E log V) time total, assuming non-negative edge weights.
- Again, this algorithm, as I wrote it, works just fine if you have negative edge weights but no negative cycles. In fact, it's likely to be faster than Bellman-Ford if you only have a few negative weight edges. If there are only a constant number of them, it still runs in O(E log V) time!
- You could also write a version that never puts a vertex back in the priority queue as CLRS

does. It will always runs in O(E log V) time, but now it may give incorrect distances if there are some negative weight edges.

Unweighted Graphs: Breadth-First Search

- So far we've seen Bellman-Ford which runs in O(VE) time if there are no negative weight cycles, and Dijkstra which runs in O(E log V) time if there are no negative weights at all.
- It's tempting to always rely on these two work horses, but in some special but common cases, we can do even better. Double check what your input looks like before you overlook these next two algorithms!
- We've already talked about the first one: suppose all your edges have weight 1 (you want to minimize the number of edges on each path).
- Here, we want to use a breadth-first search (BFS). I showed you one way to implement it a few weeks ago.
- But since we're here, I may as well show you a version that looks more like 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)
```

- The formal analysis of this algorithm is similar to Dijkstra's, but you need to be careful to argue that the queue stores vertices in increasing order of distance.
- Similar to Dijkstra, each vertex enters the queue at most once. Each queue operation and potential relaxation takes O(1) time, so the total running time is O(V + E).
- But we saw BFS before, so let's move on.

Directed Acyclic Graphs: Work in Topological Order

- Something you're less likely to have seen before is what to do when given a DAG.
- DAGs are easy to handle even with completely arbitrary edge weights. In fact, we don't have to worry about negative cycles with this case, because there aren't *any* cycles in a DAG!
- Like we saw with *longest* path, the trick here is to use dynamic programming. We'll turn the

recurrence around, though, to more closely resemble FordSSSP.

- 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,

$$dist(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{u\to v} (dist(u) + w(u\to v)) & \text{otherwise} \end{cases}$$

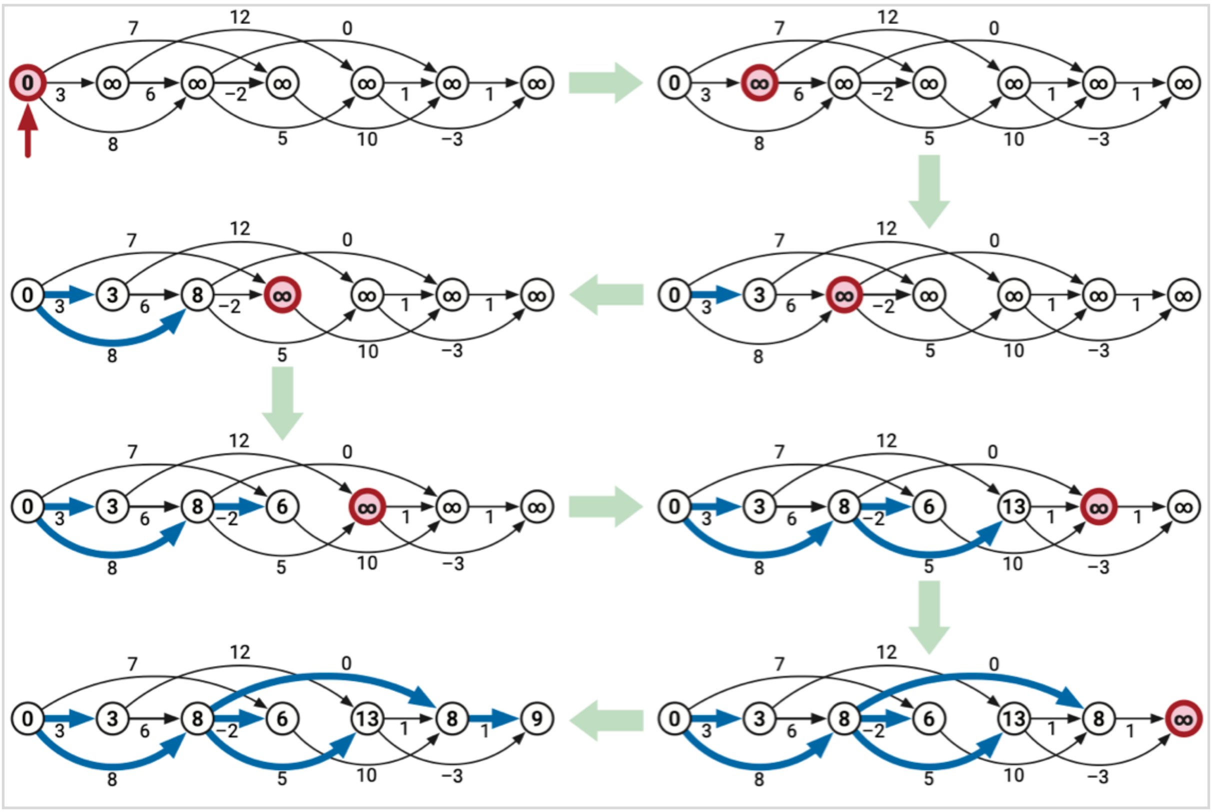
- This identity is true for *all* directed graphs, but we can't base an algorithm on it if there's a directed cycle. We'd just keep recursing backwards forever.
- But since we're working with a DAG, there exist topological orderings, and each recursive call depends only on vertices that come earlier in any such ordering.
- Here's the normal looking dynamic programming algorithm based directly off the recurrence.

```
DAGSSSP(s):
  for all vertices $v$ in topological order
      if $v = s$
          $dist(v) \leftarrow 0$
      else
          $dist(v) \leftarrow \infty$
          for all edges $u\to v$
              if $dist(v) > dist(u) + w(u\to v)$        ⟨⟨if $u\to v$ is tense⟩⟩
                  $dist(v) \leftarrow dist(u) + w(u\to v)$     ⟨⟨relax $u\to v$⟩⟩
```

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

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

- 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 only worked with outgoing edges in the previous algorithms. 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. This version also runs in O(V + E) time.
- Next Tuesday, why settle for a single source, when we can work with them all!