# CS 6363.005.19S Lecture 15—March 12, 2019

Main topics are #single_source_shortest_paths.

## Prelude

- Homework 3 is due **Thursday** March 14.
- Jiashuai is holding a guest lecture on matroids on Thursday.
- I'm holding extra office hours this afternoon from 2 to 3 since I'll be out town that day.

## Prim-Jarník Algorithm

- Last time, I put this algorithm up on the board for computing minimum spanning trees. I want to spend a couple minutes analyzing it before we move on to shortest paths.
- **[time permitting, write your own version that sets everything to infinity except priority(s) = 0 and edge(s) = emptyset. then it's more clear that we're using an adjacency list]**

```
JARNÍK(V, E, s):
    JARNÍKINIT(V, E, s)
    JARNÍKLOOP(V, E, s)

JARNÍKINIT(V, E, s):
    for each vertex v ∈ V \ {s}
        if vs ∈ E
            edge(v) ← vs
            priority(v) ← w(vs)
        else
            edge(v) ← NULL
            priority(v) ← ∞
        INSERT(v)

JARNÍKLOOP(V, E, s):
    T ← ({s}, ∅)
    for i ← 1 to |V| − 1
        v ← EXTRACTMIN
        add v and edge(v) to T
        for each neighbor u of v
            if u ∉ T and priority(u) > w(uv)
                edge(u) ← uv
                DECREASEKEY(u, w(uv))
```

- There are O(E) DecreaseKey operations, O(V) Inserts, and O(V) ExtractMins.
- Using a normal binary heap for the priority queue, operations takes O(log V) time each, so the algorithm takes O(E log V) time total.
- But, we can get fancy and use something called a Fibonacci heap instead. These handle DecreaseKeys in O(E) time on average across all the operations, so the whole thing takes O(E + V log V) time instead.
- But this is probably slower in practice unless you have very large dense graphs.

## 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. 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 trees are best described for directed graphs and are themselves directed away from their 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 an 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.
- Like with minimum spanning trees, there's nothing saying we can't have 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 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; i.e. it's not well-defined.
- If there are no negative weight cycles, though, then 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 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
```

- Now, I could prove correctness for this algorithm, but I can't say too much about running time, because that depends heavily on the order in which we relax edges.
- In particular, there are different relaxation orders depending on the time of graph and weights you use.
- If all weights are 1 (your graph is unweighted), use a BFS in O(V + E) time.
- If the graph is a DAG, use the dynamic programming algorithm I showed you but with a min, again in O(V + E) time.
- There are two more common algorithms. To avoid repeating myself too much, I'll prove correctness and analyze running more-or-less separately for the two special cases.
- One observation we'll need for both algorithms, though, is that if dist(v) ≠ infinity, then it 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).
- In particular, dist(v) is always *at least* the shortest path distance from s to v.
- We can use a similar induction proof to show the predecessors give shortest paths when no edges are tense, but I'll only focus on distances today.

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

- The first algorithm was independent discovered by many many many different researchers, but by now everybody has decided to call it Dijkstra's algorithm.
- It's similar to Prim-Jarník. We'll use a priority queue on vertices with the key of vertex v being dist(v). We repeatedly extract the minimum distance vertex and relax all outgoing edges. When dist(w) changes for some vertex w, we'll insert it in the priority queue or decrease it's key.

```
DIJKSTRA(s):
  INITSSSP(s)
  INSERT(s, 0)
  while the priority queue is not empty
      u ← EXTRACTMIN( )
      for all edges u→v
          if u→v is tense
              RELAX(u→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 it computes shortest paths as long as there are no negative *cycles* in the graph.
- But 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_i ≤ d_j. (Vertices are extracted in non-decreasing order of distance.)
  - Fix some i. By induction, it suffices to prove d_{i+1} ≥ d_i, meaning the distances are non-decreasing.
  - 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 = dist(u_i) ≤ dist(u_{i+1}) = d_{i+1}.
- Lemma: Each vertex is extracted from the priority queue at most once.
  - Suppose v = u_i = u_k for i < k.

    - dist(v) never increases and v is reinserted in the queue only when dist(v) decreases, so d_k < d_i. But that contradicts the previous lemma.
- Lemma: When Dijkstra ends, dist(v) is the length of the shortest path from s to v for every vertex v.
    - For any vertex v, consider some shortest path v_0 ➔ v_1 ➔ … ➔ v_ell where v_0 = s and v_ell = 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, we set dist(v_{j-1}) and at some point 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 the iteration. Either way
        - dist(v_j) ≤ dist_{v_{j-1}) + w(v_{j-1} + v_j) ≤ L_{j-1} + w(v_{j-1} ➔ v_j) = L_j.
    - In particular, dist(v) ≤ L_ell = the length of the whole path.
    - Again, dist(v) is at least the shortest path distance and therefore equal to it.
- Just like Prim-Jarník, we have V Insert and ExtractMin operations and E DecreaseKey operations. With a min-heap that all takes O(E log V) time. With a Fibonacci heap, it's only O(E + V log V) time.
- Again, this algorithm, as I wrote it, works just fine if you have negative edge lengths but no negative cycles. In fact, it's likely to be faster than the next algorithm I present if you only have  a few negative length edges.
- You could also write a version that never puts a vertex back in the priority queue as CLRS does, but then it's incorrect if there's some negative length edges.

## If All Else Fails: Bellman-Ford

- OK, so what if you have some negative weights and you don't have a DAG and you want to prove a good performance guarantee?
- Again, this algorithm was proposed by many people, but everybody calls it Bellman-Ford now.
- We just relax all tense 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)
```

- This algorithm is somehow a bit easier to analyze.
- 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{≤i-1}(u) + w(u → v) = dist≤i(v). Again, dist(v) does not increase after that, although it may decrease further by the time the loop ends.
- This lemma is true even if there are negative length cycles!
- Again, dist(v) 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.
- That said, maybe there are negative length cycles and your distances are not shortest path distances. You can do yet another proof by induction to show there will always be a tense edge if there are any dist values that are too high.
- So there will be a tense edge after those V - 1 iterations, and we can modify the algorithm slightly to detect negative cycles.

```
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.