CS 6363.003.21S Lecture 17—April 6, 2021

Main topics are #all-pairs_shortest_paths.

All-Pairs Shortest Paths

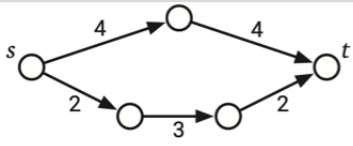
- Last week, we looked at various algorithms for computing single source shortest paths. But if you're trying to compute shortest paths in advance, say for a map, you may not know which sources you really need. So why not work with them all!
- This is the all-pairs shortest path problem. We want to compute dist(u, v), the length of the shortest path from u to v for all u and v.
- To keep things simple today, we'll assume there are no negative weight cycles. But to keep things interesting, we will allow for negative weight edges.
- So, there's an obvious algorithm for this problem. Compute single source shortest paths from every vertex!

```
OBVIOUSAPSP(V, E, w):
  for every vertex s
      dist[s, ·] ← SSSP(V, E, w, s)
```

- But depending on what kind of graph you have and what algorithm you call, this approach leads to different running times:
  - Unweighted: Use breadth-first search in V * O(E) = O(VE) = O(V^3) time.
  - A DAG: Use the DAG algorithm we just discussed in V * O(E) = O(VE) = O(V^3) time.
  - Non-negative edge weights: Use Dijkstra in O(V E log V) = O(V^3 log V) time.
  - And if all else fails: Use Bellman-Ford in V * O(VE) = O(V^2 E) = O(V^4) time.
- The last case is worse than the others. Can we get around O(V^3) time with negative length edges?

Johnson's Algorithm

- Just like in single source shortest paths, those negative weight edges are slowing us down.
- It would be great if we could just reweight all the edges so they were non-negative, but it's not obvious how to do this correctly.
- For example, we can't just add the same value to every edge. We'd be increasing the length of paths with many edges more than we would be for paths with few edges.
- For example, the shortest path changes if we add 2 to every edge in this graph.



- But there's a more complicated reweighing scheme we can use that is safe. It's often

attributed to Johnson ['73], but others came up with similar ideas.

- We'll associate with each vertex v a *price* pi(v) and define a new weight function

$$w'(u \to v) = \pi(u) + w(u \to v) - \pi(v)$$

- Imagine paying a tax of pi(u) to leave u but then getting an entrance gift of pi(v) when you finish traveling the edge.
- Obviously, the weight of paths will change. But the surprising thing is that the weight of every u,v-path u ––> v changes by the same amount! For every consecutive pair of edges u ➔ v ➔ w, we'll receive the entrance gift for v and then immediately pay it back as taxes. All the gifts and taxes for intermediate vertices cancel out, and we're left with

$$w'(u \rightsquigarrow v) = \pi(u) + w(u \rightsquigarrow v) - \pi(v).$$

- And if all the u,v-paths are changing by the same amount, the shortest u,v-path must remain shortest.
- So what would be great is if we could find prices that guarantee the modified weights are non-negative.
- This is the idea behind Johnson's algorithm.
- It begins by computing shortest paths from some vertex s to all other vertices using Bellman-Ford. If there's a negative cycle we just abort.
- Then, we set

$$w'(u \to v) = dist(s, u) + w(u \to v) - dist(s, v).$$

- The new weight of each edge u ➔ v is non-negative. Otherwise, u ➔ v would be tense and we wouldn't have shortest paths.
- But maybe there isn't a vertex s that can actually reach every other vertex, meaning we'll never find meaningful distances from s.
- We'll just add a new vertex s with 0 weight edges going *from* s to every other vertex. Then none of the original paths in G are actually affected, but we can start Johnson's algorithm.

```
JohnsonAPSP(V, E, w) :
  ⟨⟨Add an artificial source⟩⟩
  add a new vertex s
  for every vertex v
      add a new edge s→v
      w(s→v) ← 0
  ⟨⟨Compute vertex prices⟩⟩
  dist[s, ·] ← BellmanFord(V, E, w, s)
  if BellmanFord found a negative cycle
      fail gracefully
  ⟨⟨Reweight the edges⟩⟩
  for every edge u→v ∈ E
      w'(u→v) ← dist[s, u] + w(u→v) − dist[s, v]
  ⟨⟨Compute reweighted shortest path distances⟩⟩
  for every vertex u
      dist'[u, ·] ← Dijkstra(V, E, w', u)
  ⟨⟨Compute original shortest-path distances⟩⟩
  for every vertex u
      for every vertex v
          dist[u, v] ← dist'[u, v] − dist[s, u] + dist[s, v]
```

- That last line fixes the computed distances so they're based on the original edge weights.
- We spend O(V) time adding s, O(VE) time running Bellman-Ford, O(E) time reweighing edges, and O(VE log V) time doing all those Dijkstra runs.
- That's O(VE log V) = O(V^3 log V) time total!

Dynamic Programming

- I like Johnson's approach, because it uses some ideas that are important in optimization, but there is another approach we can take based on dynamic programming. If the graph is dense (E = Omega(V^2)), then the dynamic programming approach leads to simpler and faster algorithms.
- One "obvious" recursive definition of dist(u, v) is the following. Just like before.

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

- But if we're not in a DAG, the recursive calls suggested by this recursive definition may keep going back and back and back around a directed cycle. We're stuck in an infinite loop!
- We need *something* that gets smaller in each recursive call so we know the recursion bottoms out.
- Earlier, we analyzed Bellman-Ford by considering shortest paths with at most i edges. Let's take inspiration from this analysis by putting the number of edges into our recursively

defined function.

- Let dist(u, v, ell) denote the length of the shortest path from u to v that uses at most ell edges.
- Either the path uses exactly ell edges or it uses at most ell - 1 edges.
- So, we have the following recursive function:

$$dist(u,v,\ell) = \begin{cases} 0 & \text{if } \ell = 0 \text{ and } u = v \\ \infty & \text{if } \ell = 0 \text{ and } u \neq v \\ \min \left\{ \begin{array}{c} dist(u,v,\ell-1) \\ \min\limits_{x \to v} (dist(u,x,\ell-1) + w(x \to v)) \end{array} \right\} & \text{otherwise} \end{cases}$$

- If there are no negative length cycles, then every shortest path uses at most V - 1 edges, so dist(u, v) = dist(u, v, V - 1).
- We can memoize this recurrence by looping over all ell from 0 to V - 1 and then over all u and then all v.
- This strategy was described by Alfonso Shimbel in 1943, so Erickson calls it ShimbelAPSP.

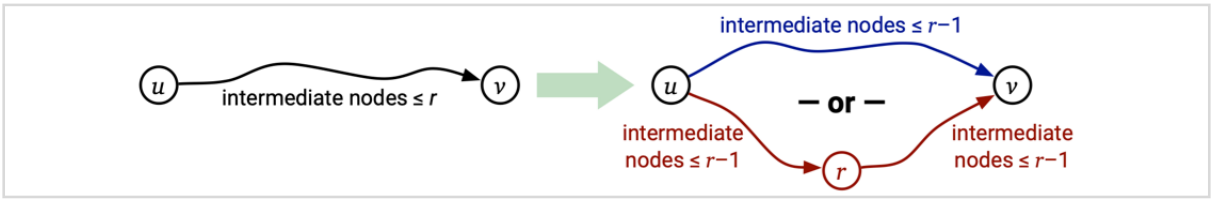
```
SHIMBELAPSP(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v, 0] ← 0
          else
              dist[u, v, 0] ← ∞

  for ℓ ← 1 to V − 1
      for all vertices u
          for all vertices v ≠ u
              dist[u, v, ℓ] ← dist[u, v, ℓ − 1]
              for all edges x→v
                  if dist[u, v, ℓ] > dist[u, x, ℓ − 1] + w(x→v)
                      dist[u, v, ℓ] ← dist[u, x, ℓ − 1] + w(x→v)
```

- Each edge is considered once per value of ell and u, so the whole thing takes O(V^2 E) = O(V^4) time. Oh, it didn't get any better.
- If you look at the recursive definition, the variable u doesn't change. We're really just computing shortest paths from each vertex u to all vertices v separately. In fact, computing the shortest paths using at most ell edges from u is essentially another way to describe Bellman-Ford with u as the source. So we shouldn't be surprised by ShimbelAPSP having the same running time as V runs of Bellman-Ford.
- One improvement we can make is not to guess the previous vertex on each path but instead to guess the middle vertex. Then the value for ell is cut in half. We'll only need to consider O(log V) different powers of 2 for ell, leading to a final running time of O(V^3 log V) which is the same time we got for Johnson's algorithm. See Erickson for details.

Floyd-Warshall

- But the best improvement comes from using a different third variable for our recursive definition.
- Instead of tracking how many edges appear in a path, we'll instead track *which vertices* are allowed to appear in the path.
- Number the vertices arbitrarily from 1 to V.
- pi(u, v, r) := the shortest path from u to v where every *intermediate* vertex is numbered at most r.
- And now define dist(u, v, r) to be the length of pi(u, v, r).
- Path pi(u, v, |V|) is the true shortest path from u to v, because it is allowed to use any of the vertices. Therefore, dist(u, v) = dist(u, v, |V|).
- If r = 0, we can't have any intermediate vertices. So either pi(u, v, 0) = u ➔ v or it is not defined.
- But what about r > 0? Either pi(u, v, r) uses intermediate vertex r or it doesn't.



- If it doesn't, then pi(u, v, r) = pi(u, v, r - 1) and dist(u, v, r) = dist(u, v, r -1).
- If it does, then it contains a subpath from u to r and a subpath from r to v so pi(u, v, r) = pi(u, r, r - 1) followed by pi(r, v, r - 1). In this case, dist(u, v, r) = dist(u, r, r - 1) + dist(r, v, r - 1).
- Naturally, pi(u, v, r) uses the better of the two options.

$$dist(u,v,r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min \begin{Bmatrix} dist(u,v,r-1) \\ dist(u,r,r-1) + dist(r,v,r-1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

- We need to solve V x V x V = Theta(V^3) subproblems, but it takes only constant time for each. So the algorithm will take Theta(V^3) time.
- Jeff calls this algorithm KleeneAPSP (clay knee), because Kleene discovered this recursive pattern first while studying finite automata.

```
KLEENEAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)

  for r ← 1 to V
      for all vertices u
          for all vertices v
              if dist[u, v, r − 1] < dist[u, r, r − 1] + dist[r, v, r − 1]
                  dist[u, v, r] ← dist[u, v, r − 1]
              else
                  dist[u, v, r] ← dist[u, r, r − 1] + dist[r, v, r − 1]
```

- We can clean this algorithm up a bit. First, we really only need to maintain the shortest paths from each u to v we've found so far, not which specific vertices they were allowed to go through.
- Also, we don't need to keep track of the specific vertex numbers as long a we loop through all the vertices.
- With these two changes, we end up with an algorithm usually contributed to Floyd and Warshall:

```
FLOYDWARSHALL(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)

  for all vertices r
      for all vertices u
          for all vertices v
              if dist[u, v] > dist[u, r] + dist[r, v]
                  dist[u, v] ← dist[u, r] + dist[r, v]
```

- This version of the algorithm still runs in O(V^3) time, but only uses O(V^2) space. A formal proof of correctness involves a similar induction proof to the one we used for Bellman-Ford. Namely, we can prove that after the rth iteration of the outer for loop, variable dist(u, v) is at most the value dist(u, v, r). I'll spare you the details.