CS 4349 Lecture—October 30, 2017

Main topics for #lecture include #APSP.

Prelude

- Homework 8 due Wednesday, November 1st.

All Pairs Shortest Paths

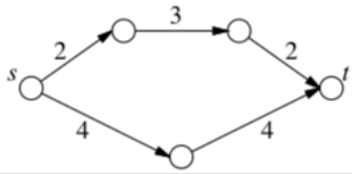
- Again, let G = (V, E) be an edge-weighted directed graph.
- Last time, we were discussing algorithms for the single source shortest path problem.
- In these algorithm, we actually computed shortest paths from a single vertex to every other vertex in the graph.
- So today, we're going to take this idea one step further and compute shortest paths from every vertex to every other vertex.
- In other words, we want to compute a V x V distance matrix where dist[u][v] is equal to the length of the shortest path from u to v. Many paper maps contain a matrix like this for quickly looking up distances between major cities.
- We'll focus just on computing these distances today. Computing the actual shortest paths isn't that much harder.

Reweighing

- So there's an obvious algorithm for solving this problem. Just run single source shortest paths from every vertex in the graph!

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

- Of course, the running time depends upon which SSSP algorithm we choose.
- If we use Shimbel's then this will take V x O(VE) = O(V^2 E) = O(V^4).
- If the edge weights are non-negative, we could use Dijkstra's with Fibonacci queues in time V x O(E + V log V) = O(VE + V^2 log V) = O(V^3).
- 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.
- Let c(v) be the *cost* of vertex v.
- Set w'(u ➔ v) := c(u) + w(u ➔ v) - c(v). Imagine paying a tax of c(u) to leave vertex u, but when you enter v you get a gift of c(v).
- But think about any path u - - - > v = u ➔ v_1 ➔ v_2 ➔ … ➔ v_k ➔ v, its modified weight is w'(u - - - > v) = c(u) + w(u ➔ v_1) - c(v_1) + c(v_1) + w(v_1 ➔ v_2) - c(v_2) + … + c(v_k) + w(v_k ➔ v) - c(v) = c(u) + w(u - - - > v) - c(v). For every vertex along the path we get a gift for visiting but then immediately pay it back in the tax.
- But since every u to v path has the same first and last term, the shortest one by the new weights is the shortest one for the original weights.
- So what would be great is if we could find costs making the modified weights non-negative and then run Dijkstra's algorithm.
- This is the idea behind Johnson's algorithm.
- It begins by computing shortest paths from some vertex s to all other vertices using Shimbel's algorithm. If there's a negative cycle we just abort.
- Then, we set w'(u ➔ v) := dist(s, u) + w(u ➔ 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.
- OK, there's one more detail I'm leaving out. 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 and infinite weight edges going the other way. Then no shortest paths are actually affected, but we can start Johnson's algorithm.

```
JohnsonAPSP(V, E, w):
    create a new vertex s
    for every vertex v
        w(s→v) ← 0
        w(v→s) ← ∞
    dist[s, ·] ← Shimbel(V, E, w, s)
    if Shimbel found a negative cycle
        fail gracefully
    for every edge (u, v) ∈ E
        w'(u→v) ← dist[s, u] + w(u→v) − dist[s, v]
    for every vertex u
        dist[u, ·] ← Dijkstra(V, E, w', 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 Theta(V) time adding s, Theta(VE) time running Shimbel, Theta(E) time reweighing edges, and O(VE + V^2 log V) time doing all the Dijkstra runs for O(VE + V^2 log V) time total.

Dynamic Programming

- But, that algorithm is rather complicated, especially because we're using Fibonacci queues.
- Maybe we can use dynamic programming to do something more simple.
- Let's make life a bit easier for the rest of the lecture. Let's assume there are no negative length cycles. Also w(u ➔ v) = infinity if edge u ➔ v doesn't exist and w(v, v) = 0.
- Now, there's an easy recursive definition of a shortest path based on the last edge along the path.
- dist(u, v) = { 0 if u = v, min_{x ➔ v} (dist(u, x) + w(x ➔ v)}.
- Unfortunately, this recurrence may never terminate. We may end up trying to compute dist(u, v) while trying to compute dist(u, x).
- But we wrote a dynamic programming algorithm on Wednesday that avoided this issue.
- There, we also used a parameter telling us the maximum number of edges lying on the paths were considering.
- Let dist(u, v, k) be the length of the shortest path from u to v using at most k edges. We really want to compute dist(u, v, V - 1) for every pair u, v.
- dist(u, v, k) =
  - 0 if u = v
  - infinity if k = 0 and u ≠ v
  - min_{x ➔ v} (dist(u, x, k-1) + w(x ➔ v)) otherwise
- k decreases in each recursive call, so an algorithm based on this recurrence will terminate.
- Using dynamic programming, we note there O(V) choices for each of u, v, and k, but it takes O(V) time to look at all the incoming edges for a recursive problem, so O(V^4) time total.
- Ah, but we only look at each edge once per pair u and k, so its really O(V^2 E) time. Still not great.
- The iterative algorithm proposed for this recurrence was first described by Shimbel. It's actually just O(V) instances of his single source algorithm, so the running time shouldn't be surprising.

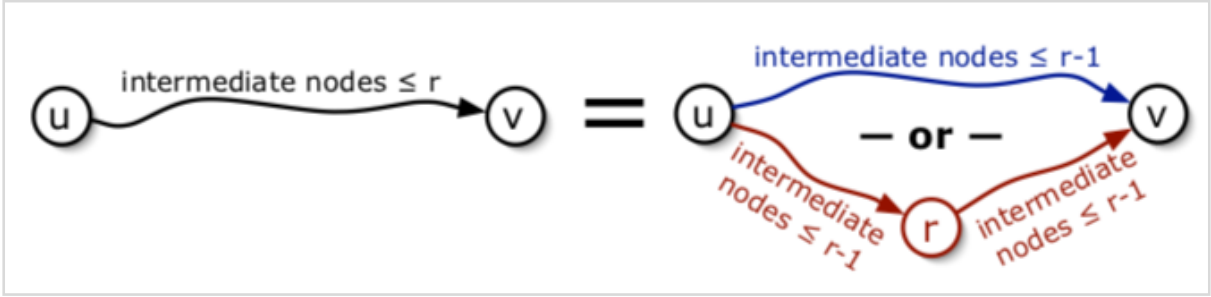
Dynamic Programming + Divide-and-Conquer

- Now, there's a simple way to make a much faster algorithm. Right now we're decreasing k by 1 in recursive calls by guessing the last *edge* along the path.

- Instead, what if we tried combining divide-and-conquer with dynamic programming. A shortest path of at most k vertices can be broken *anywhere* into two shortest paths, so let's guess the middle vertex and break it there.
- dist(u, v, k) =
  - w(u ➔ v) if k = 1
  - min_x (dist(u, x, k/2) + dist(x, v, k/2)) otherwise
- I'm assuming k is a power of 2 in this recurrence, but we can always just use dist(u, v, 2^{ceil{log V}}) for dist(u, v).
- OK, so now there are O(log V) choices for k.
- There's O(V^2 log V) subproblems, and each takes O(V) time to solve, so O(V^3 log V) overall using dynamic programming.

Floyd-Warshall

- But there's one more trick we can try.
- As always, versions of the algorithm I'm about to describe were proposed by several people, but most everybody calls this the Floyd-Warshall algorithm.
- So the main problem with the above recurrences is that we have to spend time guessing something in a min. Either the last edge along the shortest path or some vertex lying in the middle.
- We'll remove our need to make many guesses by picking a different third parameter.
- Instead of using the length of a path, let's place a limit on which vertices the path is allowed to contain.
- Number the vertices 1 through V.
- Let dist(u, v, r) be the length of the shortest path where every intermediate vertex (every vertex except u and v) is numbered 1 through r.
- dist(u, v, 0) = w(u ➔ v) since the path can't use *any* intermediate vertices.
- And dist(u, v) = dist(u, v, V).
- So, how do we recursively compute dist(u, v, r)? Well, the shortest path either has intermediate vertices 1 through r -1 or it contains vertex r at some intermediate point.



- So, dist(u, v, r) =
  - w(u ➔ v) if r = 0
  - min {dist(u, v, r-1), dist(u, r, r-1) + dist(r, v, r-1)} otherwise

- A dynamic programing algorithm takes Theta(V^3) time.

```
FLOYDWARSHALL(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]
```

- Like in the dynamic programming version of Shimbel's algorithm I showed last week, we can simplify things a bit.
- The main idea behind the change is that we don't need to *require* only intermediate vertices 1 through r-1 or even recognize that they're numbered. We'll just change those last three loops so they sound more like a suggestion. "Hey, maybe the current distance from u to v isn't minimized because you failed to include shortest paths that include vertex r, so why don't you check for that?"

```
FLOYDWARSHALL2(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]
```