Main topics for lecture include single-source shortest paths and all-pairs shortest paths.

Directed Acyclic Graphs: Work in Topological Order

- We’re not quite done with single source shortest paths! But the algorithm I saved for last provides a nice transition to our next topic.
- 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 single source shortest paths in 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,
  \[
  \text{dist}(v) = \begin{cases} 
  0 & \text{if } v = s \\
  \min_{u \rightarrow v} (\text{dist}(u) + w(u \rightarrow v)) & \text{otherwise} 
  \end{cases}
  \]
- 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, 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)
          dist(v) ← dist(u) + w(u → v)  \(\text{relax } u \rightarrow v\)

```

- And here’s an algorithm that uses our relax procedure to actual compute the pred values also.
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 with non-negative weights consider outgoing edges of a vertex $u$ after we compute the true distance to $u$. We can do the same here:

By the time we process a vertex $v$, we have already checked every edge $u \rightarrow v$ and so $\text{dist}(v)$ will be correct to the real shortest path distance to $v$.
All-Pairs Shortest Paths

- So that’s single source shortest paths. But what if you want to know the shortest paths from every vertex.
- 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 cycles, but there could be negative weight edges.
- We can also compute values pred(u, v), the vertex immediately before v on the shortest path from u to v. But by now you hopefully see that actually computing these pred values requires only tiny changes over computing the dist values.
- 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 may lead to some pretty bad 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. If we use Fibonacci heaps, that goes down to O(V(E + V log V)) = O(V^3) time.
  - Otherwise: 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 that O(V^3) with negative length edges?
- I’ll show you one way to do it based on dynamic programming.
- One “obvious” recursive definition of dist(u, v) is the following. Just like before.

\[
dist(u, v) = \begin{cases} 
  0 & \text{if } u = v \\
  \min_{x \to v} \left( \text{dist}(u, x) + w(x \to v) \right) & \text{otherwise}
\end{cases}
\]

- 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 ell edges or it uses at most ell - 1 edges.

So, we have the following recursive function:

\[
dist(u, v, \ell) = \begin{cases} 
0 & \text{if } \ell = 0 \text{ and } u = v \\
\infty & \text{if } \ell = 0 \text{ and } u \neq v \\
\min \left\{ \begin{array}{l}
\min_{x \rightarrow v} (dist(u, x, \ell - 1) + w(x \rightarrow v)) \end{array} \right. & \text{otherwise}
\end{cases}
\]

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

\[
dist[u, v, 0] \leftarrow 0
\]

else

\[
\text{dist}[u, v, 0] \leftarrow \infty
\]

for \( \ell \leftarrow 1 \) to \( V - 1 \)

for all vertices u

for all vertices \( v \neq u \)

\[
\text{dist}[u, v, \ell] \leftarrow \text{dist}[u, v, \ell - 1]
\]

for all edges \( x \rightarrow v \)

if \( \text{dist}[u, v, \ell] > \text{dist}[u, x, \ell - 1] + w(x \rightarrow v) \)

\[
\text{dist}[u, v, \ell] \leftarrow \text{dist}[u, x, \ell - 1] + w(x \rightarrow v)
\]

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 \( O(V^3 \log V) \). 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 vertex except u and v is numbered at most r.
- In particular, \( \pi(u, v, V) \) is the true shortest path from u to v since it can use any of the vertices.
- If \( r = 0 \), we can’t have any intermediate vertices. So either \( \pi(u, v, 0) = u \rightarrow v \) or it is not defined.
- Now, either \( \pi(u, v, r) \) uses intermediate vertex r or it doesn’t.

\[
\begin{align*}
\text{dist}(u, v, r) &= \begin{cases} 
    w(u \rightarrow v) & \text{if } r = 0 \\
    \min \left\{ \text{dist}(u, v, r - 1), \text{dist}(u, r, r - 1) + \text{dist}(r, v, r - 1) \right\} & \text{otherwise}
\end{cases}
\end{align*}
\]

- If it doesn’t, then \( \pi(u, v, r) = \pi(u, v, r - 1) \).
- 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 \( \pi(r, v, r - 1) \).
- So now, let \( \text{dist}(u, v, r) \) be the length of \( \pi(u, v, r) \).

- Again, the shortest path from u to v may use any vertex, so \( \text{dist}(u, v) = \text{dist}(u, v, V) \).
- We need to compute \( V \times V \times V = \Theta(V^3) \) values, but it takes only constant time for each. So the algorithm will take \( \Theta(V^3) \) time.
- Jeff calls this algorithm \texttt{KleeneAPSP} (clay knee), because Kleene discovered this recursive pattern first while studying finite automata.

\[
\begin{align*}
\text{KleeneAPSP}(V, E, w):
    & \text{for all vertices } u \\
    & \quad \text{for all vertices } v \\
    & \quad \quad \text{dist}[u, v, 0] \leftarrow w(u \rightarrow v) \\
    & \text{for } r \leftarrow 1 \text{ to } V \\
    & \quad \text{for all vertices } u \\
    & \quad \quad \text{for all vertices } v \\
    & \quad \quad \quad \text{if dist}[u, v, r - 1] < \text{dist}[u, r, r - 1] + \text{dist}[r, v, r - 1] \\
    & \quad \quad \quad \quad \text{dist}[u, v, r] \leftarrow \text{dist}[u, v, r - 1] \\
    & \quad \quad \quad \text{else} \\
    & \quad \quad \quad \quad \text{dist}[u, v, r] \leftarrow \text{dist}[u, r, r - 1] + \text{dist}[r, v, r - 1]
\end{align*}
\]

- We can clean this algorithm up a bit. We 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.

- We also don't need to keep track of the specific vertex numbers as long as we loop through all the vertices.

This version still runs in $O(V^3)$ time.

- This is the cleaned up version of that dynamic programming algorithm. It's usually referred to as Floyd-Warshall. A formal proof of correctness involves a similar induction proof to the one we used for Bellman-Ford. Namely, we can prove by induction that after the rth iteration of the outer loop, $dist(u, v)$ is at most $dist(u, v, r)$. I'll spare you the details.

```
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 $dist[u, v] > dist[u, r] + dist[r, v]$
                    $dist[u, v] ← dist[u, r] + dist[r, v]$
```