Main topics are `single_source_shortest_paths` and `all-pairs_shortest_paths`.

**Prelude**

- Homework 4 will be released soon and is due Tuesday April 9.

**Single Source Shortest Paths**

- Let’s finish shortest paths today. You’re given a directed graph $G = (V, E, w)$ where $w : E \to \mathbb{R}$ is a weight function that may be negative in places. We want to compute the shortest path tree from $s$ and the distances from $s$ to every other vertex.
- $\text{dist}(v)$ is the length of a tentative shortest $s$ to $v$ path, or infinity if we haven’t found one yet.
- $\text{pred}(v)$ is the predecessor of $v$ in the tentative shortest $s$ to $v$ path, or Null if we haven’t found one yet.

```
INITSSP(s):
    \text{dist}(s) \leftarrow 0
    \text{pred}(s) \leftarrow \text{Null}
    \text{for all vertices } v \neq s
    \quad \text{dist}(v) \leftarrow \infty
    \quad \text{pred}(v) \leftarrow \text{Null}
```

- Call an edge $u \rightarrow v$ tense if $\text{dist}(u) + w(u \rightarrow v) < \text{dist}(v)$.

```
RELAX(u\rightarrow v):
    \text{dist}(v) \leftarrow \text{dist}(u) + w(u \rightarrow v)
    \text{pred}(v) \leftarrow u
```

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

```
FordSSP(s):
    INITSSP(s)
    while there is at least one tense edge
        RELAX any tense edge
```

- The order you relax edges determines which algorithm you’re using. We’ll finish up Dijkstra’s algorithm and then do another. We’ll only worry about proving we have the correct distances today.
- Lemma: If $\text{dist}(v) \neq \infty$, then it is the length of some walk from $s$ to $v$.
- In particular, $\text{dist}(v)$ is always at least the shortest path distance from $s$ to $v$.

**No (or few) Negative Edges: Dijkstra’s Algorithm**
Lemma: For all \( i < j \), we have \( d_i \leq d_j \). (Vertices are extracted in non-decreasing order of distance.)

Lemma: Each vertex is extracted from the priority queue at most once.

Lemma: When Dijkstra ends, \( \text{dist}(v) \) is the length of the shortest path from \( s \) to \( v \) for every vertex \( v \).

For any vertex \( v \), consider some shortest path \( v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_{\ell} \) where \( v_0 = s \) and \( v_{\ell} = v \). Let \( L_j \) be the length of the subpath \( v_0 \rightarrow \ldots \rightarrow v_j \). We'll prove by induction on \( j \) that \( \text{dist}(v_j) \leq L_j \).

- \( \text{dist}(v_0) = \text{dist}(s) = 0 = L_0 \).
- Consider \( j > 0 \). By induction, we set \( \text{dist}(v_{j-1}) \) and at some point Extract \( v_{j-1} \) from the queue. At that moment either \( \text{dist}(v_j) \leq \text{dist}(v_{j-1}) + w(v_{j-1} + v_j) \) already or we set \( \text{dist}(v_j) = \text{dist}(v_{j-1}) + w(v_{j-1} + v_j) \) by the end of the iteration. Either way
  - \( \text{dist}(v_j) \leq \text{dist}(v_{j-1}) + w(v_{j-1} + v_j) \leq L_{j-1} + w(v_{j-1} + v_j) = L_j \).
- In particular, \( \text{dist}(v) \leq L_{\ell} = \) the length of the whole path.
- Again, \( \text{dist}(v) \) is at least the shortest path distance and therefore equal to it.

[pretty sure I did this already] 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, even though it may extract some vertices multiple times.

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.

This algorithm is somehow more straightforward to analyze, at least in hindsight.

Let \( \text{dist}_{\leq i}(v) \) denote the length of the shortest walk in \( G \) from \( s \) to \( v \) with at most \( i \) edges. So \( \text{dist}_{\leq 0}(s) = 0 \) and \( \text{dist}_{\leq 0}(v) = \infty \) for all \( v \neq s \).

Lemma: For every vertex \( v \) and non-negative integer \( i \), after \( i \) iterations we have \( \text{dist}(v) \leq \text{dist}_{\leq i}(v) \).

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 \( \text{dist}_{\leq i}(v) \).
- If \( W \) has no edges, it goes from \( s \) to \( s \), meaning \( v = s \) and \( \text{dist}_{\leq i}(v) = 0 \). \( \text{dist}(s) \leftarrow 0 \) in \( \text{InitSSSP} \) and \( \text{dist}(s) \) never increases, so \( \text{dist}(s) \leq 0 \).
- Otherwise, let \( u \rightarrow v \) be the last edge of \( W \). After \( i - 1 \) iterations, \( \text{dist}(u) \leq \text{dist}_{\leq i-1}(u) \).
- In the \( i \)th iteration, we consider edge \( u \rightarrow v \). Either \( \text{dist}(v) \leq \text{dist}(u) + w(u \rightarrow v) \) already or we set \( \text{dist}(v) \leftarrow \text{dist}(u) + w(u \rightarrow v) \). Either way, \( \text{dist}(v) \leq \text{dist}_{\leq i-1}(u) + w(u \rightarrow v) = \text{dist}_{\leq i}(v) \). Again, \( \text{dist}(v) \) 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, \( \text{dist}(v) \) 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 \( \text{dist}(v) \) 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 \( \text{dist} \) 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.
This version runs in $O(VE)$ time even if there are negative cycles.

**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 $\text{dist}(u, v)$, the length of the shortest path from $u$ to $v$ for all $u$ and $v$.
- So, there’s an obvious algorithm for this problem. Compute single source shortest paths from every vertex!

**ObviousAPSP($V, E, w$):**

for every vertex $s$

$\text{dist}[s, \cdot] \leftarrow \text{SSSP}(V, E, w, s)$

- But with Bellman-Ford, that takes $\Theta(V^2 E) = O(V^4)$ time. Dijkstra’s with non-negative edge lengths would take $O(VE + V^2 \log V) = O(V^3)$. 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 $\text{dist}(u, v)$ is the following:

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

- The shortest path ends with some edge $x \to v$, and we need a shortest path to $x$.
- But to compute $\text{dist}(u, x)$, we may need to know $\text{dist}(u, v) + w(x, v)$. 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 $\text{dist}(u, v, \text{ell})$ denote the length of the shortest path from $u$ to $v$ that uses at most $\text{ell}$ edges.
Now we have the following recursive function:

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

- If there are no negative length cycles, then every shortest path uses at most \( V - 1 \) edges, so \( \text{dist}(u, v) = \text{dist}(u, v, V - 1) \).
- Each edge \( x \rightarrow v \) is considered once for computing each of \( V \times V - 1 \) different values \( \text{dist}(u, v, \ell) \) where \( \ell \leq V - 1 \), so it would take \( \Theta(V^2 E) = O(V^4) \) time to fill a table based on this recurrence.
- But if you look at the recursive definition, the variable \( u \) doesn’t change. We’ll 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 just Bellman-Ford again. Can we do better?