Main topics for this lecture include single-source shortest paths and breadth-first search.

No (or few) Negative Edges: Dijkstra’s Algorithm

- Let’s continue talking about single source shortest paths today.
- Recall, the idea for all the algorithms 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.
- We start by calling InitSSSP(s).

\[
\text{InitSSSP}(s):
\]

- \( \text{dist}(s) \leftarrow 0 \)
- \( \text{pred}(s) \leftarrow \text{NULL} \)
- for all vertices \( v \neq s \)
  - \( \text{dist}(v) \leftarrow \infty \)
  - \( \text{pred}(v) \leftarrow \text{NULL} \)

- Call an edge \( u \rightarrow v \) tense if \( \text{dist}(u) + w(u \rightarrow v) < \text{dist}(v) \).
- We want to relax tense edges to represent our newly found shorter path.

\[
\text{Relax}(u \rightarrow v):
\]

- \( \text{dist}(v) \leftarrow \text{dist}(u) + w(u \rightarrow v) \)
- \( \text{pred}(v) \leftarrow u \)

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

\[
\text{FordSSSP}(s):
\]

- \( \text{InitSSSP}(s) \)
- while there is at least one tense edge
  - \( \text{Relax} \) any tense edge

- The first algorithm we’ll talk about today was independently discovered by 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 \( \text{dist}(v) \). We repeatedly extract the minimum distance vertex and relax all outgoing edges. When \( \text{dist}(w) \) changes for some vertex \( w \), we’ll insert it in the priority queue or decrease its key so that we can later relax any of its outgoing edges that may have become tense.
Now, this is an instance of Ford’s general strategy, so I claim (without full proof) that 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. Just like a breadth-first search.

Let’s analyze this algorithm and prove correctness assuming no negative weight edges. Let $u_i$ be the vertex returned by the $i$th ExtractMin call (so $u_1 = s$) and $d_i$ be $\text{dist}(u_i)$ 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 $d_i \leq d_j$. (Vertices are extracted in non-decreasing order of distance.)

- Fix some $i$. We’ll show $d_{i+1} \geq d_i$, meaning the distances are non-decreasing.
- If $u_i \rightarrow u_{i+1}$ is relaxed during the $i$th iteration, then afterward $d_{i+1} = \text{dist}(u_{i+1})$
= dist(u_i) + w(u_i \rightarrow u_{i+1}) \geq 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) \leq dist(u_{i+1}) = d_{i+1}.

- Lemma: Each vertex is extracted from the priority queue at most once.
  - We pull vertices out in non-decreasing order of distance, but distance label dist(v) never increases. Once we pull v from the queue once, we’ll never get an opportunity to pull it again.

- Lemma: When Dijkstra ends, dist(v) is the length of the shortest path from s to v for every vertex v.
  - This proof is almost identical to the one for breadth-first search:
    - For any vertex v, consider some 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 dist(v_j) \leq L_j.
    - dist(v_0) = dist(s) = 0 = L_0.
    - Consider j > 0. By induction, at some point Extract v_{j-1} from the queue. At that moment either dist(v_j) \leq dist(v_{j-1}) + w(v_{j-1} \rightarrow v_j) already or we set dist(v_j) = dist(v_{j-1}) + w(v_{j-1} \rightarrow v_j) by the end of the iteration. Either way
      - dist(v_j) \leq dist(v_{j-1}) + w(v_{j-1} \rightarrow v_j) \leq L_{j-1} + w(v_{j-1} \rightarrow v_j) = L_j.
    - In particular, dist(v) \leq L_ell = the length of the whole path.
    - Again, dist(v) is at least the shortest path distance since it is the length of some walk. It's also at most the shortest path distance, so it's equal.
  - And again, the pred pointers are set correctly.
  - 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.
It’s completely mystifying that this algorithm can be efficient. After all, we’re not doing
But it turns out the analysis is actually more straightforward than either breadth-first search or Dijkstra’s algorithm.

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 \( \text{dist}_0(s) = 0 \) and \( \text{dist}_0(v) = \infty \) 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 \( \text{dist}_\leq 0(v) = 0 \). \( \text{dist}(s) \leftarrow 0 \) 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 \( \text{dist}(v) \leq \text{dist}(u) + \text{w}(u \rightarrow v) \) already or we set \( \text{dist}(v) \leftarrow \text{dist}(u) + \text{w}(u \rightarrow v) \). Either way, \( \text{dist}(v) \leq \text{dist}_\leq (i-1)(u) + \text{w}(u \rightarrow v) = \text{dist}_\leq i(v) \). 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. As I said last week, there will always be a tense edge in this case, even after \( V - 1 \) iterations. We can modify the algorithm slightly to detect negative cycles.

\[
\text{BellmanFord}(s)
\]
\[
\text{InitSSSP}(s)
\]
\[
\text{repeat} \quad V - 1 \ \text{times}
\]
\[
\text{for every edge} \quad u \rightarrow v
\]
\[
\text{if} \ u \rightarrow v \ \text{is tense}
\]
\[
\text{RELAX}(u \rightarrow v)
\]
\[
\text{for every edge} \quad u \rightarrow v
\]
\[
\text{if} \ u \rightarrow v \ \text{is tense}
\]
\[
\text{return} \quad \text{“Negative cycle!”}
\]

This version runs in \( O(VE) \) time even if there are negative cycles.