SSSP Special Case 1: No cycles

One important special case of Single-Source Shortest Paths algorithms is the acyclic case. If we have a DAG $G$, then we can choose a good ordering, such that we only have to relax each edge once.

Choosing an ordering We would like to pick an ordering such that every time we relax an edge from vertex $u$ to $v$, $u.d$ is at its minimum (i.e. is equal to $\delta(s,u)$). DAGs provide us a special case in which we can be sure that we’ve dealt with all incoming edges to a node $u$ before relaxing edges outwards from it. To do this, we make use of a topological sort, which gives us an ordering of nodes for which we’re guaranteed to have all edges oriented from earlier nodes to later nodes. Given this ordering, we iterate over nodes in order starting with $s$, and relax all outgoing edges from each successive node.

Correctness We can sketch out a proof by induction for why this is guaranteed to correctly determine shortest paths. Our inductive hypothesis will be that on step $i$, the first $i$ nodes in the topological sort starting with $s$ are fully relaxed (i.e. $\text{TOPO}[j].d = \delta(s,j)$, for all $j$ between the $\text{INDEX}(s)$ and $\text{INDEX}(s) + i$). Node $s$ is started off with distance 0, and because the graph is acyclic this can never be reduced, so the base case is true. We next assume that our hypothesis is true through step $n$ and show that this implies our hypothesis on step $n + 1$. Because all edges go from earlier to later in the topological sort, all paths to node $n + 1$ from $s$ consist of some path from $s$ to a node before $n + 1$ plus an edge from this node to node $n + 1$. From our hypothesis all nodes before $n + 1$ are already labelled with their optimal values. Once we start step $n + 1$, all edges out of these nodes have also been relaxed (note that these edges were all relaxed after the node they originate from reached its optimal value). Thus, all optimal paths to $n + 1$ have been considered, and the minimum has been chosen. At step $n + 1$, then, node $n + 1$ reaches its optimal value. Thus, by induction our algorithm iteratively reaches the optimal distance for each node in the topological sort.

Runtime The runtime of the algorithm consists of the runtime of the topological sort, $O(V + E)$, plus the relaxation of every edge exactly once, $O(E)$, giving us $O(V + E)$ in total.

SSSP Special Case 2: No negative weight edges

Another commonly seen special case of Single-Source Shortest Path algorithms is the case of a graph with all non-negative edge weights. The advantage of this graph is that all paths have non-increasing total weights.

The algorithm To take advantage of this special case graph, we perform a modified BFS. Rather than using a Queue in the implementation, we choose to use a Priority Queue instead, such that the
path we choose to expand on each iteration is the least weight path seen so far. All other aspects of the algorithm are the same as BFS.

**Correctness and Runtime** These concepts will be explored in more detail in the upcoming lecture. An intuitive rationale for the correctness is that the nodes expanded are guaranteed to be at their optimal values because all other paths on the queue have equal or higher length paths (and are non-decreasing).

**SSSP Special Case 3: Bidirectional Search (Highly branched graphs)**

Suppose we have a graph $G$ in which the number of vertices distance $d$ away from any given vertex is roughly $b^d$ (in other words, our graph has a branching factor of $b$). An example of a graph like this which we’ve seen in class is the Rubik’s cube state graph. If we have $n$ total vertices in this graph, then we can roughly estimate the diameter of the graph to be $\log_b n$ if we assume that the last layer outwards dominates the number of nodes.

**The algorithm** Bidirectional search is an optimization on Dijkstra’s algorithm which attempts to improve performance in this highly branching situation, if we’re looking for a single path from $s$ to $t$. The idea is simple: run a forward Dijkstra’s search from $s$ in parallel with a reverse Dijkstra’s search from $t$ and stop once the two searches meet (see below for more detail on the stopping condition).

**Stopping condition** Though it may be tempting to assume that can stop as soon as the two searches connect at an edge $(p, q)$ and simply add the weights of $p.s$ (the distance from $s$ to $p$), $q.t$ (the distance from $t$ to $q$), and $w(p, q)$, this would result in incorrect paths in cases where $w(p, q)$ is very large. Instead, we do the following check every time we find an edge between the forward exploration and reverse exploration:

1. Update a tracking variable, $\mu$, to be $\min(\mu, p.s + w(p, q) + q.t)$.
2. Find the minimum, $l_s$ of $v.s$ over every unexpanded node $v$ in the forward queue
3. Find the minimum, $l_t$ of $u.t$ over every unexpanded node $u$ in the reverse queue
4. If $l_s + l_t \geq \mu$, then all remaining paths on the queue cannot possibly reduce the shortest path $\mu$ any further, so stop.

**Correctness** We can prove the correctness of the algorithm using contradiction. Suppose there is some path $p$ from $s$ to $t$ such that the weight of the path is smaller than $\mu$. Pick an edge $(v, w)$ such that $\delta(s, v) < l_s$ and $\delta(w, t) < l_t$. If this is the case, both $v$ and $w$ must have been processed with $v$ in the forward search and $w$ in the reverse search. When the second one of $v$ and $w$ was scanned, $\mu$ would have been set to $\delta(s, v) + w(v, w) + \delta(w, t) = w(p) < \mu$. This is a contradiction.