Review: General Structure of Shortest Paths

- \( v.d \) - The weight of the current shortest path from \( s \) to \( v \).
- \( v.\pi \) - The parent vertex of \( v \) in the current shortest path.
- \( w(u, v) \) is the weight of the edge from vertex \( u \) to vertex \( v \)
- \( \delta(u, v) \) is the weight of the shortest path from vertex \( u \) to vertex \( v \)

Initialize: for \( v \in V \):
- \( v.d \leftarrow \infty \)
- \( v.\pi \leftarrow \text{NIL} \)
- \( s.d \leftarrow 0 \)

Main:
- repeat
  - select edge \((u, v)\) [somehow]
  - “Relax” edge \((u, v)\)
    - if \( v.d > u.d + w(u, v) \):
      - \( v.d \leftarrow u.d + w(u, v) \)
      - \( v.\pi \leftarrow u \)
  - until all edges have \( v.d \leq u.d + w(u, v) \)

Shortest Paths in DAGs

This is an example of a weighted directed graph. Since there are no cycles, we do not have to worry about negative weight cycles. We know that there will be no path from a vertex \( u \) to another vertex \( v \) occurring before \( u \) in the topological order. The first step in finding the shortest path, therefore, should be to topologically sort the graph, in \( O(V + E) \) time.

Now we go through each vertex starting from the beginning, and relax each outgoing edge. This will guarantee that each edge only need be relaxed once, making the total runtime \( O(V + E) \).

Example to go over in class if need be on top of next page (if need be).

Dijkstra’s Algorithm

The most used shortest paths algorithm for weighted directed graphs. It is sneakily the most intuitive, even though its implementation is more complicated than that of DAG shortest paths and Bellman-Ford. Imagine we drop a huge colony of ants onto the source vertex \( s \) and each goes along a single possible edge from \( s \). Whenever an ant arrives at a vertex, it splits into several different ants each following a different edge going outwards.

Each ant walks at the same speed, so when an ant reaches a vertex for the first time, we know that it will have followed the shortest path to the vertex. We then mark a vertex the first time an ant
arrives with the distance the ant travels, $\delta(s, u)$. Any other ant arriving at the same location can be obliterated, because we only really care about the first ant that arrives. This ensures that we do not check worthless paths.

To turn this into an actual algorithm, we maintain a frontier of visited vertices. Each of these vertices in the frontier will have the invariant that the have been visited by ants (their shortest path has been determined from $s$). We then pick the closest vertex to $s$ and relax its edges, thereby expanding the frontier. From here, we get our familiar algorithm:

Dijkstra $(G, W, s)$  //uses priority queue $Q$
  Initialize $(G, s)$
  $S \leftarrow \emptyset$
  $Q \leftarrow V[G]$  //Insert into $Q$
  while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)  //deletes $u$ from $Q$
    $S = S \cup \{u\}$
    for each vertex $v \in \text{Adj}[u]$
      do RELAX $(u, v, w)$  ← this is an implicit DECREASE_KEY operation

**Priority Queues**

An important part of Dijkstra’s algorithm is the use of the priority queue, a data structure which allows us to extract the current minimum element very quickly, and also to decrease the value of any element in the data structure. This sounds awfully similar to heaps, which we learned about in this class. One of the biggest uses of heaps is actually for Dijkstra’s algorithm! (And other very similar graph algorithms).

We recall that the most common implementation of the heap is using an array, which gives us the following complexity:

Binary min-heap:
\[ \Theta(lg V) \text{ for extract min} \]
\[ \Theta(lg V) \text{ for decrease key} \]

In Dijkstra, we call extract-min once for every vertex in the graph, and for every vertex we relax its outgoing edges once, such that the total number of relaxations we do is \( O(E) \). The runtime for Dijkstra using a binary heap would then be \( O(V \log V + E \log V) \).

How could we speed this up? Well turns out, binary heaps are not the most efficient heaps. Fibonacci heaps (covered in 6.854) operate in \( \Theta(lg V) \) for extract min and \( \Theta(1) \) for decrease key, in amortized time. The runtime of Dijkstra will then be \( O(V \log V + E) \). We know this is an optimal implementation of a priority queue because of the \( O(n \log n) \) sorting bound.

**Comparison of Sorting Algorithm Runtimes**

- **Unweighted Graph** - BFS: \( O(V + E) \) worst case
- **DAG** - DAG Shortest Paths: \( O(V + E) \) worst case
- **Weighted Graph (Non-negative weights)** - Dijkstra’s Algorithm: \( O(V \log V + E \log V) \) worst case. \( O(V \log V + E) \) amortized.
- **General Graph** - Bellman-Ford: To be continued...