• DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

• Main topics:
  – Speedups using priority queue.
  – Dijkstra’s Algorithm and its relation to BFS

• This week in a nutshell:
  – Cut property: edge in MST iff its minimum on some cut.
  – Kruskal’s algorithm: sort by weights, greedy.
  – Prim’s algorithm: start with single vertex, greedily add minimum edge in cut between reached and unreached vertices.
  – Running time: straight implementation $O(nm)$ for Kruskal, $O(n^2)$ for Prim.
  – Union find: merge smaller into bigger, $O(\log n)$ per union, $O(m \log n)$ Kruskal.
  – Priority queue: Prim’s algorithm in $O(m \log n)$ as well.
  – Can also use priority queue to speed up shortest path with positive edge weights, $O(m \log n)$: Dijkstra’s algorithm.

• Comments from Last Time:
  – Where do the runtimes come from?

We turn our attention back to shortest paths. We still work with directed graphs, but only consider positive edge lengths now. Note that this includes the unit length case, specifically our algorithm is equivalent to breadth-first-search (BFS) in this case.

Recall that shortest path algorithms rely on updating a set of distance estimates $d[u]$. The core step is:

$$d[v] \leftarrow \min\{d[v], d[u] + l_{u \rightarrow v}\},$$

and the Bellman-Ford algorithm essentially repeatedly performs this across all edges $u \rightarrow v$ $O(n)$ times.

The fact that the lengths are positive implies the following:

**Lemma 0.1.** A vertex $u$ can only update vertices $v$ with $d[v] > d[u]$. 
This motivates us to come up with Dijkstra’s algorithm, which is:

1. Initialize \( d[s] = 0, d[u] = \infty \) for all \( u \neq s \).
2. Mark all vertices \( u \) as unupdated.
3. While there are unupdated vertices with \( d[u] < \infty \) remaining.
   (a) Find \( u \) with minimum \( d[u] \) among unupdated vertices.
   (b) Mark \( u \) as updated.
   (c) Update all neighbors \( u \rightarrow v \).

The correctness of this algorithm comes from the observation that \( d[u] \) of the vertices that we pick to update is increasing: each \( d[u] \) can only update the \( d[v] \) of some vertex to a value that’s \( d[u] + l_{u\rightarrow v} \geq d[u] \). So we update \( u \), subsequent steps cannot improve its value, and its \( d[u] \) value is final.

The algorithm as stated runs in \( O(n^2) \) time: \( O(n) \) outer iterations, \( O(n) \) to find \( u \), and \( O(n) \) to perform the updates. It can be sped up to \( O(m \log n) \) time by putting all the vertices into a priority queue, which is a data structure that supports:

1. insert / delete / update an element,
2. find the element with minimum key,

Note that we only update each vertex at most once, so each \( v \) has its value changed by \( u \rightarrow v \) at most once. So the total number of updates is \( \sum_u |N(u)| = O(m) \).

There are many ways of implementing a priority queue so all operations are \( O(\log n) \) time (findmin actually takes \( O(1) \) in our implementations). You are not required to know how it operates, just the interface of insert/delete/query. A simple way to reason about a priority queue is to view insert/delete as changing keys. Then the keys are partitioned in a divide and conquer manner: we partition the data evenly, and store the minimum of each half and repeat. Note that the minimum of a set can be recomputed from the minimum of its two subsets in \( O(1) \) time, assuming that those values are correct.

Every time we update, we update the minimum of all the subsets containing it in a bottom up manner, maintain correctness of the value inductively. As the sizes of the sets are geometrically increasing, each node can belong to at most \( O(\log n) \) of them, giving the \( O(\log n) \) cost per update.

Finally, notice that on a unit weight graph, the algorithm is exactly Dijkstra’s algorithm: the unvisited vertices have distances either \( d[u] \), \( d[u] + 1 \), or \( \infty \). At each step, the only update that we can make is turning \( \infty \)s into \( d[u] + 1 \)s, so a queue suffices for maintaining which vertex to update next.