Last time, we discussed all-pairs shortest path algorithms, and developed an algorithm as well as analyzed its complexity.

Further we considered the Floyd-Warshall algorithm which organized complexities to be efficient.

Uses the Bellman principle to find shortest paths using known shortest paths, building on them.

Consider graph $G$ with vertices $V = \{1, \ldots, n\}$.

Define a sub-function $\text{ShortPath}(i, j, k)$ that returns the shortest path from $i$ to $j$ using only vertices only from the set $\{1, 2, \ldots, k\}$ as intermediate points.

Given this function, the goal is to find the shortest path from each $i \neq j$ using only vertices in $\{1, 2, \ldots, k\}$. For each pair, the true shortest path could be either:

1. path that only uses vertices in $\{1, \ldots, k\}$
2. the path that goes from $i$ to $k$, then back to $j$.
   (Which is concatenation of known shortest paths).

Algorithm: compute $k = k_1, k_2, \ldots$ until done. Found it is $O(n^3)$.

Now what if we don't care about all-pairs, but only about distance from $s$ to all other vertices.

**Breadth-first search algorithm**
Basic idea of BFS is we initially know

s has distance 0 from itself, all other distances are unknown.

next find all neighbors of s, which by definition have distance 1.
next find all neighbors of those vertices, but excluding those already visited.

Basic fact is: every vertex whose shortest distance from s is d has
neighbour whose shortest distance from s is d-1.

So again we are using a dynamic programming-like principle.

Suppose we already know the distance to every vertex on the subgraph that is at steps or less
from s. Neighbors of edge nodes are at most d+1 steps from s, but could be
less, if there is another shorter path through neighbour.

But we already know whether there is a shorter path to any particular
vertex, since by hypothesis we know the distance to every vertex d steps or less.

How to implement?

Simple:
1. Create array of a elements to store distance from s to each vertex.
   Initialize at 00 for s, and e.g. NaN for others.

2. Create variable d to keep track of when we are in BFS, d=0.

3. Find all vertices that are distance d from s, by going through distance
   array, one by one.

4. Find all neighbors of these vertices and check each one to see
   if its distance from s is NaN.

5. if number of neighbors with unknown distance is zero, stop, otherwise set distance
   to d+1

6. d = d+1.
Complexity analysis:

- $O(n)$ to set up array.
- $O(n)$ for each iteration, set $r$ iterations so $O(rn)$.

Within iteration, need to investigate vertex or distance $d$, to check if neighbor has unknown vertex, so $O(m)$.

So total time including setup is $O(n + rn + m)$.

What is $r$? The maximum distance from source vertex to any other vertex, which in worst-case is diameter where worst case is $n$.

So $O(n + rn + m)$

Is there better implementation?

The two run-time parts of basic implementation is going through list of distance to first vertices that are distance from $s$.

This is unsatisfactory since few vertices are distance $d$.

What if we store list of vertices at distance $d$? Maintain hash table.

$\Rightarrow$ Use FIFO queue.

Algorithm:

1. Place label of source $s$ in first element of queue, set read pointer to point 1, set write pointer to point 2, second element, first empty in distance array, set distance $b(s) = 0$, allocate $s - 5 = N\cdot N$.

2. If read/write pointers point to same element of queue, the done, otherwise, read write label from element pointed to by read pointer and decrease that pointer by one.

3. First distance $d$ for that vertex by looking in distance array.

4. Go through each neighboring vertex in turn and look up its distance in the distance array also. If unknown, assign distance $d + 1$, store label in queue, at element pointed to by write pointer, increment with pointer repeat from step 2.
Finding Shortest Paths

algorithm finds for only finds distances, not actual paths

so, construct shortest path tree.

If we have weights/lengths on edges

use Dijkstra's algorithm.