Last 2 classes:

DFS: useful for connectivity
- connected components of undirected graphs
- topologically sorting a DAG (order by TV post #)
- SCCs of general directed graph
  (take reverse of G, vertex with highest post # in source SCC of GR
  = sink SCC of G)

Today: BFS - useful for distances.

breadth first search.

Use queue instead of a stack/recursion

BFS

DFS

enqueue
dequeue

Queue: [ ] ... [ ]

implement using a linked list with pointers to front of list
FIFO = first-in first-out & back of list
**BFS(G, s)**: need to specify start vertex s.

**Input**: Directed or undirected $G = (V, E)$ in adjacency list representation and $s \in V$.

**Output**: for all $w \in V$, $\text{dist}(w) = \min \# \text{ of edges to go from } s \text{ to } w$.

For all $w \in V$, $\text{dist}(w) = \infty$

$\text{dist}(s) = 0$

$Q = \{ s \}$ (create a queue containing $s$)

While $Q \neq \emptyset$:

$w = \text{dequeue}(Q)$

For all $(w, z) \in E$:

If $\text{dist}(z) = \infty$

Then $\text{enqueue}(Q, z)$

$\text{dist}(z) = \text{dist}(w) + 1$

**Running time**: $O(n+m)$

$n = |V|$, $m = |E|$
Generalize BFS to allow positive lengths on edges.

Let \( l(e) \) = length of edge \( e \).
Assume \( l(e) > 0 \).

For a path \( P = w_0 \rightarrow w_1 \rightarrow w_2 \rightarrow \cdots \rightarrow w_k \)

define its length as \( l(P) = \sum_{i=0}^{k-1} l(w_i, w_{i+1}) \).

Let \( \text{dist}(v, w) \) = length of shortest path from \( v \) to \( w \).

Goal: for given \( s \in V \), find for all \( w \in V \), \( \text{dist}(s, w) \).

BFS solves it when \( l(e) = 1 \) for all \( e \in E \).

First, suppose every \( l(e) \) is a positive integer.
Then can replace edge \( e \) of length \( l(e) \)
by a path of length \( l(e) \)
& give each new edge length 1.
Run BFS on the new graph to get shortest path lengths in the original graph.

Problem: running time depends on lengths \( l(e) \) (since graph size depends on \( l(e) \) & these may be HUGE.

-Much of the time when running BFS on this new graph we're processing "dummy" vertices that we added.

-Can we only consider times when we process vertices in the original graph?

Solution: set "alarm clocks" for times of interest.
Example:

When exploring, see A, B, F, then E to D.

Jump to time 4: see A
Jump to time 10: see B
Jump to time 12: see C.

Idea: each vertex has an alarm clock. When we explore a new path to a vertex w, then check if need to adjust w’s clock.
High-level algorithm:

Use \( \text{dist}(w) \) for \( w \) is alarm time.
Set \( \text{dist}(s) = 0 \) & for all \( w \neq s \), \( \text{dist}(w) = \infty \).
\( T = 0 \).

Repeat until no more alarms:
Increase \( T \) to next alarm, say for vertex \( w \).

for every \( (w, z) \in E \):
if \( \text{dist}(z) > \text{dist}(s) + l(w, z) \)
then \( \text{dist}(z) = \text{dist}(s) + l(w, z) \).

How to maintain alarms?

Use min-heap data structure (priority queue)

It maintains a set of elements (corresponding to vertices of the graph)
each element has a key (= alarm clock time)
Min-heap data structure:

Basic operations:

- **Insert** \((H, v, \text{dist}(v))\)
  - add element \(v\) to \(H\) with key \(\text{dist}(v)\)

- **DecreaseKey** \((H, v, \text{dist}(v))\)
  - for \(v\) in \(H\), decrease its key to \(\text{dist}(v)\)

- **DeleteMin** \((H)\)
  - return the element in \(H\) with smallest key & delete it from \(H\).

If \(\leq n\) elements in \(H\),
then \(O(\log n)\) time per operation.
Dijkstra(G,s):

\[ \text{Input: } G = (V,E) \text{ with } l(e) > 0 \text{ for } e \in E, s \in V. \]

\[ \text{Output: for all } w \in V, \text{ dist}(w) = \text{ length of shortest } s \rightarrow w \text{ path } \]

\[ \text{Prev}(w) = \text{ parent of } w \text{ on this path.} \]

\[ \text{for all } w \in V, \begin{cases} \text{dist}(w) = \infty \\ \text{Prev}(w) = \text{ NULL} \end{cases} \]

\[ \text{dist}(s) = 0 \]

\[ H = \emptyset \]

\[ \text{for all } w \in V, \text{ Insert}(H, w, \text{dist}(w)) \]

\[ \text{While } H \neq \emptyset: \]

\[ w = \text{ delete-min}(H) \]

\[ \text{for all } (w,z) \in E: \]

\[ \text{if } \text{dist}(z) > \text{dist}(w) + l(w,z) \]

\[ \text{then } \begin{cases} \text{dist}(z) = \text{dist}(w) + l(w,z) \\ \text{Prev}(z) = w \\ \text{Decrease-key}(H, z, \text{dist}(z)) \end{cases} \]
Running time:

$N = |V|$, $m = |E|$

$N$ inserts & deletes (one per vertex)

$\leq m$ decrease keys.

$O(\log n)$ time per operation.

$\Rightarrow O((n+m) \log n)$ time.

assuming $G$ is connected
then $m \geq n-1$ and it's $O(m \log n)$ time.
How to implement min-heaps?

Complete binary tree so \( p(i) = \lfloor \frac{i}{2} \rfloor \)

Min-heap property: \( A[\lfloor \frac{i}{2} \rfloor] \leq A[i] \)

Then min key is at the root easy to find but then need to restructure after deleting.

Insert: add at end & then bubble-up (swapping with parent if strictly smaller)

\( \leq 1 \) leaf-root path so \( O(\log n) \) time.

Decrease-key: reduce key & then bubble-up.

Delete-Min: remove root, take last element to root & sift-down (change current node with smallest of children if bigger & then repeat from the new position).