Last 2 classes:

DFS: useful for connectivity

- connected components of undirected graphs
- topologically sorting a DAG
  (order by ↓ post #)
- SCCs of general directed graph
  (take reverse of G, vertex with highest post # in source SCC of G^R = sink SCC of G)

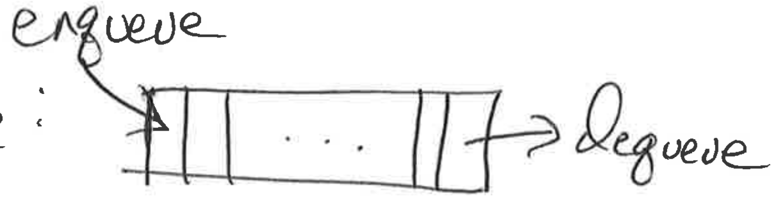
Today: BFS – useful for distances.

BFS = breadth first search.

Use queue instead of a stack/recursion

queue = BFS, stack/recursion = DFS.

Queue:



implement using a linked list with pointers to front of list & back of list.

FIFO = first-in first-out.

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 $w \in V$, $dist(w)$ = min # of edges to go from s to w.

for all $w \in V$, $dist(w) = \infty$
$dist(s) = 0$
$Q = \{s\}$ (create a queue containing s)
While $Q \neq \emptyset$:
    $w = dequeue(Q)$
    for all $(w,z) \in E$:
        if $dist(z) = \infty$
            then [$enqueue(Q,z)$
                   $dist(z) = dist(w) + 1$

Running time: $O(n+m)$

$n = |V|$, $m = |E|$

Generalize BFS to allow positive lengths on edges.

Let $\ell(e)$ = length of edge e.

Assume $\ell(e) > 0$.

For a path $P = w_0 \to w_1 \to w_2 \to \cdots \to w_k$

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

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

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

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

First, suppose every $\ell(e)$ is a positive integer.

Then can replace edge e of length $\ell(e)$

by a path of length $\ell(e)$

& give each new edge length 1.

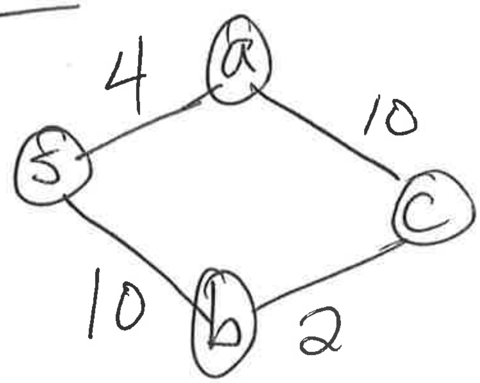
Run BFS on the new graph to get shortest path lengths in the original graph.

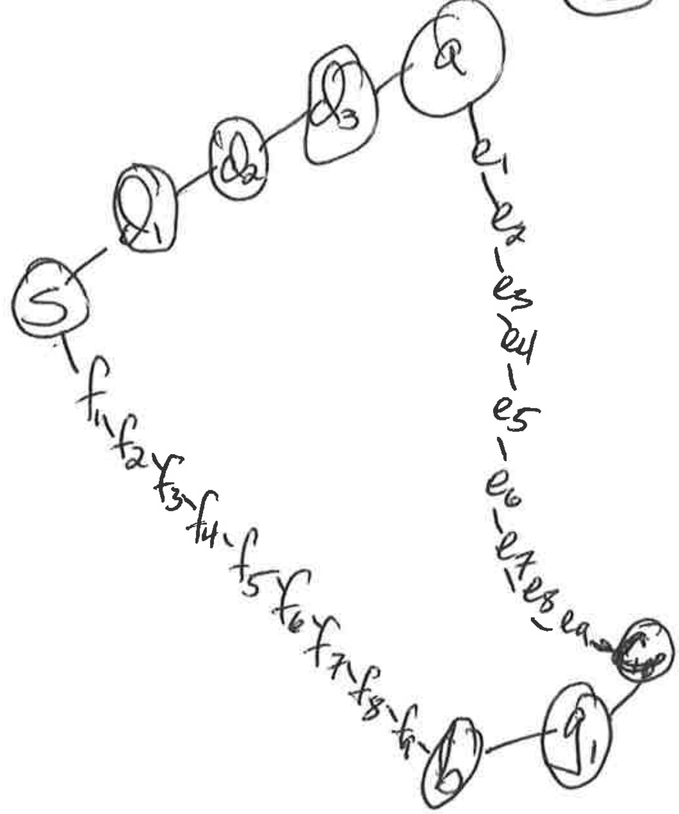
Problem: running time depends on lengths $l(e)$ (since new graph size depends on $l(e)$) & these may be HUGE.

- 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 s,
see d_1 & f_1
then f_2 & d_2
⋮

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 $dist(w)$ for w's alarm time.

Set $dist(s)=0$ & for all $w \neq s$, $dist(w)=\infty$.

$T=0$.

Repeat until no more alarms:

Increase T to next alarm, say for vertex w.

for every $(w,z) \in E$:

if $dist(z) > dist(s) + \ell(w,z)$

then $dist(z) = dist(s) + \ell(w,z)$.

How to maintain alarms?

Use min-heap data structure (priority queue)

H 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$(H, v, dist(v))$
  - add element v to H with key $dist(v)$.
- DecreaseKey$(H, v, dist(v))$
  - for v in H, decrease its key to $dist(v)$
- 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, ℓ, s):

input: $G=(V,E)$ with $\ell(e)>0$ for $e\in E$, & $s\in V$.
output: for all $w\in V$, $dist(w)$ = length of shortest $s \rightsquigarrow w$ path
$prev(w)$ = parent of w on this path.

for all $w\in V$, $\begin{cases} dist(w) = \infty \\ prev(w) = NULL \end{cases}$

$dist(s) = 0$

$H = \emptyset$

for all $w\in V$, Insert$(H, w, dist(w))$

While $H \neq \emptyset$:
  w = deletemin(H)
  for all $(w,z)\in E$:
    if $dist(z) > dist(w) + \ell(w,z)$
    then $\begin{cases} dist(z) = dist(w) + \ell(w,z) \\ prev(z) = w \\ \text{DecreaseKey}(H, z, dist(z)) \end{cases}$

Running time:

$n = |V|$, $m = |E|$.

n inserts & deletes (one each 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: $A[p[i]] \le A[i]$

then min key is at the root

easy to find but then need to restructure after deleting.

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

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

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

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