Week 2 L1

Dijkstra's shortest path algorithm.

Input: directed graph $G_p = (V, E)$

- Each edge has non negative length
- source vertex s

Output: for each $v \in V$, compute

$L(v)$ = length of shortest s-v path

Assumptions:

1. [for convenience], consider strongly connected, $\exists$ s-v path for $\forall v$
2. $l_e \geq 0$, $\forall l_e \in E$ (l_e: edge length (for convenience))

WHY DO WE NEED ANOTHER SHORTEST PATH ALGO? CAN'T WE ALREADY USE BFS IN $O(n)$?

Yes. IF $l_e = 1$ for every edge.

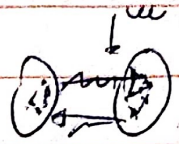
Question: why not replace each edge by multiple edges of length 1.

Stupid question

DIJKSTRA'S ALGORITHM

Initialize:

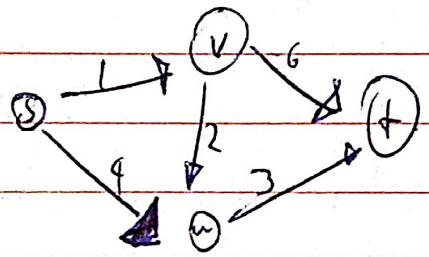
- $X = \{s\}$ #(vertices processed so far)
- $A[s] = 0$ #(computed shortest path distance)
- $B[s] =$ empty path (computed spot) // This is only for show not included in a real implementation



- while $X \neq V$:
  - among all edges $(v,w) \in E$ with $v \in X$, $w \in X$, pick the edge (v^*, w^*) that minimizes the following criterion:
    $A[v] + l_{vw}$ (Dijkstra's greedy criterion)
  - add w^* to X
  - set $A[w^*] = A[v^*] + l_{v^*w^*}$
  - set $B[w^*] = B[v^*] \cup [v^*, w^*]$?

WEEK 2 L2

EXAMPLE



First iteration: all edges leaving to outside X

Dijkstra's criterion: (s,v) has score of $0+1$, (s,w) $= 0+4$

So we go to s,v

Second iteration:

All edges leading to outside of X : (s, w), (v, w), (v, t)

Dijkstra's score for each : (0 + 4), (1 + 2), (1 + 6)

We then add w to X, etc etc (A[w] = 1 + 2 = 3

Third iteration

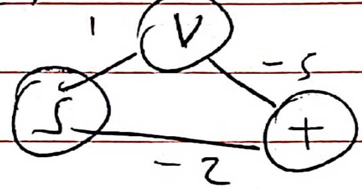
All edges: (v, t), (w, t) greedy (1 + 6) (3 + 3)

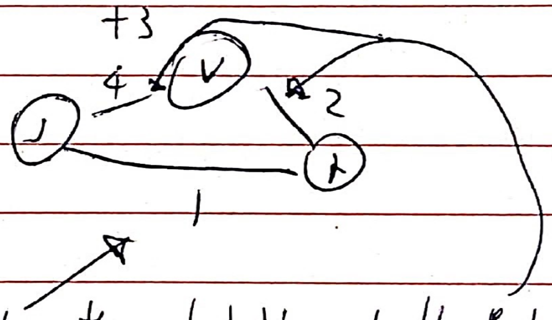
Bring in t,

A(t) = 3 + 3 = 6

~~Breaks~~ The while loop now breaks due to X = V

Can't reduce negative length to non negative by adding a large number to all lengths because:





WEEK 2 C3

WHY IS DIJKSTRA'S ALGO CORRECT

Proof by induction:

Base case: $A[1] = L[1] = 0$ correct ✓

Then on and on. Seems pretty simple.

Induction hypothesis: all previous iterations correct.

→ In current iteration, we set $B[v^*] = B[v] \cup (v^*, w$
computed length

DIJKSTRA AND RUNNING TIME

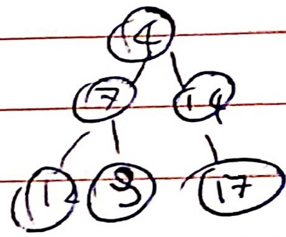
Naively appears to be $O(mn)$ (roughly)

HEAP OPERATIONS

Raison d'être of heap = perform Insert, Extract-min in $O(\log n)$ time

[rest of video assumes familiarity with heaps]

- Conceptually, a perfectly balanced binary tree
- heap property: at every node, key ≤ children's keys



- Extract-min by plucking off the root, bubble down
- Insert via bubbling down

Also: will need ability to delete from middle of heap.

Because you maintain heap as pbbt, height ≈ $\log_2 n$

All operations run in $O(\log_2 n)$ time.

TWO INVARIANTS

elements in heap = vertices of V-X

For $v \notin X$

key(v) = smallest Dijkstra's greedy ... #2

knockout tournament.

two rounds: 1 At first each vertex runs a local round to find smallest edge from X pointing to it, put them on the heap

2 then min (heap) finds best out of these local winners. We'll be handed the min on a silver platter, root of heap

to maintain #2

that $\forall v \notin X$, key[v] = smallest ~~~~ greedy score of all edges (u,v), $u \in X$

- When w is extracted from heap (added to X):

  - for each edge $(w,v) \in E$:

    - ~~insert~~ if $v \in V - X$
      - delete from heap
      - recompute key[v] = min(key[v], A[w] + l_{wv})
      - re insert v into heap

        (maybe it slides down)

Running Time is $O(\log n)$ for each heap ~~~~ operation

- (n-1) ~~~~ extract mins

- each edge (v,w) results in at most one Delete/insert combo if v added to X first.

So # of heap operations is $O(n+m) = O(m)$

[illegible]

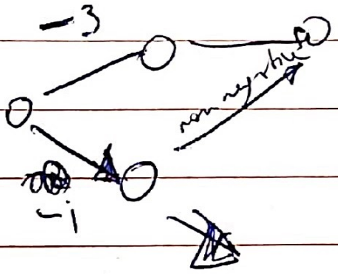
Problem [illegible]

✓

✓

✗

✗



Can't go back, so yeah. Can add positive constant to all ~~paths~~ edges

~~[illegible]~~

Insert is normally log n, but could be n

Exmin $O(1)$

Will always terminate.

with negative shift mixed results

Opposite for priority queue

Loop forever.

Largest item $O(n)$

Median same

7th smallest ✓ $\log(n)$