Midterm

give each node a starting value of ∞ as its distance from root! Give root's value 0!

Zhou, Arthur
405999589

1) Weighted & undirected

a) • Pick an arbitrary node as the root of our shortest path tree

• Create 2 empty sets: one including the FINALIZED vertices and the other being the set containing all nodes within the tree, add the root to the FINALIZED set

• Find all neighbors of our root vertex

  ◦ Choose the one that has the min. distance from the root (min. weighted edge)

  ◦ Add it to the finalized set, then update the distances of all adjacent vertices

    ▪ IF the sum of the distance from root and the min. edge that we added is less than the node's distance value, update the distance

repeat

• Repeat until all vertices have been visited, return the tree, node values, or its distance from source

b) Proof by induction:

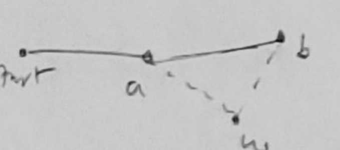
BASE CASE: For n=1 nodes, it's obviously correct as distance is 0

ASSUME: For n=k-1, Dijkstra's returns an optimal path to each node

Prove it works for n=k:

BWOC: Let's say Dijkstra's chose a worse path

CASE: Assume (a,w) and (w,b) is more optimal than (a,b) but we choose (a,b)



(a,w) + (b,w) < (a,b)

↓ since (b,w) must be positive

(a,w) < (a,b)

But by Dijkstra's algorithm we would've selected the path with w since (a,b) is heavier, so it's a CONTRADICTION!

Because we prove the induction step by contradiction, we proved all 3 steps of Induction thus verifying Dijkstra's validity!

c) To implement a heap, begin by inserting the values of all vertices in the minheap. Remember we initialized them as 0 as source and ∞ as all others. When we check the adjacent edges we update the values of all adjacent vertices and we replace those values in the heap with our new values. The minheap will let us extract the min. ~~value~~ node in log v time where v is # of vertices.

We repeat this process for each edge we add to the shortest path tree, so it's

$\boxed{O(e \cdot \log v)}$

2) GOAL: complete in $O(n)$ time!

BB = Blackbox

- Make an empty vector $v = \{\}$
- Int $i=1$
- while $i \leq n$:
  - UNION v and input sequence
  - Take the union and k and put it into the BB
    - IF YES:
      - i++
    - ELSE:
      - if $i==0$, the subset we're looking for doesn't exist!
      - else, push back input_sequence[i-1] into v and i++
- After while loop, input both v and k to the BB to check
  - If YES, we found solution and return v
  - IF NO, add the last element of the input to v
- Return v

Time Complexity: Since our BB iterates thru the input sequence linearly and uses the BB at each increment of i until $i > n$, our complexity is $\boxed{O(n)}$

Proof: Induction

~~BASE CASE: For only one element of input, we can just check that against k~~

~~Assume: it works for $n = k-1$~~

~~Induction step: If we add 1 more element then we simply put $n = k-1$ as the process done within the while loop, our final check in checking if v and k matches will determine if we need the final value, since we proved the while and final check works we return the correct subset!~~

proof not needed apparently

4) undirected & connected

- Perform DFS from an arbitrary node
  - As you traverse down each edge, direct it outwards from the current node if the neighboring vertex hasn't been visited
    - If the neighbor vertex is visited direct the edge towards the current vertex to ensure the in degree = the out degree, then mark the neighbor vertex as UNVISITED so if it has more than 2 edges we can properly calibrate the other ones.

repeat

- Repeat until all edges directed.

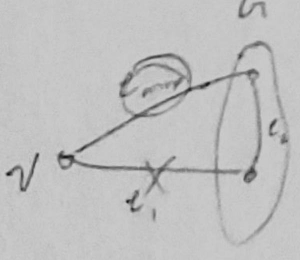
Time Complexity: $\boxed{O(v+e)}$ since we're doing DFS, we iterate through every node and degree which takes constant time to process for both.

3) We know: connected, weighted, and undirected

- If there's only 1 weighted edge added, keep the MST as is but add the edge since it's our only option to get to vertex v.
- If there's more than 1 weighted edge added, perform BFS to identify the cycles made by adding the new edges. We know a cycle is FOR SURE added since the graph is connected
  - For each cycle, identify its e_{max} weighted edge
    - Remove e_{max} from MST T to break the cycle,
    - Replace e_{max} w/ the edge connecting v to the vertex
      - Add this edge to the MST T to maintain its spanning property
- Return the modified MST T

Time Complexity: $\boxed{O(n+e)}$ where e = # of edges and n = # of nodes total since we're making constant operations on the existing MST T and the only complex action we are performing is BFS to identify a cycle.

Proof: BWOC, Imagine that we choose to include e_{max} in our final tree over a lighter weight in the cycle CLAIMING it makes the overall MST lighter



Let's say we chose e_{max} over e_1, now the MST is

$$T' = T \underbrace{- \overset{\text{weight of } e_1}{\{e_1\}} + \overset{\text{weight of } e_{max}}{\{e_{max}\}}}_{\text{we know this is positive since } e_{max} > e_1}$$

Let's say

$e_{max} \in T'$

$e_{max} \notin T$

But this means that T' is heavier since this is positive

Contradicting to our claim at the start!

As a result the MST cannot contain e_{max}.

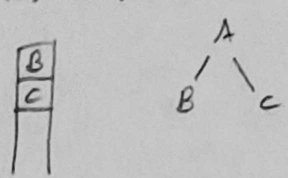
5)

Start: Put A into stack

stack:

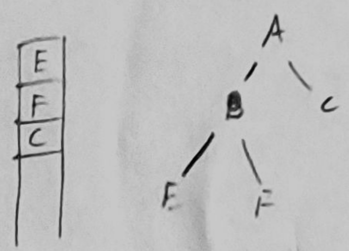
pop A, put A's neighbors in

stack:



pop top, put B's unvisited neighbors in:

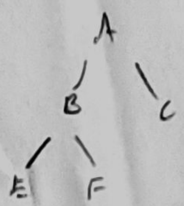
stack:



pop E, put E's neighbors; there are none

pop F, put F's neighbors; there are none

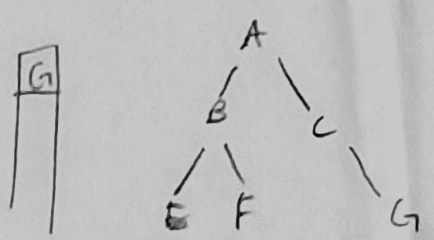
our graph still looks like this:



stack:

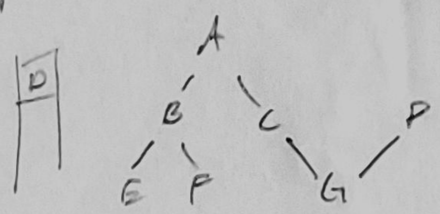
pop C, put C's unvisited neighbors:

stack:



pop G, put G's unvisited neighbors:

stack:



stack is empty, we can end because the graph is connected!