Nafis Abeer

EC 330 HW 8

1) a) Maybe-mst(G):

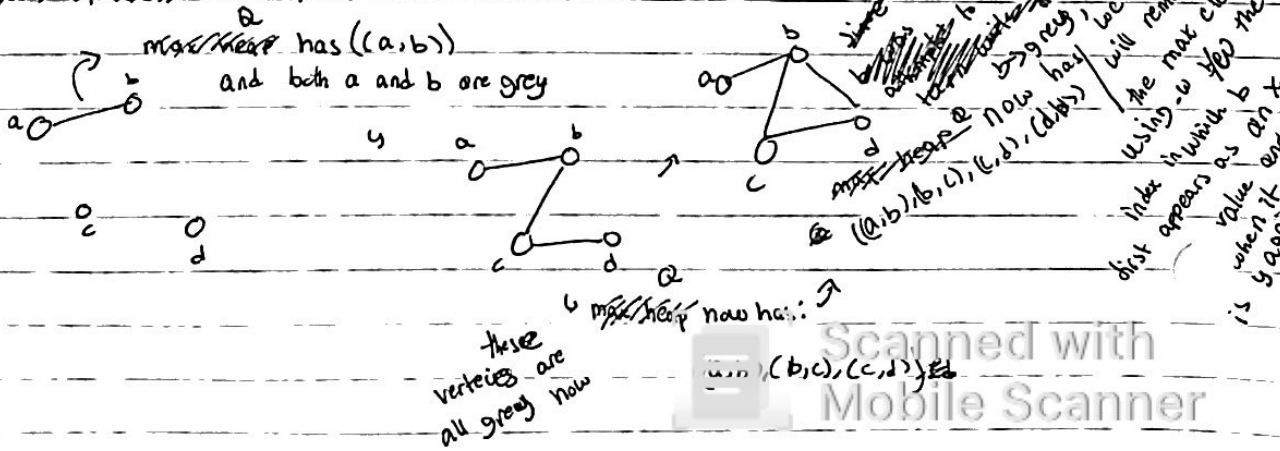
while there is a cycle C in G:

Remove the maximum weight edge in C.

Because of the fact that this algorithm only removes the maximum weighted edge in a cycle, the graph still stays connected after the edge removal, therefore if we eliminate the maximum weighted edges from all the cycles, we will always end up with a minimum spanning tree as all the remaining edges are crucial for ensuring that the graph is connected.

b) Maybe-MST is similar to MST-KRUSKAL, only MST-Kruskal builds a MST by not including edges that try to connect two vertecies that are a part of the same set (hence avoiding cycles) where as we are given an already connected graph consisting of cycles and are told to remove the maximum-weighted edge within the cycles, hence removing cycles.

→ I will assume that my input is a connected and undirected graph G=(V,E) in which E is a vector of pairs of every possible vertex-vertex connections, (u,v) and each vertex-to-vertex connection has a weight property (u,v).w. I will use DFS and DFS-visit to visit every single vertex and I will use a data structure to keep track of the neighboring edges. This may look some thing like:



```
DFS_MAYBE_MST (G)
for each vertex u ∈ G.V → O(V)
    u.color = WHITE
for each vertex u ∈ G.V → O(V)
    Q = { }
    if u.color == WHITE
        DFS-VISIT (G, u, Q)
```

```
DFS_Visit (G, u, Q)        → Θ(E)
u.color == GRAY
for each v ∈ G.adj[u]
    Q = Q ∪ ((u, v))
    if v.color == WHITE
        DFS-VISIT (G, v, Q)
    if v.color == GRAY
        Find the maximum (a_n, b_m) ∈ (w) value
        in Q between when v first appears
        as (a) in (a,b) in Q and now
        when it appears as (b) in (a,b)
        and remove that (a_n, b_m) combo from
        G.E.,
u.color = BLACK
Q = Q - all (a,b) combos that appear
    after u appears as (a),
```

this step just emties the Q data structure as we go. If we reach end of DFS-visit recursion calls, Q ends up being empty.

this step looks something like:

Q = ((a,b), (u, something), (something, something), ...)

this step deletes the cycle-and removes the max edge in a cycle.

If a vertex is grey that means it was being visited and it been entered into Q as (v, something) and it has now cycled back and instead of has been entered as (something, v) into Q. All the edges between (v, something) and (something, v) form a cycle C so by removing the (something, something), w with the maximum (w) from E, we have removed the maximum weighted edge in a cycle.

inside C

→ We need Q to be a data structure that allows us to find max value b/w certain indices.

if its a max-heap it will take $\log k$ (k being size of index to index) to extract max.

overall runtime: O(V·E)

+ whatever time is required to probe through Q and remove max elements b/w indices. + time to remove pairs from G.E.

If Q is max heap and there are C cycles!

Runtime: $O(V(E) + C \cdot \lg E)$

assuming Q has E edges in the worst case

2) MST-prim(G, w, r)
for each $u \in G.V$
$u.key = \infty$
$u.\pi = NIL$
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$ → |V| times
$u = Extract\text{-}min(Q)$ → $\Theta(V)$ time for linked lists
for each $v \in G.Adj[u]$ → O(E) times
if $v \in Q$ and $w(u,v) < v.key$
$v.\pi = u$
$v.key = w(u,v)$

done in constant time by keeping a bit for each vertex for weather or not it is in Q.

↳ this implies that we are making an edge and inserting to another linked list keeping track of keys of vertices of the growing minimum spanning tree and this happens in constant O(1) time for linked lists.

Use unordered linked lists as the data structure for keys (v), the minimum weight of any edge connecting v to a vertex in the tree.

- using Insert in unordered linked list : takes $\Theta(1)$ time and use it to build linked list of every vertex-vertex (edge) key which is what happens at Q = G.V in O(V) time

→ (V-Q)

*Q is all the keys not in the MST and V-Q is all the keys inside MST

total runtime:

~~$O(|V| \cdot V + E \cdot 1)$~~ → *This explanation in the book was not very clear, although the book example used a min heap, the had the same for loop inside a while loop, yet they put their final runtime for prims as $O(V \lg V + E \lg V)$ instead of $O(V(\lg V + E \lg V))$. Since we have to use a linked list the lg V for extract min changes to V and lg V for decrease key simply changes to O(1) for insertion but it is still unclear why our answer is not $O(V(V+E \cdot 1))$*

$= O(V^2 + E)$ ← matter of but I'm assuming

$= O(V^2)$ this is true for simple graphs → $E = O(V^2)$ and that the for loop runs for each while loop so

for trees $E = V - 1$ for connected graph $E = \Omega(V-1)$

total run time: $O(|V| \cdot V + V \cdot E)$ → 1 leaving my answer in terms of E.

~~$= O(V^2 + V \cdot E)$~~

~~$= O(V^3)$~~

RUNTIME: $\Theta(V(V+E))$

3) Bellman-Ford (G, w, s) → original

```
Initialize-single-source(G, s)
for i = 1 to |G.V| - 1
    for each edge (u,v) ∈ G.E
        Relax (u, v, w)
for each edge (u,v) ∈ G.E
    if v.d > u.d + w(u,v)
        return false
Return true
```

```
Initialize-single-source(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

original

```
Relax (u,v,w)
if v.d > u.d + w(u,v)
    v.d = u.d + w(u,v)
    v.π = u
```

New Relaxx will take in count_vector as an input arguement and update its values based on if statement. Count_vector(v) holds how many shortest paths from s to v have been encountered.

Our new Belmont-Ford Algorithm is assumed to always return true and the second outer for loop is the same,

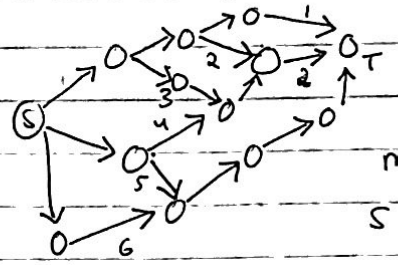
```
Bellman-Ford-New (G, w, s)
Initialize-single-source (G, s)
Initialize countvector ∀ V as 0dices
for i = 1 to |G.V| - 1
    for each edge (u,v) ∈ G.E
        Relax-New (u, v, w, count_vector)
return count-vector
```

```
Relax-New(u, v, w, count_vector)
if v.d > u.d + w(u,v)
    v.d = u.d + w(u,v)
    v.π = u
    count_vector(v) = 1
elseif v.d == u.d + w(u,v)
    count_vector(v) += 1
```

the count_vector holds how many shortest path exist from the source to each vertex V.

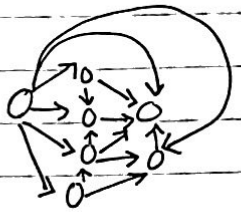
* I honestly don't understand the Boolean component of this algorithm and it does not seem necessary for the promt so I am just excluding it.*

4) Since it is a directed graph you can have multiple ways to reach t from s without forming a cycle.



6 paths to reach T in this case

In the worst case every vertex could have many different ways of reaching t from it and S can be connected to every vertex:



Every vertex can be connected to every vertex in front of it

```
BFS(G, s, t)
for each vertex u ∈ G.V - {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Initialize vector to hold paths called PATHS
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each v ∈ G.adj[u]
        if v == t
            PATHS.append (s ⇝ u → v)
        else if v.color == white
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
Return PATHS
```

→ only works if s is connected to any vertex that is connected to t.

$O(|V|+|E|) = O(b^d)$

where

b = branching factor of graph or average out factor

and d = distance from start node

d = n → can be n nodes away

and assuming each node has 2 children on average

run time is:

$O(2^n)$

5) Problem could be solved by using either DFS to detect cycle or Disjoin sets. I will be using disjoint sets:

→ Necessary supplemental algorithms:

Make-Set (x)
x.p = x
x.rank = 0
↳ O(1)

Find-Set (x)
if x ≠ x.p
x.p = Find-Set (x.p)
return x.p
↳ O(m α(n)) = O(m lg n) to run it m times

Link (x, y)
if x.rank > y.rank
y.p = x
else x.p = y
if x.rank == y.rank
y.rank = y.rank + 1 → O(1)

Union (x, y)
Link (Find-Set (x), Find-Set (y))
↳ O(m lg n)

~~all these algorithms take these constant time~~

algorithm to delete an edge →

```
delete_edge (vector<int> adj[], int u, int v) {
    for all neighbors of u
        if v is a neighbor erase the connection
    for all neighbors of v
        if u is a neighbor, erase the connection
}
```
→ O(v) → linear time

↳ Our Algorithm will create a disjoint set by creating a new set for each vertex. It will then loop through all the edges of the given graph (tree T) and for each edge it will connect the two sets if they are not already a part of the same set. If An edge attempts to connect two vertices that are already a part of the same set, we remove that edge because the two vertices are already connected as a part of a tree.

```
void Remove_Extra_Edge (Graph G{V,E})
v∈for all V { make-set (v) } → V · O(1) = O(V)
e∈ for each edge (u,v) in E → E · lg(V)
    if find-set (u) != find-set (v) → lg(V)
        union(u,v)
    else
        delete-edge ( G, u, v ) → O(V)
```

overall runtime : O(E lg V)

can beat a runtime of $O(n^2)$

6) Social Distancing : cell (i,j) = 1 if some one is standing there

↳ sd has to ~~bigger~~ smaller than manhattan distance b/w any 2 points w/ 1.

→ An Adjacency matrix can be used to hold the grid.

↳ // vector<vector<int>> adj-matrix (n+1, vector<int> (n+1, 0));

If All the positions are given in a vector of pairs, loop through this vector and set the corresponding coordinates in the matrix to one.

```
/* People_standing = [(1,2), (3,4), (7,6), ....]
   for (int i=0; i < People_standing.size(); i++){
       adj_matrix [People_standing[i].first][people_standing[i].second] = 1;
   } */
```

Given The above statements our algorithm would take the size of the matrix (n), the (people_standing) vector, and the social distancing requirements (sd), as its 3 argument inputs.

→ initialize a vector of pairs to hold "sd violators" and an adjacency matrix ↳ comments above

loop through the rows and columns of the adjacency matrix and if your coordinates [row][column] combo contains a 1, start a new set of for loops that run from row-sd to row+sd and column-sd to column+sd.

```
for (int row=0; row < n+1; row++){
    for (int column=0; column < n+1; column++){
        if (adj_matrix[row][column] == 1){          // have a way to make sure
            for (int row2 = row-sd; row2 < row+sd; row2++){   // negative indices aren't used
                for (int column2 = column-sd; column2 < column+sd; column2++){
                    if (adj_matrix[row2][column2] == 1){
                        sd_violators.push_back ((row,column),(row2,column2));
                        // if two people are too close, record their coordinates
                    }
                }
            }
        }
    }
}
```

Runtime $O(n \times n \times sd \times sd) = O(n^2 \cdot sd^2)$

space complexity $(n \times n)$

→ The sd_violators vector will hold any coordinates that are a manhattan

loop through each row and for each row loop through the columns. If a 1 is encountered, inside the loops, loop through all the rows and columns again. Each time another 1 is encountered, calculate the manhattan distance b/w the second set of rows and columns and the first set. If Sd happens to be bigger then these calculated values add both set of rows and columns to sd-violators. at the end, output the sd-violators vector of pairs as the coordinates of those points that violate the social distancing requirement.

```
for (row = 0 to n){
  for (column = 0 to n){
    if (adj_matrix[row][column] == 1){
      for (row2 = 0 to n) {
        for (column2 = 0 to n) {
          if (adj_matrix[row2][column2] == 1) {
            if (Sd > (abs|row2-row| + abs|column2-column1)) {
              if ((row, column) not in sd_violators)
                sd_violators.push_back(row, column)
              if ((row2, column2) not in sd_violators)
                Sd_violators.push_back(row2, column2)
            }
          }
        }
      }
    }
  }
}
```

RUNTIME: $O(n \cdot n \cdot n \cdot n) = O(n^4)$

SPACECOMPLEXITY: $O(n \times n)$