EC330 HW 8

Chase
Marvald
U18719879

1a) This algorithm MAYBE-MST does always produce a minimum spanning tree.

```
1   while there is a cycle c in G
2       remove max weight edge in c
```

This algorithm will detect a cycle remaining in G, & remove the edge with max weight. Continual removal of the largest weight edge in remaining cycles minimizes the total edge count & individual edge weight, creating a MST.

b) An efficient implementation of MAYBE-MST could stand as follows,

```
    MAYBE-MST (Graph G, int weights[]) {
1       T = ϕ
2       for edges in e                  3+4: |E| FIND-SET operations
3           T = T U {e}                 // if (FIND-SET(u) = FIND-SET(v),
4           if T has a cycle c                 cycle in T U {e})
5               e ← max_edge(c)              ↳ In other words, a DFS
6               T = T - {e}                     violation i.e.
7       return T                                duplicate grey
    }                                           set instance
```

This works by the property that a MST with a cycle represents a part of a tree which isn't minimized. In a cycle, every vertex makes contact with ≥ 2 edges, so a removal of the highest weight edge still leaves a spanning tree.

1b) Runtime of MAYBE-MST function

Initialization, Line 4, Line 5	Line 2	Line 3	Line 6
$O(1)$	$\lvert E \rvert$	edge ins queue $O(\lg V)$	del edge $O(\lg V)$

Since $|E| \geq |V|-1$, disjoint set operations take $O(V+E)\alpha(V)$ time $\Rightarrow O(E\alpha(V))$ time.

Since $\alpha(V) = O(\lg(V)) = O(\lg E)$, total run time is $O(E \lg E)$ worst case.

Since $|E| < |V|^2$, $\lg|E| = O(\lg V)$, so we can also say total run time is $O(E \lg V)$.

2) The implementation of Prim's algorithm is as follows:

1) Create a black set of vertices already in MST
2) Assign 0 to the source vertex's key, & ∞ for all the other vertices keys, arbitrarily.
3) While black set doesn't have all vertices
   - i) Pick a vertex u with min key value != BLACK
   - ii) include u to the black set
   - iii) For all adjacent vertices, if weight u-v < previous value of v, update key as weight u-v.

Given the use of an adjacency list, the run time of this algorithm is $|V|$ time to extract min in 3) $\times |V|$ in the while loop for a total of $V \times O(|V|)$, & updating the key values in 3) takes $|V|$ time $\times |E|$ times from its for loop. Overall time complexity is:

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

assuming a connected graph, $V = O(E)$

Single source Shortest Path

3) This solution effectively computes the number of shortest s-t paths in a graph with modification of the Bellman-Ford algorithm. The original algorithm finds a simple graph with $\leq |V|-1$ edges & no negative weight cycles. to find the shortest number of st edges in the graph, we perform the following,

```
int bellmanford_count(Graph G, int src){
  for v ∈ V
      A[v,0] ← (∞, 0)   // set large initial distances (arbitrary)
  A[t,0] ← (0,1)    // d[source] ← 0, src-src = 0

  for i from 0 to |V|-1:   // max # edges
      for v ∈ V:
          c ← min_{u∈V, (v,u)∈E} A(c(v,u) + A[u,i][1])
                         ↳ smallest wt path
          k ← 0
          for u ∈ V such that (v,u) ∈ E
              if c(v,u) + A[u,i][1] = c
                  k ← k + A[u,i][2]   // # instances of shortest dist
          A[v, i+1] ← (c, k) }
  }   }
}
c ← min_{0≤i≤n} A[s,i][1]
k ← 0
for i from 0 to |V|
    if A[s,i][1] = c
        k ← k + A[s,i][2]
return k
}
```

// computes smallest weight across all A[s,i], & sum the corresponding # of paths k

- Each entry $A[v,i]$ is a pair (c,k) where c = smallest weight path from $s-t$, k = total # of paths.
- Let $c = \min_{u\in V, (v,u)\in E} A(c(v,u) + A[u,i][1])$, where $A[u,i][1]$ is the first in the pair $A[u,i]$
- $A[v, i+1] = (c, \sum_{u\in S} A[u,i][2])$

4) All paths from source to destination

Assuming the graph is connected, we are able to find all paths between two nodes with DFS.

```
ALL-PATHS(Graph G, int sourcenode, int destination)
    for each vertex u in G.V
        u.color = WHITE // mark unvisited nodes
    path = 0
    for each vertex u in G.V
        if (u.color == WHITE)
            DFS-VISIT(G, sourcenode)
DFS-VISIT(Graph G, int u)
    if (u == TARGET)
        path ++
        return
    u.color = BLACK
    for each v in G: Adj[u] // for all neighbors of u
        if (v.color == WHITE)
            DFS-VISIT(G, v)
    u.color = WHITE // check for other possible walks
```

To iterate through the stack created by G: Adj[u] in DFS-VISIT, we do O(1) work per |E| iterations, with |V| extra work for the worst case, (all vertices + all edges can be used for a given path) which gives us the worst-case of $O(|V|+|E|)$ for DFS-VISIT, the standard for DFS. Assuming we are given the number of nodes (n) we can say the total worst-case runtime with ALL-PATHS calling DFS-VISIT is $\underline{O(n(|V|+|E|))}$. Initializing our vertices to WHITE takes $O(n)$ time, but $O(n) << O(n(|V|+|E|))$, so we may ignore it in the total $\underline{O(n) + O(n(|V|+|E|))}$.

5) Extra Edges

To remove any extra edges in a graph, we utilize DFS & two edge functions to remove edges and check if the graph is still connected: i.e. $V = |V|$.

1) Remove the given edge
2) Find all reachable vertices, labelling visited vertices as BLACK or false, true as WHITE, unvisited nodes, & GREY as visiting.
3) If original $V| = |V|$, removed edge set, return false for the function. The given edge is not necessary for connectivity.

memset(visited, false, sizeof(visited)) & initialization such functions will take $O(n)$ time. DFS will take its standard runtime of $O(|V| + |E|)$, used a few times in the program. If there was a preexisting cycle, this would solve the problem of creating a graph with n-1 edges. DFS would be run until it found a repeated node in the GREY visiting set, & the occurence of DFS in the removeEdge function would allow termination of the program in worst-case $O(|V|+|E|)$ time if we traverse the whole graph and there aren't any cycles ($|E| \leq |V| - 1$).

6) Social Distancing

To check if a distance given by the Manhattan distance given by $|i-k| + |j-l|$ between vertex u & v as (i,j) & (k,l), respectively, we apply the Floyd Warshall Algorithm to find all the shortest paths from each node to every other node. While doing this, we qualify any violation of our distance $|i-k| + |j-l|$ bound by an integer sd. Any two nodes closer together than the bound in our adjacency matrix will be updated as a shortest path, as distance[i][j] > dist[i][k] + dist[k][j]

in the following structure:

```
for k in range 0:|V|
    for i in range 0:|V|
        for j in range 0:|V|
            if (dist[i][k] + dist[k][j] < sd)
                dist[i][j] <- 0, flag for violation
            if vertex k shows a path < sd
```

This has a time complexity of $O(V^3)$ in every case, as we need to check our $n \times n$ adjacency matrix for any distances below the threshold.