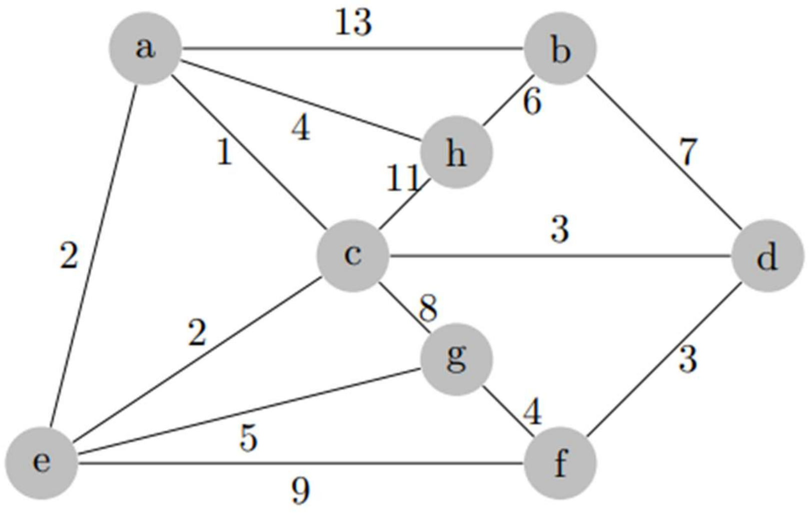
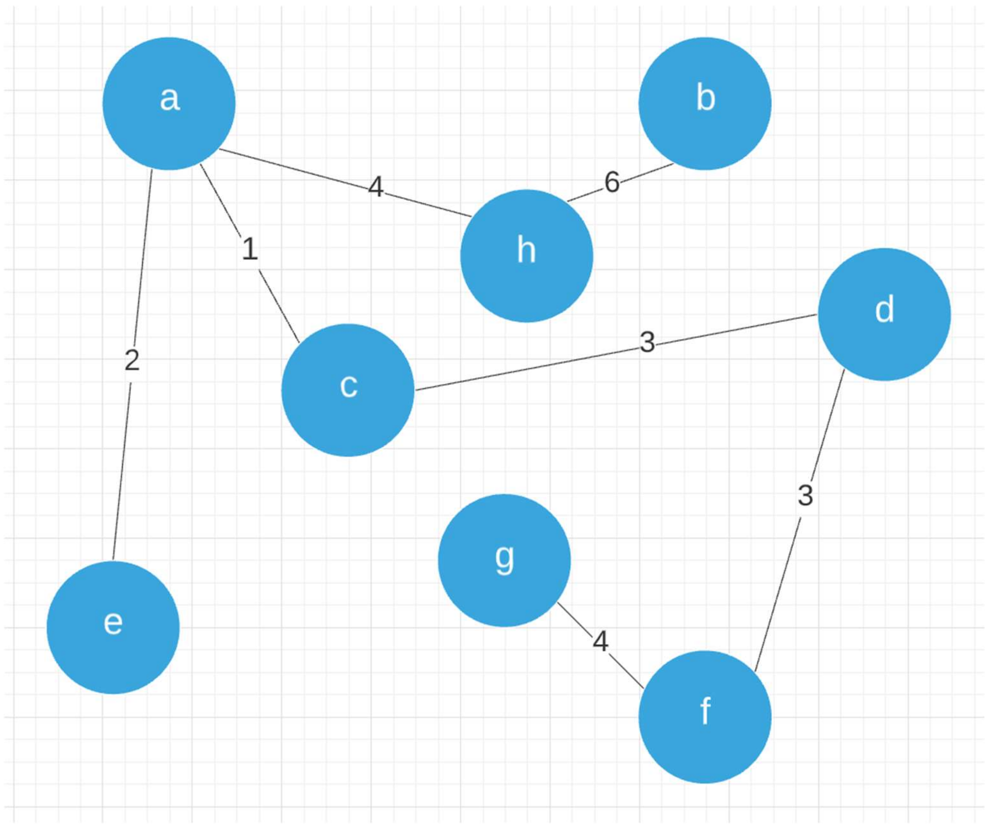
Brandon Vo

# Homework 9



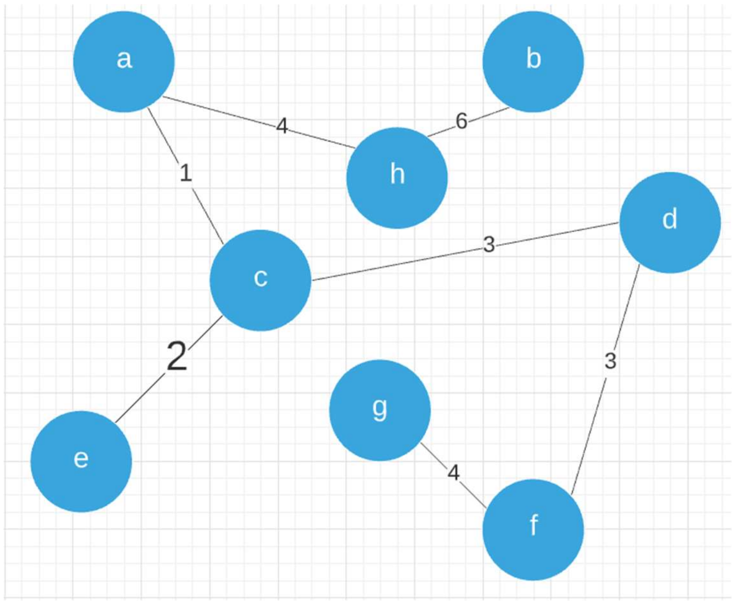
1. For the graph above:
   (a) What is the cost of a minimum spanning tree?



- Total Cost = 6 + 4 + 1 + 2 + 3 + 3 + 4 = 23

Brandon Vo

(b) How many minimum spanning trees does it have?



- We can replace (a, e) with (c, e) and maintain the same cost of the MST

There are 2 different minimum spanning trees contained in this graph.

(c) Run Kruskal's algorithm on the graph above. In what order are the edges added to the MST? For the first three edges in this sequence, give a cut that justifies its addition.

Kruskal's algorithm starts with the lowest weight first:

- (a, c) – 1
- (a, e) – 2
- (c, e) – 2 (Already a part of the set, so not added. Considered for a second possible MST.)
- (c, d) – 3
- (d, f) - 3
- (a, h) – 4
- (g, f) – 4
- (e, g) – 5 (Will form a cycle, not added)
- (b, h) – 6
- (b, d) – 7 (Will form a cycle, not added)
- (c, g) – 8 (Will form a cycle, not added)
- (e, f) – 9 (Will form a cycle, not added)
- (c, h) – 11 (Will form a cycle, not added)
- (a, b) – 13 (Will form a cycle, not added)

Brandon Vo

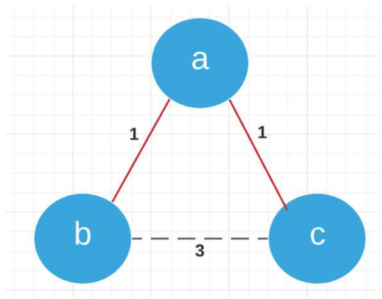
2. Prove or disprove: If a graph has a cycle, the heaviest edge in the cycle will not be part of any MST.

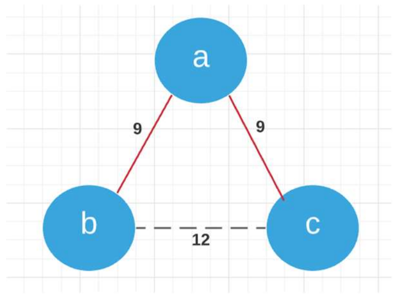
A cycle means there is more than one path reach every vertex node in the graph. If the heaviest edge is part of the cycle, it must be considered when trying to remove the cycle. When being considered, the extra edge with the largest weight will always be removed; otherwise, we will not have a minimum spanning tree.

A cycle means there is more than one path to reach all nodes in the graph, so the heaviest edge in the cycle would be unnecessary to reach the rest of the vertices.

Brandon Vo

3. Prove or disprove: Adding a constant to every edge doesn't change the MST.





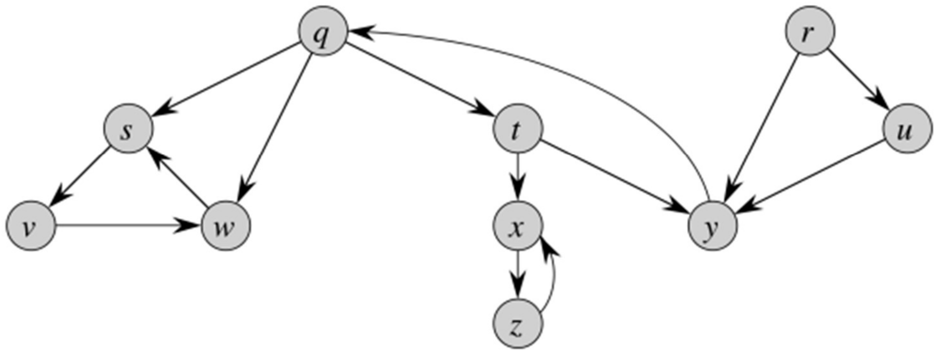
The MST path itself doesn't change because the path taken will still have the lowest cost despite the k constant being added. The entire MST will have its cost increased by (n-1)*k, but the MST path itself doesn't change.

When you cut the graph above, we will still have (a, b) and (a, c) to consider first when building the MST because they're still the smallest weights in the graph. Adding k doesn't change which edge we start with.

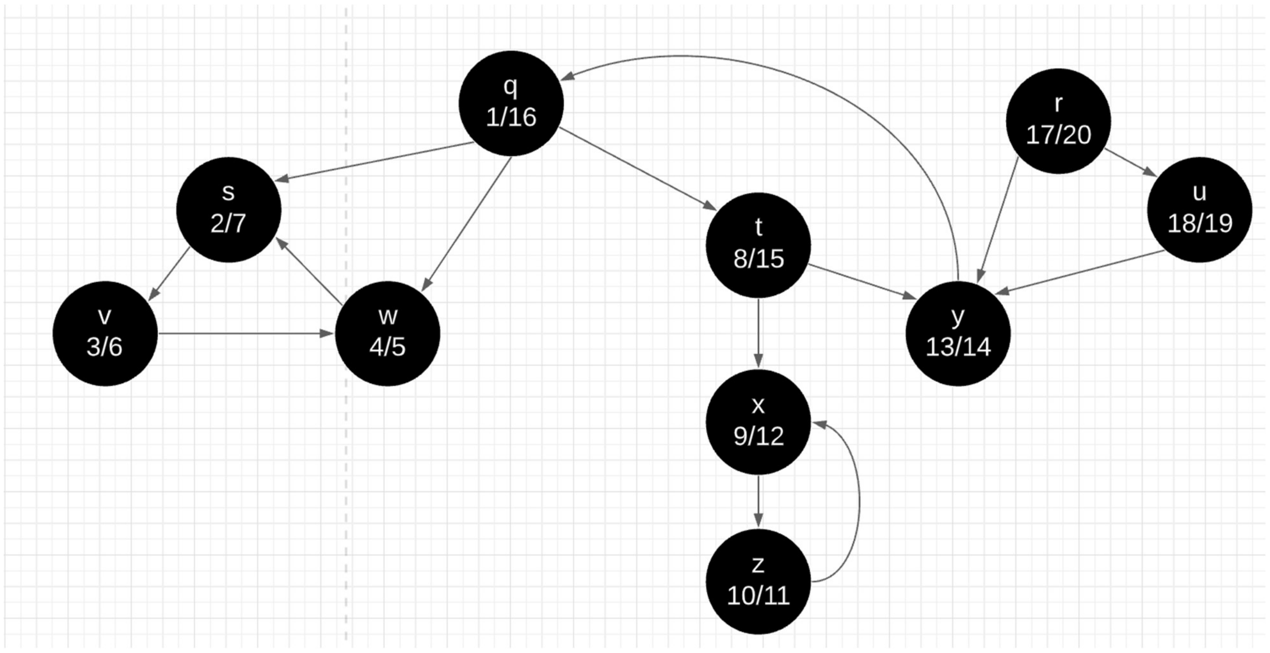
Brandon Vo

4. Run the procedure STRONGLY-CONNECTED-COMPONENTS on the graph below. Show the:
   (a) the finishing times for each node after running DFS in line 1
   (b) the DFS forest produced by line 3
   (c) the nodes of each tree in the DFS forest produced in line 3 as a separate strongly connected component.

Assume the loop of lines 5−7 of the DFS procedure (page 604 in CLRS) considers the vertices in alphabetical order and assume the adjacency list is ordered alphabetically.



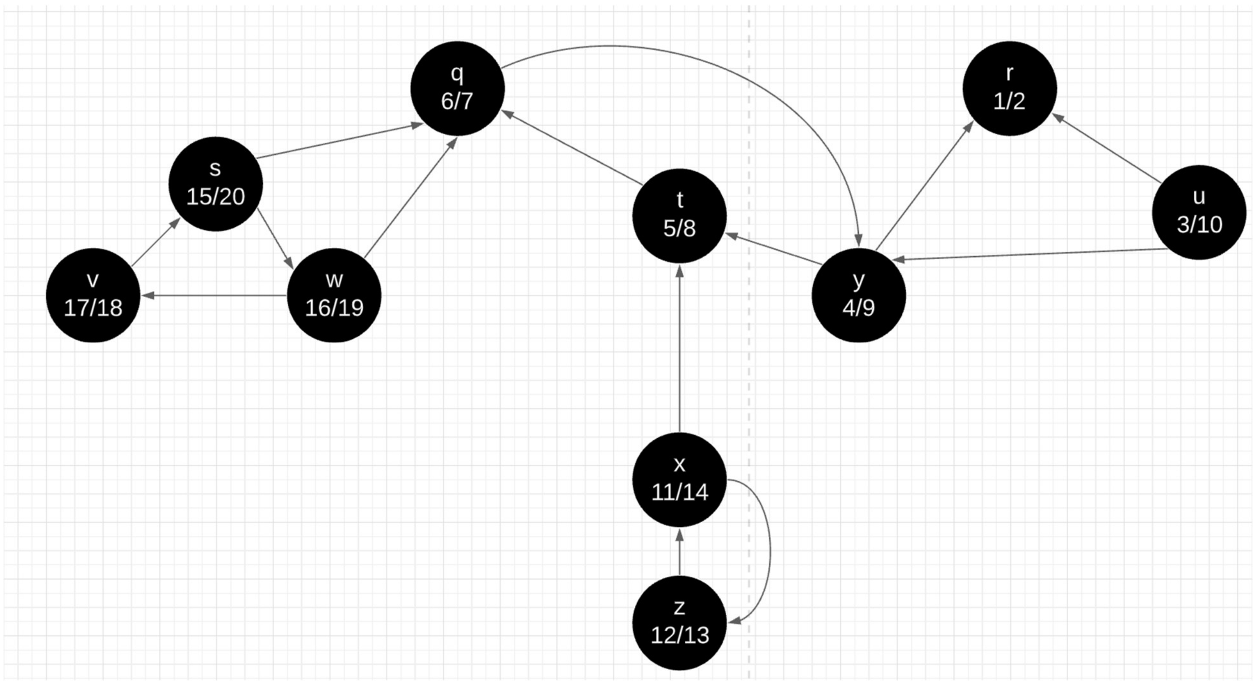
a) Running STRONGLY-CONNECTED-COMPONENTS
1) Calling DFS(G) to compute finish times



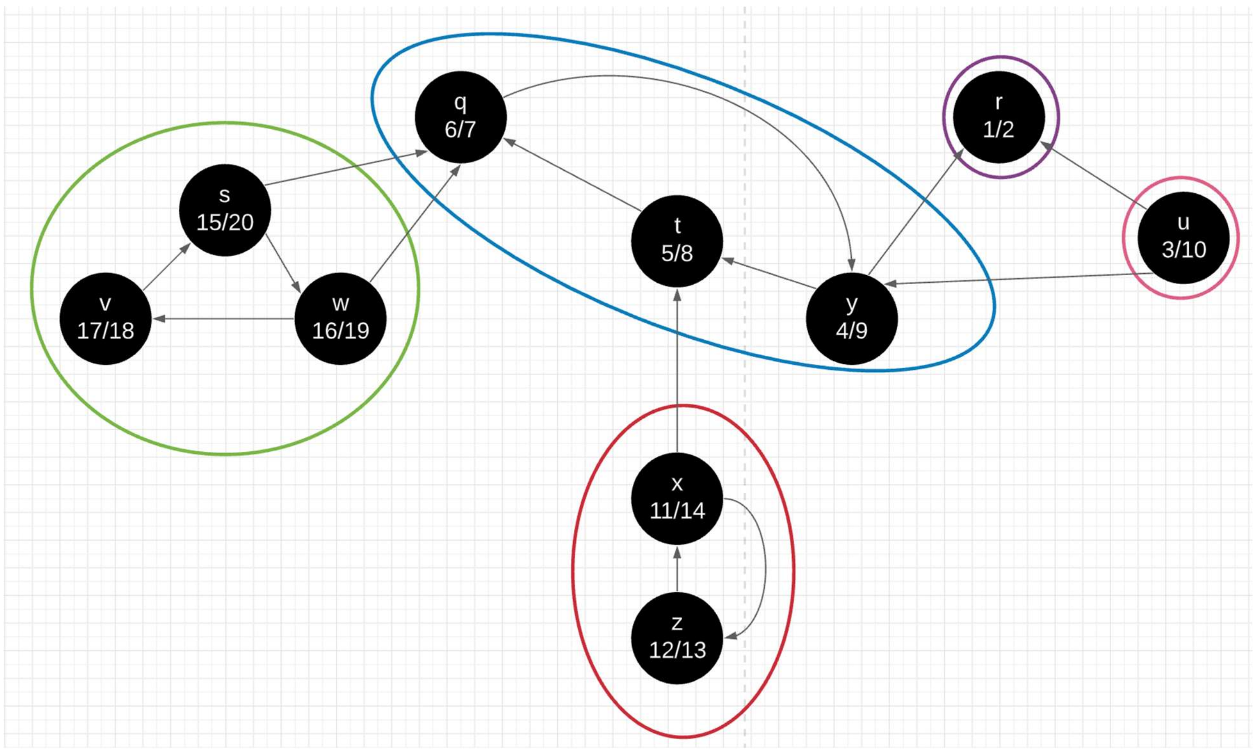
Brandon Vo

b) The DFS forest produced:

Computing DFS($G^T$)

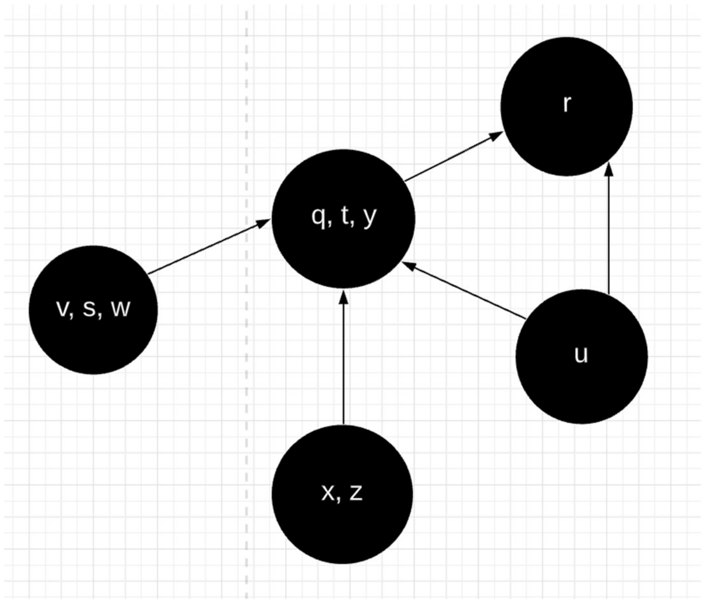


c) Showing the separate SCC subgraphs



Brandon Vo

5. Create $G^{SCC}$ the component graph, of the graph in question 4. See page 617 in CLRS for the definition of a component graph.



Brandon Vo

6. (15 points) In a small town, the mayor has the budget to resurface n streets.

   He would like you to resurface the streets so that the maximum number of people will use the new streets. He has a graph of the existing streets and on average how many times per day they are used.

   You want to renew the most used streets. You are given a graph G with all the streets as edges and intersections as nodes.

   Design an algorithm to determine which of the streets to work on. Determine and then justify the run time of your algorithm. Prove your algorithm works

- Assuming that for this question, the paths will be undirected because streets are generally bi-directional
- We can invert the minimum spanning tree into a maximum spanning tree by running Kruskal's algorithm. To do so, we can simply run Kruskal's algorithm but take the negative weight -w when sorting our edges.
    - When sorting in non-decreasing order, it will choose the edges that originally have the maximum weight and build a Spanning Tree based off those edges.
- The time spent is the same as Kruskal's algorithm's time complexity which is O(E log V)

- We will always have the most used street because we're organizing based off the inverted values of their weights.
- Kruskal's algorithm sorts based off nondecreasing order so the edge with the largest weight will also have the smallest inverted weight $-w$.
    - The smallest inverted weights will always be chosen and be used to build several forests used.
    - These forest will not have cycles and will use the next largest edge as a cut edge to bridge the connection between two disconnected subgraphs.
- For Kruskal's we start our MST with a single path and continue building our MST through the next largest edge found. Each loop iteration retains the Maximum Spanning Tree property and always includes the next largest weight available to be added.

```
MAX-KRUSKAL(G, w)
A = ∅
for each vertex u ∈ G.V
    MAKE-SET(v)
//Make the weight negative.  This would change the assortment of edges to be in decreasing order, and as such, we would be organizing based off largest weight rather than smallest weight
```

Brandon Vo

```
Sort the edges of G.E into nondecreasing order by weight -w        //Use -w instead of w
//The rest of the algorithm will be the same but uses negative weight to sort the edges in decreasing order
For each edge (u, v) $\in$ G.E, taken in nondecreasing order by weight -w
    if FIND-SET(u) $\neq$ FIND-SET(v)
        if A < (u, v).distance
            A = (u, v)
return A
```

Brandon Vo

7. After providing the mayor from question 6 his answer, suddenly a new restaurant opened in town selling cake. This changed to traffic on one street. It now was 30 cars/hour instead of 2 cars/hour. Design an algorithm to update which streets should be repaved if exactly one street $n_i$ was changed from $w_i$ to $w'_i$, the mayor wants a generic approach as he is worried about rapidly adapting to further developments in his community.

What is the running time of your algorithm? Justify your run time. Prove your algorithm works

- When we linked every node together using Kruskal's algorithm, we have been given a set of maximums and the subtrees that are tied to each maximum.
- Prim's algorithm starts within a node and builds a tree that spans the vertices in v.
- We can add the extra edge to the graph. This will create a cycle as two nodes will now both be pointing at each other
- We can then run DFS on that node to determine if we need to change our MST path.
- This would take O(|V|+|E|) time as the worst case would require us to traverse the entire MST cycle to find an edge to remove.

- The algorithm works because we have 4 cases when modifying an edge:
    - If an edge is in the maximum spanning tree and has its weight increased, then it will remain in the Maximum Spanning Tree due to still meeting the requirements. Maximum Spanning Tree Property will still be maintained.
    - If an edge is not in the Maximum Spanning Tree and has its weight decreased, then the resulting edge will still not meet the requirements to be in the Maximum Spanning Tree and doesn't need to be considered. Maximum Spanning Tree Property will still be maintained.
    - If an edge is in the Maximum Spanning Tree and has its weight decreased, then it has to be checked to see if it should remain in the Maximum Spanning Tree.
        - While the edge is not in the MST, the vertex is a part of the MST and should have another edge that is a part of the MST.
        - If we remove our updated edge, it would create two separate SCC graphs that need a cut edge along one of the graph's paths to reconnect the MST.
        - We can cut the updated edge and use DFS to find a suitable path to reconnect the two graphs.
        - If the updated edge is still the largest possible cut edge, then it will be readded. Otherwise, it will be replaced by a different edge to create a different path.
        - In either case, the Maximum Spanning tree property will be retained after adding or readding a cut edge back into the MST as the DFS search will always bring up the largest possible edge.
    - If an edge is not in the Maximum Spanning Tree and its weight increases, then it has to be considered if its eligible to be in the MST.
        - While this edge was originally not in the MST, the vertex should be in the MST, meaning that it has an edge that is in the MST.
        - This means that adding the extra edge into the MST will create a cycle

- We can run DFS and check for back-edges as a sign of a cycle and check if it's the smallest edge found in the cycle's subgraph
- We can remove the next smallest edge which was part of the cycle, which would eliminate the cycle and restore our MST's acyclic property with the new edge if it was small enough to remain included.
- Regardless of which edge is removed, the cycle will be eliminated and the edge with the largest possible weight will remain in the MST. This means that the Maximum Spanning Tree property will be maintained.

```
UPDATE-ROAD(G, u, r, X)
//u is the node where the restaurant opened at
//v is the road where the traffic has changed
//X is the new distance we're changing (u, v) to

If (u, r) ∈ MST  //If this road was in the maximum spanning tree
    If (X > 0) //If we're increasing the weight
        Return  //Do nothing as it's already in the MST
    Else      //We're decreasing the weight
//Temporarily remove the updated edge from the MST and search the graph again for the lowest possible cut edge
        Set (u, r) = X
        Remove (u, r) from the MST
//Splits into two separate GCCs which we must find the connected edge of
        Maximum = (u, r)
        //Search the graph G for a cut edge that can be used to reconnect the two SCC graphs
        While v in DFS(MST)
            If FIND-SET(u) != FIND-SET(v)
                UNION(u, v)

Else if (u, r) ∉ MST //Not in the MST
//Then we have to check if increasing the weight makes it viable to be in the MST
```

Brandon Vo

```
    If (X > 0)
        Insert (u, r) with distance X to the graph          //This will create a cycle
//We have created a cycle which can be solved by removing the smallest edge in the graph
        Minimum = X
        While v in DFS(MST)      //This would take O(V log V) time
            If (u, v) in DFS(MST) has a back edge //Cycle found
                If (u, v)'s distance < Minimum
                    Minimum = (u, v)
        Remove the edge of (u, v) found in Minimum      //This will remove the cycle and restore
the MST with its maximum property
    Else
        Return  //Not in the MST and decreasing weight means it won't change the MST.
```

Brandon Vo

8. Your Aunt Mary is having a party! She lives in California, so it is warm enough to have a picnic outside. She has invited n people.
   When it is time to eat, she would like to make it easy for everyone to start eating.

   She has decided to spread out k blankets on the ground. Therefore, she would like you to group the guests into k groups. To keep the conversation flowing cleanly she wants people who are in a group to share a blanket (note the blankets are really large so many people can sit on them, however her party takes place over an impressive 20 square miles of territory ). In other words, she wants to cluster groups of people together such that the distance between groups represented on a 2-d plane is maximized. Due to prior difficulties (uncle Bob got himself lost for 5 months), she has the geo-location of every member of the party ready for analysis in real time. Group the people into k clusters C1, C2, . . . , Ck such the minimum distance between two people which are in different clusters is maximized.

   Design an efficient algorithm to divide the guests. Aunt Mary wants to know the list of people in each group.

   Determine the running time of your algorithm using big-Oh notation. Justify your running time.

- This question is asking how to maximize the distance between every blanket of guests.
- We have to maximize the distance between every group of people
- We can represent a blanket k as a node with the distance being an edge of v
- When presented with a list of people, we can use Kruskal's algorithm mixed with union-find to group up the closest people out of n possible
  - The two closest people will form a pair and those pairs will pair up with the other closest pair found
- These people will group up and merge their groups with other groups until they have enough people to form k clusters
- As a result of the merge, the clusters we would end up with would have the maximum possible distance from each other
- We're running Kruskal's algorithm up until we have K clusters in terms of vertices.
  - The time to sort all edges would remain to be O(E log E)
  - The time to process the vertices would be O(N – K), we would have to run Union merge O(N - K) times to reach K clusters, but our edges outnumber the number of vertices as the number of edges between all people would be $E = N^2 - 1$.
  - The process to merge based on the smallest edge would be O(E log V) due to using Kruskal's algorithm
  - Our run time would be O(E + E log E) where V > K and N >= K

Proof:

- Because Union-Find merges people based on the smallest relative distance to the nearest person, every time we merge, the minimal distance that group has relative to the other would increase as we're merging the smallest distance in the graph.
  - Every time we merge, we remove the smallest path to another potential group.

  - Whenever we merge, we also change the number of groups we have from n to n – 1 to n – 2 and so on until n = k clusters.
  - When we finish merging into k clusters, the only distances remaining would be the distances that weren't small enough to be merged in UNION-FIND, meaning that we maximized the minimal possible distance.
- During our loop, we maintain the property of having only the maximum possible distance remaining between each group of people.
  - After each loop, we merge a group which removes the smallest weight found in the graph. This property is maintained after each loop iteration, meaning that we will keep the maximum possible distance for each loop iteration.

```
GUEST-LIST(G, List, k)

//Preprocessing, place every element in the graph

For every element X in List

        MAKE-SET(X)

        Build X onto graph G with one or more edges V with distance D


//We merge based on closest distance, O(E log E)

Sort the edges of G.E into nondecreasing order by distance d


N = List.length

While (n > k)     //Keep merging until we get k clusters, O(N – K)

   Minimum = NIL

   For each (u, v) taken from the sorted list //O(E) time

         If FIND-SET(u) ≠ FIND-SET(v) and (u, v) has a distance d > Maximum

                Minimum = v

   EndFor

   UNION (u, Minimum)//Merge the two closest people into a group, O(V time)

   N--   //Two clusters have merged, leaving us with one less cluster

EndWhile
```