CS535 Design and Analysis of Algorithms - Assignment 1

Batkhishig Dulamsurankhor - A20543498

September 6, 2024

Problem 1

Solution:

Let's prove that these two statements are equivalent by deriving one statement from another.

1. If a connected subgraph is on n vertices has $n-1$ edges, then the connected subgraph must have exactly one path between every pair of vertices.

   Let's say we have a connected subgraph $G = (V, E)$ that has n vertices and $n-1$ edges, but G also has a pair of vertices that have more than one distinct path between them. This implies that there is a cycle/cycles in the graph which makes it possible to connect the vertices by more than one path. However, a graph with cycle but has the property a cannot exist since having a cycle will result in a loose vertex and a disconnected subgraph. Therefore, property a and b must be both true.

2. If a connected subgraph have exactly one path between every pair of vertices, then the connected subgraph is on n vertices has $n-1$ edges.

   If there exists only one path between every pair of vertices, the graph must be acyclic as we have proven above. For the subgraph to be connected, it must follow the basic property of a tree, which is having $n-1$ edges for n number of vertices.

Problem 2

Solution:

1. The correctness of Prim's algorithm

   Prim's algorithm is a greedy algorithm that after each iteration we choose the vertex that has smallest weight edge to the result set from the unvisited set of edges and remove edges from cutset if they are connected to the vertex, and add the the rest of the connected edge to the cutset. In class, we kept the cutset using heap data structure.

   Firstly, the resulting subgraph will always be a connected tree, since each vertex will be added only once to the visited set (the result set) and it is impossible for a back edge to be added to the result for the reason that a vertex can be added to the result only once.

   Also, on each iteration, the result set will always be a MST for the current result set. We can argue that it is not a MST and some edge e can be replaced with an edge e' where $weight(e) > weight(e')$ and both connected to vertex v. However, when we added the vertex v to the result set, we added the minimum weight edge in the cut set and minimum weight edge e is more than e' which results in contradiction and e' must be chosen. This is true for every iteration, including the last one that gives the final answer.

2. The correctness of Boruvka/Sollin's algorithm

   Boruvka/Sollin's algorithm is also a greedy algorithm. We start off by choosing the smallest weight edge for all vertices, and merge the connected edges to treat them as single edges. Then, run the same process on the new graph and keep repeating it until we get a MST.

   Similar to above proof, at any given moment, resulting tree/trees will always be MSTs. Because, we are choosing the minimum edge for vertices/merged vertices. By combining trees/vertices together with the smallest weighted edge between them, we are building bigger and fewer trees as long as the given graph is connected. Since the number of edges will always be less than the number of vertices and we prevent cycles by removing them and treating the merged edges as one to not choose an edge connects to itself again, we can conclude that the result after each step is a tree. This is valid for every iteration, so by conduction the algorithm will give MST.

Problem 3

Solution:

MST in the original graph remains the same after we add $-min_e w(e)$ to all the edges. This is because to calculate MST, the relative differences between the edge weights are used and not their individual values.

Let's say vertex v is connected to vertex u by edges e_1, e_2, e_3 with the corresponding weights $-4, -2, 3$. The minimum edge between the two vertex would be e_1, because it has the least weight. Now let's add $-min_e w(e) = 4$ to each of the vertex, and the modifed edges are e_1', e_2', e_3' will have $0, 2, 7$ respectively. Here, the minimum edge is e_1' and even after modification, we still have the same edge.

$$
\begin{aligned}
&(w(e_1) + (-min_e w(e))) - (w(e_2) + (-min_e w(e))) \\
&\quad = (w(e_1) - min_e w(e)) - (w(e_2) - min_e w(e)) \\
&\quad = w(e_1) - min_e w(e) - w(e_2) + min_e w(e) \\
&\quad = w(e_1) - w(e_2)
\end{aligned}
$$

Since all algorithms to calculate MST, like Kruskal's, Prim's and Boruvka/Sollin's etc, involves relative difference between edge weights, MST in the original graph will stay the same after the modification.

Problem 4

Solution:

If the edge weights are integers in the range from 1 to W for some constant W, we can optimize the algorithm by using Bucket queue[1], instead of a normal priority queue. In the bucket queue, the priorities must be integers, and it is particularly suited to applications in which the priorities have a small range.[2]

In this algorithm, insertion and decrease key operations are constant time($O(1)$), which is a much more efficient compared to $O(log(n))$ priority queue's operations. The time complexity of the algorithm will decrease down to $O(V + E)$, because these operations take constant time.

Problem 5

Solution:

1. As we have discussed in the class, the number of iteration in the algorithm is $log(V)$ because the number of vertices is reduced logarithmically every iteration. The number of edges that we have to

calculate stays the same. It means that the time complexity to calculate G and G' is $O(E)$ for both, thus the running time of the algorithm is $O(Elog(V))$.

If we assume that the Algorithm 1 ran for k phases, the overall running time would be $O(kE)$ since the number of edges we have to calculate for each phase is E in worst case scenario.

2. After running k phases, we have to connect T trees instead of the initial problem where we had to connect V vertices. According to Kruskal's algorithm, determining if an edge form a cycle is $O(logV)$, in our case it is $O(logT)$. In worst case scenario, we still have to iterate over all of the edges E for all phases. So the time complexity of the second part (Kruskal's algorithm) would become $O(Elog(T))$. Thus, the total running time of the problem would be $O(kE + Elog(T))$.

3. I don't think there is a universally optimal value of k. Because the performance depends on many factors, namely the number of vertices, edges, density of the graph etc. However, Boruvka's algorithm quickly reduces the number of trees and Kruskal's algorthm is efficient when there are fewer edges to process for finding cycles.

   A good value for k seems to be the number that halves the workload for both algorithms. After running k phases, the number of vertices left would be approximately $n/2^k$ because the number of vertices is halved on each iteration. Switching to Kruskal's when we are left with half of the vertcies is trivial, that means $k = 1$ and it can be useful when the number of vertices is very few. Since we are dealing with logarithmic number of total iterations, switching when the number of vertices is around $\sqrt{n}$ will remain around equal workload for both.

   $n/2^k = \sqrt{n}$
   $n/\sqrt{n} = 2^k$
   $2^k = \sqrt{n}$
   $k = log(\sqrt{n})$

   The time complexity of the algorithm will be $O(Elog(\sqrt{n}) + Elog(T)) = O(Elog(\sqrt{n}T))$.

Problem 6

Solution:

Only an edge is removed from the communication MST so that we are left with 2 trees and we need to find the minimum weighted edge that connects these two trees.

To efficiently solve this problem, we can modify Kruskal's algorithm. The bottleneck for Kruskal's algorithm here is to sort the edges which takes $O(ElogE)$ time. Since we are trying to find only one edge, we can skip the sorting part. First, we need to initialize the union-find datastructure on the two trees. We keep track of the minimum weight value. Then for all edges, we execute the find operation and whenever we find an edge that connects these two trees, we compare its weight with the minimum weight and replace it if the new value is smaller. At the end of the iteration, we will be left with the answer. However, if the minimum value is the same as we initialized it, we have no edge that connects these two trees. In other words, the only edge that connected these trees has been removed in the beginning. The following is the pseudocode for this algorithm.

```
min=MAX_VALUE
uf = UNION_FIND(set1, set2)
for u,v,w in E: //u and v are the vertices and w is the weight
    if uf.find(u) != uf.find(v):
        if min > w:
            min=w
```

if min==MAX_VALUE:

    return no edge

else:

    return min

The time complexity of initializing union-find takes $O(V)$. Find operation takes amortized $O(\alpha(V))$, so for all edges it takes $O(\alpha(V)E)$, where $\alpha(V)$ is negligible. Therefore, running time of the algorithm is $O(V + E)$.

Problem 7

Solution:

If we compute the smallest edge in the cut-set for each vertex (label) instead of maintaining the cut-set for the edges, finding the smallest edge will now become finding the vertex with the smallest label.

Let's say we have S that is the objective and R from which we have to add vertices to S. Let's say a vertex v is added to S. Diffferent from the classical approach, we don't have to delete any edges apart from the edge that is connected to v this time. Because we have the label of the vertex, which is the smallest weight that connected it to S. However, we would still have to add edges to the cut-set if there exist edges such that $u, v \in R$. We can add new vertex labels only if it's not present in the cut-set. If a vertices u is already present in the cut-set, we have to update its label in case if the weight is less.

This algorithm is more efficient if the number of edges are much more compared to the number of vertices in the graph G. Because, we are reducing the heap size by eliminating the unnecessary edges before we adding them to the cut-set, thus lowering redundant calculations. The time complexity of the algorithm is $O(VlogE)$ since updating values in priority queue is $O(logE)$ and we are iterating through every vertices.

References

[1] Bucket queue, *Wikipedia*, `https://en.wikipedia.org/wiki/Bucket_queue`

[2] Skiena, Steven S. (1998), *The Algorithm Design Manual*, Springer, p. 181.