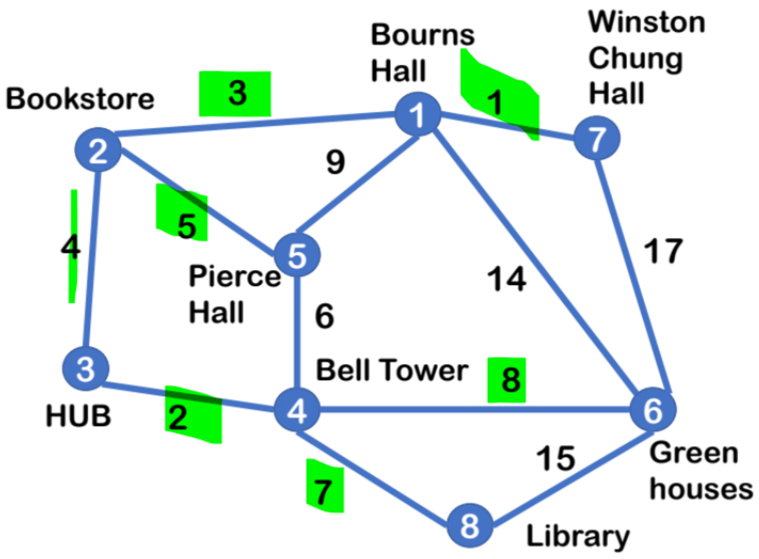
CS141, Fall 2021, Assignment #4 **Due: 11:59pm, Wed 11/17, 2021**
Name: Rishab Dudhia **Student ID: 862141444**

1 Decorating UCR

1. As shown in the given map below, I have highlighted 7 edges, or roads, to be decorated. To make my selection, I sorted the edges by weight from least to greatest. Then starting at the lightest weight edge, I chose to decorate each road, edge, as long as choosing that edge does not create a cycle within the vertices being connected by the chosen edges. Once all vertices, buildings, can be reached by a decorated road, we have finished our task.



2. We know that the road between Bourns Hall and Winston Chung Hall will always be decorated because it is always in the optimal solution of the problem. The problem states that there must be at least one decorated path that connects any two buildings. Then, let the set S contain the set of edges in our solution. If the edge connecting Bourns Hall and Winston Chung Hall is not in S then, there must be a path connecting these two buildings in S. Since we cannot use the edge directly connecting these two vertices, we must travel through at least one vertex (along at least 2 roads) to achieve a path between Winston Chung Hall and Bourns Hall. Therefore, with all edges having positive weights, the minimum path weight to connect the two buildings must be at least 2 units (each road length ≥ 1). However, we know that the weight of the edge between Winston Chung Hall and Bourns Hall is 1 unit, so including this edge in S instead of a different subset of E to connect Bourns Hall and Winston Chung Hall will always be a part of the optimal solution to minimize the cost of decorating roads on campus.

3.

Step 1. Sort all the roads by road length from least to greatest, store them in a list E.

Step 2. Scan all the roads in E in order, and decide if that road will be selected or not. How do you decide if a road will be chosen to decorate?
We know to select a road if the road does not result in a cycle in the map with the previous connected buildings and the new connected buildings. We can use Depth First Search to find a cycle in a given graph, however, with Kruskal's algorithm, we create disjoint sets to remember

super vertices. Then, when checking an edge, we make sure that the buildings connected by the road do not belong to the same super vertex, in turn ensuring that no cycle occurs.

Step 3. You are using a table to help you simulate the algorithm (see below). The first row has been shown as an example of the first step of the algorithm. Fill in the table of simulating the Kruskal's algorithm:

Step	endpoint 1	endpoint 2	weight	Decorate?
Step 1	1	7	1	yes
Step 2	3	4	2	yes
Step 3	2	1	3	yes
Step 4	2	3	4	yes
Step 5	2	5	5	yes
Step 6	5	6	6	no
Step 7	4	8	7	yes
Step 8	4	6	8	yes
Step 9	1	5	9	no
Step 10	1	6	14	no
Step 11	6	8	15	no
Step 12	6	7	17	no

4. The total cost of Kruskal's Algorithm is $O(m \log n)$. We can see that sorting the list of edges by weight from least to greatest costs $O(m \log m)$. Using the union-find data structure, we know that we can iterate through the list of edges with a check that costs $\alpha(n)$ to make sure that a given road only connects a super vertex to another super vertex, a super vertex to an unvisited vertex, or an unvisited vertex to another unvisited vertex (all three scenarios result in a new super vertex containing the two prior elements). Once we know the edge is valid, we can use the union function of cost $\alpha(n)$ to complete one of the aforementioned scenarios to create a new super vertex. The cost of $\alpha(n)$ is relative to constant, so the cost of these functions can be seen as having a cost of $O(1)$. Then, after sorting the list of edges, the remainder of the program has cost $O(m)$ since it iterates over all edges. Since $O(m \log m)$ is greater than $O(m)$ the algorithm has a complexity of $O(m \log m)$. We also know that a completely dense graph will have maximum n^2 edges, and it follows that

$$\begin{aligned} m &\leq n^2 \\ \log m &\leq \log n^2 \\ &\leq 2 \log n \\ \log m &= O(\log n) \end{aligned}$$

Therefore, we can confidently say that the total cost of Kruskal's Algorithm is $O(m \log n)$.

5. (a) The Decrease-Key function allows us to change the *key* value for the passed in vertex which signifies the tentative distance of the passed in vertex from the current vertex. This way, when we are at current node u, we will look at all adjacent nodes and the cost it takes to get from u to each adjacent node v. Then, if the cost of the edge from u to v is less than the *key* of vertex v and v is still in the priority queue Q, Decrease-Key will be called on v passed with the new *key* value for v as the weight of edge (u, v).

   (b) Once we have chosen the edge $(7, 1)$, our current node will be node 1 since we have just selected this edge from node 7. Then, we update the tentative distances of all unvisited neighbors, v, of node 1 (neighbors who are not part of the solution) using Decrease-Key if the *key* value of node v is more than the weight of the edge $(1, v)$. Then, after updating the *key* values of all the unvisited neighbors of node 1, we choose to visit the neighbor k with the lowest *key* value. Therefore, we have then chosen edge $(1, k)$, the edge with the lowest weight connecting the current node to an unvisited node. We will continue to update tentative distances relative to each current node as we choose edges to create our Minimum Spanning Tree.

(c) To update the tentative distance of a node v, two requirements must be satisfied. To calculate the tentative distance, we need the current node u from which the distance is calculated. Before calculating the distance, we must check to see if the vertex v is still part of the priority queue (the vertex has not be a part of the solution yet). Then, the *key* value of v must be greater than the weight of the edge (u, v). If both of these conditions are met, the tentative distance of a node v can be updated.

(d) Simulation of Prim's algorithm:

Step	tentative distance								the road you choose
	1	2	3	4	5	6	7	8	to decorate
Step 1	1	$+\infty$	$+\infty$	$+\infty$	$+\infty$	17	0	$+\infty$	(7,1)
Step 2	0	3	$+\infty$	$+\infty$	9	14	0	$+\infty$	(1,2)
Step 3	0	0	4	$+\infty$	5	14	0	$+\infty$	(2,3)
Step 4	0	0	0	2	5	14	0	$+\infty$	(3,4)
Step 5	0	0	0	0	5	8	0	7	(2,5)
Step 6	0	0	0	0	0	8	0	7	(4,8)
Step 7	0	0	0	0	0	8	0	0	(4,6)
Step 8	0	0	0	0	0	0	0	0	done
Step 9									
Step 10									
Step 11									
Step 12									

(e) The total cost of Prim's Algorithm using a binary heap as the priority queue would be $O((m+n)\log n)$. We can see how we achieve this cost from the following. First, from the starting node s, we must relax the tentative distances of each neighbor of s (the neighbor of the current node at any time will be v and the tentative distance of any node i would be $\delta(i)$) and mark s as the parent to each neighbor because we must remember what parent has resulted in the tentative distance for v. Since, initially, all tentative distances, besides that of the starting node, are ∞, all neighbors of s will be relaxed and, therefore, will have s be marked as their parent. Then for all v's of s (also the current node u), we will push the edges (u, v) onto our priority queue, which has cost $O(\log n)$. The top of the priority queue will contain the neighbor of s that has the smallest tentative distance. This will be popped from the priority queue (cost $O(\log n)$) and will become the new current node u. We will look at the neighbors of u, defined as v, and they will be classified as follows: settled, visitied, or unvisited. Node s would be settled since it has been chosen and had its child chosen. The neighbors of s are visited since their tentative distances are not ∞, and if neighbors$(u) \cap$ neighbors(s) is not empty then these neighbors of u are visited. Any neighbor of u with tentative distance ∞ means that it is unvisited. We will visit all of the neighbors of u and rest the respective nodes that have a tentative distance larger than the weight of edge (u, v), and we will mark u as the parent of all nodes that were rested by u. We will also push any unvisited nodes onto the priority queue that will now be visited due to u. This will cost $O(\log n)$ to modify an element in the binary heap and also to push an element onto the binary heap. Since we will continue this process for each edge of each node, there will be at most $(m+n)$ operations to modify elements in the binary heap. Therefore, the cost would be $O((m+n)\log n)$ for the while loop which clears the priority queue (checking each node).

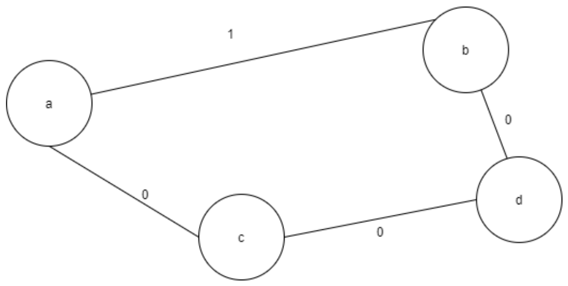
2 Will your MST or SSSP change? (10pts)

1. (a) Given a graph $G = (V, E)$, the set T is the set of edges in the MST of G. If the weight of each edge in E is doubled, resulting in a graph G', the set of edges T will still represent the MST of the new graph G'. We can use Prim's algorithm on both graphs, G and G', to prove this statement. We know Prim's algorithm will result in the set T since these are the edges of the MST of G (given in the problem). Running Prim's algorithm on G', we know that we still need to pick the same number of edges as the number of edges in T, since it will take $n - 1$ edges to connect all n nodes. We pass the initial check that T', the edges in the MST of G', must have the same size as T for the sets to be equal. Then, Prim's algorithm selects the same edges in the same order on G' as it does on G. This happens because at each node, the choice made will always be the same (whether on G or G'), since each edge's weight in G' will be just be doubled than the same edge in G. Then, the same edge (edge connecting the same two vertices) in both G and G' will have the lowest weight in their situations, so the edge will be added to both T and T', respectively. Therefore, once Prim's algorithm finishes running on G', the edges in T' will be the same exact edges in T. Thus, $T = T'$ and we can confidently say that T will be the MST of both graphs G and G' if G' is simply the graph G with the weights of all edges doubled.

   (b) We can also use the same logic from part (a) of the problem to prove that T, the MST of G, will be the MST of G' if G' is graph G with all edges' weight incremented by one. Also, we know that there must be $n-1$ edges in the MST of any graph of n nodes. Then, the weight of each spanning tree of G' will be the weight of each corresponding spanning tree of G plus a constant, $n-1$. The constant is $n-1$ because there are $n-1$ edges used in any of the spanning trees of G' and each edge has a weight of 1 more than the corresponding edge in any of the corresponding spanning trees of G. Then, if the minimum spanning tree of the original graph G contains the edges in set T, the minimum spanning tree of G' will contain the same corresponding edges in T, but the cost of the MST of G' will be the cost of the MST of G plus $n - 1$ (where n is the number of vertices, i.e. the number of elements in V). Since only the same edges are used in the MST's of both G and G', T must be the MST of both graphs.

2. (a) If each edge in the graph G has its weight doubled, the resulting graph is graph G'. Then, from any vertex $s \epsilon V$ where V is the set of vertices in G (and therefore the set of vertices in G'), the set of edges that represent the path from s to $t \epsilon V$ is P. It is obvious that if every edge weight is multiplied by a factor of 2, then every path from s to t would have a cost that is two times the cost of the corresponding path from s to t in the original graph G. Therefore, the shortest path from s to t in the original graph will be the shortest path from s to t in G' with double the cost.

   (b) We know that if each edge weight of graph G is incremented by 1, the SSSP from s to t would not necessarily be the same for the new graph G' as the SSSP for G. An example is shown below.



   From the graph seen above, currently the SSSP from a to b would be $P = \{(a, c), (c, d), (d, b)\}$. The cost of this SSSP would be 0 since the edge weight of each edge in P is 0. If each edge weight is increased by 1, the SSSP from a to b of the new graph above would be $P' = \{(a, b)\}$ because the weight of edge (a, b) would be 2 while the weight of the old path in the new graph would be 3. Therefore, P will not always be the SSSP of G if the edge weights were increased by 1.