Please enter your name and uID below.

Name: Devin White

uID: u0775759

Collaborators, if any, and how you collaborated:

Submission notes

- Due at 11:59 pm (midnight) on Thursday, Nov 17th.
- Solutions must be typeset using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for this problem set, you are allowed to collaborate in detail with your peers, as long as you cite them. However, you must write up your own solution, alone, from memory. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Borůvka's Edge Contraction)

   The algorithm to perform Boruvka's contraction is so:

   (a) initialize a graph dictionary G, and a global array that will hold the minimum edges contracted to form the MST(1 edge per vertex contracted), and a 3rd array we will call parent, which will be initialized to begin with each node being its own parent.

   (b) Parse the input, adding each vertex and edge to the graph in adjacency list format(or assume that our adjacency list input IS already a dictionary), such that each vertex has a list of directly connected adjacent vertices and their weights. For example, if v1 was connected to v2 with a weight of 1 and connected to v3 with a weight of 2, then $(v1 : (v2 : 1, v3 : 2))$ as well as $(v2 : (v1 : 1))$. and $(v3 : (v1 : 2))$ and so on

   (c) After our graph is constructed, we perform the normal boruvka O(V+E) nested loop, in order to go through every single vertex one by one, getting the minimum weight edge from each vertex, and storing them in a node array such that min[i] = the vertex and weight of the minimum edge connected to Vi, and so on.

   (d) upon finding and storing the minimum weights of each edge, we perform our contraction. The contraction can be done by looping through the original graph combining the vertices connected by minimum edges like so: We loop through the vertexes in order, looking at the vertex stored in min[v]. The adjacency lists of each of these vertices is combined such that if u→v then u is our new "super-vertex", containing all the edges originally in both U and V, at which point we can then delete v from the graph and change min[v] to 0, to represent that the minimum edge between u and v is "gone" , as its connections are already stored in u by this point. We also at this time will set parent[v] = u. This will be used later to change the edges previously belonging to V to belong to the new "super vertex U" which will be done in a later loop. Each time a vertex is contracted to U, min[u] will be incremented with the weight of the newly contracted edge.

   (e) We then loop through each vertex and each edge once again, looking for duplicated edges in our new "super" vertices by comparing the values within the node . If any duplicate edges are found, the larger weighted edge is removed, and the smaller edge is kept. While performing this loop, we also look for any edges that originally connected the new super Vertex, and remove those as well as changing any incidences of the old V to the new "Super U"(in other words, if V0 and V1 are contracted, any V1 in any other adjacency list will be altered to be V0 instead, with the same relevant lengths, with any duplicates with higher weights once again being deleted from the graph) This is done by assigning each vertex v to be parent[v], which if the vertex hasn't yet combined, will remain V, otherwise it will change to be the new super vertex U. We are removing any nodes within the dictionary that have the same parent as the key using our parent[] array. That is, if our original vertices were V0 and V1, then in our contracted vertex parent[0] = 0 and parent[1] = 0; Therefore we check if any node.vertex = parent within our dictionary values, if it does, remove it.

   Ultimately by the end of the first pass, we are left with a dictionary containing only our "super vertices" as the keys, and all edges originally owned by the 2 vertices that made up this "super vertex" as the values. with these edges represented as nodes of (target vertex, weight).

   This algorithm runs in O(V+E) time, because while there are several sequential runs of loops which are nested, checking every vertex and every edge, and manipulating our dictionary each time, adding and removing elements from a dictionary runs at a constant O(1) time in a best case, and O(v) in a worst case in the event it has to be expanded(which shouldn't ever need to happen, as it will only ever reach a V size of keys. Similarly, even though we end up performing something like 3 or 4 O(V+E) loops, O(4*(V+E)) is still ultimately O(V+E) due to constants not being a super relevant factor.

2. (Negative Bellman-Ford)

   for the questions below, the "initial bellman-ford algorithm" refers to these steps: we start with an initial array of distances,with the index of the array corresponding to the respective vertex and an array called parent, both of size V. We initialize distance[0] to 0, and then We loop through each of these vertices and initialize every other distance, from distance[1] to distance[v] to an initial value of -1, while initializing every index in parent to -1. After this, we then loop through all vertices from 1 to v-1 and all edges connected to these vertices. if $dist[v] > dist[u] + weight(e)$ then $dist[v] = dist[u] + weight(e).and parent[v] = u$.

   (a) We begin using the initial Bellman-Ford algorithm.
   It's from this point that our algorithm starts to differ from the generic bellman-ford algorithm. We loop once again though all the edges(e) and check if there are any edges that can still be relaxed, by checking if $dist(v) > dist[u]+weight(e)$, essentially repeating the same process as above. If there are any edges which can be relaxed, then there is a negative cycle.
   If there is a negative cycle
   we create a new dictionary/graph which holds the vertices in the negative cycle, and add u and v to this cycle. We then loop, while the parent[u] is not equal to v, we add parent[u] to our dictionary, and assign u = parent[u]. Once this looping operation is completed, we simply return the dictionary containing our set of vertices
   if there is no negative cycle
   we create a new graph dictionary, which contains initially all the vertices from G and empty edge lists, which we will call G2. We then loop through each vertex, if the vertex is not the starting vertex, we add the edge parent[u]→ u in G2. When we are finished, we simply return G2.
   This algorithm runs in O(V+E) time. the initial Bellman ford step loops through the vertices 2 times, which removing the constant is V, and the Edges E once. Once we get to our differing section, if a negative cycle is found, the Edges from U → V are looped through again, making 2*V + 2 * E, which is still O(V+E). If there is no negative cycle, we still loop through the edges again which is still O(V+E) run time.

   (b) As before, We begin using the initial Bellman-Ford algorithm
   It's from this point that our algorithm starts to differ from the generic bellman-ford algorithm. We loop once again though all the edges(e) and check if there are any edges that can still be relaxed, by checking if $dist(v) > dist[u]+weight(e)$, essentially repeating the same process as above. If there are any edges which can be relaxed, then there is a negative cycle.
   If there is a negative cycle
   we create a new list which holds the vertices in the negative cycle, we then loop through every edge, and check if $distance[v] > distance[u] + weight(u, v)$, if it is, we add v to our list. after doing this, we loop through each vertex in our list and perform a Depth-first-search on the vertex edges, marking each node along the way as visited and setting distance[v] as 1. We can then simply return 1.
   if there is no negative cycle
   we can simply return the distance computed from the initial Bellman-ford algorithm, as this will be the correct length of the shortest path. Note:(Edge contraction **MAY** be able to be done more easily with a graph consisting of a dictionary of dictionaries, as this will guarantee that each vertex can only ever have a single edge to any other given vertex.) This algorithm runs in O(V+E) time, the initial bellman-Ford step loops through the vertices 2 times, which removing the constant is v, and loops through the edges E once. If there is no negative cycles, it simply returns the distance, for a total of O(V+E). There is a negative cycle, we once again loop through the edges, which again

is E, and once again loop through the vertexes, which is again V. We perform a DFS on all the vertexes in our list, which DFS is known to perform O(V+E) this gives us something of a total of O(3*(V+E)) = O(V+E)