DAA Homework 5

Karthik Ragunath Ananda Kumar - KXA200005
Indukuri Vijay Anand Varma - VXI210000

Problem 1

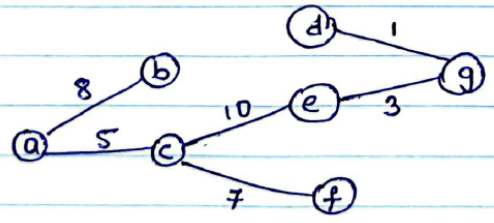
1a)

Active	Null	s	a	d	b	c	e
s	0	0	0	0	0	0	0
a	∞	3	3	3	3	3	3
b	∞	∞	7	7	7	7	7
c	∞	∞	12	10	9	9	9
d	∞	6	5	5	5	5	5
e	∞	∞	∞	12	11	10	10

1b)

s → a
a → d
a → b
b → c
c → e
s → d
a → c
d → c
d → e
b → e

1c) MST:



1d) Weight of fifth edge added when running Prim's algorithm is 3 → (e, g)
Weight of fifth edge added when running krukal's algorithm is 8 → (a, b)

Problem 2 - Updating the MST (16 points)
Let G = (V,E) be an undirected and connected graph, with edge weights w : E → R. For simplicity, assume all edge weights are distinct. You are already given the MST, T, of G. Now suppose the weight w(e) of a single edge e ∈ G is decreased to some new value w(e) −c for some c > 0. This produces a new graph weighted graph G. Show how to use T to compute the MST, T, of Gin linear time.

Algorithm:

Solution can be represented in 2 cases:

Case 1:
If the edge for which weight is reduced is already in original MST T, then modified MST T' will be same as the original MST

Case 2:
If the edge for which weight is reduced is not in original MST T, then we add the reduced edge weight back into original MST T which will form a cycle. Then, we will resolve the cycle by removing the edge with maximum weight in the cycle which can be done in linear time.

Suppose, let the edge with reduced weight is added between vertices i and j. We can perform dfs starting from vertex i and keep track of max edge weight and corresponding edge and stop once we reach vertex_j (since edge from vertex_i to vertex_j is part of cycle, we will definitely reach this case). Then the edge with maximum weight in the paths between vertex_i and vertex_j is removed.
Since we added one edge to MST from vertex_i to vertex_j, there will be two paths from vertex_i to vertex_j and we must remove the edge with maximum weight along these paths.
After the dfs traversal is completed, we can iterate through edges and remove the edge we found to have max weight in the paths from vertex_i to vertex_j.

PseudoCode:

```
DFS(cur_vertex, vertex_i, vertex_j, visited, edges):
    if edges[vertex] == None:
        return None

    # One of the base cases when there are no edges originating from a vertex
    max_edge_weight = -inf
    max_edge = None
    for each edge (vertex_dest, weight) in edges[vertex]:
        if visited[vertex_dest] == False:
            if vertex_dest != vertex_j:
            # Another base case when we reach vertex_j we no longer have to traverse
                                                                    #further
```

```
                returned_weight = DFS(vertex_dest, vertex_i, vertex_j, visited, edges)
            else:
                returned_weight = None
            local_edge = (cur_vertex, vertex_dest)
            local_edge_weight = weight
            if returned_weight == None:
                if local_edge_weight > max_edge_weight:
                    max_edge_weight = local_edge_weight
                    max_edge = local_edge
            else:
                max_edge_weight_along_path, max_edge_along_path = returned_weight[0],
    returned_weight[1]
                if  max_edge_weight_along_path > local_edge_weight and
    max_edge_weight_along_path > max_edge_weight:
                    max_edge_weight = max_edge_weight_along_path
                    max_edge = max_edge_along_path
                elif local_edge_weight > max_edge_weight_along_path and
    local_edge_weight > max_edge_weight:
                    max_edge_weight = local_edge_weight
                    max_edge = local_edge

    visited[cur_vertex] = True
    if max_edge != None:
        return (max_edge_weight, max_edge)
```

calling function:

```
max_edge_weight, edge_to_remove = DFS(vertex_i, vertex_i, vertex_j, visited, edges)
# We start at vertex_i and find the edge with max weight edge in the path from vertex_i to
vertex_j)

remove(edge_to_remove) from MST # By traversing through MST
```

Time Complexity:

Overall Time Complexity = O(1 + (V + E)) = O(V + E)
1 - to add one edge to MST T'
(V + E) - DFS to find edge with max weight along the path vertex_i to vertex_j since we added an edge between vertex_i and vertex_j

---

Problem 3 - Cycle Killer (16 points)
For both parts of this problem you are given an undirected and connected graph G = (V,E), with distinct weights on the edges. (Note the edge weights can be zero, negative, or positive.)

(a) Give an efficient algorithm to compute the maximum weight spanning tree in G.

Algorithm Logic:
To compute the Maximum Spanning Tree we just have to replace minimum priority queue used in the original version of Primm's algorithm to compute Minimum Spanning Tree with maximum priority queue (pushes maximum value to head of queue) which will give Maximum Spanning Tree.

Pseudo Code:

We start from arbitrary vertex 's' in G

```
MAX_PRIMS(s, visited, edges):
    """
    s - arbitrary source vertex
    edges - hashmap which contains tail of a vertex and weights of edges with keys being vertex origin
    visited - hashmap which indicates whether a vertex is visited or not (be default, it is initialized
              to False for all vertices
    """
    Init empty MAXIMUM priority queue, max_priority_Q
    put (None, s, 0) in max_priority_Q
    while max_priority_Q not empty:
        (parent, vertex) = max_priority_Q.pop()
        if visited[vertex] == False:
            visited[vertex] = True
            parent[vertex] = parent
            for each edge (vertex_dest, weight) in edges[vertex]:
                if visited[vertex_dest] == False:
                    put (vertex, vertex_dest, w) in max_priority_Q
return parent
```

Time Complexity:

Overall Time Complexity of Max Spanning Tree computed using Primm's Algorithm is same as Min Spanning Tree - O(E * log (V))

(b) A subset F ⊆ E is called a cycle killer set if it contains at least one edge from every cycle in G, or in other words removing F kills all cycles. Give an efficient algorithm to compute the cycle killer set with minimum total weight.

Algorithm Logic:

We can use the Max Spanning Tree algorithm given above to compute the Killer set with minimum total weights.
The cycle killer set can be computed as edges in only Graph G(V,E) but not in Max Spanning Tree MST(V, F)
Therefore, cycle_killer_set = set.difference(E, F)

Why is this correct?

This is correct because Max Spanning Tree gives you tree with MAXIMUM possible edge weights possible and adding further edges will form a cycle and hence will be part of cycle_killer_set (F)

PseudoCode:

```
def compute_cycle_killer_set(s, visited, edges):
    “””
    s - arbitrary source vertex
    edges - hashmap which contains tail of a vertex and weights of edges with keys being vertex origin
    visited - hashmap which indicates whether a vertex is visited or not (be default, it is initialized
              to False for all vertices
    “””
   parent_edges = MAX_PRIMS(s, visited, edges) # O(E * log (V))
   edges_max_MST = parent_edges.items() # O(E)
   original_edges = []
   for key, val in edges.items(): # O(E)
          original_edges.append(key, val[0])

   cycle_killer_set = difference(original_edges, edges_max_MST)
   # O(E) (if we use set difference operation which computes difference between two hashes)
   return cycle_killer_set
```

Time Complexity - O(E * log (V))

Problem 4 - SSSP (24 points)
Let G = (V,E) be a directed graph with positive edge weights, and let s be an arbitrary vertex of G. Your roommate claims to have written a program which computes the single source shortest path tree from s. Specifically, for every vertex v ∈ V , your roommate

tells you the pred(v) and dist(v) values output by the program. Unfortunately, your roommate is not that reliable and so you want to check their solution. Give an O(|V| + |E|) time algorithm to check whether the given values correctly describe some shortest path tree of G. For simplicity you can assume that all vertices in the graph are reachable from s, and hence for every vertex in your roommates solution dist(v)= ∞, and moreover s is the only vertex such that pred(s) = NULL.

Algorithm Logic:

Let G = (V, E) be a directed graph which has positive edges and let S be an arbitrary source vertex of Graph G. It also has a single source shortest path from S to every vertex in G.
The roommate provides pred(x) which gives the parent of vertex x and dist(x) which gives distance of vertex x from source. To verify the correctness of roommate's analysis we can traverse the vertices in a dfs manner and on each vertex we can check verify two conditions:

1. for a vertex v,
   dist[v] = dist[pred[v]] + weight[pred(v) -> v]
2. for each edge v->x originating from v,
   dist[x] <= dist[v] + weight(v -> x)

We would also use a boolean array to ensure that we visit each vertex only once. Hence, the total run time of the algorithm will be O(|V| + |E|)

PseudoCode:

```
def verifySSSP(v, pred, dist, weight, visited, edges):
    """
    v - current_vertex
    pred - hashmap which contains predecessor of parent of each vertex
    dist - hashmap which contains distance of each vertex from source (s)
    weight - hashmap which contains weights of edges with keys being a pair of form (u,v) where
             u indicates head of an edge and v indicates tail of an edge
    visited - hashmap which indicates whether a vertex is visited or not (be default, it is initialized
             to False for all vertices
    node - hashmap which contains details on edges starting from a vertex (v)
    """
    visited[v] = True
    if dist[v] != dist[pred[v]] + weight[(pred[v], v)]:
        return False
    for (v, x) in edges[v]:
        if dist[x] > dist[v] + weight[(v,x)]:
            return False
        if visited[x] == False:
            verifySSSP(x, pred, dist, weight, visited, edges)
    return True
```

```
verified_val = verifySSSP(source_vertex, pred, dist, weight, visited, edges)

if verified_val == True:
    print("Roommate has provided the right info on pred(v) and dist(v) for each each vertex v, hence he is reliable")

else:
    print("Roommate has not provided the right info on pred(v) and dist(v) for each each vertex v, hence he is unreliable")
```

If we call verifySSSP starting from the source vertex, if the function returns True value, then the roommate's info on pred(v) and dist(v) for all vertices v in Graph G is reliable, else he is not reliable.

Time Complexity:
O(|V| + |E|) since each edge originating from each vertex is checked only once.

Problem 5 - Best Edge (24 points)

Let G =(V,E) be a directed graph with positive edge weights, let s and t be vertices of G, and let H = (V,F) be a subgraph of G obtained by deleting some edges, that is F $\subset$ E. Suppose we want to reinsert exactly one edge from E \ F back into H, so that the shortest path from s to t in the resulting graph is as short as possible. Describe and analyze an algorithm that chooses the best edge to reinsert, in O(|E|log|V |) time.

Algorithm Logic:

Let G = (V, E) be a directed graph with positive edge weights.
Let s and t be arbitrary source and destination vertices of the Graph G.

Let H = (V, F) be a sub-graph obtained by deleting some edges from G, hence F is a subset of E.

We want to reinsert exactly one edge from E into H such that it minimizes the shortest path from s to t in the resulting graph.

Let there are n edges which are present in E and not in H, E \ F given by -
[(u1, v1), (u2,v2), …., (un, vn)]

To find the best possible from E \ F which minimizes the distance from s to t in the resulting graph, we can do the following:

1. Find the shortest path distance from source 's' to all vertices from where edges originate in

the list of edges in E \ F and create a hashmap (hashmap_s) of distance from source.
Basically, we create a hashmap of distance from s to u1, s to u2, …, s to un

To perform this, we can perform a Dijkstra's SSSP algorithm from source 's' in Graph H to all vertices u1, u2, .., un and store their distances in a hashmap.

2. Similarly we can reverse edges in Graph G and perform Dijkstra's SSSP algorithm from 't' in Graph H to all vertices v1, v2, …, vn in E \ F edge list and store their distances in a hashmap (hashmap_t).

3. Then we can find the best edge to add from the list of edges in E \ F by finding the edge which has
min_value {hashmap_s[u] + weight(u -> v) + hashmap_t[v]}

Why is this correct?

The above mentioned algorithm is correct because the best case edge depends not only on its weight but the shortest distance of each edge's head in E \ F set (u1, u2, …, un) from the source, the weight of the edge in itself (weight(u1 -> v1), weight(u2 -> v2), …, weight(un -> vn)) and the shortest distance from each edge's tail in E \ F set (v1, v2, …, vn) to tail.
Thus, the best case edge is computed using min_value {hashmap_s[u] + weight(u -> v) + hashmap_t[v]} among all edges in the E \ F set.

PseudoCode:

```
def best_edge_to_insert(s, t, H, weight, left_over_edges):
    “””
    s - source vertex
    t - destination vertex
    H (V, F) - subgraph formed from G (V, E) by removing edges from it such that F is subset of E
    left_over_edges - edges in the set E \ F
    weight - hashmap with keys as pairs (u, v) which contains weights of edges from u -> v as values
    “””
    hashmap_s = Dijkstra(H, s) # performing dijkstra on Graph H starting from ‘s’
    reversed_graph = reverse_edges(H)
    hashmap_t = Dijkstra(reversed_graph, t) # performing dijkstra on reversed_graph from ‘t’
    min_combo_weight = inf
    best_edge = None
    for (u, v) in left_over_edges:
        local_weight = hashmap_s[u] + weight[(u,v)] + hashmap_t[v]
        if min_combo_weight > local_weight:
            local_weight = min_combo_weight
            best_edge = (u, v)
```

return best_edge

Time Complexity:
O(2* E * log (V) + E + E)
where 2* E * log (V) - performing Dijkstra twice on graph G starting from ‘s’ and graph G with edges reversed starting from ‘t’
E - performing edge reversal operation on G
E - loop through edges in set E \ G

Therefore, overall time complexity is O(E * log(V))
