DAA Assignment 4

1. Purpose: Learn about Euler tours, practice describing algorithms. Please solve Problem 22-3 (12 points) on page 623.

a. Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex v ∈V.

A cycle that traverses each edge of Graph G exactly once is called an Euler tour. But the tour might not be a simple cycle all the time. It can be decomposed into a set of edge-disjoint simple cycles that follow Euler tour until a vertex is encountered for the second time. With this Euler tour completes a first simple cycle. And if the Euler tour is not yet completed, continue the same process by removing the edges of the first cycle. This process can be repeated multiple times until the entire Euler tour has been completed. In each of the resulting simple cycles each vertex v has inDegree(v)=outDegree(v)=1 indicating one incoming edge and one outgoing edge and isolated vertices have inDegree(v)=outDegree(v)=0 indicating no incoming and outgoing edge. Adding all the in and out degrees over all cycles and isolated vertices proves that if Graph G has an Euler tour, then the inDegree(v)=outDegree(v) for all the vertices v.

We prove the same in reverse that if inDegree(v)=outDegree(v) for all vertices v, then we can pick up any vertex u for which inDegree(u)=outDegree(u)>=1 and can create a cycle that contains u and traverses every edge exactly once. To prove this claim, let us start by placing vertex u on the cycle and choose any leaving edge of u, for example (u, v). Now we add v on the cycle, since inDegree(v) = outDegree(v)>=1. We can pick some leaving edge of v and continue visiting edges and vertices. Each time we pick an edge, we can exclude it from further consideration. At each vertex other than u, at the time we visit an incoming edge, there must be an unvisited outgoing edge, since inDegree(v) = outDegree(v) for all vertices v. The only vertex for which there might not be an unvisited outgoing edge is u, since we started the cycle by visiting one of u's outgoing edges. Since there's always an outgoing edge we can leave from all vertices other than u, and eventually the cycle must return and terminate in u. During this procedure, we have removed an equal number of in and out edges at every vertex of the cycle. If there are still edges left in the graph we can choose a vertex x on our first cycle that still has undeleted edges. If no such vertex exists, then the graph is not connected which is in contrast to our assumption. Following the same arguments as above we can construct a second cycle, third cycle and so on and splice it together with the first cycle until no further edges remain. This way we can prove that G has an Euler tour.

b. Describe an O(E)-time algorithm to find an Euler tour of G if one exists. (Hint: Merge edge-disjoint cycles.)

PseudoCode:

```
eulerTour(G):
    D = {}                                          // empty list
    S = (any vertex v ∈G.V, NIL)
    while S is not empty
        S.remove(v, location in D)
        C = visit(G, S, v)
        if location in D == NIL
            D = C
        else join C into D just before location in D
    return D

visit(G,S,v)
    C = {} //empty sequence of vertices
    u = v
    while outDegree(u) > 0
        let w be the first vertex in G.Adj[u]
        G.Adj[u].remove(w)                    // remove w from adjacency list of u
        outDegree(u)--                        // decrementing outDegree of u
        C.add(u)                              // add u on to end of C
        if out-degree(u) > 0
            S.add(u, u's location in C)
        u = w
    return C
```

Textual Description:

Let G be a graph represented by adjacency lists and we work with a copy of the adjacency lists, so that as we keep visiting each edge we can remove it from its adjacency list. The singly linked list form of the adjacency list will solve our purpose. The output of this algorithm is a doubly linked list D of vertices which on reading in list order will give an Euler tour. The algorithm constructs D by finding cycles and joining them into D. By using doubly linked lists for cycles and the Euler tour, joining a cycle into the Euler tour takes constant time. We also maintain a singly linked list S, in which each list element consists of two parts: A vertex v and a pointer to some appearance of S. Initially, S contains one vertex, which may be any vertex of G.

Proof of Correctness:

Loop Invariant: In each iteration the euler tour finds a simple cycle and merges it into the larger cycle D. With this we will be able to see if a euler tour can be performed for all the vertices in the graph.

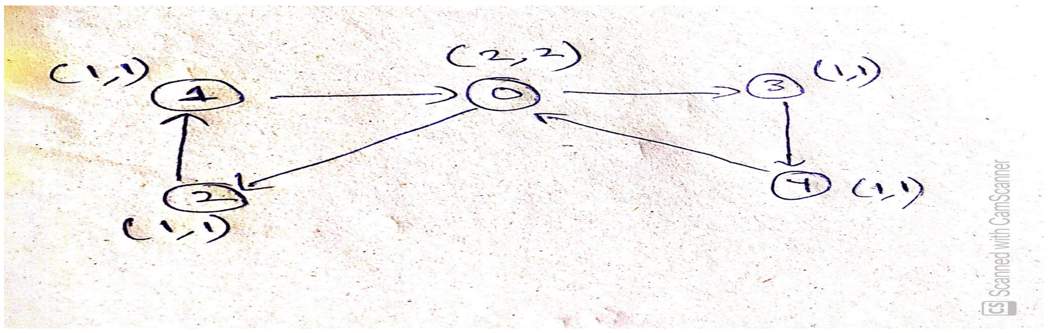
Initialization: Initially, The use of NIL in the initial assignment to S ensures that the first cycle C returned by visit(G,S,v) becomes the current version of the Euler tour D.

Maintenance: All cycles returned by visit(G,S,v) thereafter are joined into D. We assume that whenever an empty cycle is returned by visit(G,S,v), joining it into D leaves D unchanged. Each time that eulerTour(G) removes a vertex v from the list S, it calls visit(G, S, v) to find a cycle C, possibly empty and not simple, that starts and ends at v. The cycle C is represented by a list that starts with v and ends with the last vertex on the cycle before the cycle ends at v. eulerTour(G) then joins this cycle C into the euler tour D just before some appearance of v in S. When visit(G,S,v) is at a vertex u, it looks for some vertex w such that the edge (u, w) has not yet been visited. Removing w from Adj[u] ensures that we will never visit (u, w) again. visit(G,S,v) adds u onto the cycle C that it constructs. After removing edge (u, w), if vertex u still has any outgoing edges then u along with its location in C, is added to S.

Termination: The cycle construction continues from w, and it ceases once a vertex with no unvisited leaving edges is found. Using the argument from part (a) at that point, this vertex must close up a cycle. At that point, the cycle C is returned. It is possible that a vertex u has unvisited leaving edges at the time it is added to list S in visit(G,S,v), but by the time that u is removed from L in eulerTour(G), all of its outgoing edges have been visited. In this case, the while loop of visit(G,S,v) does not execute, and visit(G,S,v) returns an empty cycle.

Example:
Consider the following adjacency matrix.
[[0,1,0,0,0],
[0,0,1,0,0],
[0,0,0,1,1],
[1,0,0,0,0],
[0,0,1,0,0]]

This above matrix when represented as a graph with both indegree and outdegree of each vertex, looks like the below diagram. From the diagram it is evident that for every vertex has inDegree(u)=outDegree(u) which indicates that euler tour is found for this graph according to the proof in part(a). Hence this is a graph G which has a valid Euler-Tour.



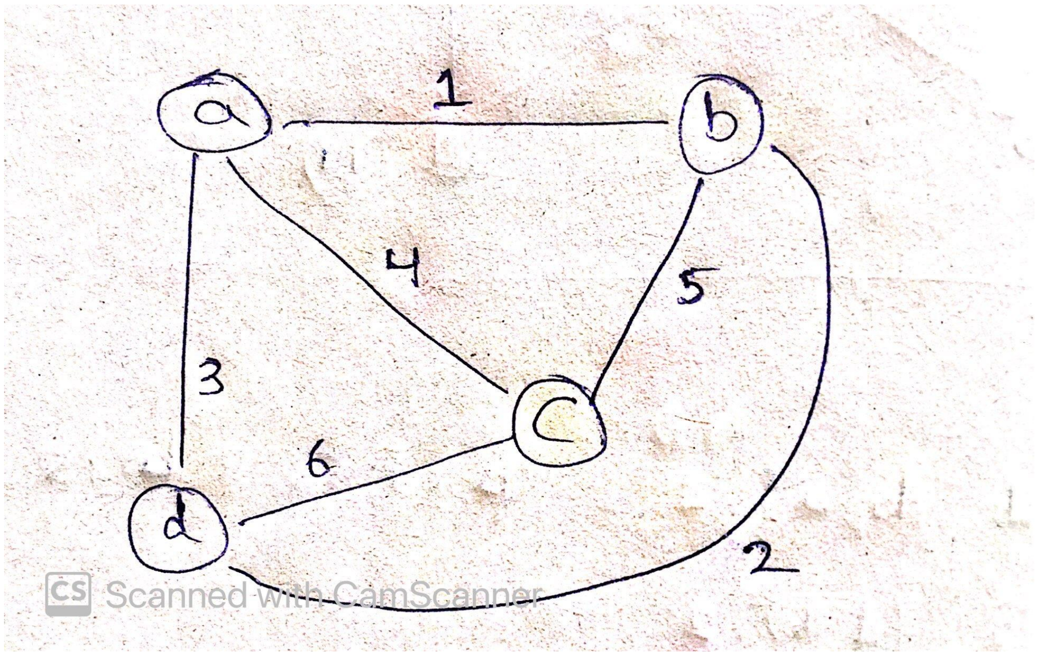
Time Complexity:
Once the list S is empty it indicates that every edge has been visited. The resulting cycle D is then an Euler tour. To see that eulerTour(G) takes O(E) time, observe that because we remove each edge from its adjacency list as it is visited, no edge is visited more than once. Since each edge is visited at some time, the number of times that a vertex is added to S and then removed from S is at most |E|. Thus, the while loop in eulerTour(G) executes atmost E iterations. The while loop in visit(G,S,v) executes one iteration per edge in the graph, and so it executes at most E iterations as well. Since adding vertex u to the doubly linked list C takes constant time and then joining C into D takes constant time. Thus the entire algorithm takes O(E) time.

Space Complexity: The space complexity of this algorithm is also O(E).

2. Purpose: Reinforce your understanding of MST algorithms, and practice algorithm design (16 points). Let T be the Minimum Spanning Tree of a graph G=(V,E,w). Suppose g is connected, |E|≥|V|, and that all edge-weights are distinct. Denote T* the MST of G and ST(G) be the set of all spanning trees of G. A second-best MST is a spanning tree T such that w(T) =min{w(T): T ε ST(G)-{T*}}.

a. (4 points) Show that T* is unique, but that the second-best MST T2 need not be unique.

Yes, the second-best MST T2 need not be unique. To see that the second best minimum spanning tree need not be unique, we consider the following example graph on four vertices. Suppose the vertices are {a,b,c,d}, and the edge weights are as follows:



Then, the minimum spanning tree has weight 7, but there are two spanning trees of the second best weight, 8 as shown in the below figure.

(a, b) : 1, (b, a) : 1 (b, c) : 5, (c, b) : 5

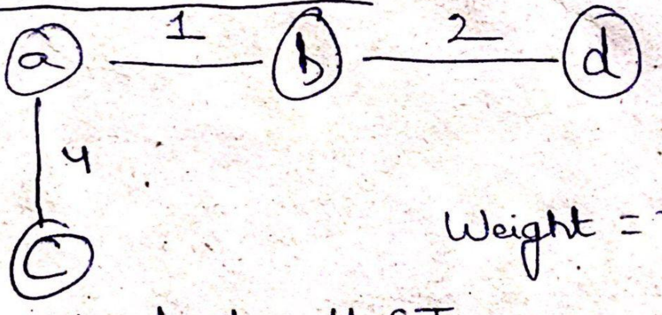
(b, d) : 2, (d, b) : 2

(a, d) : 3, (d, a) : 3

(a, c) : 4, (c, a) : 4

(c, d) : 6, (d, c) : 6

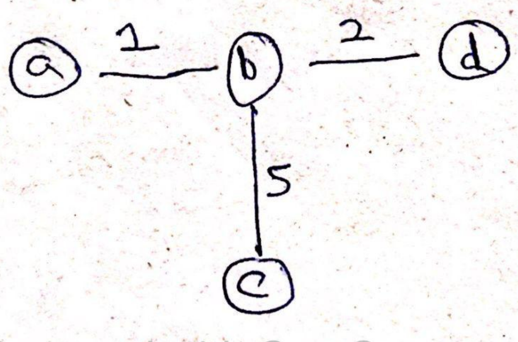
first best M.S.T

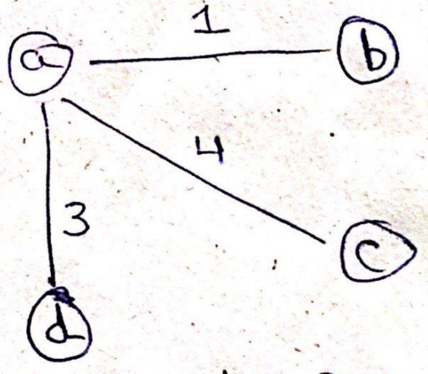
① 

Weight = 7.

Second best M.S.T

②





Weight = 8 Weight = 8

b. (4 points) Prove that G contains an edge (u,v) ε T* and another edge (s,t) ∉ T* such that (T*-{(u,v)}) ∪ {(s,t)} is a second-best minimum spanning tree of G.

Yes. In order to achieve a second-best minimum spanning tree we should find an edge (s,t) which is not in T* and replace it with an edge in T* say (u,v) such that (T*-{(u,v)}) ∪ {(s,t)} is a spanning tree and weight difference of {(s,t) - (u,v)} is minimum otherwise it is not possible. We are trying to show that there is a single edge swap that can demote our minimum spanning tree to a second best minimum spanning tree. In obtaining the second best minimum spanning tree, there must be some cut of a single vertex away from the rest for which the edge that is added is not light, otherwise, we would find the minimum spanning tree, not the second best minimum spanning tree. Call the edge that is selected for that cut for the second best minimum spanning tree (s, t). Now, consider the same cut, except look at the edge that was selected when obtaining T*, call it (u,v). Then, we have that if consider T*−{(u,v)}∪{(s,t)}, it will be a second best minimum spanning tree. This is because if the second best minimum spanning tree also selected a non-light edge for another cut, it would end up more expensive than all the minimum spanning trees. This means that we need for every cut other than the one that the selected edge was light. This means that the choices all align with what the minimum spanning tree was.

c. (6 points) Please use Kruskal's algorithm to design an efficient algorithm to compute the second-best minimum spanning tree of G.

PseudoCode:

```
Kruskal_MST(G):
    sum = 0
    A <- Φ
    for each vertex v do
        MAKE-SET(v)
    Sort E by weight
    for each edge (u, v) ∈E taken in
        Nondecreasing order of weight do
            If FIND-SET(u) != FIND-SET(v)
            then
                A<- A U {(u,v)}
                sum += UNION(u,v)
    T =(V,A)
    return (T, sum)
```

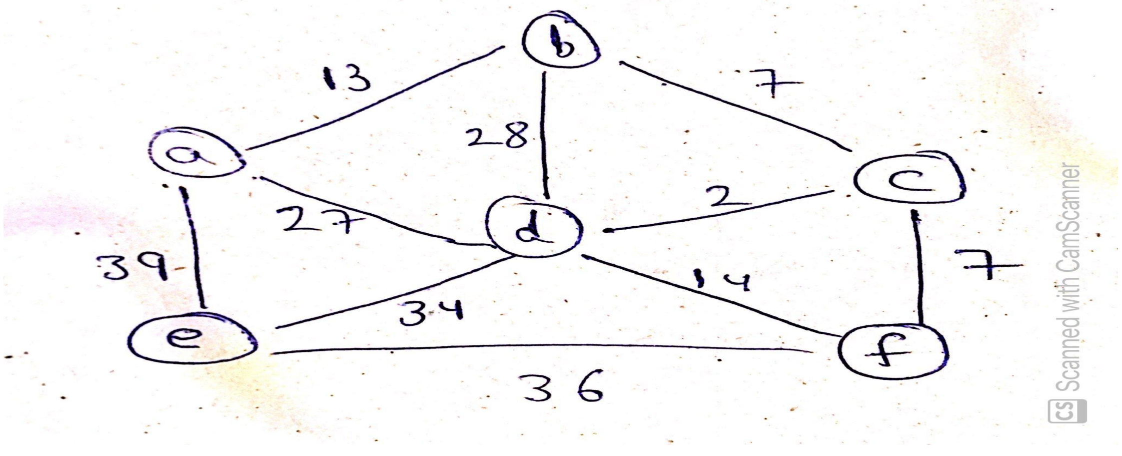
```
T = Kruskal_MST(G)
sum = 0
second_best_sum = INT_MAX
second_best_tree = T
for each edge (u,v) in T do
    original_Graph = G
    original_Graph.remove((u,v))
    T.remove((u,v))
    MST, sum = Kruskal_MST(original_Graph)
    if (second_best_sum > sum)
        second_best_tree, second_best_sum = MST, sum
    sum = 0
print(second_best_tree)
print(second_best_sum)
```

Explanation:

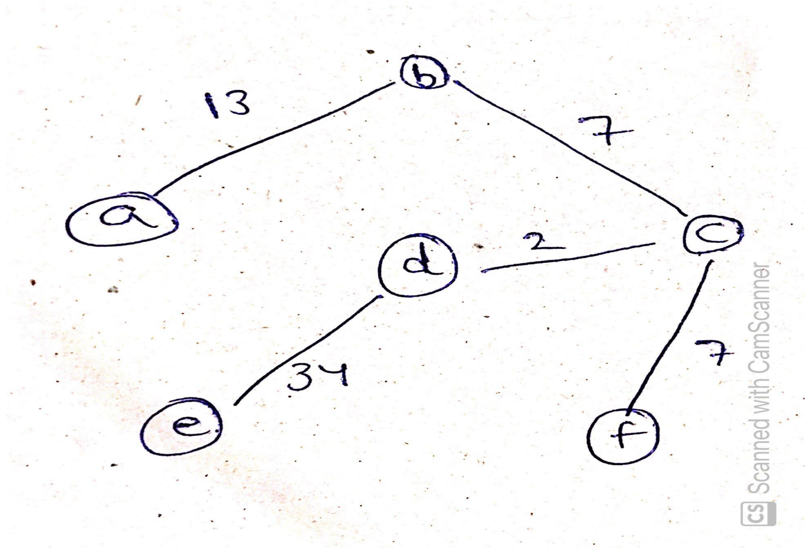
In the above algorithm we use kruskal's algorithm to calculate the minimal spanning tree T for the given graph G. Once the minimum spanning tree is calculated, we pick each edge from the minimum spanning tree, temporarily we exclude if from the main graph edge list and calculate the new MST for the main graph using remaining edges. Similarly we repeat this process considering all the edges in the MST and we take the best one with the 2nd minimum weight sum. This gives us the second minimum spanning tree.

Example:

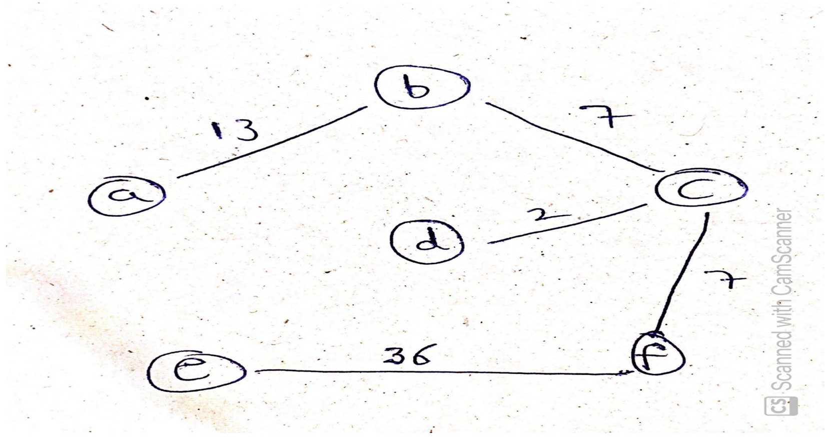
Consider the below tree.



Minimum spanning tree is of length 63.



The second best spanning tree is of length 65.



Proof of correctness:

Loop Invariant: The algorithm in each iteration tries to find a minimum spanning tree using kruskal's algorithm and keeps track of the minimal one among all the iterations.

Initialization: Initially we find the minimum spanning tree for the given graph using kruskal's algorithm and assume that this is the second-best minimum spanning tree too.

Maintenance: During the execution of each iteration which is done by excluding each edge from the MST we find the minimum spanning tree in the original graph with remaining edges. And we keep on doing the same process considering all the edges in the MST and keep track of the best

MST among all the solutions. This way we will be able to find the second-best minimum spanning tree.

Termination: Once all the iterations are done we will be left out with the second best minimum spanning tree with all the connecting vertices and sum of their weights.

Time Complexity: In this algorithm we sort all the edges in O(Elog(V)) time and find the minimal spanning tree in O(E) time. Then for each edge in the MST we temporarily exclude if from the edge list so that it cannot be chosen and again try to find MST using remaining edges. We do this same process for all the edges in O(VE) time. So, the overall time complexity will be O(ElogV+E+VE) = O(VE).

Space Complexity: Space complexity of this algorithm is O(|E| + |V|), since disjoint set data structure takes O(|V|) space to keep track of the roots of all the vertices and another O(|E|) space to store all edges in a sorted manner.

3. Purpose: Reinforce your understanding of Dijkstra's shortest path algorithm, and practice algorithm design (16 points). Suppose you have a weighted, undirected graph G with positive edge weights and a start vertex s.

a. (6 points) Describe a modification of Dijkstra's algorithm that runs (asymptotically) as fast as the original algorithm, and assigns a binary value usp[u] to every vertex u in G, so that usp[u]=1 if and only if there is **a unique shortest path from s to u**. We set usp[s]=1. In addition to your modification, be sure to provide arguments for both the correctness and time bound of your algorithm, and an example.

Explanation:
As discussed in the class we can update the unique shortest path from s to u .i.e usp[u] during the RELAX operation. In order to do that we will have to consider the following three cases during RELAX(u,v):

1) minDistance[u] + weight(u,v) < minDistance[v]: This condition in the code indicates that a new and shorter path to vertex v is found from the starting vertex. But this will be unique only if the current shortest path to vertex u is also unique. So we update the usp[v] as usp[u] when this condition is satisfied.
2) minDistance[u] + weight(u,v) = minDistance[v]: Here in this case a shortest path of same length is found to the vertex v from the starting vertex s. Since this has appeared twice and is not unique we update the usp[v] as 0.
3) minDistance[u] + weight(u,v) > minDistance[v]: Since the new path is not shorter than the existing minimum distance to vertex v, we will not consider updating usp[v] in this

case.

Before calling the RELAX(u,v) we initialize the usp[s] to 1 which indicates true, and basically means that a unique shortest path to vertex s exists since it is the starting vertex and will rewrite the RELAX function as follows for the remaining vertices:

PseudoCode:

```
RELAX(u, v)
1   if minDistance[u] + weight(u,v) < minDistance[v]  then
2       minDistance[v] = minDistance[u] + weight(u,v)
3       usp[v] = usp[u]
4   else if minDistance[u] + weight(u,v) == minDistance[v]  then
5       usp[v] = 0
6   endif
end RELAX
```

Proof of Correctness:
Loop Invariant Condition: At any point of time in the execution of the Dijkstra's algorithm, usp[u] is 1 i.e true if and only if there is exactly one path of weight minDistance[u] from s to u. If there are duplicate paths then usp[u] is 0 .i.e false.

Initialization: Initially we initialize the usp[s] as 1 which indicates true, and basically means that a unique shortest path to vertex s exists since it is the starting vertex. And also for the remaining vertices we initialize usp[u] as 0 assuming that the vertex doesn't have a unique shortest path from s and will update later if there is a change during the execution of the algorithm.

Maintenance: During the execution of the algorithm since minDistance[u] = δ(s, u) when u is added to a set of vertices to which shortest path from s is known(S), usp[u] will be 1 .i.e true if there is a unique shortest path from s to u and usp[u] = 0 if there is no unique shortest path from s to u. Similarly the algorithm executes for all the vertices and updates the corresponding value of usp[u] for each vertex in the given undirected graph.

Termination: At the end of the algorithm, we will have usp[u] filled with either 1 or 0 for all the vertices which indicates if there exists a unique shortest path or not between the starting vertex and that particular vertex u selected.

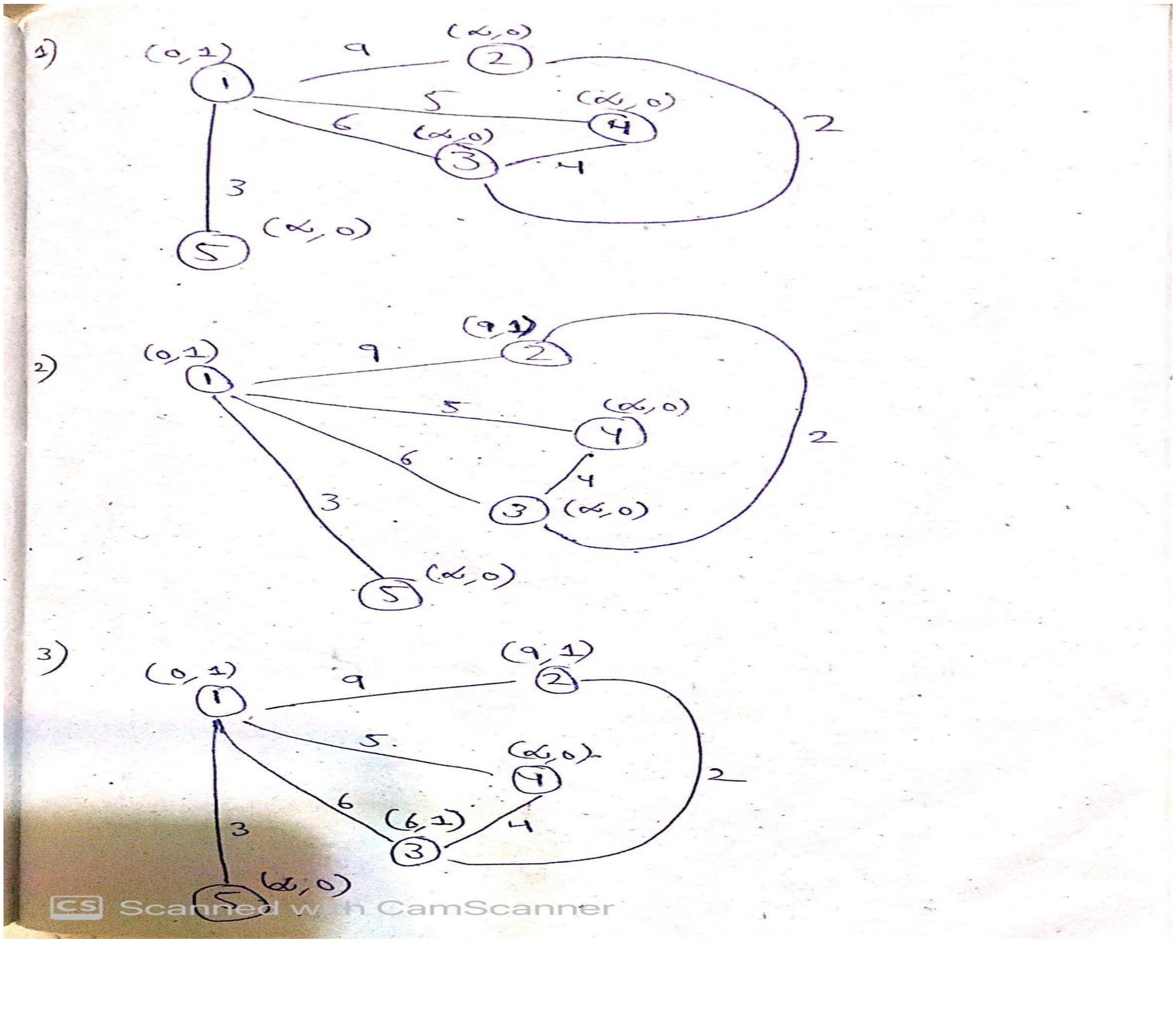
Time Complexity: The time complexity of this algorithm is exactly the same as the original Dijkstra algorithm which is O(V + ElogV) where V is the number of vertices and E is the number of Edges. The only modification we've made is to add a constant amount of operations in RELAX(u, v), which would have no impact on the time complexity of the original Dijkstra's algorithm. Therefore, the time bound for the whole algorithm would remain unchanged.

Space Complexity: The Disjkstra's shortest path algorithm implementation using a MinHeap , which in turn uses an array for storing the heap of vertices V, and also we use an array to store the values of the shortest distance for every node in the graph, so the space complexity will be O(V) + O(V) = O(2V) which is equivalent to O(V).

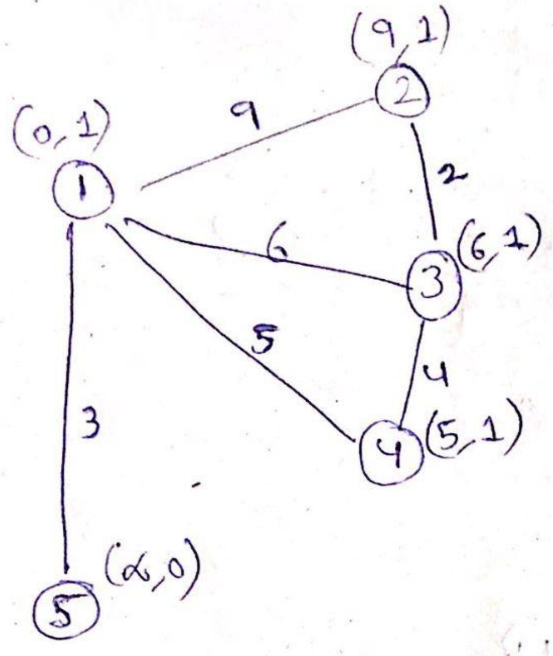
Example:
Let us consider an undirected graph with the vertices as below:
Weighted Adjacency Matrix = [[1, 2, 9], [1, 3, 6], [1, 4, 5], [1, 5, 3], [3, 2, 2], [3, 4, 4]]

The above vertices in [u,v,w] format where u is source vertex, v is destination vertex and w is weight of the vertex. The below diagrams show the values of (d[u], ups[u]) for all the vertices in each step of Dijkstra's algorithm, where d[u] is the shortest path distance from starting vertex to u at that point of time and ups[u] is a value either 0 or 1 which indicates if a unique shortest path exists for a vertex u from the start vertex s or not.

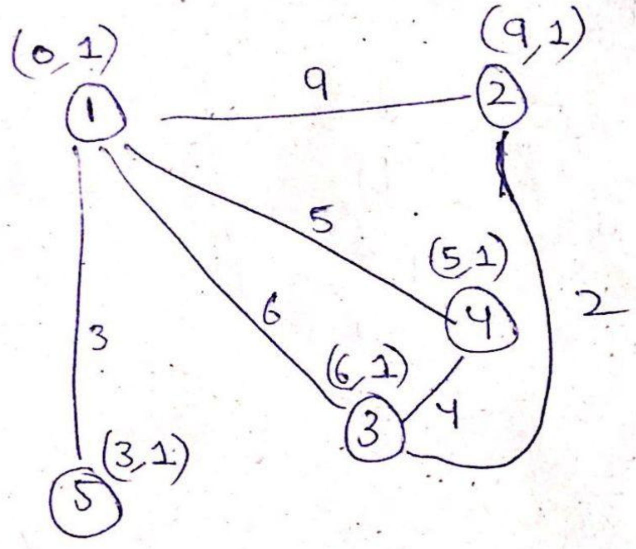
So the final answer for this problem is: [[0, 1], [8, 1], [6, 1], [5, 1], [3, 1]]



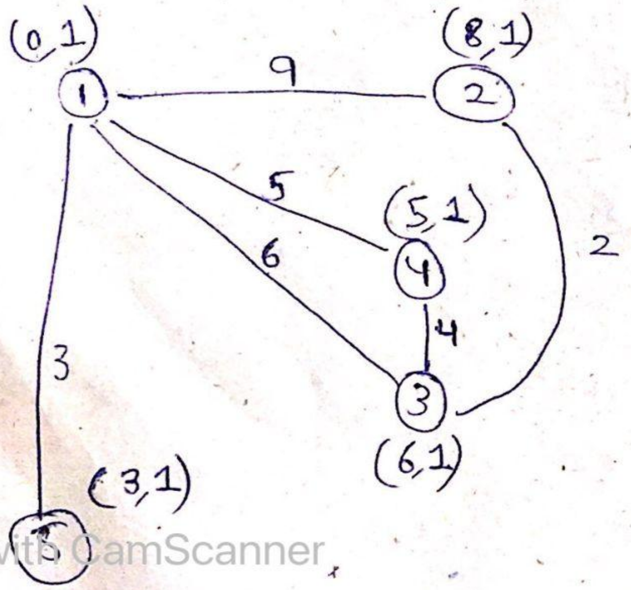
4)


(0,1)
1
9
(9,1)
2
2
6
(6,1)
3
5
4
3
(5,1)
4
(∝,0)
5

5)


(0,1)
1
9
(9,1)
2
5
(5,1)
4
2
6
3
(6,1)
3
4
(3,1)
5

6)


(0,1)
1
9
(8,1)
2
5
(5,1)
4
6
2
4
3
3
(6,1)
(3,1)
5