DAA HW 4

Problem 1

An Euler tour of a strongly connected, directed graph G=(V, E) is a cycle that traverses each edge of G exactly once, although it may visit each vertex more than once.

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

Ans:

To prove the given statement let's use the proof by contradiction.

1) Case 1 - Consider the case where the in-degree is greater than the out-degree for some vertex v$\in$V.

   In this case, at some point of time all the outgoing edges will be visited but some incoming edges would be left to be visited. If we start from any arbitrary vertex v, as there are fewer outgoing edges, after some point all the outgoing edges will be exhausted and some incoming edges will still be left for exploration at some vertex. To visit those we need to traverse through the already visited outgoing edges from that vertex. This clearly doesn't follow the concept of Euler's tour. Thus an euler tour is not possible if the in-degree of vertex is greater than the out-degree of vertex for all the vertices in the graph.

2) Case 2 - Consider the case where the in-degree is lesser than the out-degree for some vertex v$\in$V.

   In this case, the incoming degree of the vertex is less than the out-degree of the vertex. This case just seems like a transpose of a graph and we are traversing backward in the original graph. Since the incoming edges are less, after some point we would have visited all the incoming edges to a vertex and there would be some outgoing edges from the vertex which are not yet visited. So to visit those remaining outgoing edges, the only way to reach that vertex is by traversing the already visited incoming edge. Thus we will get stuck and hence it is not possible to return to the source vertex. Because to reach there we might have to travel through the list of already visited edges. This violates the concept of Euler's tour.

3) After analyzing both the cases, our assumptions are wrong. So for any vertex v$\in$V, we cannot have an unequal number of in-degree and out-degree.
4) Let's prove that G has an Euler tour if in-degree(v) = out-degree(v) for all vertices v.

   We can pick any vertex u for which in-degree(u) = out-degree(u)>=1 and create a cycle (not necessarily simple) that contains u and traverses every edge exactly once. To prove this claim, let us start from vertex u on the cycle, and choose any

outgoing edge of u, say (u, v). Now we put v on the cycle. Since in-degree(v) = out-degree(v)>=1. We can pick some outgoing edge of v and continue visiting edges and vertices. Each time we pick an edge, we can remove it from further consideration. At each vertex other than u, we are considering an equal number of incoming edges and outgoing edges, since in-degree(v) = out-degree(v) for all vertices v. Since we started the cycle by visiting one of u's leaving edges, to complete the cycle there has to be one incoming edge to vertex u. Hence in-degree(u) = out-degree(u) is also true. 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 u2 on our first cycle that still has un-deleted edges - if no such vertex exists the graph is not connected, in contrast to our assumption. Following the same arguments as above we can construct a second (third, ...) cycle and splice it together with the first cycle until no further edges remain.

5) Hence, a graph G has a Euler tour if and only if in-degree(v) = out-degree(v) for each vertex v belongs to V.

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

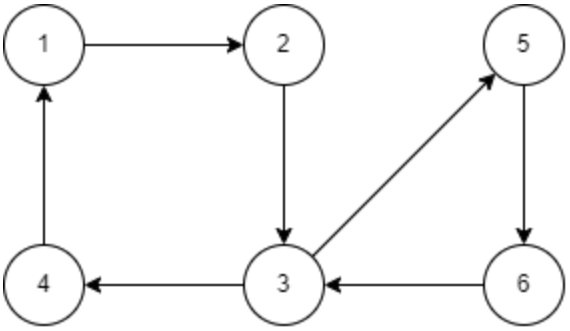
Ans:

1) Pseudo code -

```
Euler-Tour(G)
      Initialize an empty stack S to store vertices
      Initialize empty Edge_list to store edges to visit in Euler tour
      Find-Euler-Tour(G, S)
      u = S.pop()
      while S is not empty
            v = S.pop()
            Edge_list.append((u, v))
            u = v
      return Edge_list

Find-Euler-Tour(G, S)
      u = choose any random vertex from graph G
      current_path = [u]
      while current_path do
            u = current_path[len(current_path)-1]
            if Adj[u] is not empty
                  v = Adj[u].pop()
                  current_path.append(v)
            else
                  S.push(u)
```

```
                current_path.pop()
```

2) Textual description -
    a) Assumptions -
        i) Graph is a directed graph
        ii) Stack S is passed by reference to the function Find-Euler-Tour
        iii) The algorithm assumes that given graph has Euler Tour
    b) In this algorithm, stack S is used to store vertices in reverse order to visit. When we get the final stack S containing all the vertices after executing the Find-Euler-Tour algorithm, we can get the Euler tour stored in Edge_list by popping each vertex from the stack and adding an edge between the top vertex and its consecutive vertex from the stack.
    c) The Find-Euler-Tour algorithm is used to find the sequence of vertices to form a Euler tour.
        i) u is any random vertex from the Graph
        ii) Current_path is used to store the vertices whose adjacency list is not empty.
        iii) If the adjacency list of a particular vertex is not empty that means we have not traversed all the incoming and outgoing edges from that vertex.
        iv) For a vertex, if the whole adjacency list is visited then we can add that vertex to the stack S and remove it from the current_path.
        v) In the end, we get all the adjacency lists of vertices are empty and current_path empty. Hence, our algorithm terminates. And stack S contains all the vertices in the order in which the Euler tour forms.
    d) Algorithm Euler-Tour -
        i) Initialize empty stack S and empty edge list to store edges from Euler tour
        ii) Find Euler tour vertices sequence
        iii) For each vertex u from the stack S, add an edge from (u, v) in the edge_list, where v is the next vertex in the stack.
        iv) Return Edge_list
    e) Algorithm Find-Euler-Tour -
        i) Select any random vertex u from the Graph.
        ii) Add vertex u to the current_path list
        iii) While the current_path list is not empty, get vertex u as the last vertex from current_path list.
        if adjacency list of vertex u is not empty, then remove one vertex from the adjacency list of vertex u and append that vertex to the current_path list.
        Else, push vertex u on the top of the stack S
3) Example -



a) Let the adjacency list for a graph G be as follows -
   Adjacency List - vertex 1 - [2]
   vertex 2 - [3]
   vertex 3 - [4, 5]
   vertex 4 - [1]
   vertex 5 - [6]
   vertex 6 - [3]
   Initially, Stack S is empty.
   Edge_list = []

b) In algorithm Find-Euler-Tour,
   Let u = vertex 1,
   current_path = [1]

c) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 1
   ii) Adj[1] is not empty,
   Hence, v = Adj[1].pop() = 2
   iii) current_path = [1, 2] and
   Adj[1] becomes empty

d) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 2
   ii) Adj[2] is not empty,
   Hence, v = Adj[2].pop() = 3
   iii) current_path = [1, 2, 3] and
   Adj[2] becomes empty

e) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 3
   ii) Adj[3] is not empty,
   Hence, v = Adj[3].pop() = 5
   iii) current_path = [1, 2, 3, 5] and
   Adj[3] becomes [4]

f) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 5
   ii) Adj[5] is not empty,
   Hence, v = Adj[5].pop() = 6
   iii) current_path = [1, 2, 3, 5, 6] and

Adj[5] becomes empty

g) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 6
   ii) Adj[6] is not empty,
   Hence, v = Adj[6].pop() = 3
   iii) current_path = [1, 2, 3, 5, 6, 3] and
   Adj[6] becomes empty

h) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 3
   ii) Adj[3] is not empty,
   Hence, v = Adj[3].pop() = 4
   iii) current_path = [1, 2, 3, 5, 6, 3, 4] and
   Adj[3] becomes empty

i) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 4
   ii) Adj[4] is not empty,
   Hence, v = Adj[4].pop() = 1
   iii) current_path = [1, 2, 3, 5, 6, 3, 4, 1] and
   Adj[4] becomes empty

j) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 1
   ii) Adj[1] is empty,
   Hence, S = {1}
   iii) current_path = [1, 2, 3, 5, 6, 3, 4]

k) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 4
   ii) Adj[4] is empty,
   Hence, S = {1, 4}
   iii) current_path = [1, 2, 3, 5, 6, 3]

l) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 3
   ii) Adj[3] is empty,
   Hence, S = {1, 4, 3}
   iii) current_path = [1, 2, 3, 5, 6]

m) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 6
   ii) Adj[6] is empty,
   Hence, S = {1, 4, 3, 6}
   iii) current_path = [1, 2, 3, 5]

n) Current_path is not empty, so in a while loop,
   i) u = current_path[-1] = 5
   ii) Adj[5] is empty,
   Hence, S = {1, 4, 3, 6, 5}
   iii) current_path = [1, 2, 3]

o) Current_path is not empty, so in a while loop,
    i) u = current_path[-1] = 3
    ii) Adj[3] is empty,
        Hence, S = {1, 4, 3, 6, 5, 3}
    iii) current_path = [1, 2]
p) Current_path is not empty, so in a while loop,
    i) u = current_path[-1] = 2
    ii) Adj[2] is empty,
        Hence, S = {1, 4, 3, 6, 5, 3, 2}
    iii) current_path = [1]
q) Current_path is not empty, so in a while loop,
    i) u = current_path[-1] = 1
    ii) Adj[1] is empty,
        Hence, S = {1, 4, 3, 6, 5, 3, 2, 1}
    iii) current_path = []
r) Current_path becomes empty, so we go back to the driver function Euler-Tour,
   u = S.pop() = 1
   S = {1, 4, 3, 6, 5, 3, 2}
s) S is not empty,
    i) v = S.pop() = 2
    ii) Edge_list = [(1, 2)]
    iii) u = 2
    iv) S = {1, 4, 3, 6, 5, 3}
t) S is not empty,
    i) v = S.pop() = 3
    ii) Edge_list = [(1, 2), (2, 3)]
    iii) u = 3
    iv) S = {1, 4, 3, 6, 5}
u) S is not empty,
    i) v = S.pop() = 5
    ii) Edge_list = [(1, 2), (2, 3), (3, 5)]
    iii) u = 5
    iv) S = {1, 4, 3, 6}
v) S is not empty,
    i) v = S.pop() = 6
    ii) Edge_list = [(1, 2), (2, 3), (3, 5), (5, 6)]
    iii) u = 6
    iv) S = {1, 4, 3}
w) S is not empty,
    i) v = S.pop() = 3
    ii) Edge_list = [(1, 2), (2, 3), (3, 5), (5, 6), (6, 3)]
    iii) u = 3
    iv) S = {1, 4}
x) S is not empty,

i) v = S.pop() = 4
ii) Edge_list = [(1, 2), (2, 3), (3, 5), (5, 6), (6, 3), (3, 4)]
iii) u = 4
iv) S = {1}

y) S is not empty,
i) v = S.pop() = 1
ii) Edge_list = [(1, 2), (2, 3), (3, 5), (5, 6), (6, 3), (3, 4), (4, 1)]
iii) u = 1
iv) S = {}

z) S is empty,
So return Edge_list [(1, 2), (2, 3), (3, 5), (5, 6), (6, 3), (3, 4), (4, 1)]

4) Time complexity -
   a) For Find-Euler-Tour algorithms,
      i) current_path is not empty until all the edges are traversed using the adjacency list of each vertex.
      ii) So the number of basic operations performed in the while loop are O(E) times, where E is the number of edges in the graph.
   b) Hence, Time complexity = O(E)
   c) After executing Find-Euler-Tour algorithm,
      i) We add the edges in the edge_list using the stack S which contains the vertices in order to visit to complete the Euler tour.
      ii) The number of basic operations performed in emptying the stack S is O(E) as the Euler tour traverses every edge so to cover all the edges we may need to visit the vertices more than once.
   d) Hence the total time complexity of Euler-Tour = O(E + E) = O(E)
5) Space complexity -
   a) Stack S holds the vertices as many edges are there. So space required by the stack S is O(E)
   b) The current_path contains the list of vertices until their adjacency lists are not empty, so it can hold all the vertices which cover all the edges. Hence, space required by current_path is O(E+1) = O(E)
   c) Hence, the extra space required by our algorithm is O(E+E+1) = O(E)

Problem 2

Let T be the Minimum Spanning Tree of a graph G=(V, E, w). Suppose G is connected, $|E| \geq |V|$, 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 $w(T)=\min\{w(T): T \in ST(G)- \{T^*\}\}$.

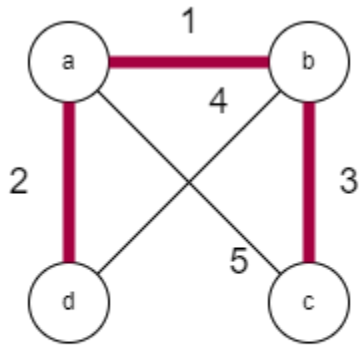
1. Show that T* is unique but that the second-best MST T2 need not be unique.

Ans:

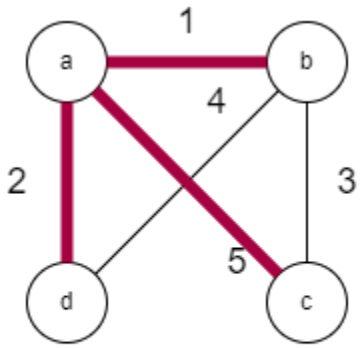
1) To prove - T* is unique -

a) Since all edge weights are distinct, for every cut of G, there is a unique light edge crossing the cut.
b) Let's consider that there are two distinct minimum spanning trees, T* and T*', of G.
c) T* and T*' are distinct because there is at least one edge (u, v) from T* which is not present in T*'
d) If we remove and edge (u, v) from T*, then T* becomes disconnected which results in a cut (S, V-S).
e) The edge (u, v) is the only light edge crossing the cut (S, V-S) and (u, v) is not in T*', so each edge in T*' that crosses the cut (S, V-S) must have weight strictly greater than w(u, v).
f) Let the unique edge (x, y) in T*' crosses the cut (S, V-S) and lies on the cycle that will form if we add (u, v) to T*' and edge (x, y) is common in both the trees.
g) By our assumption, we know that $w(u, v) < w(x, y)$(as (u, v) is a light edge we have considered) then we can replace an edge (x, y) with an edge (u, v) in T*' which results in a spanning tree with weight strictly less than w(T*').
h) Thus, T*' was not a minimum spanning tree, contradicting our assumption that the graph G has two unique MSTs.
i) Hence, T* is unique.

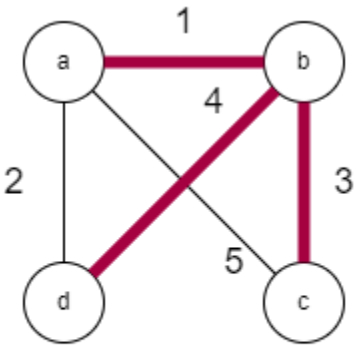
2) To prove - The second-best MST T2 need not be unique.
a) Suppose there are two second-best MST T1 and T2 present for a graph G, then the weight of both the trees must be the same $w(T1) = w(T2)$.
b) T1 and T2 are distinct because there is at least one edge (u, v) from T1 which is not present in T2
c) In some cases, this might happen that the sum of weights of two or more edges in T1 can be equal to the sum of weights of exactly the same number of edges in T2 and the remaining all edges are the same.
d) Hence, we can replace the set of edges from T1 by the set of edges in T2, where the sum of weights of those edges from T1 is the same as the sum of weights of edges in T2 and the number of those edges are equal in both the trees. This can be done because the other edges are common.
e) Hence it is possible to have multiple second-best minimum spanning trees.
f) Here, we have a weighted, undirected graph G with all unique edges and unique minimum spanning tree of weight 6 and two second-best minimum spanning trees with weight 8 -



Minimum spanning tree



second best minimum spanning tree



second best minimum spanning tree

2. 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.

Ans:

Let T2 be a second-best minimum spanning tree of G

1) Case 1 - To prove by replacing one edge from T* we can get T2
   Every minimum spanning tree has |V| - 1 edges, so the second best minimum spanning tree must have at least one edge which is not in the minimum spanning tree. If the second-best minimum spanning tree has exactly one edge (s, t) which is not in the minimum spanning tree, then it has the same set of edges as the minimum spanning tree except the one edge (s, t) which replaces the edge (u, v) of minimum spanning tree.
   Here, T2 = (T*- {(u,v)}) ∪ {(s,t)} is proved.
2) Case 2 - To prove that by replacing two or more edges from T* we cannot get a second best minimum spanning tree.
   a) Let T* and T2 differs by two or more edges. So there are at least two edges in the set T* - T2 and let (u, v) be the edge in T* - T2 with minimum weight.
   b) If we add (u, v) in T2 then we will get a cycle c. This cycle contains an edge (x, y) which is in T2 - T*.
   c) We claim that w(x, y) > w(u, v). To prove this by contradiction -
      i) Let's assume that w(x, y) < w(u, v). If we add an edge (x, y) to T*, we get a cycle c'. This cycle contains an edge (u', v') which is in T* - T2.
      ii) Therefore, we get a spanning tree T*' = (T* - {(u', v')}) U {(x, y)} by replacing edge (u', v') by (x, y) and we must have w(u', v') < w(x, y), otherwise T*' would be a spanning tree with weight less than w(T*).

iii) We get w(u', v') < w(x, y) < w(u, v) which contradicts with our choice of (u, v)

d) Since (x, y) and (u, v) in T2 would be on a common cycle c, then after replacing (x, y) by (u, v) we get a spanning tree with set of edges (T2 - {(x, y)}) U {(u, v)} whose weight is less than w(T2). Ans this tree differs from the T* because this tree differs from T2 by only 1 edge.So we have formed a spanning tree whose weight is less than w(T2) but is not T*. Hence T2 was not a second-best minimum spanning tree.

3) Hence, by cases 1 and 2 we proved that (T*- {(u,v)}) ∪ {(s,t)} is a second-best minimum spanning tree of G.

3. Use Kruskal's algorithm to design an efficient algorithm to compute the second-best minimum spanning tree of G.

Ans:

1) Psedocode -

```
Second-Best-MST(G, w)
      A = Φ
      for each vertex v ∈ G.V
            MAKE-SET(v)
      sort the edges of G.E into nondecreasing order by weight w
      for each edge (u,v) ∈ G.E, taken in nondecreasing order by weight

            if FIND-SET(u) != FIND-SET(v)
                  A = A ∪ {(u, v)}
                  UNION(u, v)
      second-best-MST-weight = MAX_VALUE
      SB-MST = Φ
      for each edge (x, y) ∈ A
            weight = 0
            current-MST = Φ
            for each vertex v ∈ G.V
                  MAKE-SET(v)

            for each edge(u,v) ∈ G.E(taken in nondecreasing order by
weight)
                  if edge (x, y) = edge (u, v)
                        Continue
                  else
                        if FIND-SET(u) != FIND-SET(v)
                              current-MST = current-MST ∪ {(u, v)}
                              weight = weight + w(u, v)
                              UNION(u, v)
```

```
        //check if number of edges in current-MST is |V|-1
        if current-MST.size() != |V|-1
              continue

        if second-best-MST-weight > weight
              SB-MST = current-MST
              second-best-MST-weight = weight

  return second-best-MST-weight, SB-MST
```

2) Textual Description -
    a) Let A - minimum spanning tree f graph G
       SB-MST - second-best MST of the graph G
       second-best-MST-weight - weight of the second-best MST of graph G
    b) First, Sort the edges of graph G and find MST using Kruskal's algorithm
    c) Then For each edge in MST, temporarily exclude it from the edge list (so that we cannot choose it)
    d) For each edge in MST, temporarily exclude it from the edge list so that we cannot choose it for MST
    e) Repeat the above for all edges in MST and take the best one which has the minimum weight.
    f) Algorithm Second-Best-MST(G, w) -
        i) Initialize an empty minimum spanning tree A
        ii) Perform MAKE_SET() operation for each vertex v ∈ V
        iii) Sort all the edges of the graph G in non-decreasing order of weights.
        iv) For each edge from the sorted edges of graph,
            (1) If the root of both the vertices is not same then add that edge in the minimal spanning tree and do union of (u, v)
        v) Initialize second-best-MST-weight to 0 and an empty minimum spanning tree SB-MST
        vi) For each edge(x, y) from the minimum spanning tree A,
            (1) Initialize the weight = 0 and current-MST empty set
            (2) Perform MAKE_SET() operation for each vertex v ∈ V
            (3) For each edge(u, v) from the sorted edges of graph,
                (a) Continue if the edge(x, y) = edge(u, v)
                (b) Else
                    (i) If the root of both the vertices is not same then
                        1. add that edge in the current-MST
                        2. Add weight of (u, v) in weight

3. union(u, v)

(4) If current-MST size is not equal to |V|-1, then continue for next iteration

(5) If second-best-MST-weight > weight then assign weight to second-best-MST-weight and assign current-MST to SB-MST

vii) Return second-best-MST-weight, SB-MST

3) Example -

a) The graph G has vertices V = {a, b, c, d} and edges E = {(a, b, 1), (b, c, 3), (a, d, 2), (b, d, 4), (a, c, 6)}

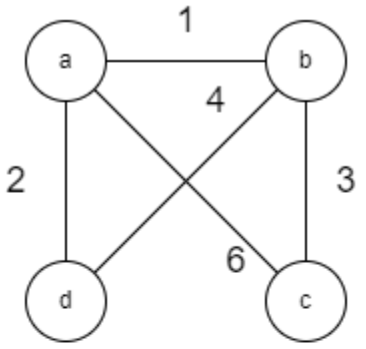
b) After sorting all the edges according to the weight, we get E = {(a, b, 1), (a, d, 2), (b, c, 3), (b, d, 4), (a, c, 6)}

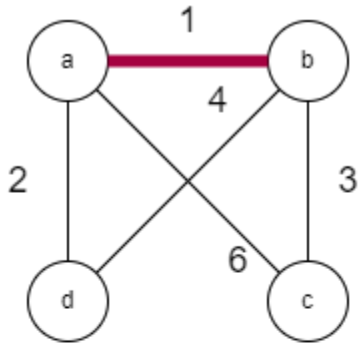
c) FIND(a) != FIND(b) so add an edge in MST

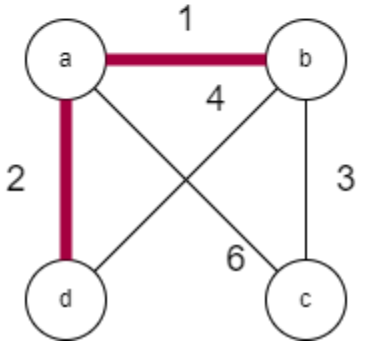
d) FIND(a) != FIND(d) so add an edge in MST

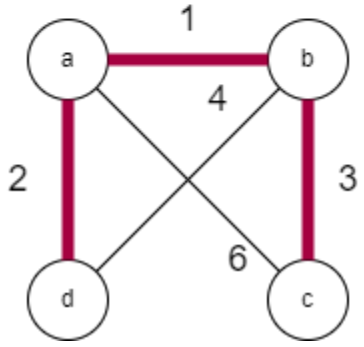
e) FIND(b) != FIND(c) so add an edge in MST

f) So MST is formed using Kruskal's algorithm -







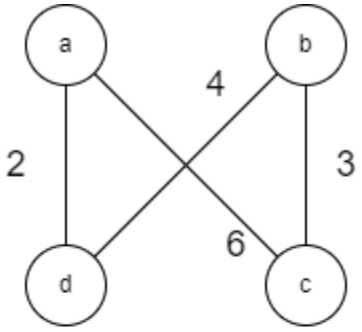


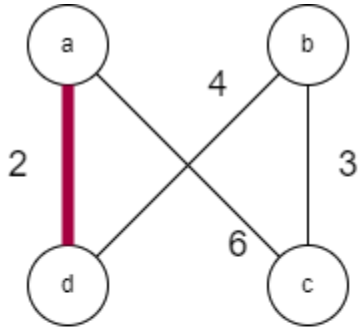
g) second-best-MST-weight = MAX_VALUE and SB-MST is empty

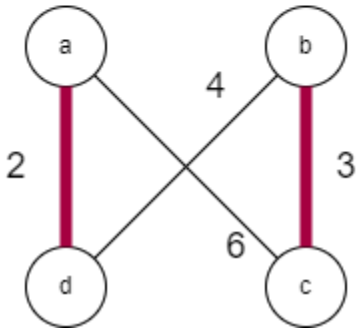
h) Edges in A = {(a, b, 1), (a, d, 2), (b, c, 3)}

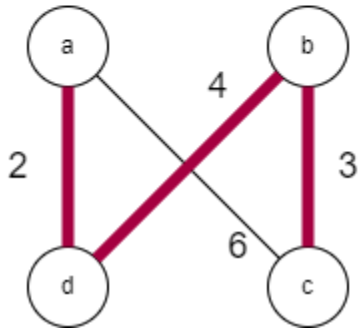
i) For an edge (a, b, 1),
Current-MST is formed by using all the edges in graph G except this

(a, b, 1) edge



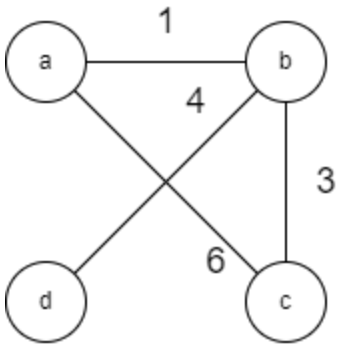


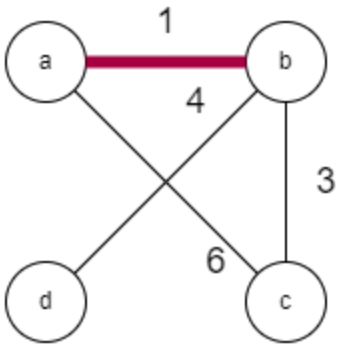


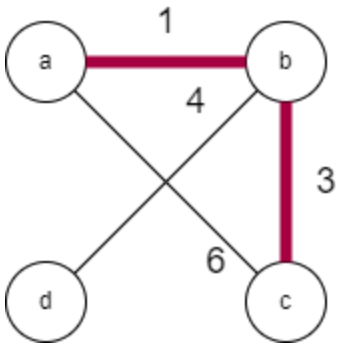


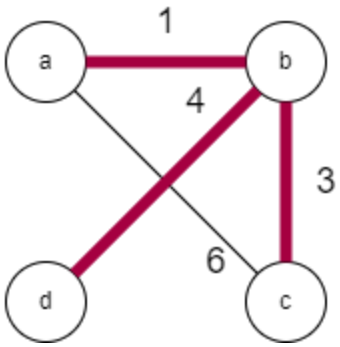
j) second-best-MST-weight = 2+3+4 = 9 and SB-MST is {(a, d, 2), (b, c, 3), (b, d, 4)}

k) For an edge (a, d, 2),
Current-MST is formed by using all the edges in graph G except this (a, d, 2) edge



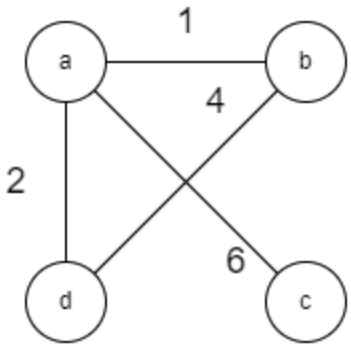


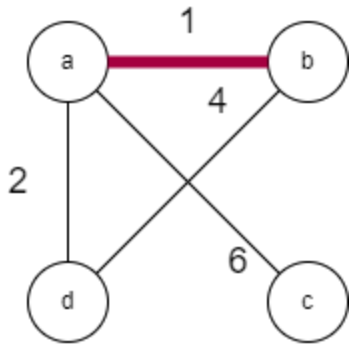


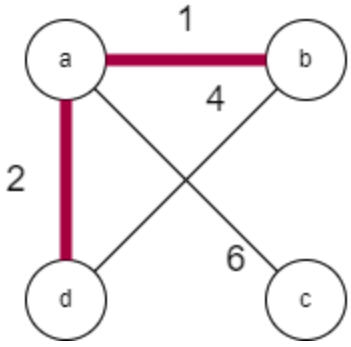


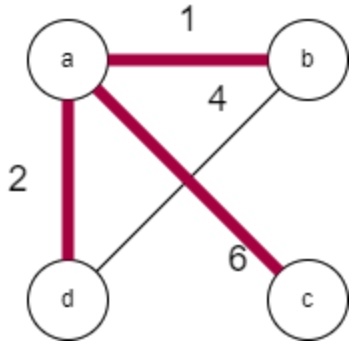
l) second-best-MST-weight = 1+3+4= 8 and SB-MST is {(a, b, 1), (b, c, 3), (b, d, 4)}

m) For an edge (b, c, 3),
Current-MST is formed by using all the edges in graph G except this (b, c, 3) edge









n) Since weight of current-MST is 1+2+6 = 9 Hence no changes in second-best-MST-weight and SB-MST
o) Hence, second-best-MST-weight=8 and SB-MST {(a, b, 1), (b, c, 3), (b, d, 4)} is returned.

4) Time complexity -
   a) Time required for MAKE-SET() operations for all vertices in the graph is O(V)
   b) Time complexity to sort all the edges in the graph is O(E*lg(E))
   c) For all the edges, we perform basic operations in FIND-SET() and UNION() by using union by rank and path compression heurstics. We perform O(E) FIND-SET() and UNION() operations on the disjoint-set forest.
   d) Along with MAKE-SET() operations, FIND-SET() and UNION() operations take total O((V+E)α(V)) where α is the very slowly growing function which is α(V) <= 4. So this can be seen as linear time in O(V+E)
   e) The MST contain |V|-1 edges, so for every edge in MST we find the new MST using all the edges except that edge.
   f) For every edge from the MST we need to perform O(V) MAKE-SET operations and O(E) FIND-SET() and UNION() operations. Hence total number of basic operations and MAKE-SET, FIND-SET() and UNION()

operations performed using union by rank and path compression heurstics is $O((|V|-1)*((V + E)*(\alpha(V)))) = O(VE\alpha(V)) \approx O(VE)$ where α is the very slowly growing function which is $\alpha(V) <= 4$.

g) Hence, the total time complexity of our algorithm is $O(E*\log(E) + (V+E)\alpha(V) + VE\alpha(V)) = O(VE\alpha(V)) \approx O(VE)$

5) Proof of correctness -

Theorem - Our algorithm correctly finds the second-best MST

a) For our theorem to prove, we need to prove that by using kruskal's algorithm, we get the minimum spanning tree.

b) By using Proof by induction,

i) Let T be the spanning tree we get from Kruskal's algorithm,

ii) Let T* be the minimum spanning tree.

iii) If T=T* Kruskal's algorithm finds the minimum spanning tree correctly.

iv) If $T \neq T^*$ then there exist an edge $e \in T^*$ of minimum weight that is not in T.

v) Further, $T \cup \{e\}$ contains a cycle C such that -

(1) Every edge in C has weight less than wt (e) (This follows from how the algorithm constructed T)

(2) There is some edge g in C that is not in T*. (Because T* does not contain the cycle C)

vi) Consider the spanning tree $T2 = T \setminus \{e\} \cup \{g\}$

(1) And $wt(T2) \geq wt(T)$. (We exchanged an edge for the one that is no more expensive.)

vii) We can redo the same process with T2 to find a spanning tree T3 with more edges in common with T*. By induction, we can continue this process until we reach T*, from which we see $wt(T) \leq wt(T2) \leq wt(T3) \leq \cdots \leq wt(T^*)$.

viii) Since T* is a minimum weight spanning tree, then these inequalities must be equalities and we get that T is a minimum weight spanning tree.

c) Hence, in our algorithm, we find the other minimum spanning trees by using Kruskal's algorithm by excluding one edge at a time from the minimum spanning tree received from Kruskal's algorithm.

d) This way we get all the spanning trees except the best spanning tree of the graph. And Hence by choosing the minimum weight spanning tree from those spanning trees, we get our second-best minimum spanning tree.

Problem 3

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

1. 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.

Ans:

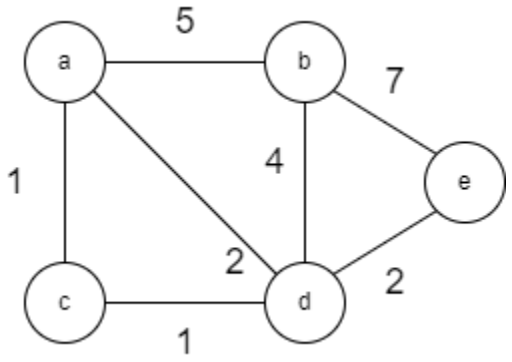
1) Pseudocode -

```
def INITIALIZE()
      for each v ∈ V do
            d[v] ← ∞
            π[v] ← nil
            usp[v] ← 0
      d[s] ← 0
      usp[s] ← 1

def RELAX(u, v, w)
      if d[u]+w < d[v]
            d[v] = d[u] + w
            π[v] = u
            usp[v] = 1
            if usp[u] = 0
                  usp[v] = usp[u]
            DECREASE-KEY(Q, v, d[v])
      else if d[u]+w = d[v]
            usp[v] = 0

def DIJKSTRA_ALG
      INITIALIZE()
      S ← Φ
      Add all vertices to a priority queue Q, on key value d[v].
      while not EMPTY(Q) do
            u = EXTRACT-MIN(Q)
            S = S U {(u)}
            for each v in adj_list[u] do
                  if v not in S then
                        RELAX(u, v, w(u, v))
      return d, usp
```

2) Textual description -
    a) In the above pseudocode, the DIJKSTRA_ALG is exactly similar to the Dijkstra's original algorithm with some additional code.
    b) One basic intuition is that we know there are more than one shortest paths to a vertex if the calculated relax value is equal to the value already present in the distance array.
       i.e, d[u]+w = d[v]
    c) So if the above condition is true for some vertex v then we will know that there exists multiple shortest paths to reach from source vertex to vertex v.
    d) Therefore, if the above condition is true then, we will set the usp value of vertex v to 0. Otherwise it will be 1.
    e) If the relax value is lesser than the value present in the distance array for some vertex v then we simply overwrite the value in the distance array and set the usp value of that vertex v to 1.
    f) But if the vertex u in relax function has usp value 0 which means that there exist multiple shortest paths from source vertex s to u then for our vertex v also we will set the usp value as 0 because we can reach from s to v in multiple shortest paths of s to u and then by traversing edge from u to v.
    g) Therefore, we have to check every time when the d[v] gets updated, what is the value of its parent's usp.
    h) If usp[u] = 0 then usp[v] will also be zero.
    i) Thus for every possible case our algorithm performs correctly by setting the accurate usp values.
    j) Algorithm -
        i) Initialize distance array to infinity, parent array to nil and usp array to 0 for every vertex v from the graph.
           Set d[source_vertex] = 0 and usp[source_vertex] = 1
        ii) Initialize set S to an empty set which is used to store the set of vertices whose shortest distance from source vertex is calculated.
        iii) Add all the vertices in priority queue based on distance from source vertex
        iv) While priority queue is not empty -
            (1) Get the minimum element u from priority queue
            (2) Add it in set S
            (3) Traverse adjacency list of u. If the vertex v from the adjacency list of u is not in set S then perform RELAX operation on vertex v
        v) We get the distance array d which stores the shortest distance from source vertex to any vertex v from the graph and an array

usp which gives the value 1 if there is a unique shortest path, otherwise 0.

3) Example -



a) Consider above graph,
Where, w(a,b) = 5
w(a,d) = 2
w(a,c) = 1
w(b,d) = 4
w(b,e) = 7
w(c,d) = 1
w(d,e) = 2

b) Consider vertex a as source vertex
S = {}

	a	b	c	d	e
d	0	inf	inf	inf	inf
usp	1	0	0	0	0

c) Iteration1 -
S = {a}
vertex a is extracted from the queue and added to S. It's adjacency list is traversed.
After performing the RELAX operation on vertices b, c, and d we get,

	a	b	c	d	e
d	0	5	1	2	inf
usp	1	1	1	1	0

d) Iteration 2 -
S = {a, c}
vertex c is added to the S. Its adjacency list is traversed.
d is now reachable in distance 2 from a. One via direct edge to d and one via c.
So usp[d] becomes 0.

	a	b	c	d	e
d	0	5	1	2	inf
usp	1	1	1	0	0

e) Iteration 3 -
S = {a, c, d}
vertex d is added to the S. It's adjacency list is traversed.
Vertex e is updated for the first time. But its usp value is set to 0. This is because we are going to vertex e via vertex d which has usp 0.
Hence, usp[e] = 0

	a	b	c	d	e
d	0	5	1	2	4
usp	1	1	1	0	0

f) Iteration 4 -
S = {a, c, d, e}
vertex e is added to the S. It's adjacency list is traversed.
We cannot RELAX d because it is in set S.
For vertex b, d[b] < d[e]+w(b,e) Hence no change in d[b] and usp[b]

	a	b	c	d	e
d	0	5	1	2	4
usp	1	1	1	0	0

g) Iteration 5 -
S = {a, c, d, e, b}
vertex b is added to the S. It's adjacency list is traversed.

We cannot RELAX vertices a, d, and e because they are in set S.
Hence, we get our final array -

	a	b	c	d	e
d	0	5	1	2	4
usp	1	1	1	0	0

h) Here we got the usp value for every vertex. Our algorithm works correctly.

4) Time Complexity -
   a) Insertions of |V| vertices in a priority queue take the time of Vlg(V)
   b) Then we pop the edge with minimum weight from the queue. This results in the heapify operation internally. We have to do this for every edge in 'E'. The time required to do it is Elg(V).
   c) In the RELAX method, the number of basic operations are performed in constant time and DECREASE-KEY operation takes O(lgV) time using binary heaps and DECREASE-KEY operation is performed O(E) times. Hence the time complexity is O(1+ElgV) = O(ElgV).
   d) Overall Time Complexity: O(ElgV + (V+E)lgV) = O(E*lgV) if all the vertices are reachable from the source vertex.

5) Proof of correctness -
   Proof by loop invariants -
   a) Initialization:
      i) At the first step, we initialize the usp of source vertex s to be 1. That states that there is a unique path from source vertex to source vertex which holds true.
   b) Maintenance:
      i) In the while loop inside the Dijkstra_Alg, a vertex is extracted from the queue which has a minimum distance from the source.
      ii) According to the algorithm described above, we mark the usp for that vertex in RELAX operation. And this vertex is added to the explored vertex list S. The usp value for this vertex will not change in the next iteration.
      iii) Thus after every iteration, we get one vertex whose usp value we add. And there will be no change in that vertex's usp value because according to the Dijkstr's algorithm, once the vertex is added in S set that means that we have found the shortest path from source vertex to that vertex and there will be no change in that. Hence we get a fixed and correct usp value.
   c) Termination:

i) After the termination of the while loop we get the usp value for all vertices that have been filled using the modified Dijkstra's algorithm described above.

d) Hence, the above algorithm performs correctly.