QUESTION 1

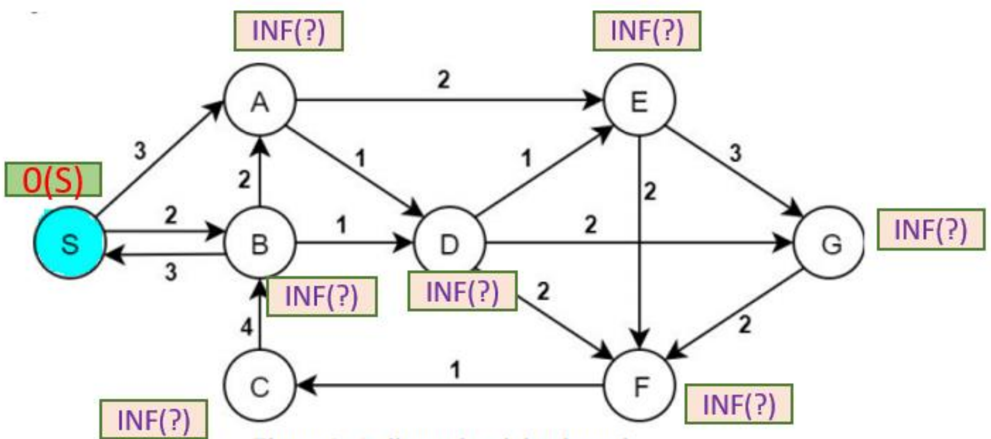


Figure 1: A directed weighted graph.

Starting from S, set the distance of node S to 0 and it's preceding vertex to S.

Every other vertex is set to distance infinity and preceding vertex is unknown.

Mark S as known

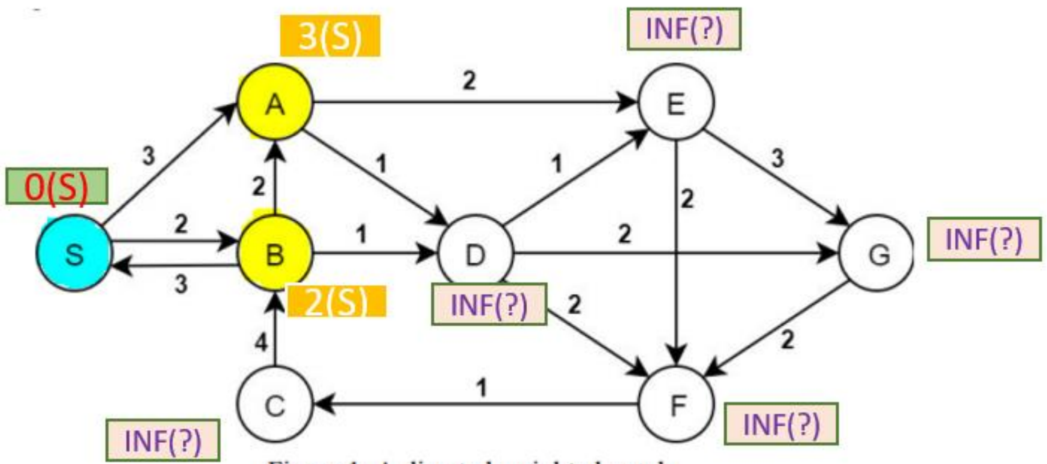


Figure 1: A directed weighted graph.

Check the unknown adjacent vertices of S, namely A and B

Assign distance to vertices A and B since 3 and 2 is smaller than infinity and update their preceding vertex as S.

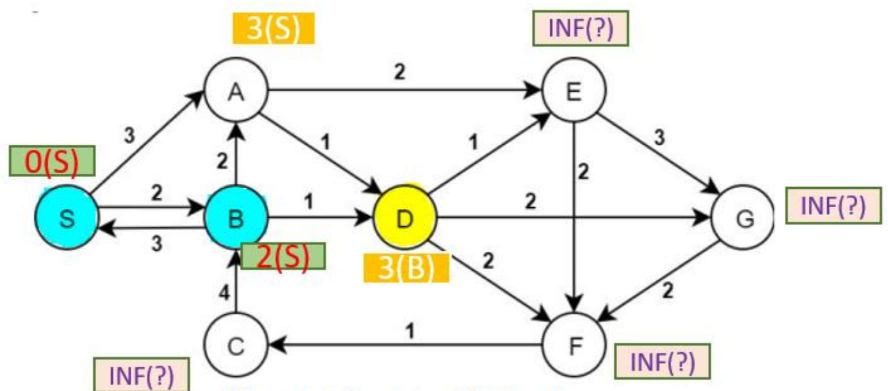


Figure 1: A directed weighted graph.

Between all the unknown vertices, chose the vertex with the least distance, which is B in this case.

Mark B as known.

Check the unknown adjacent vertices of B, namely D

Assign distance to D since 3 is smaller than infinity and update it’s preceding vertex as B

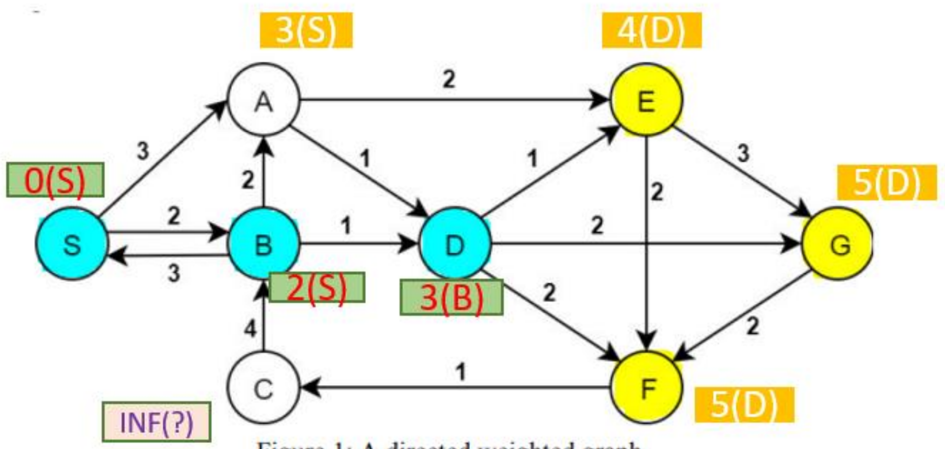


Figure 1: A directed weighted graph.

A and D has the least distance between all the unknown vertices. Since their distance is both 3, choosing one of them is an arbitrary decision.

D is chosen and it’s marked as known.

Check the unknown adjacent vertices of D, namely E, G and F

Assign distance to E, G , F and update their preceding vertex as D.

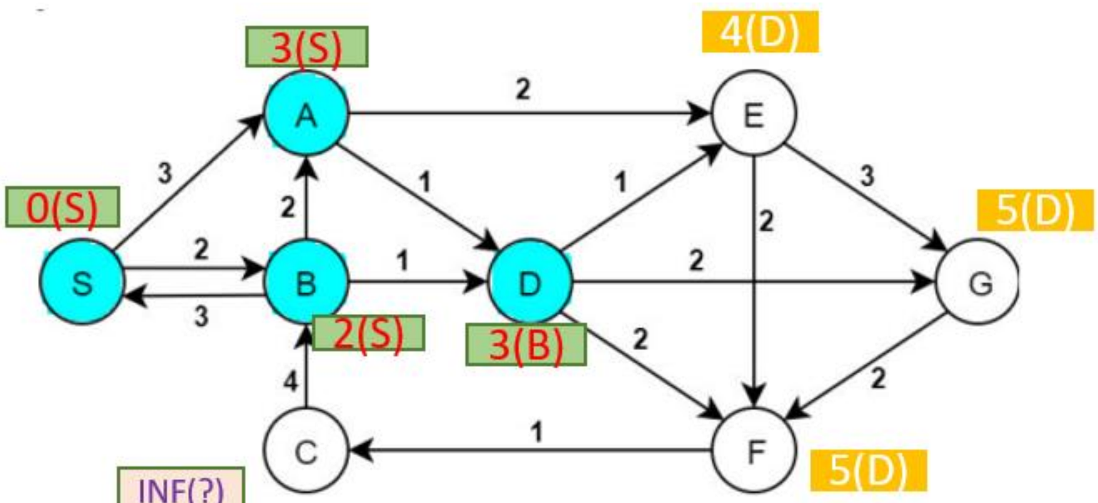


Figure 1: A directed weighted graph.

Between all the unknown vertices, A is chosen since it has the least distance.

Mark A as known

Check the unknown adjacent vertices of A, namely E

We don't update the distance and preceding vertex of E because distance from A to E (5) is larger than the current distance assigned to E (4).

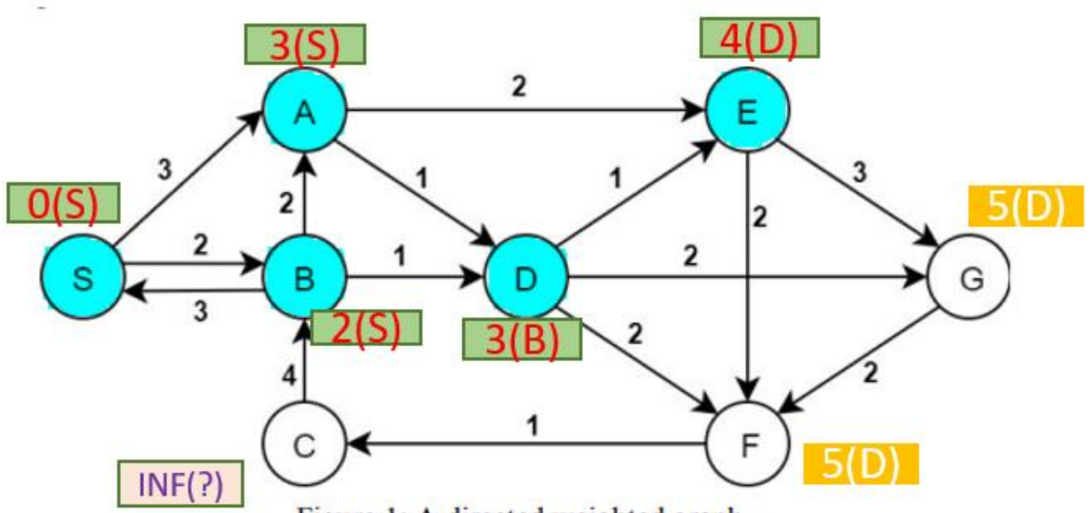


Figure 1: A directed weighted graph.

Between all the unknown vertices, E is chosen since it has the least distance.

Mark E as known.

Check the unknown adjacent vertices of E, namely G and F

We don't update the distance and the preceding vertex of G and F because their currently assigned distance is smaller than distance from E to F (6) and E to G (7)

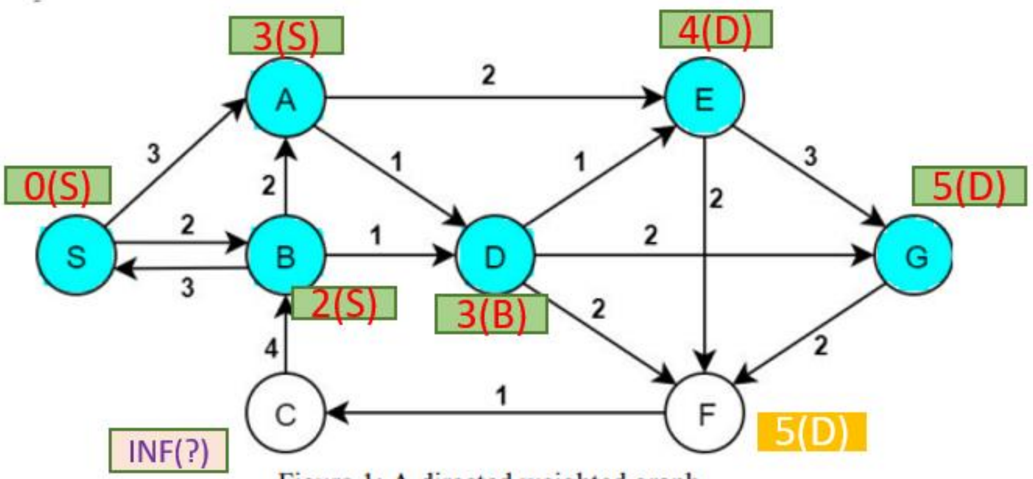


Figure 1: A directed weighted graph.

Between all the unknown vertices, G and F has the least and same distance so decision is arbitrary, G is chosen.

Mark G as known.

Check the unknown adjacent vertices of G, namely F

We don't update the distance and preceding vertex of F since it's currently assigned distance is less than the distance from G to F (7)

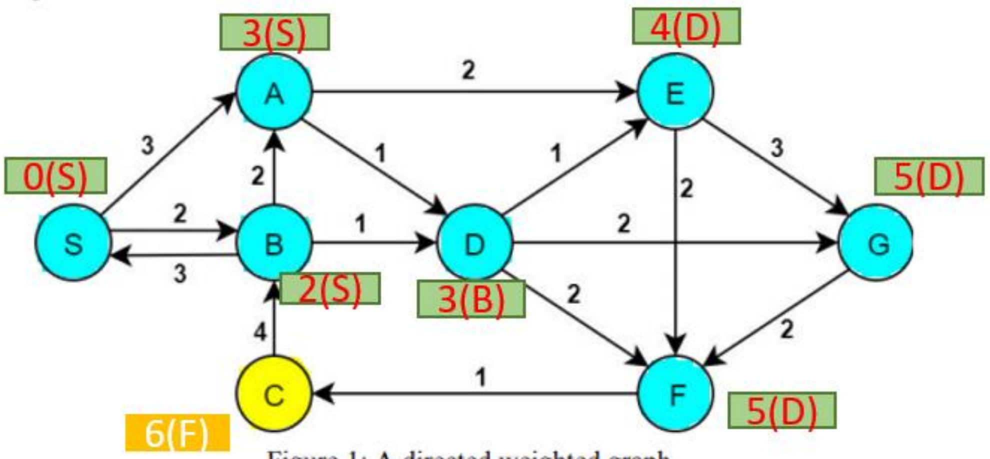


Figure 1: A directed weighted graph.

Between all the unknown vertices, F is chosen since it has the least distance.

Mark F as known.

Assign distance to unknown adjacent vertex of D, namely, C and update it's preceding vertex as F.

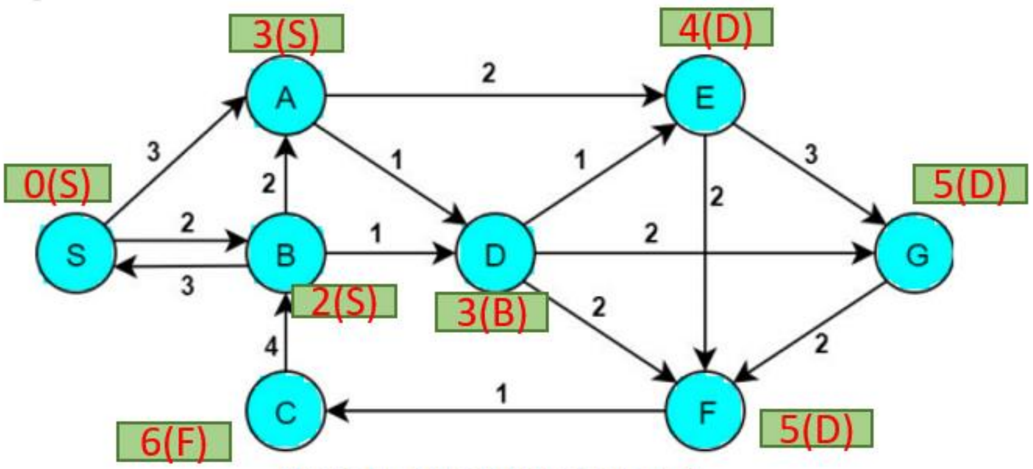


Figure 1: A directed weighted graph.

Between all the unknown vertices, C is chosen since it’s the only unknown vertex.

Mark C as known.

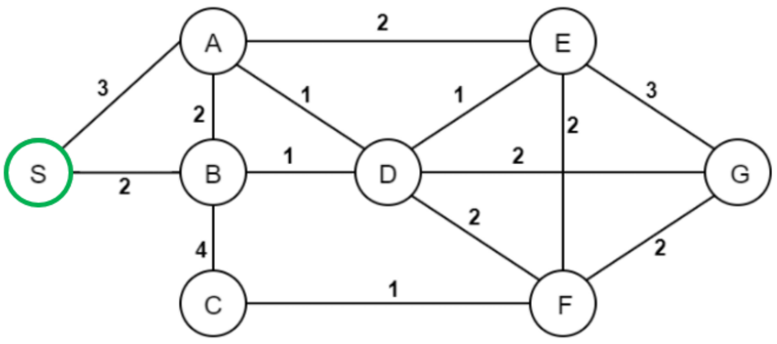
Adjacent vertex of C, namely B, is already marked as known so there is nothing to update.

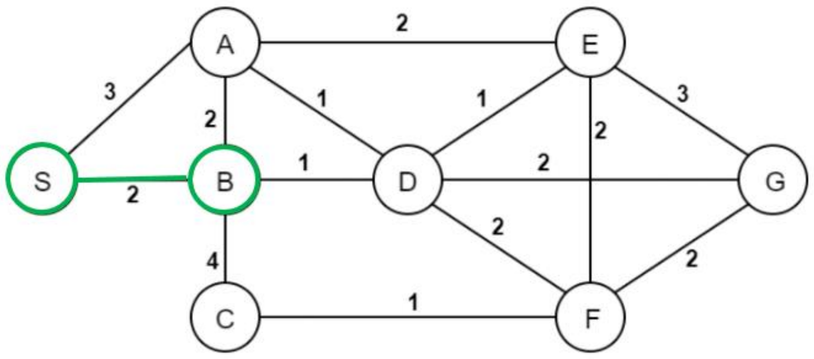
All vertices are marked as known.

QUESTION 2

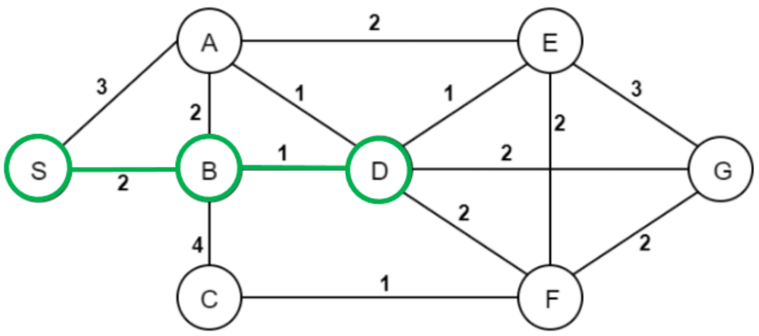
Initializing the tree from vertex S and between all the edges that connects the tree to a new vertex, we chose the edge with the minimum weight. Same edge cannot be concluded to tree more than once.

Starting from vertex S, steps taken and the Minimum Spanning Tree is given below:

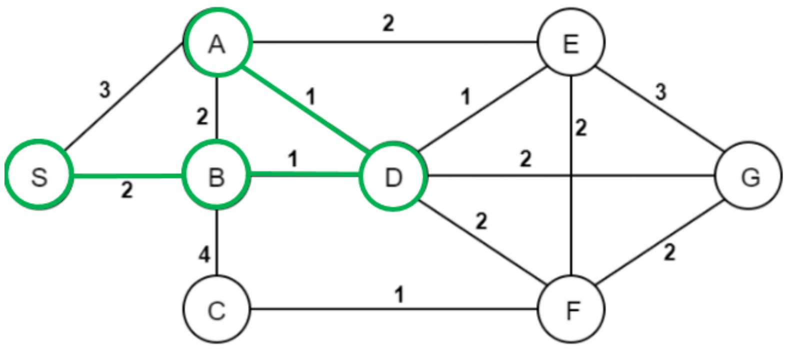




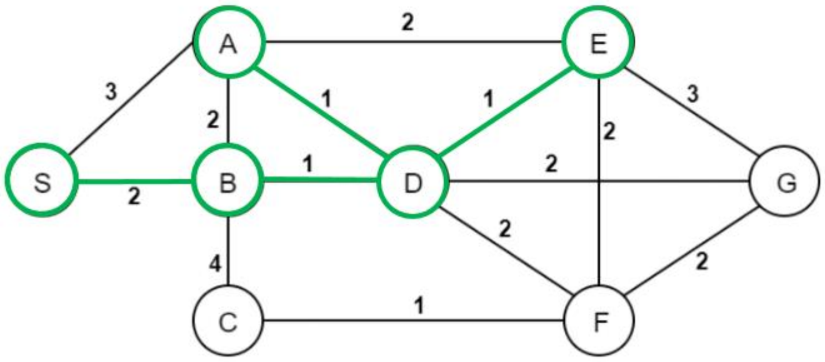
(S,B) is chosen because it has lesser weight than (S,A)



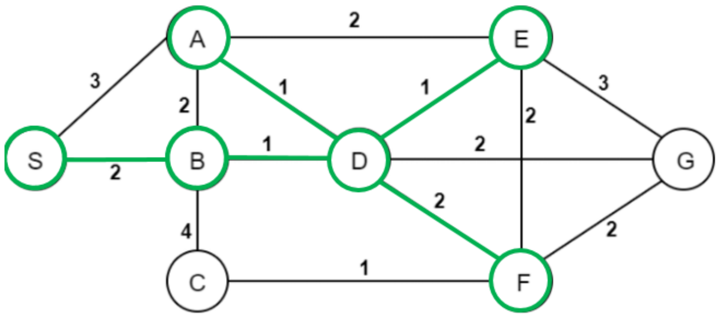
(B,D) is chosen because it has the least weight between (S,A), (B,A) and (B,D)



(D,A) and (D,E) has both weight 1 which is the least weight. Decision is arbitrary, (D,A) is chosen

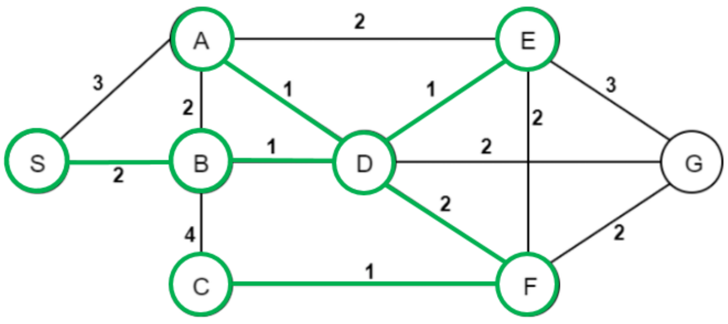


(D,E) is chosen since it has the least weight.

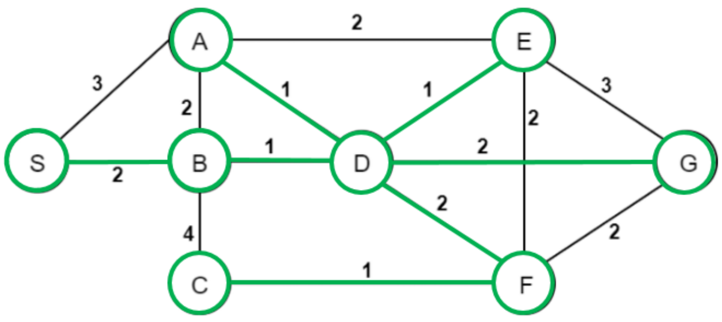


Decision between edges (E,F) and (D,F) is arbitrary, both has weight 2. Note that we cannot choose (A,E) since both A and E are already in the tree.

(D,F) is chosen.

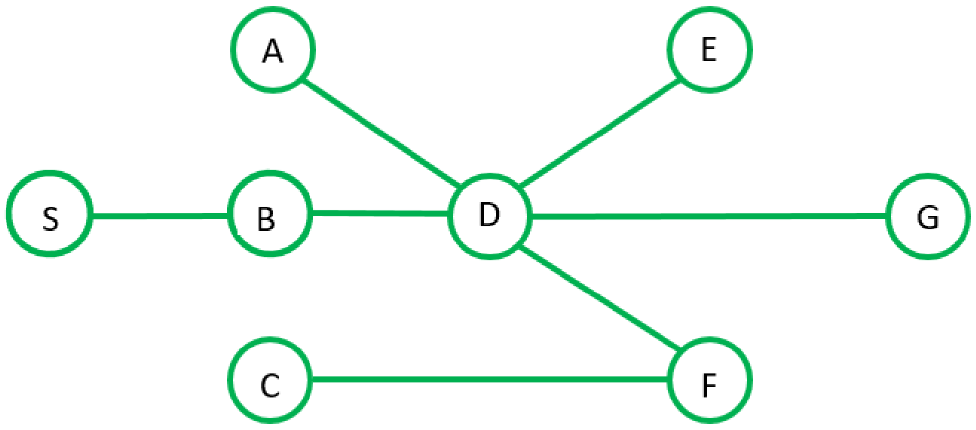


(F,C) is chosen since it has the least weight.



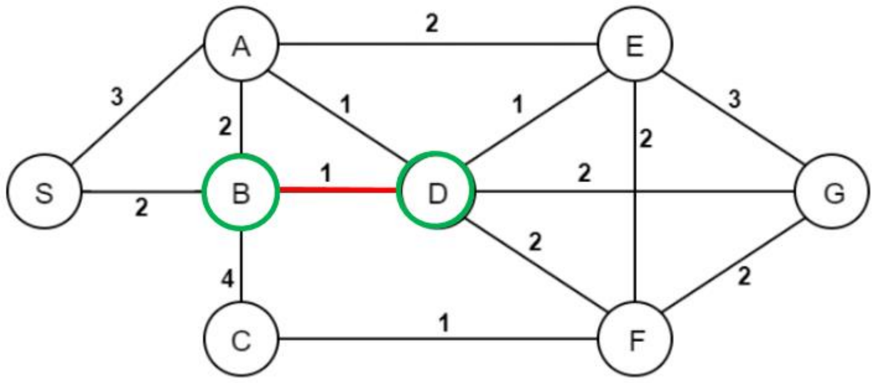
(D,G) is chosen.

Minimum Spanning Tree is given below.

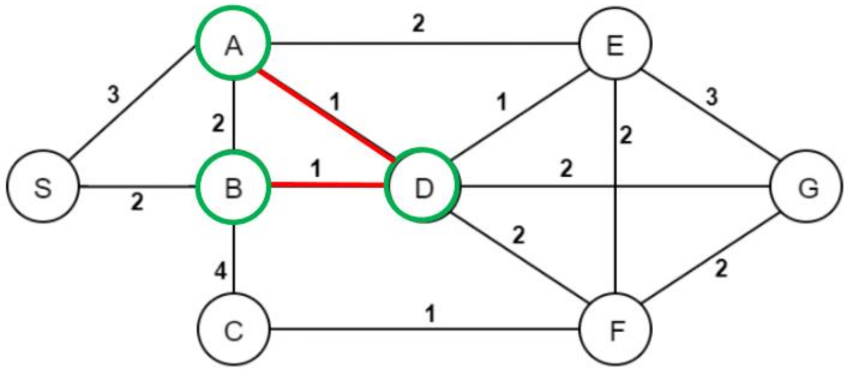


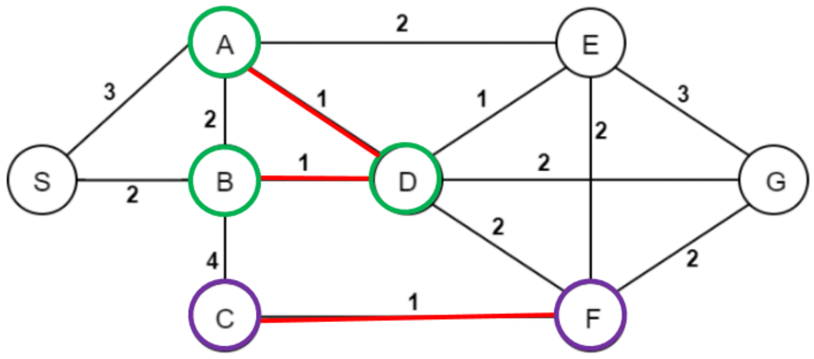
Total Cost = 2 + 1 + 1 +1 + 2 + 1 + 2 = 10

QUESTION 3

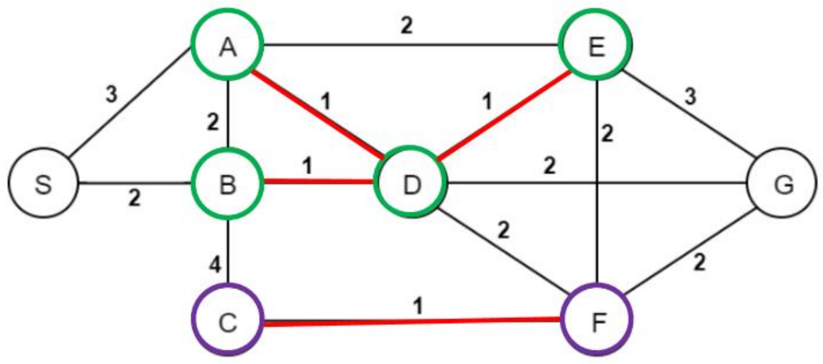


Choosing arbitrarily between the edges with least weights (1) , (B,D) is chosen



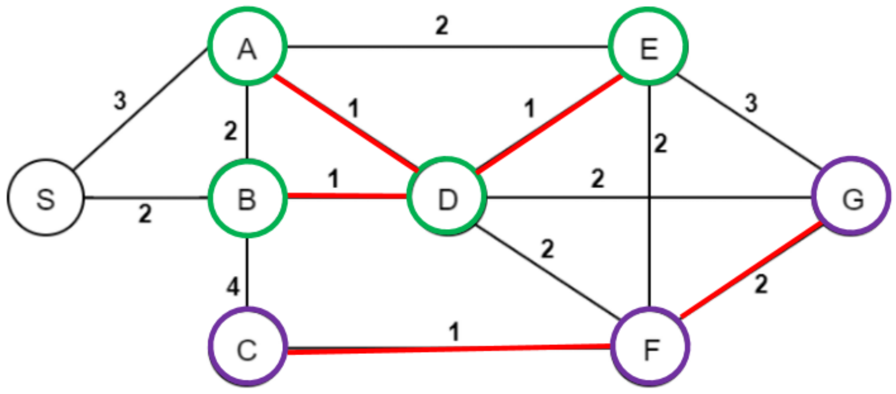


Now we have 2 different trees which are not yet connected

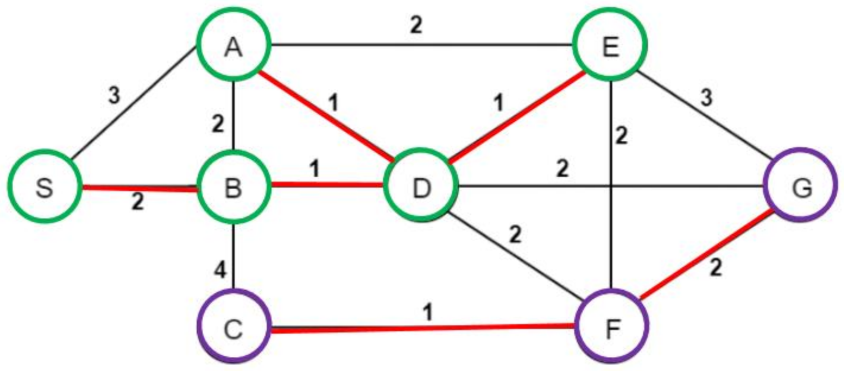


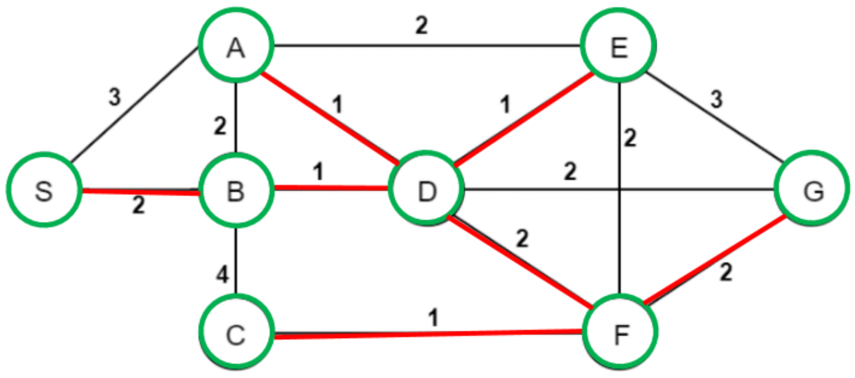
Choosing between the edges with weights 1

They don't create a cycle and two different trees are formed.



Included (F,G) to second tree since it does not create a cycle





All nodes are now in the same tree. We have a MST.

(S,A) , (A,E) and (E,G) are not included since they create cycle.

QUESTION 4

Q = {} . Queue initialized empty.

First, mark all edges as unvisited.

Then mark S as 0 since we are starting from S and distance from S to S is 0.

Enqueue S. Q = {S}

1- Check the front of the Queue, which is S.
   - Visit all unvisited adjacent vertices of S, which are A and B.
   - Label A and B as visited.
   - Mark their distance as 1.
   - Enqueue A and B. Q = { S, A, B}
   - Dequeue S. -> Q = {A, B}
2- Check the front of the Queue, which is A.
   - Visit all unvisited adjacent vertices of A, which are E and D.
   - Label E and D as visited.
   - Mark their distance as 2.
   - Enqueue E and D. Q = { A, B, E,D}
   - Dequeue A. -> Q = { B, E, D}
3- Check the front of the Queue, which is B.
   - Visit all unvisited adjacent vertices of B, there are no unvisited adjacent edges.
   - Dequeue B. Q = {E, D}
4- Check the front of the Queue, which is E.
   - Visit all unvisited adjacent vertices of E, which are G and F.
   - Label them as visited.
   - Mark their distance as 3

- Enqueue G and F -> Q = {E, D G, F}
- Dequeue E -> Q = { D, G, F}

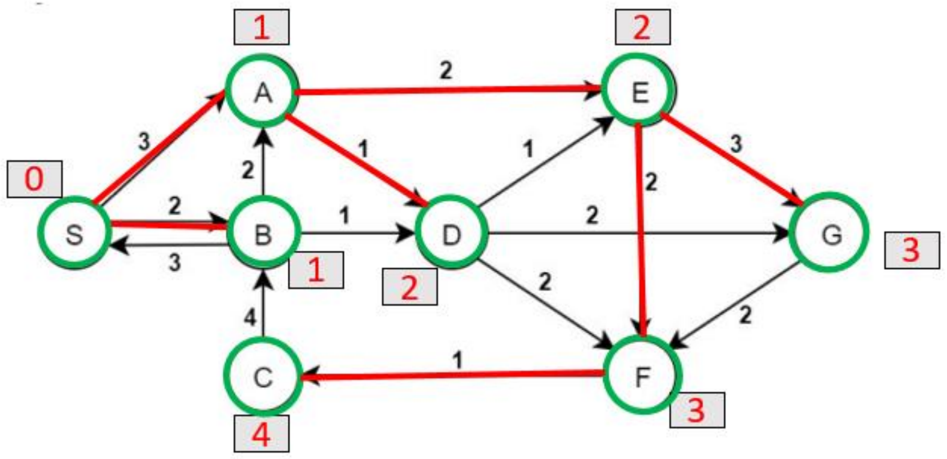
5- Check the front of the Queue, which is D.
   - Visit all unvisited adjacent vertices of D, there are no unvisited adjacent edges.
   - Dequeue D -> Q = { G, F}

6- Check the front of the Queue, which is G.
   - Visit all unvisited adjacent vertices of G, there are no unvisited adjacent edges.
   - Dequeue G -> Q = { F}

7- Check the front of the Queue, which is F.
   - Visit all unvisited adjacent vertices of F, which is C.
   - Label C as visited.
   - Mark it's distance as 4
   - Enqueue C -> Q = { F, C }
   - Dequeue F -> Q = { C}

8- Check the front of the Queue, which is C.
   - Visit all unvisited adjacent vertices of C, there are no unvisited adjacent edges.
   - Dequeue C -> Q = { }

9- Queue is empty, we are done.
   Visited nodes, their distance and the breadth first search tree is depicted below.



QUESTION 5

Part (a)

S = {} . Stack initialized empty.

First, mark all edges as unvisited.

Since we are starting from the edge S, push it into the stack. -> S = { S }

Mark S as visited

1- Starting from S

- Check an unvisited adjacent edge of S, which is B.
- Mark B as visited
- Push B into the Stack -> S = { S, B }

2- Check the top of the stack, which is B
   - Check an unvisited adjacent edge of B, which is D.
   - Mark D as visited
   - Push D into the Stack -> S = {S , B , D}

3- Check the top of the stack, which is D
   - Check an unvisited adjacent edge of D, which is F.
   - Mark F as visited
   - Push F into the stack -> S = { S , B ,D, F}

4- Check the top of the stack, which is F
   - Check an unvisited adjacent edge of F, which is C
   - Mark C as visited
   - Push C into the Stack -> S = { S , B , D , F, C }

5- Check the top of the stack, which is C
   - C does not have any unvisited adjacent edge
   - Pop C from the stack. S = { S , B, D , F}

6- Check the top of the stack, which is F
   - F does not have any unvisited adjacent edge
   - Pop F from the stack -> S = { S , B, D }

7- Check the top of the stack, which is D
   - Check an unvisited adjacent edge of D, which is E
   - Mark E as visited
   - Push E into the stack -> S = { S , B, D, E}

8- Check the top of the stack, which is E
   - Check an unvisited adjacent edge of E, which is G
   - Mark G as visited
   - Push G into the stack -> S = { S, B, D ,E ,G }

9- Check the top of the stack, which is G
   - G does not have any unvisited adjacent edge
   - Pop G from the stack -> S = { S , B, D, E }

10- Check the top of the stack, which is E
   - E does not have any unvisited adjacent edge
   - Pop E from the stack -> S = { S, B, D}

11- Check the top of the stack, which is D
   - D does not have any unvisited adjacent edge
   - Pop D from the stack -> S = { S, B}

12- Check the top of the stack, which is B
   - Check an unvisited adjacent edge of B, which is A
   - Mark A as visited
   - Push A into the stack -> S = { S, B, A }

13- Check the top of the stack, which is A

- A does not have any unvisited adjacent edge
- Pop A from the stack -> S = { S, B}

14- Check the top of the stack, which is B

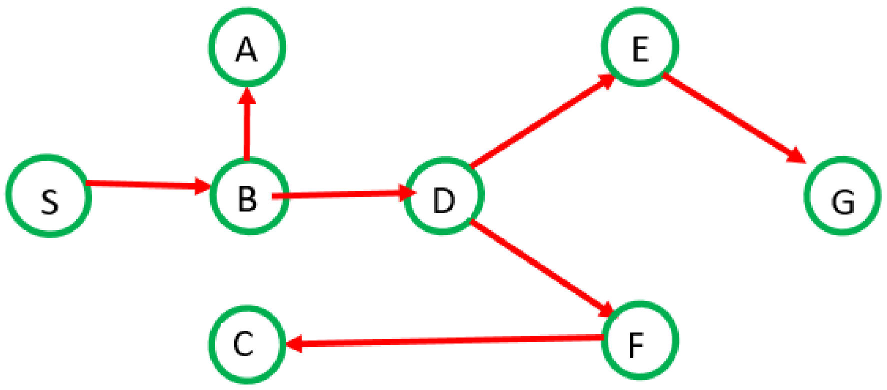
- B does not have any unvisited adjacent edge
- Pop B from the stack -> S = { S}

15- Check the top of the stack, which is S

- S does not have any unvisited adjacent edge
- Pop S from the stack -> S = { }

Stack S is empty, we are done

Depth First Search Tree is depicted below



Depth First Search Tree

Part (b)

Post – order numbers:

C: 1

F: 2

G: 3

E: 4

D: 5

A: 6

B: 7

S: 8

Part (c)

Pre – order numbers:

S: 1

B: 2

D: 3

F: 4

C: 5

E: 6

G: 7

A: 8

Part(d)

Tree Arcs : (S,B), (B,D) ,(B,A), (D,E), (E,G), (D,F), (F,C)

Cross Arcs: (A,D) , (A,E) , (G,F) , (E,F) , (C,B)

Forward Arcs: (S,A) ,

Backward Arcs: (B,S)

QUESTION 6

We select the vertex with in degree 0, print it out and then remove it. Repeat this until the end.

Let T = {} be the set of topological ordering

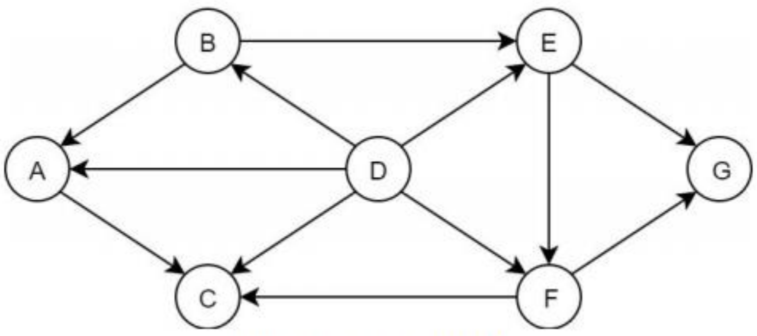
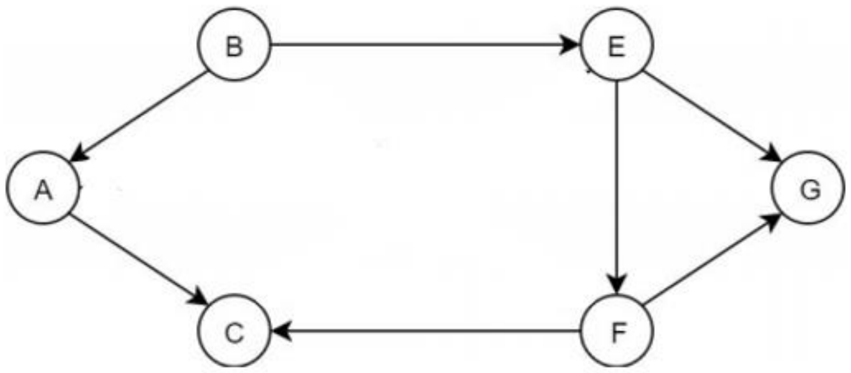


Figure 2: An example DAG.

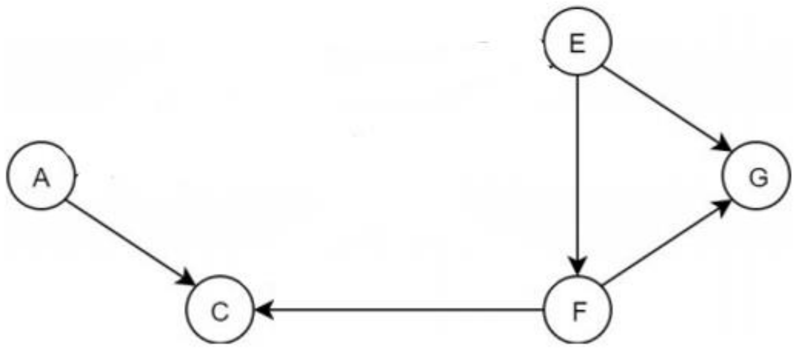
Take D, since it has in degree 0

T = { D }



Take B, since it has in degree 0

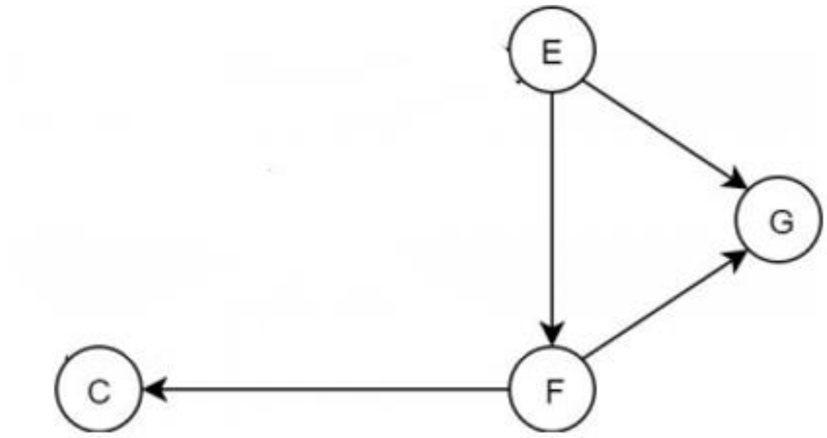
T = { D, B}



A and E both has in degree 0, decision is arbitrary.

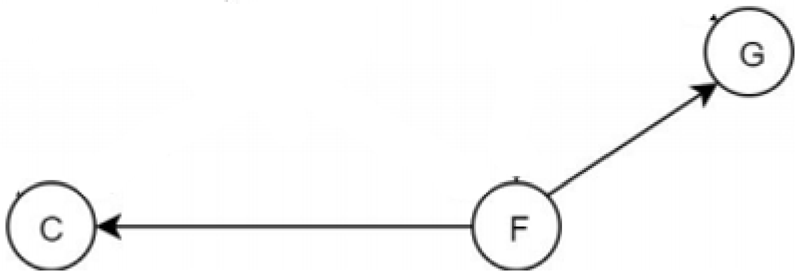
Take A

T = { D, B, A}



Take E, since it has in degree 0.

T = { D, B, A ,E}



Take F, since it has in degree 0.

T = { D, B, A, E, F}

Both G and C has in degree 0, decision is arbitrary.

Take G

T = { D, B, A, E, F, G}

Take C

T = { D, B, A, E, F, G, C}

T is our topological ordering, note that it’s not unique.