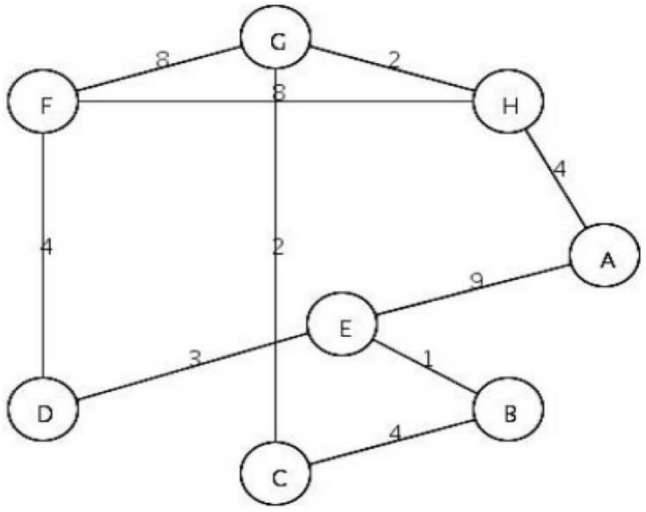
CS 300 - Data Structures

Homework 5 – 28/12/2018

Atilla Alpay Nalcaci - 19510

Question 1. Dijkstra's Algorithm

Dijkstra's Algorithm calculates shortest paths from a given vertex to all the other vertices by means of calculating the shortest path from a vertex to all its adjacent vertices at each step while progressively updating and proceeding.



Known Vertices	D_A (P_A)	D_B (P_B)	D_C (P_C)	D_D (P_D)	D_E (P_E)	D_F (P_F)	D_G (P_G)	D_H (P_H)
G	∞ (?)	∞ (?)	∞ (?)	∞ (?)	∞ (?)	∞ (?)	0 (G)	∞ (?)
G, H	∞ (?)	∞ (?)	∞ (?)	∞ (?)	∞ (?)	∞ (?)	0 (G)	2 (G)
G, H, C	∞ (?)	∞ (?)	2 (G)	∞ (?)	∞ (?)	8 (G)	0 (G)	2 (G)
G, H, C, F	∞ (?)	∞ (?)	2 (G)	∞ (?)	∞ (?)	8 (G)	0 (G)	2 (G)
G, H, C, F, A	6 (H)	∞ (?)	2 (G)	∞ (?)	∞ (?)	8 (G)	0 (G)	2 (G)
G, H, C, F, A, B	6 (H)	6 (C)	2 (G)	∞ (?)	∞ (?)	8 (G)	0 (G)	2 (G)
G, H, C, F, A, B, E	6 (H)	6 (C)	2 (G)	∞ (?)	7 (B)	8 (G)	0 (G)	2 (G)
G, H, C, F, A, B, E, D	6 (H)	6 (C)	2 (G)	10 (E)	7 (B)	8 (G)	0 (G)	2 (G)

D_v (P_u) = Tentative distance to vertex v (Last vertex to cause a change to D_v)

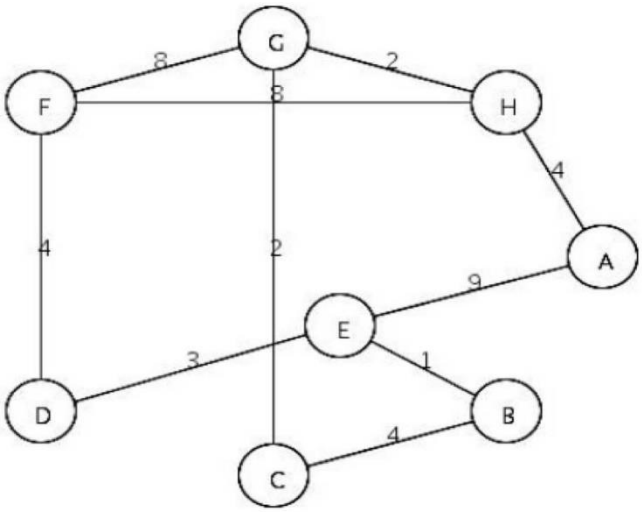
Ultimately, the Shortest Path Table corresponding to the initial figure is presented below.

Shortest Path Table:

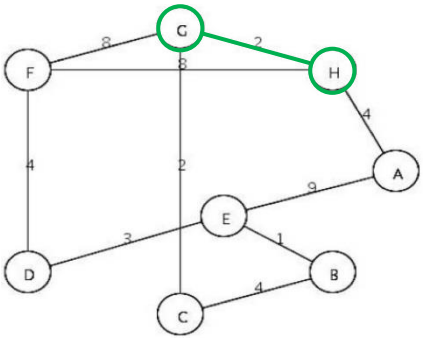
A	B	C	D	E	F	G	H
6 (H)	6 (C)	2 (G)	10 (E)	7 (B)	8 (G)	0 (G)	2 (G)

Question 2. Prim's Algorithm

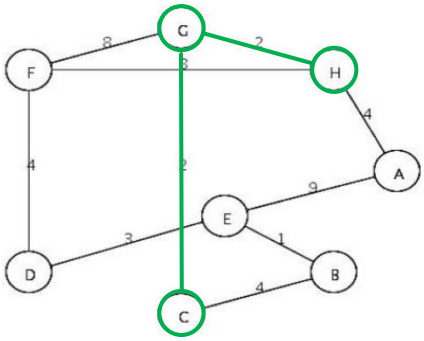
Prim's Algorithm proceeds by adding the edge which has the smallest cost among all other edges and leads to an unknown node (but is "closest" to the tree) at each step.



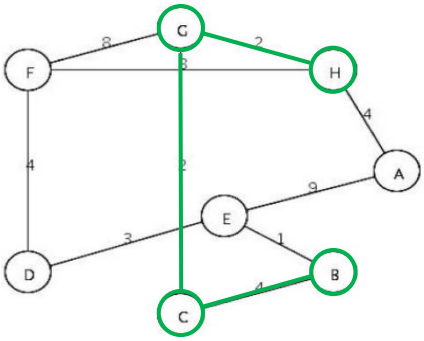
Starting from G, trace operations applied on the initial figure with respect to Prim's Algorithm are depicted below.



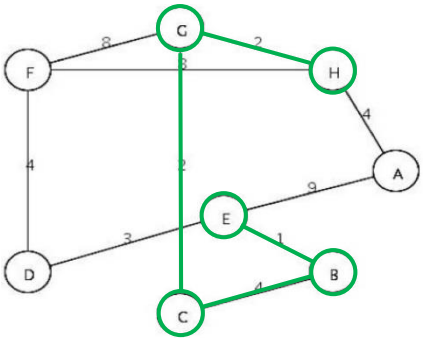
After G → H



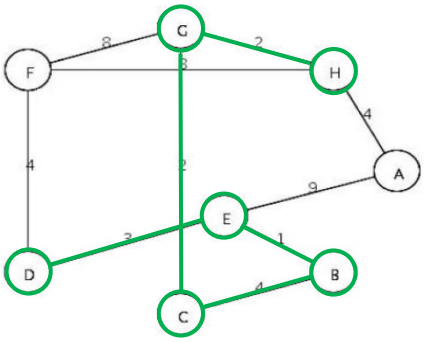
After G → C



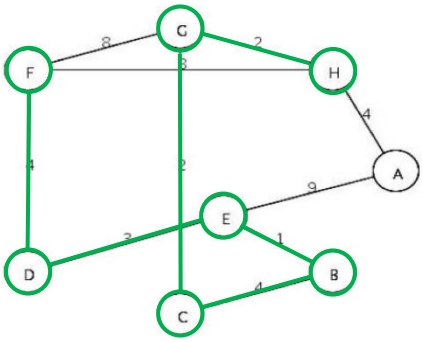
After C → B



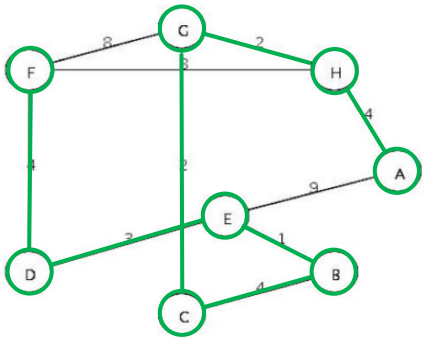
After B → E



After E → D



After D → F



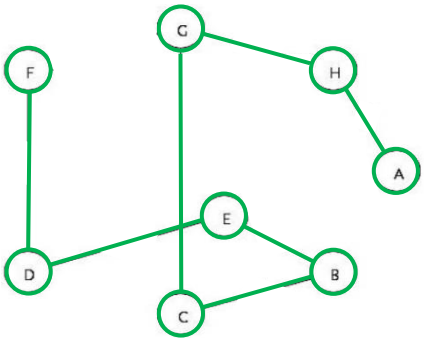
After H → A

Cost = $C_{(G,H)} + C_{(G,C)} + C_{(C,B)} + C_{(B,E)} + C_{(E,D)} + C_{(D,F)} + C_{(H,A)}$

= 2 + 2 + 4 + 1 + 3 + 4 + 4 = <u>20</u>

($C_{(u,v)}$ = Weight (cost) of edge (u, v))

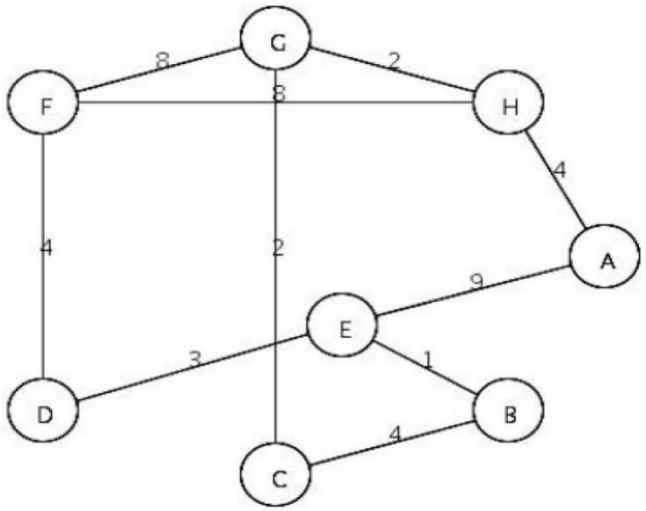
Minimum Spanning Tree in accordance with the initial figure depicted below.



Minimum Spanning Tree

Question 3. Kruskal's Algorithm

Kruskal's Algorithm proceeds by adding the edge with least possible weight as long as the addition procedure does not generate a cycle at each step.

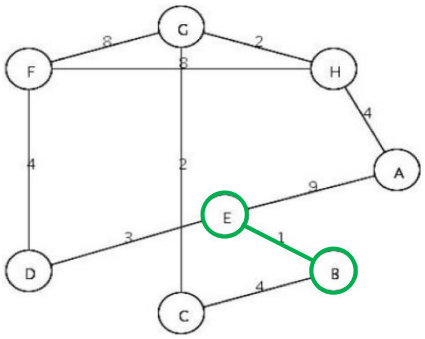


First, list all the edges on the graph with their weights in incrementing order.

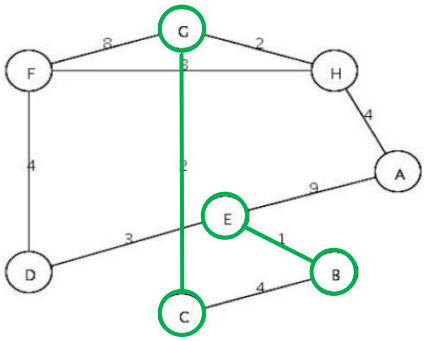
Edge	(E, B)	(G, C)	(G, H)	(D, E)	(C, B)	(H, A)	(D, F)	(G, F)	(F, H)	(E, A)
Weight	1	2	2	3	4	4	4	8	8	9

(u, v) = Edge from vertex u to vertex v

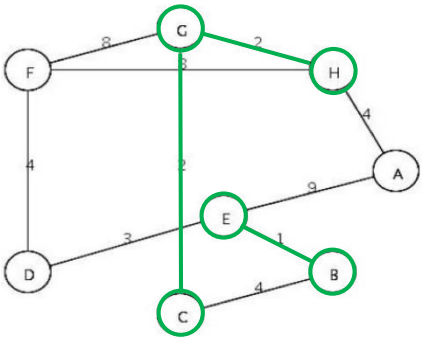
Then start generating your tree in accordance with above list. From left to right, check each edge whether if it is valid to add (e.g. if it is not creating a cycle) or if all nodes have been travelled.



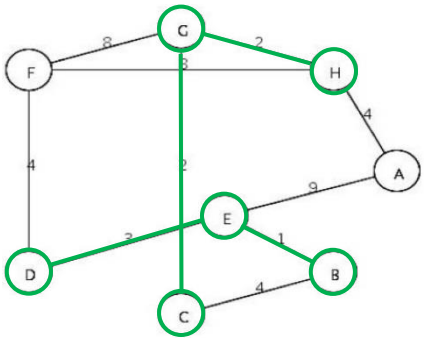
E → B | valid ☑



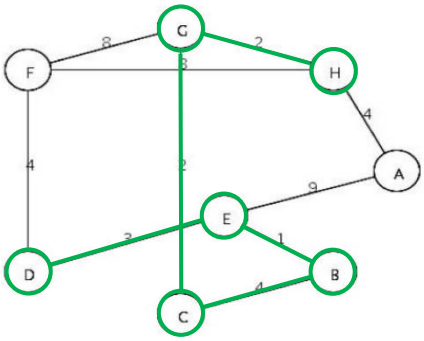
G → C | valid ☑



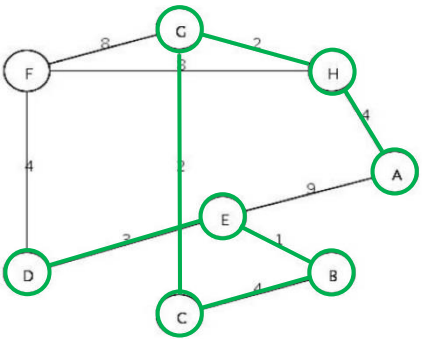
G → H | valid ☑



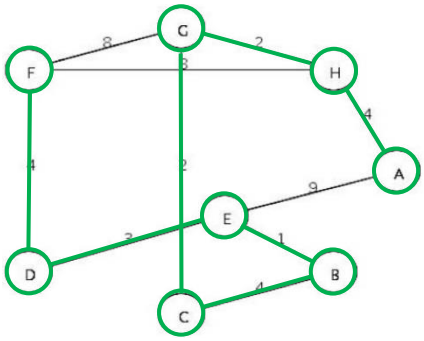
D → E | valid ☑



C → B | valid ☑



H → A | valid ☑



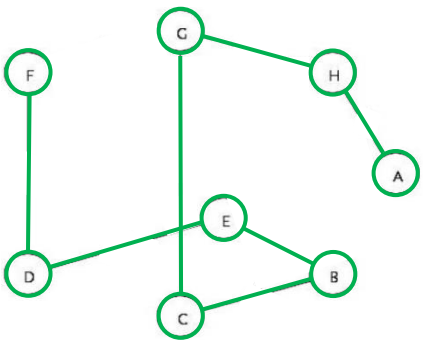
D → F | valid ☑

Tracing completed. All nodes have been travelled. Note that for this question, invalid encounters did not occur. In such case, operator should skip the invalid edge (e.g. edge that creates a cycle) and move on through the list.

Cost = $C_{(E,B)} + C_{(G,C)} + C_{(G,H)} + C_{(D,E)} + C_{(C,B)} + C_{(H,A)} + C_{(D,F)}$

$= 1 + 2 + 2 + 3 + 4 + 4 + 4 = 20$

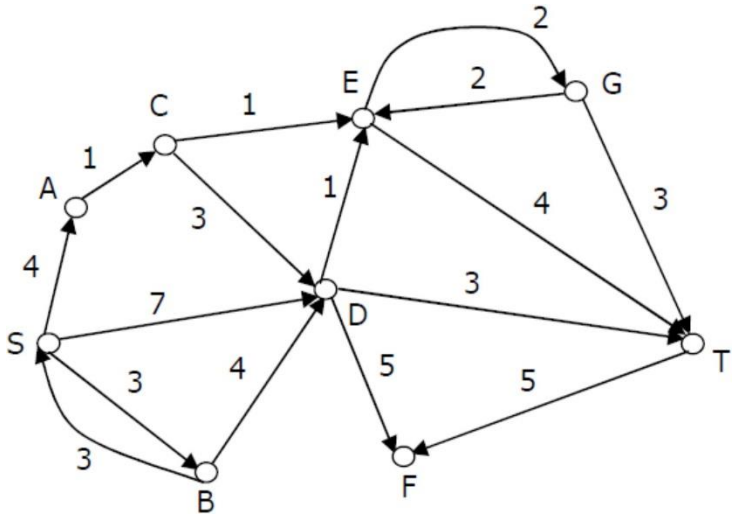
Minimum Spanning Tree in accordance with the initial figure depicted below.



Minimum Spanning Tree

Question 4. Breadth-First Search

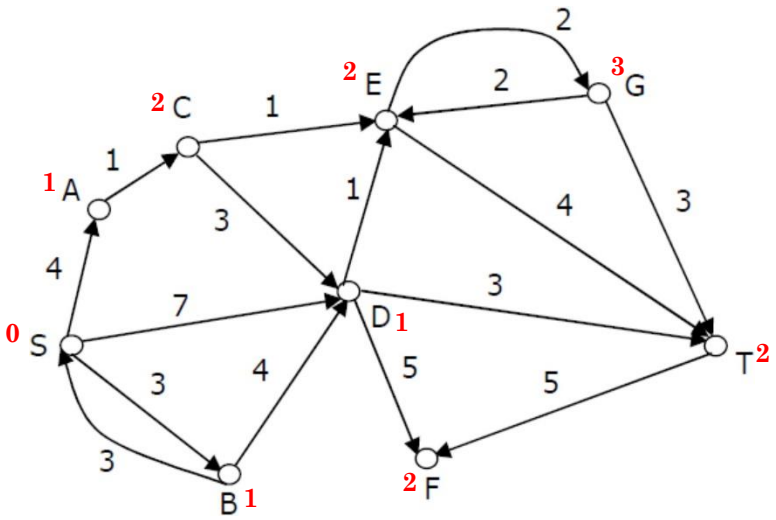
Breadth-First Search is an algorithm for traversing a tree or a graph in a breadthward motion while using a queue for storing the visited nodes (and removing them as it proceeds).



For the above figure, initial node of reference is given as S.

- From S to S, the shortest path is 0.
- Mark A, B and D as 1 (consider vertices 1 away).
- Mark C, E, F and T as 2 (consider vertices 2 away).
- Mark G as 3 (consider vertice 3 away).

Thereupon, the values are inserted as;



Trace operations with respect to Breadth-First Search algorithm for the figure are as follows:

Q refers to a Queue (as in a linear data structure)

1. Start from S
    - Enqueue S. (*Q* = {S})
    - Visit all unvisited adjacent nodes of S which are A, B and D.
    - Enqueue A, B and D. (*Q* = {S, A, B, D})
    - All descendants are visited. Dequeue S. (*Q* = {A, B, D})

2. Continue with A.
    - Visit all unvisited adjacent nodes of A which is C.
    - Enqueue C. (*Q* = {A, B, D, C})
    - All descendants are visited. Dequeue A. (*Q* = {B, D, C})

3. Continue with B.
    - Visit all unvisited adjacent nodes of B. (no unvisited descendants)
    - Dequeue B. (*Q* = {D, C})

4. Continue with D.
    - Visit all unvisited adjacent nodes of D which are E, F and T.
    - Enqueue E, F and T. (*Q* = {D, C, E, F, T})
    - All descendants are visited. Dequeue D. (*Q* = {C, E, F, T})

5. Continue with C.
    - Visit all unvisited adjacent nodes of C. (no unvisited descendants)
    - Dequeue C. (*Q* = {E, F, T})

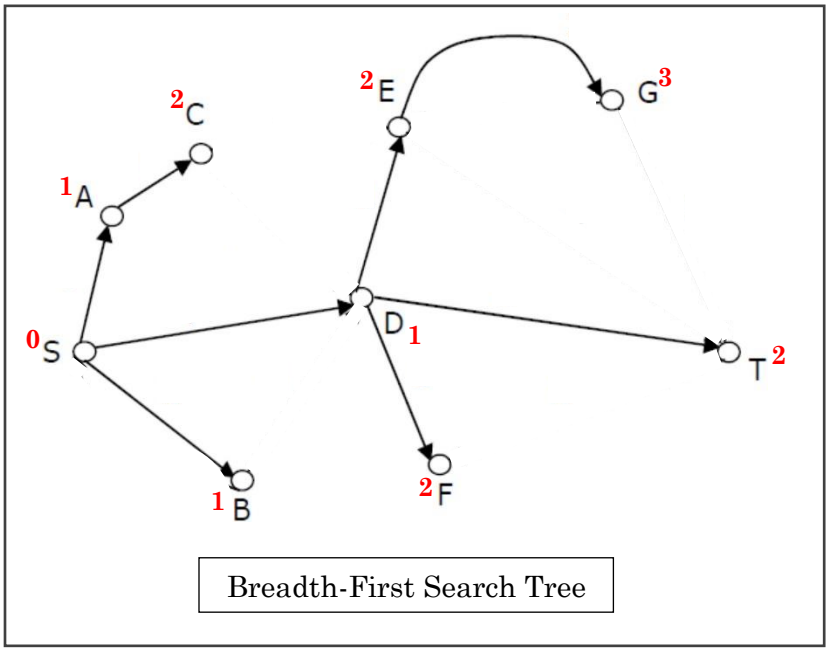
6. Continue with E.
    - Visit all unvisited adjacent nodes of E which is G.
    - Enqueue G. (*Q* = {E, F, T, G})
    - All descendants are visited. Dequeue A. (*Q* = {F, T, G})

7. Continue with F.
    - Visit all unvisited adjacent nodes of F. (no unvisited descendants)
    - Dequeue F. (*Q* = {T, G})

8. Continue with T.
    - Visit all unvisited adjacent nodes of F. (no unvisited descendants)
    - Dequeue T. (*Q* = {G})

9. Continue with G.
    - Visit all unvisited adjacent nodes of F. (no unvisited descendants)
    - Dequeue G. (*Q* = {∅})

Resulting Breadth-First Search Tree depicted below.



Breadth-First Search Tree

Antecedently, the Shortest Path Table is provided below. (Note that in overall, the queue used in this question is formed as Q = {S, A, B, D, C, E, F, T, G })

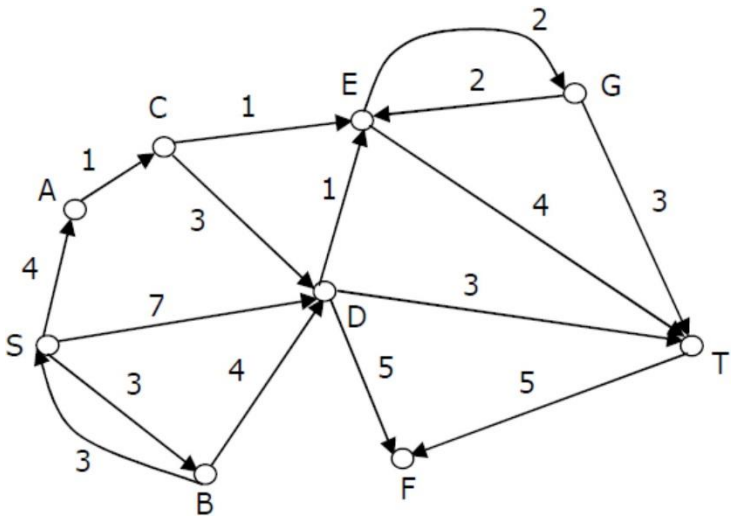
Shortest Path Table

S	A	B	C	D	E	F	G	T
(0, S)	(1, S)	(1, S)	(2, A)	(1, S)	(2, D)	(2, D)	(3, E)	(2, D)

(u, v) = Edge from vertex u to vertex v

Question 5. Depth-First Traversal, Pre-Ordering, Post-Ordering and Arcs

a) Depth-First Search is an algorithm that follows the principle of (starting from root) exploring the nodes as far as possible along each branch without backtracking. Throughout the iterative implementation, an explicit stack is used to hold visited vertices.



Trace operations with respect to Depth-First Search algorithm for the figure are as follows:

S' refers to a Stack (as in abstract data type)

1. Start from S
    - Push S. (*S'* = {S})
    - Visit one of the unvisited adjacent node of S which is B.
    - Push B. (*S'* = {S, B})

2. Continue with B.
    - Visit an unvisited adjacent node of B which is D.
    - Push D. (*S'* = {S, B, D})

3. Continue with D.
    - Visit an unvisited adjacent node of D which is F.
    - Push F. (*S'* = {S, B, D, F})
    - F has no descendants to visit. Pop F. (*S'* = {S, B, D})
    - Push T. (*S'* = {S, B, D, T})
    - T has no unvisited descendants. Pop T. (*S'* = {S, B, D})
    - Push E. (*S'* = {S, B, D, E})

4. Continue with E.
    - Visit an unvisited adjacent node of E which is G.
    - Push G. (*S'* = {S, B, D, E, G})
    - G has no unvisited descendants. Pop G. (*S'* = {S, B, D, E})

5. Backtrack to S.
    - Pop all elements up to S. (*S'* = {S})
    - Visit an unvisited adjacent node of S which is A.
    - Push A. (*S'* = {S, A})

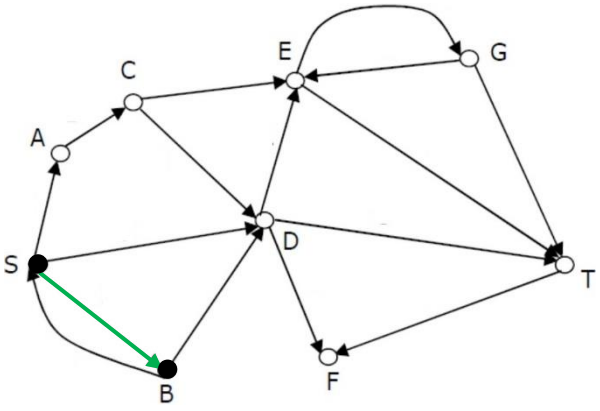
6. Continue with A.
    - Visit an unvisited adjacent node of A which is C.
    - Push C. (S' = {S, A, C})
    - C has no unvisited descendants. Pop C. (S' = {S, A})

7. Backtrack to S again.
    - Pop all elements up to S. (S' = {S, A})
    - S has no unvisited descendants. Pop S. (S' = {∅})

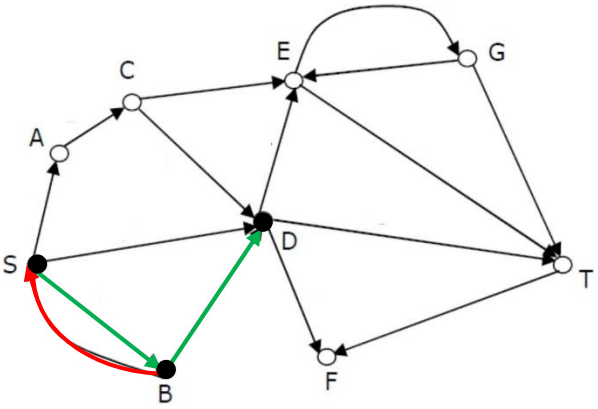
All nodes have been visited and operations are complete. An output string (if it would have been asked) would be as S, B, D, F, T, E, G, A, C.

To trace the graph in terms of Depth-First Traversal method; start at some unvisited node and visit any unvisited adjacent nodes.

Starting from S:

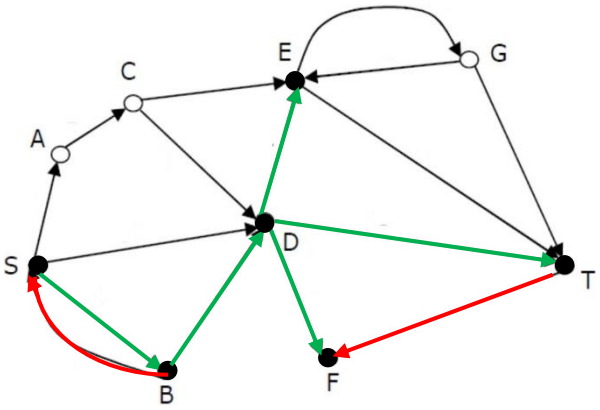


S → B | valid ☑

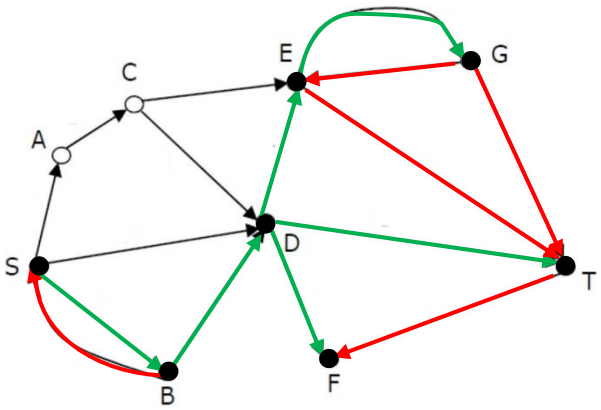


B → S | invalid ☒

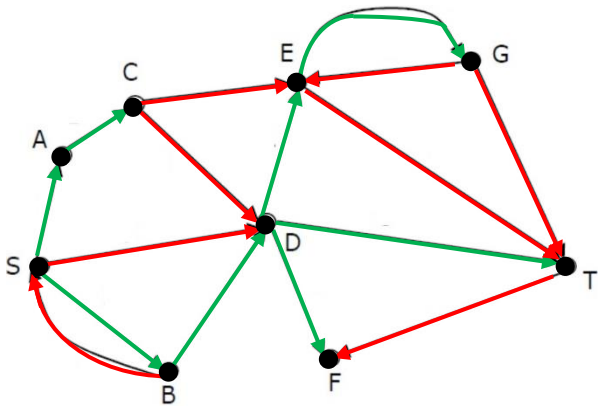
B → D | valid ☑



D → F | valid ☑
D → T | valid ☑
T → F | invalid ☒
D → E | valid ☑



E → T | invalid ☒
E → G | valid ☑
G → T | invalid ☒
G → E | invalid ☒



S → D | invalid ☒
S → A | valid ☑
A → C | valid ☑
C → D | invalid ☒
C → E | invalid ☒

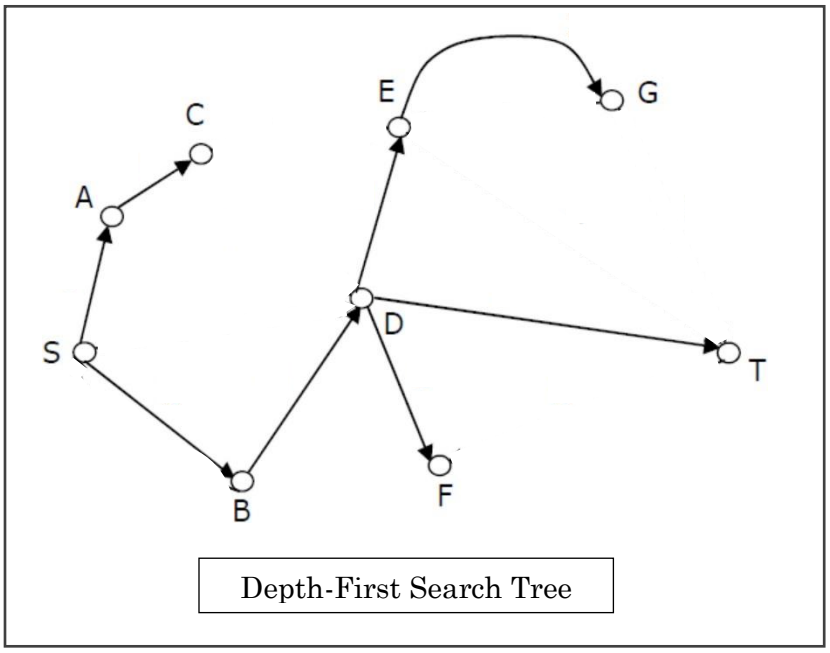
Note that;

(u, v) = Edge from vertex u to vertex v

Valid ☑: For $u \rightarrow v$, if v is an unvisited adjacent vertex of u.

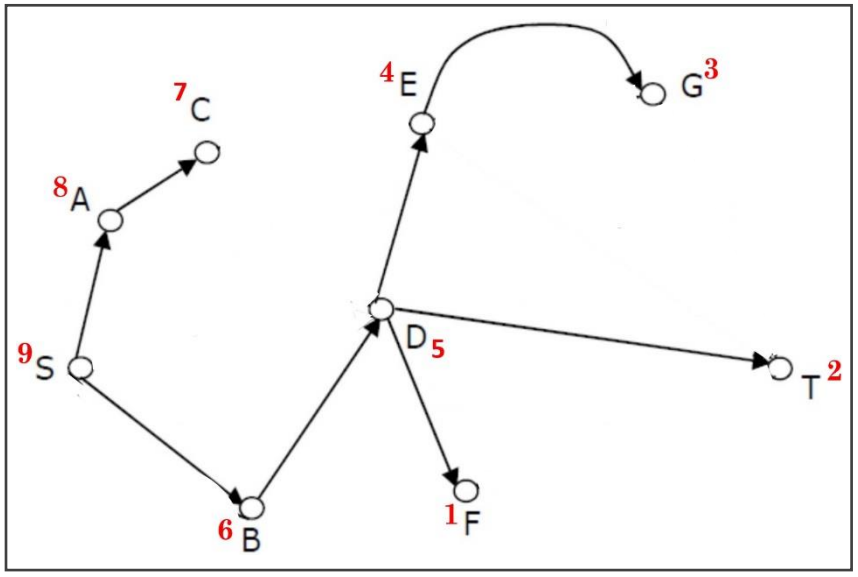
Invalid ☒: For $u \rightarrow v$, if v is a visited (e.g. already in the known vertices set) adjacent vertex of u.

All nodes have been travelled. The Depth-First Tree of the figure depicted below. Note that Depth-First Tree of a graph is not necessarily unique.



Depth-First Search Tree

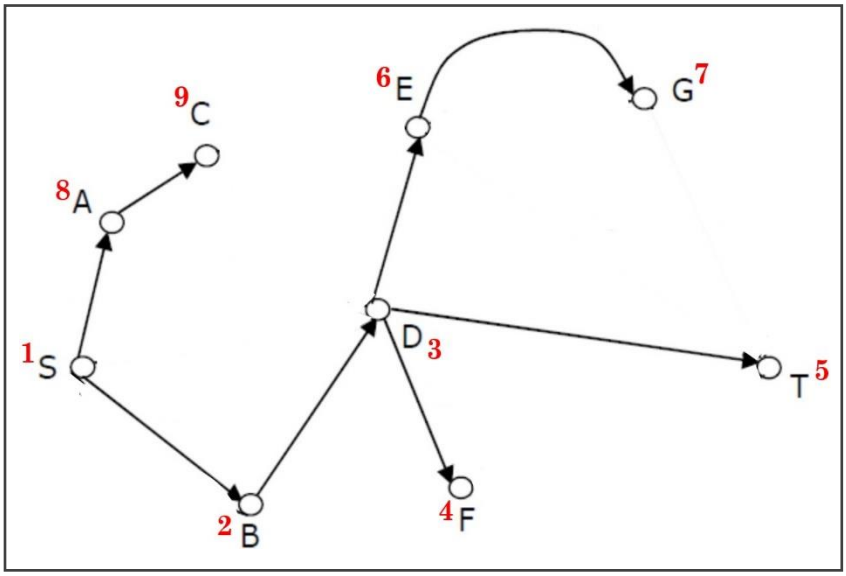
b) Post-Order numbers of each node in reference to the figure are provided below.



Post-Order Numbers

S	A	B	C	D	E	F	G	T
9	8	6	7	5	4	1	3	2

c) Pre-Order numbers of each node in reference to the figure are provided below.



Pre-Order Numbers

S	A	B	C	D	E	F	G	T
1	8	2	9	3	6	4	7	5

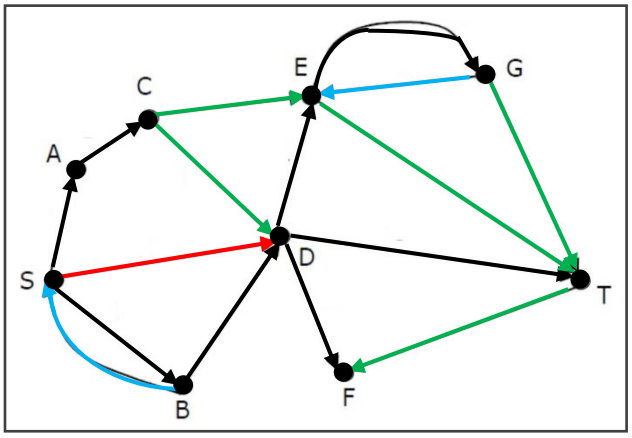
d) Depth-First Search partitions the arcs of a directed graph into tree arcs, cross arcs, forward arcs and backward arcs.

(u, v) = Edge from vertex u to vertex v

For $u \rightarrow v$

- Tree Arc: Arcs that form the Depth-First Search Tree.
- Cross Arc: v is neither a descendant nor an ancestor of u.
- Forward Arc: v is a proper descendant of u, but not a child of u in Depth-First Search Tree.
- Backward Arc: v is an ancestor of u in Depth-First Search Tree.

List of tree arcs, cross arcs, forward arcs and backwards arcs are depicted in the graph below.



tree arc | **cross** arc | **forward** arc | **backward** arc

Followingly, arcs represented as a list are provided below.

Tree	**Cross**	**Forward**	**Backward**
S → B	C → E	S → D	B → S
B → D	C → D	-	G → E
D → F	E → T	-	-
D → T	T → F	-	-
D → E	G → T	-	-
E → G	-	-	-
S → A	-	-	-
A → C	-	-	-

For $u \rightarrow v$ and PON(u) = Post-Order number of u

- If $u \rightarrow v$ is a cross arc then
  PON(v) < PON(u)

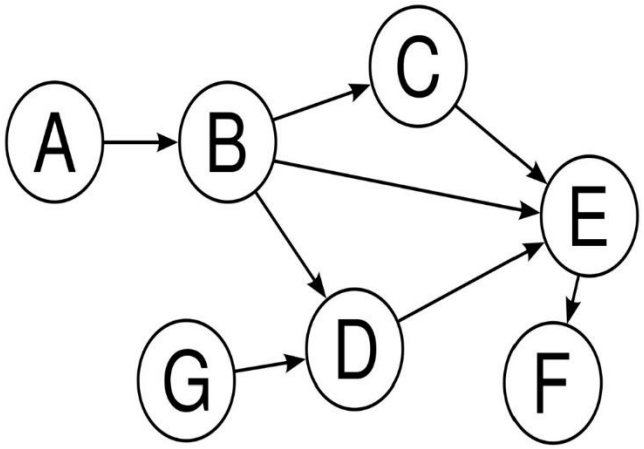
- If $u \rightarrow v$ is a forward arc then
  PON(v) < PON(u)

- If $u \rightarrow v$ is a backward arc <u>if and only if</u>
  PON(v) > PON(u) [note that u and v vertices are now switched]

Question 6. Topological Ordering

Topological Sorting is a linear ordering of vertices in a directed acyclic graph. If there is a path from u to v (*u* having in-degree 0) then v appears after u in the ordering.

(*u*, *v*) = Edge from vertex *u* to vertex *v*



Tracing of nodes are done with respect to the following set of directions.

- Select a vertex with in-degree 0.
- Do something. (Print in this case)
- Remove the vertex from the graph.
- Repeat the steps.

Trace operations with respect to Topological Sorting for the figure are as follows:

P = Set of Topological Ordering, V = Set of vertices with in-degree 0

1. *V* = {A, G}
   - Print and remove A from the graph. (*P* = {A})
2. *V* = {G, B}
   - Print and remove B from the graph. (*P* = {A, B})
3. *V* = {G, C}
   - Print and remove C from the graph. (*P* = {A, B, C})
4. *V* = {G}
   - Print and remove G from the graph. (*P* = {A, B, C, G})
5. *V* = {D}
   - Print and remove D from the graph. (*P* = {A, B, C, G, D})
6. *V* = {E}
   - Print and remove E from the graph. (*P* = {A, B, C, G, D, E})
7. *V* = {F}
   - Print and remove F from the graph. (*P* = {A, B, C, G, D, E, F})

Ultimately, *P* = {A, B, C, G, D, E, F}. Thus, Topological Ordering of this graph is A, B, C, G, D, E, F. Note that a Topological Sort is not necessarily unique.