Data Structures

Graphs

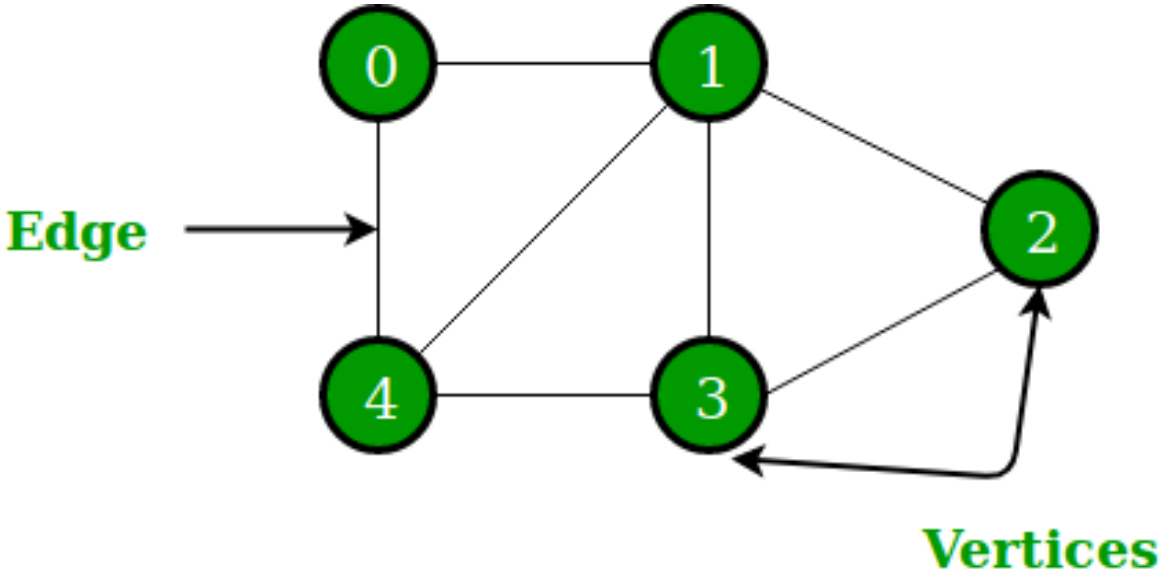
Rutvij H. Jhaveri
Computer Science & Engineering

Outline

- What is Graph
- Directed and Undirected Graphs
- Applications
- Weighted Graphs
- Spanning Tree and Minimum Spanning Tree
- Kruskal's Algorithm
- Prim's Algorithm
- BFS and DFS

Graph

- A Graph is a non-linear data structure consisting of nodes and edges.
  - The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.



Formal Definition

- A graph G = (V, E) consists of a set of vertices V = { $V_1, V_2, \ldots$ } and set of edges E = {$E_1, E_2, \ldots$ }.
- The unordered pairs of distinct vertices are called **edges** of graph G and each edge is identified with an unordered pair (V_i, V_j) of **vertices**.
- The vertices (V_i, V_j) are said to be **adjacent** if there is an edge E_k which is associated to V_i and V_j.

Terminology

- Adjacent nodes: two nodes are adjacent if connected by an edge
- Path: a sequence of vertices that connect two nodes in a graph
- **Order of the graph** = The number of vertices in the graph.
- **Size of the graph** = The number of edges in the graph.
- **Degree of a vertex of a graph** = Number of edges incident to the vertex
- What is the number of edges in a complete directed graph with N vertices?

  *N * (N-1)*
- What is the number of edges in a complete undirected graph with N vertices?

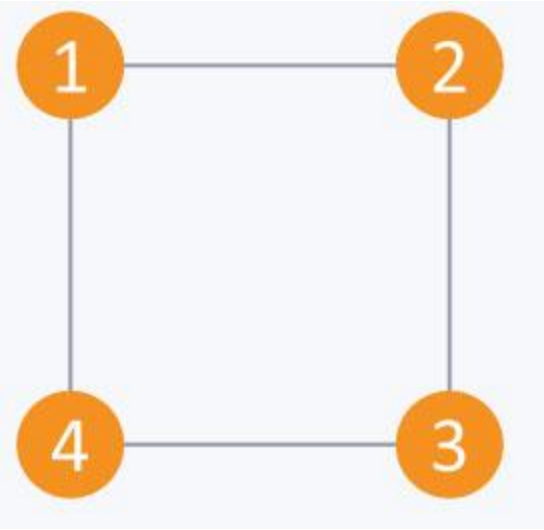
  *N * (N-1) / 2*

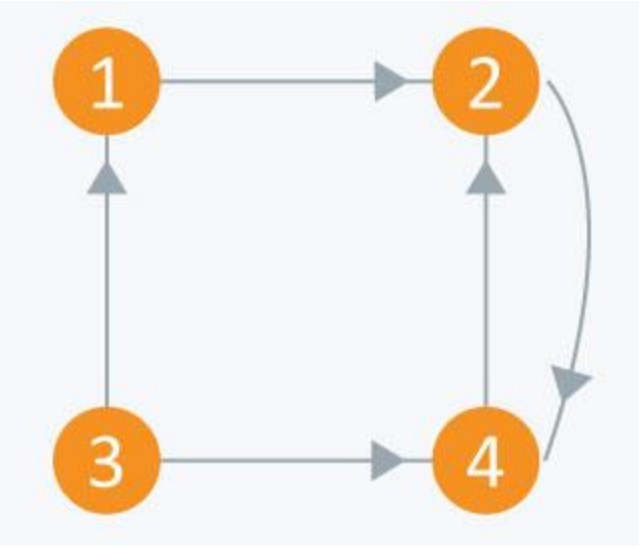
Applications

- Social network graphs: An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop and more.
- Google Maps: Various locations are represented as vertices and the roads are represented as edges and graph theory is used to find shortest path between two nodes.
- Network packet traffic graphs. Vertices are IP addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity
- Compilers. Graphs can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
- Other applications in neural networks, quantum field theory, Semantic networks and more

Undirected Vs Directed Graph

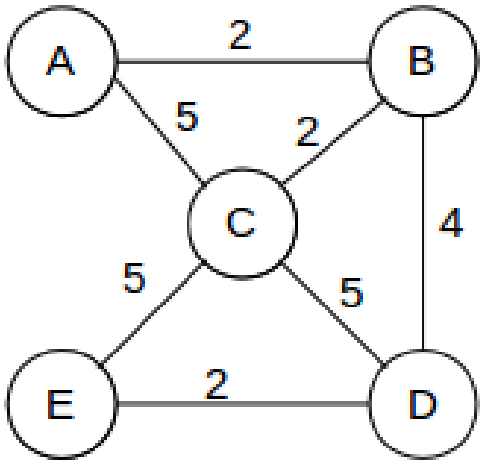
- Undirected: An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction. [Tree is an undirected graph]
- Directed: A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



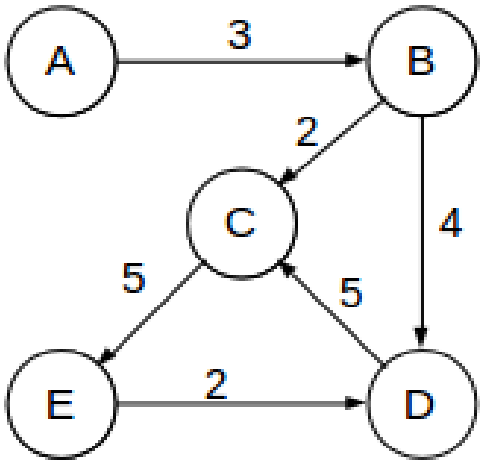


Weighted Graph

- Each edge is assigned a weight or cost.



Undirected

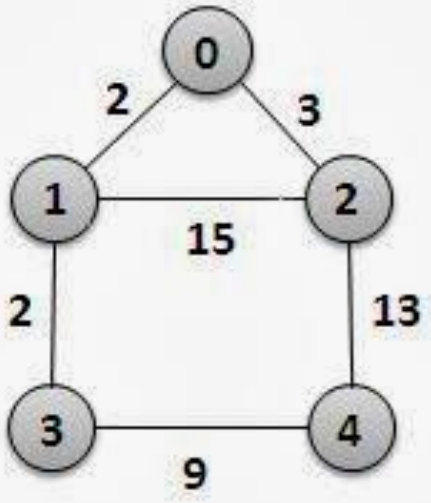


Directed

Adjacency Matrix

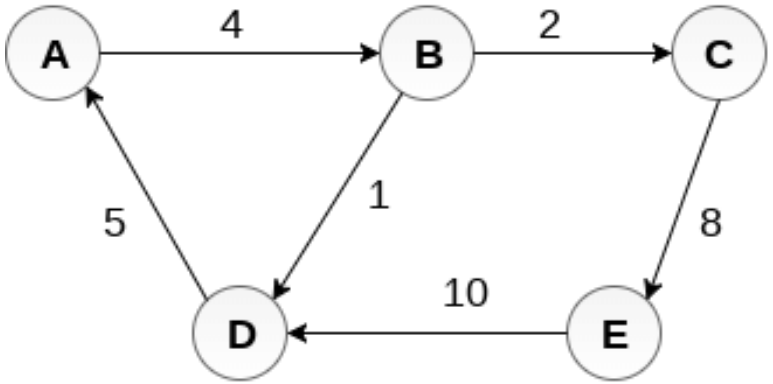
- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- Adjacency matrix for undirected graph is always symmetric, while it is asymmetric for directed graph.
- Adjacency Matrix is also used to represent weighted graphs.
- If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Adjacency Matrix: Undirected Weighted Graph



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

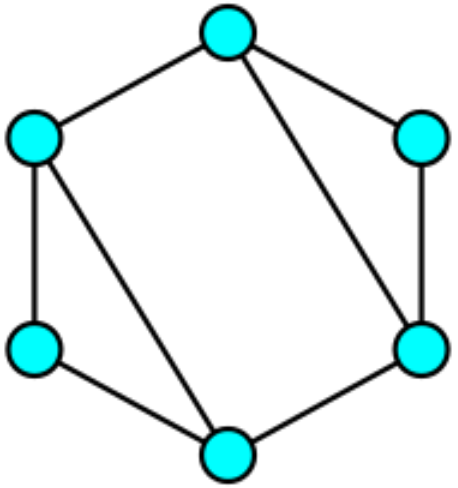
Adjacency Matrix: Directed Weighted Graph



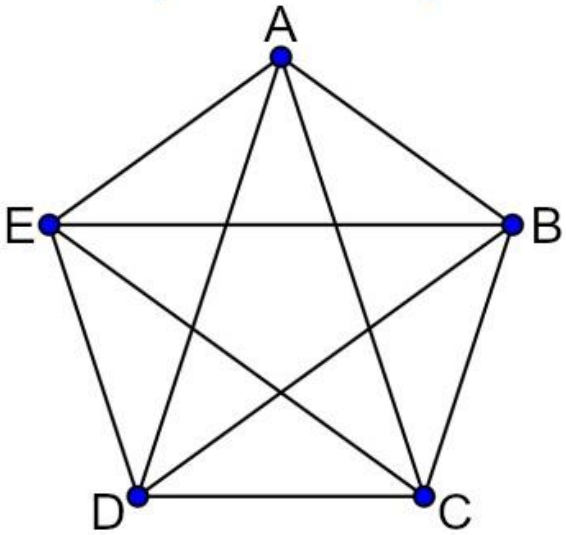
	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Connected Vs Complete Graphs

- A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph.



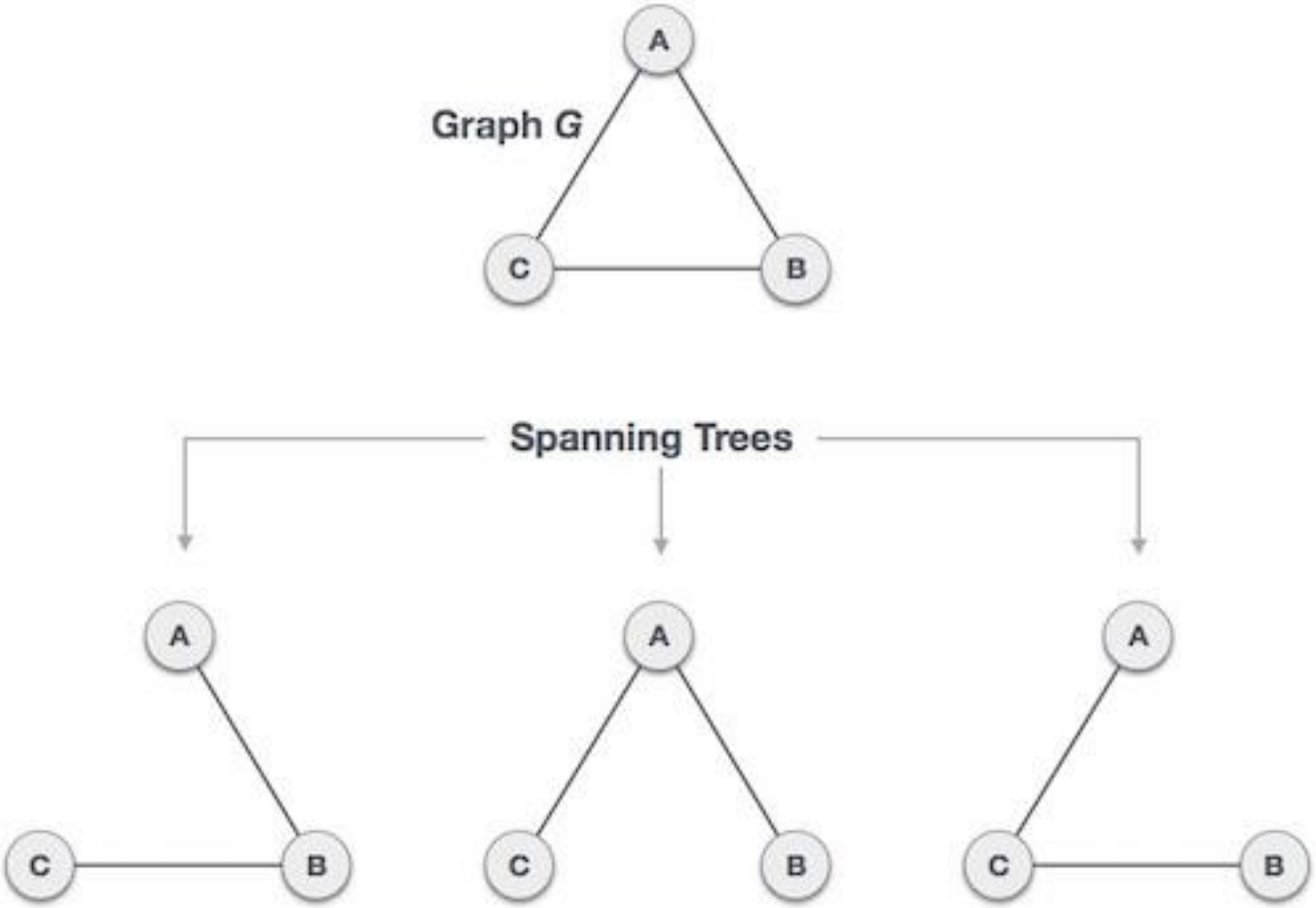
- A simple graph with 'n' mutual vertices is called a complete graph i.e. **a vertex should have edges with all other vertices**.



Spanning Tree

- Given a connected undirected graph, a ***spanning tree*** of that graph is a subgraph that is a tree and connects all the vertices together.
- Properties of a spanning tree connected to graph G:
  - Connected graph G can have more than one spanning trees
  - All possible spanning trees of graph G, have the same number of edges and vertices
  - The spanning tree does not have any cycle (loops)
  - Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**
  - Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**

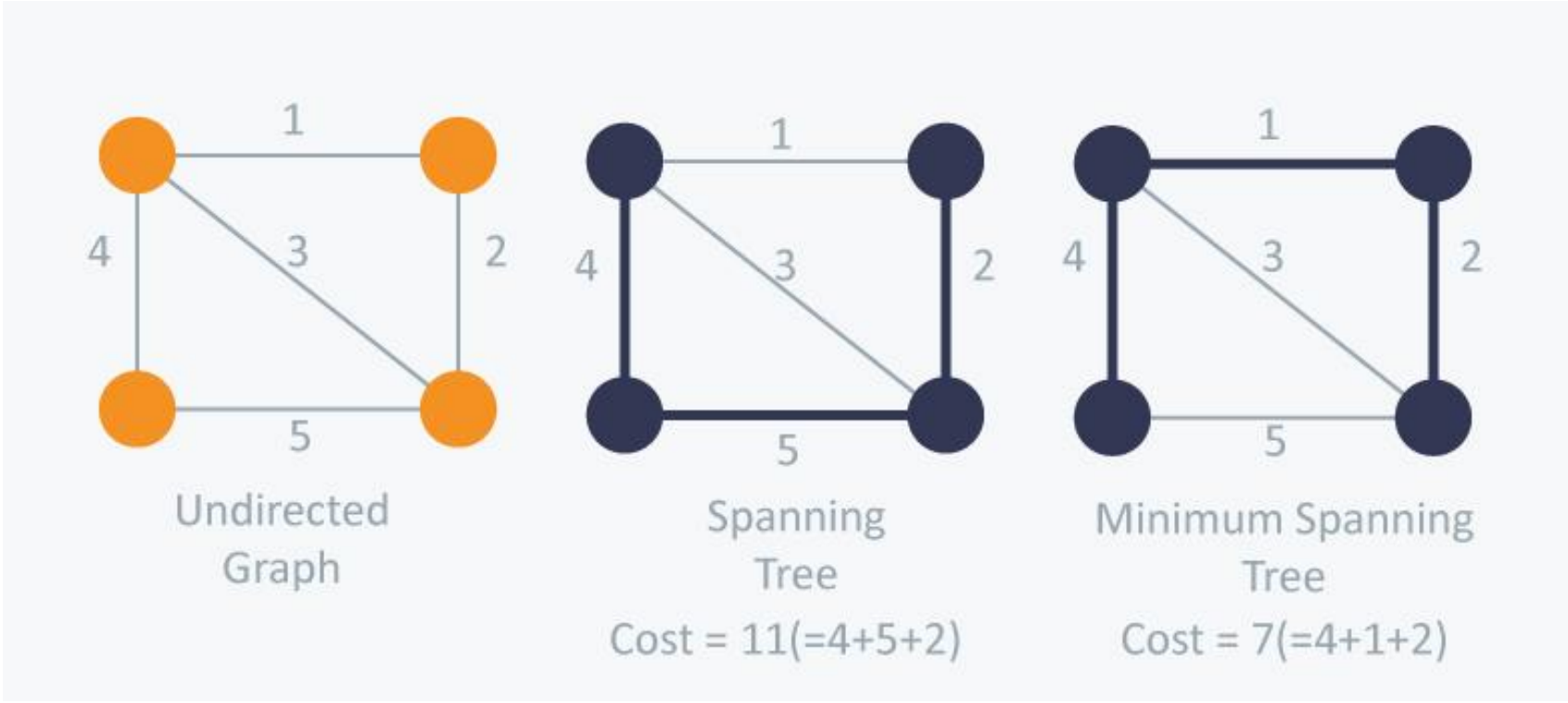
…Spanning Tree



Minimum Spanning Tree (MST)

- For a weighted Graph, the weight of a spanning tree is the sum of weights given to each edge of the spanning tree.
- A ***minimum spanning tree (MST)*** or minimum weight spanning tree for a weighted, connected undirected graph is a spanning tree with the least cost (total weight of all edges).
  - There can be more than one MST for a graph.
- **Applications**
  - **Network design**
  - **Approximation algorithms for NP-hard problems**
  - **Cluster analysis (for k-clustering problem)**
  - **…**

…Minimum Spanning Tree (MST)



Kruskal's Algorithm to Find MST

- Step 1: Sort all the edges in non-decreasing order of their weight.
- Step 2: Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
- Step 3: Repeat Step 2 until there are (V-1) edges in the spanning tree.

…Kruskal's Algorithm to Find MST

- MAKE-SET(v) **puts** v **in a set by itself**
- FIND-SET(v) **returns the name of** v**'s set**
- UNION(u, v) **combines the sets that** u **and** v **are in**

MST-Kruskal(G, w)

$A \leftarrow \emptyset$

$COST \leftarrow 0$

for each vertex $v \in V[G]$

  do MAKE-SET(v)

sort the edges of E **into nondecreasing order by weight** w

for each edge $(u, v) \in E$**, taken in nondecreasing order by weight**

  do if FIND-SET(u) $\neq$ FIND-SET(v)

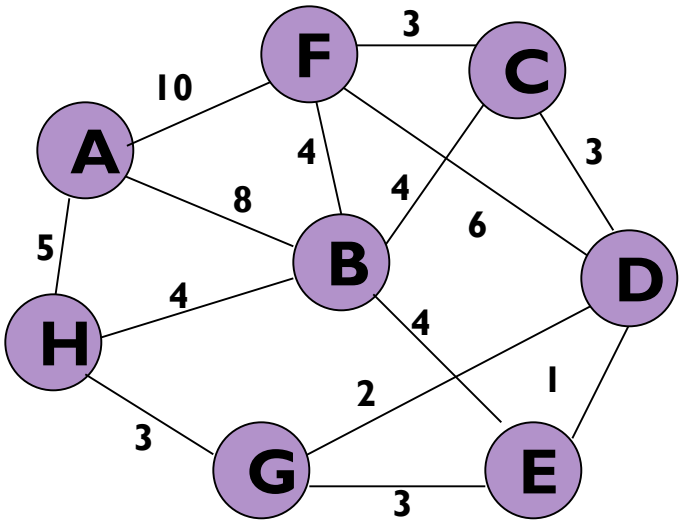
    then $A \leftarrow A \cup \{(u, v)\}$

      UNION(u, v)

      $COST \leftarrow COST + w(u, v)$

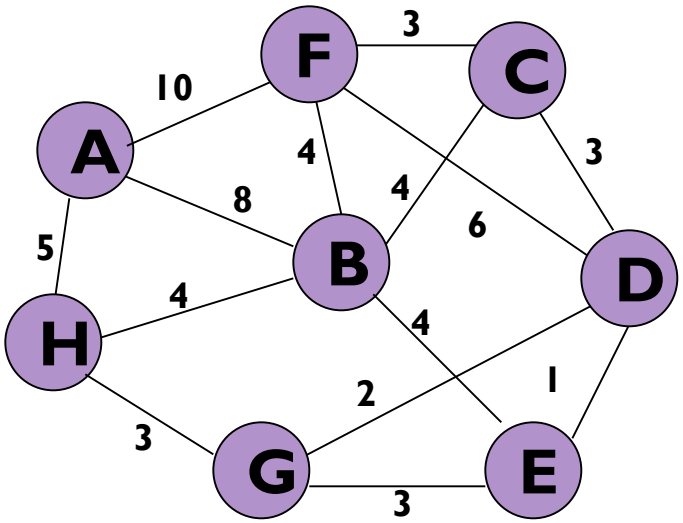
return A, $COST$

Example: Kruskal's Algorithm



Consider an undirected, weight graph

…Example:Kruskal's Algorithm

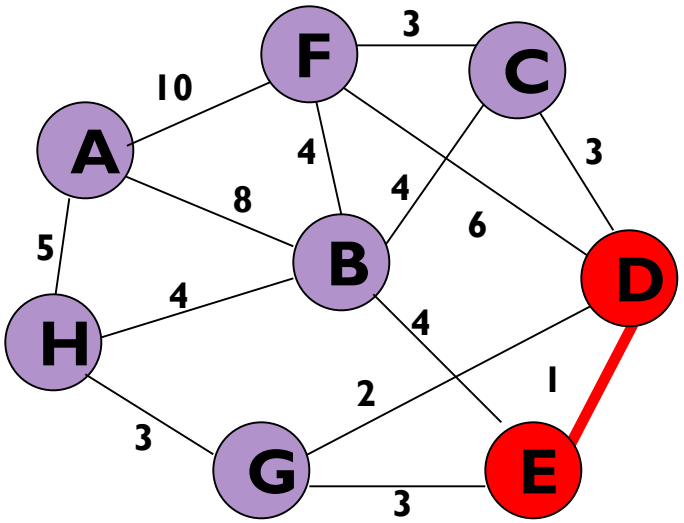


Sort the edges by increasing edge weight

edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm

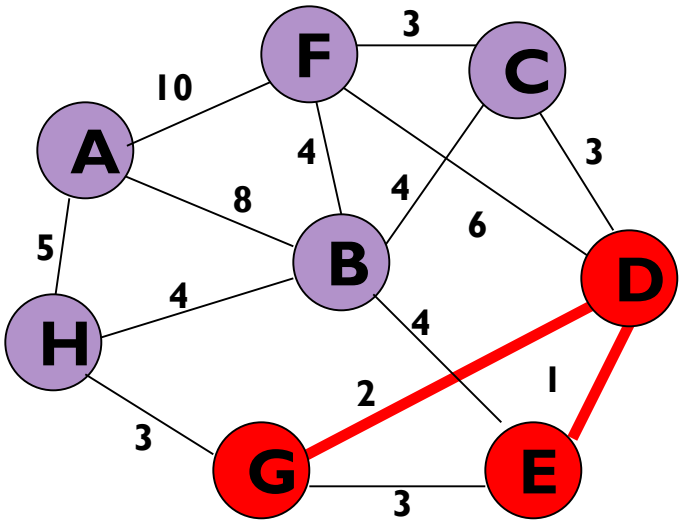


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm

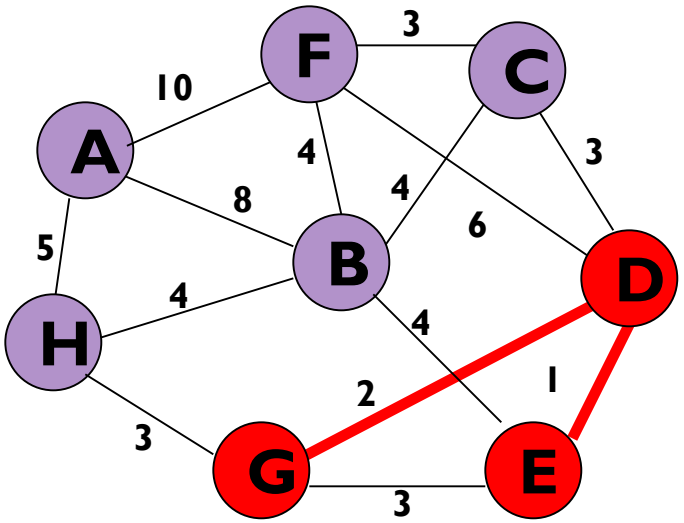


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm



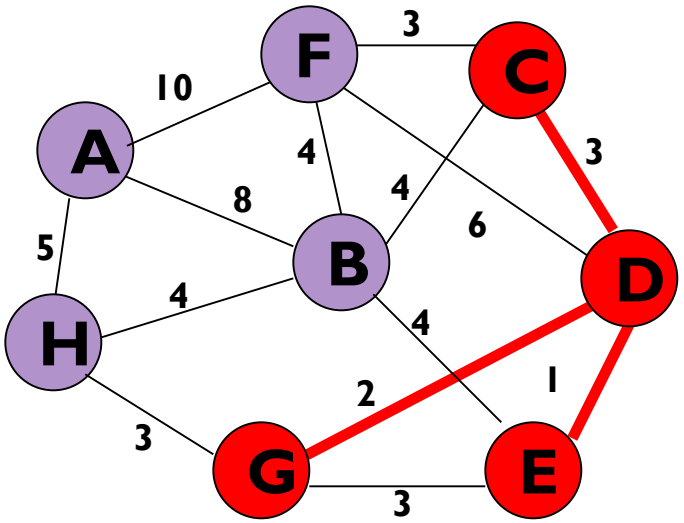
Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Edge (E,G) would create a cycle

…Example:Kruskal's Algorithm

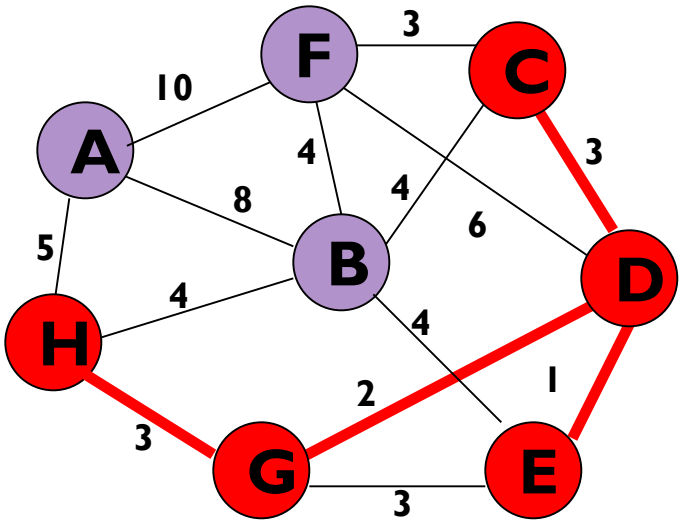


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm

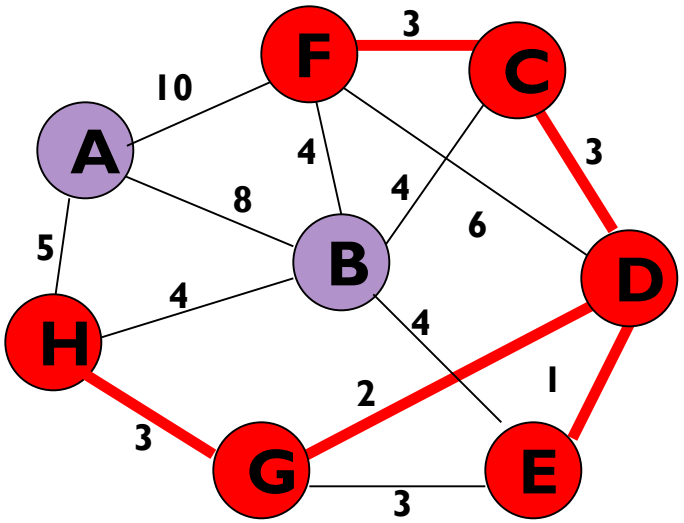


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm

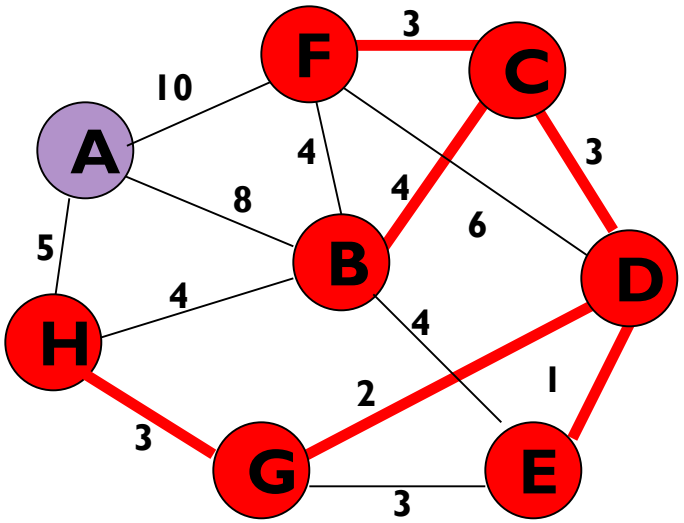


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm

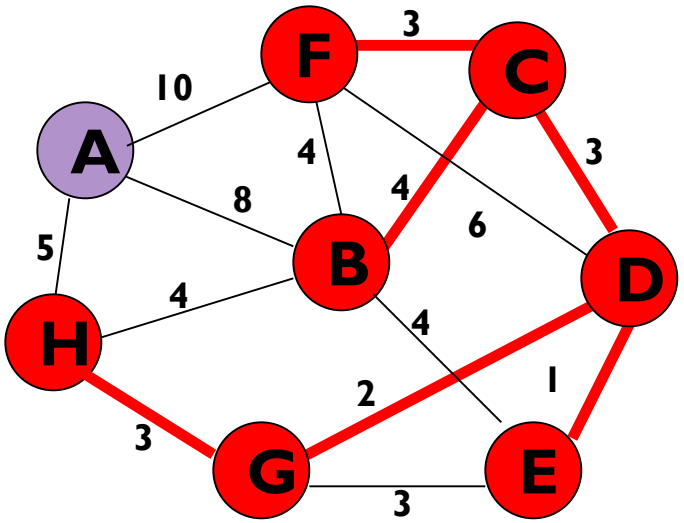


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm

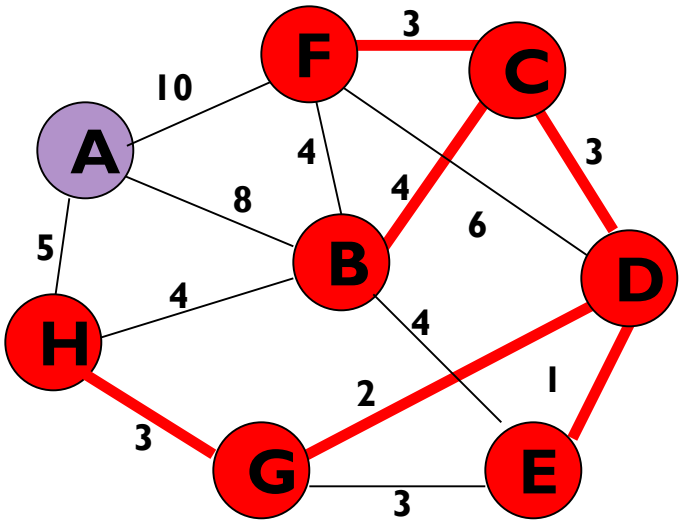


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm

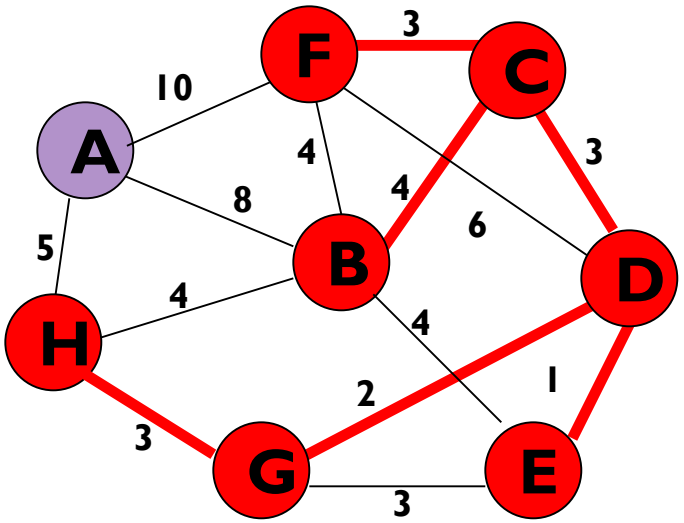


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm

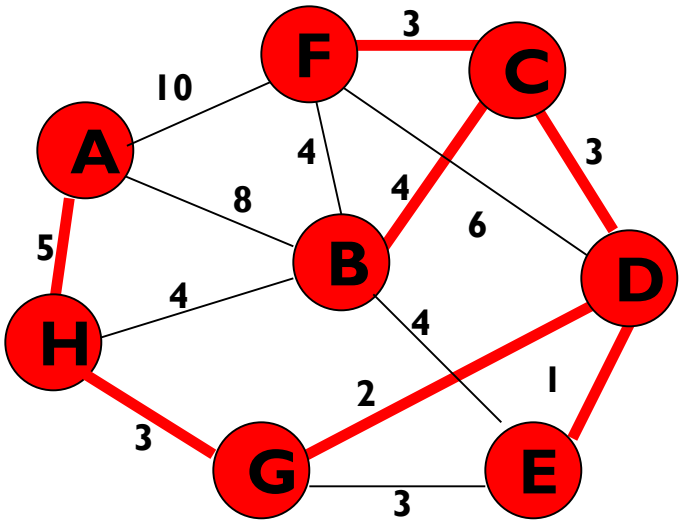


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm

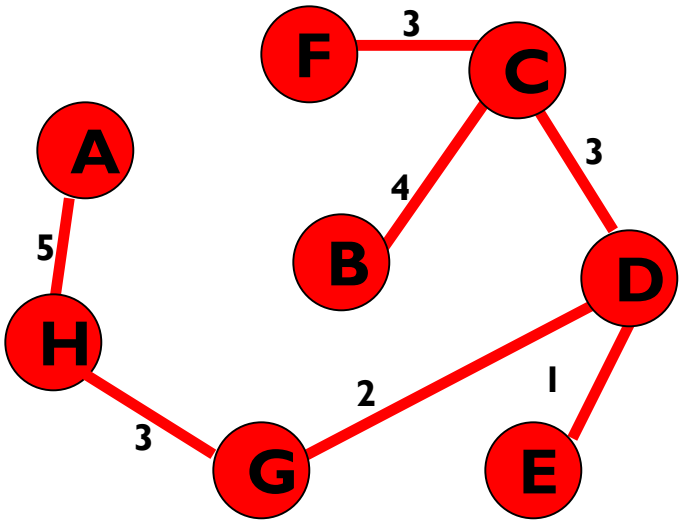


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

…Example:Kruskal's Algorithm



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

} not considered

We have now V-1 edges i.e. 7

Total Cost of MST = $\Sigma\, d_v = 21$

Prim's Algorithm to Find MST

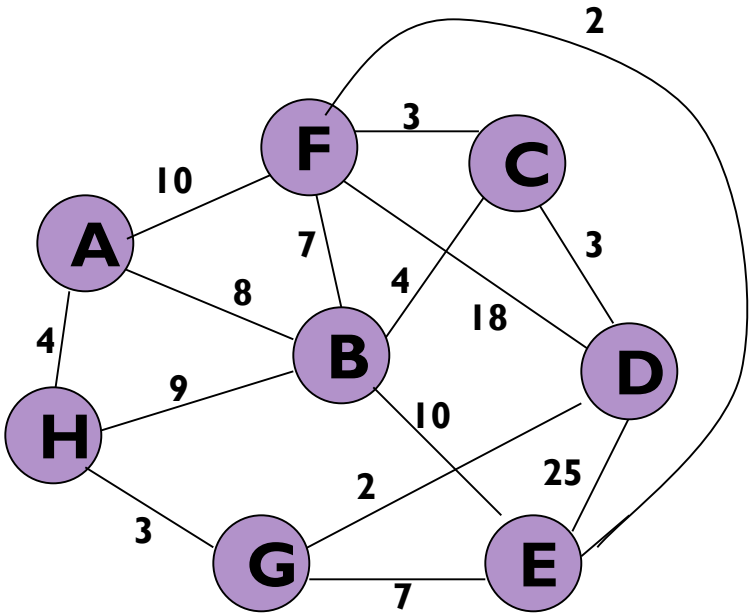
- Step 1: Initialize the minimum spanning tree with a vertex chosen at random.
- Step 2: Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree.
- Step 3: Keep repeating step 2 until we get a minimum spanning tree.

…Prim's Algorithm to Find MST

- INSERT(v) **puts** v **in the structure**
- EXTRACT-MIN() **finds and returns the node with minimum key value**
- DECREASE-KEY(v, w) **updates (decreases) the key of** v

```
MST-Prim(G, w, r)
 1  for each u ∈ V[G]
 2        do key[u] ← ∞
 3           π[u] ← NIL
 4  key[r] ← 0
 5  Q ← V[G]
 6  while Q ≠ ∅
 7        do u ← EXTRACT-MIN(Q)
 8           for each v ∈ Adj[u]
 9                 do if v ∈ Q and w(u, v) < key[v]
10                       then π[v] ← u
11                            key[v] ← w(u, v)
```

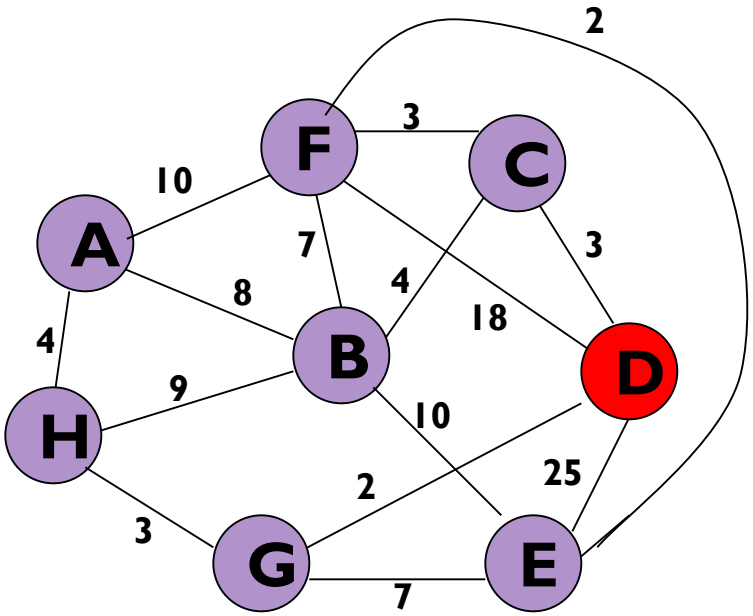
Example: Prim's Algorithm



Initialize array

	K	d_v	p_v
A	F	∞	–
B	F	∞	–
C	F	∞	–
D	F	∞	–
E	F	∞	–
F	F	∞	–
G	F	∞	–
H	F	∞	–

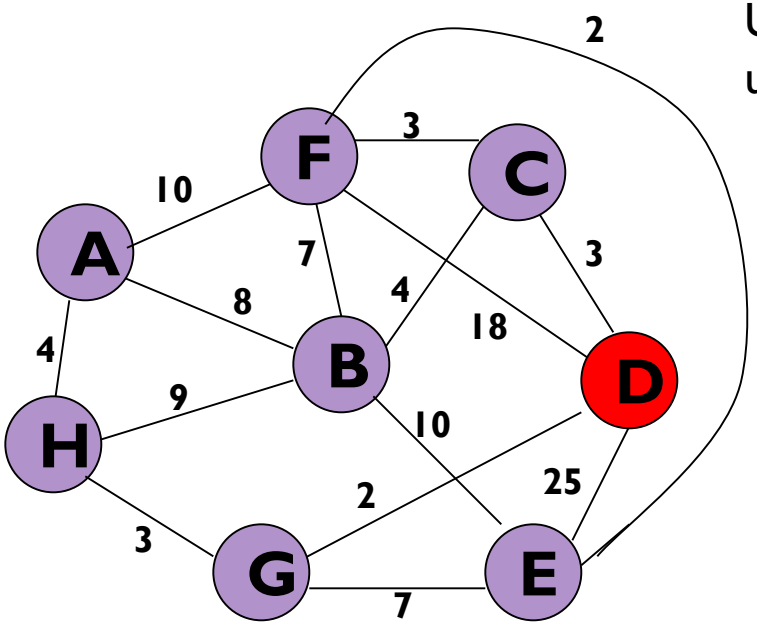
…Example:Prim's Algorithm



Start with any node, say D

	K	d_v	p_v
A			
B			
C			
D	T	0	–
E			
F			
G			
H			

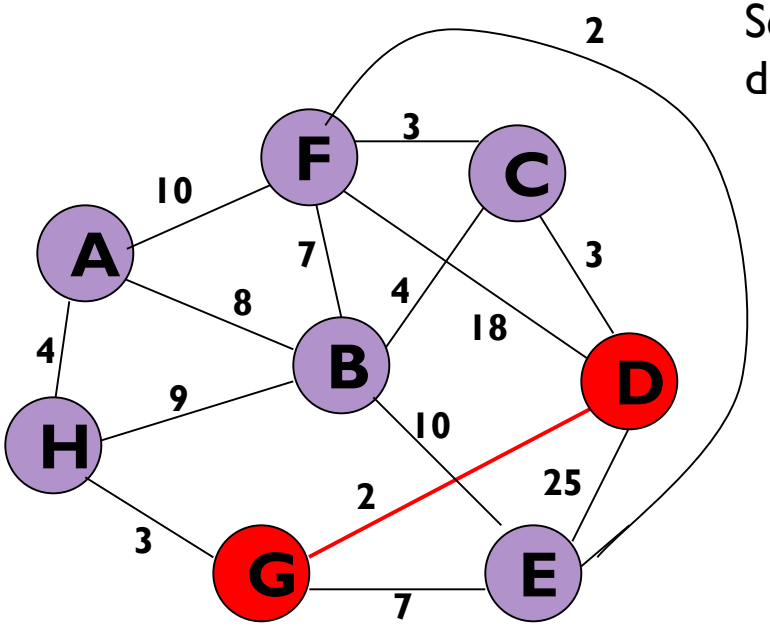
…Example:Prim's Algorithm



Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G		2	D
H			

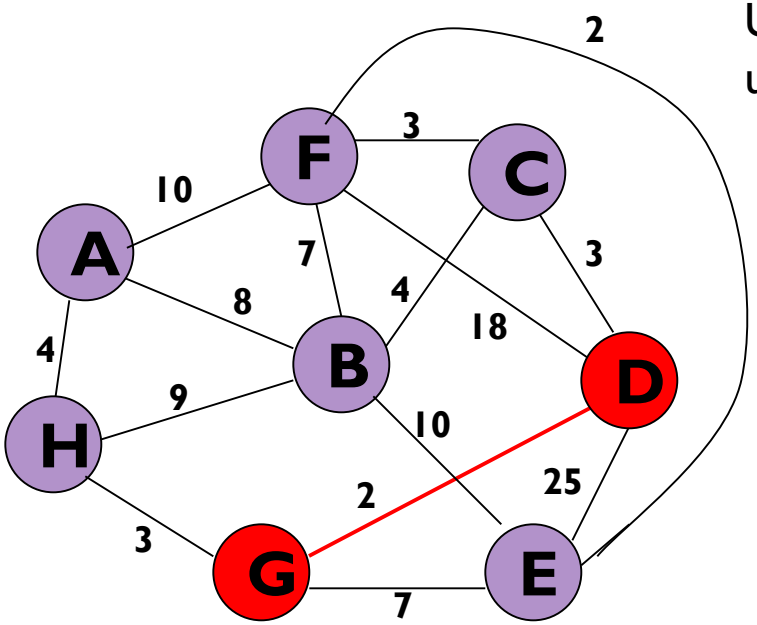
…Example: Prim's Algorithm



Select node with minimum distance

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G	T	2	D
H			

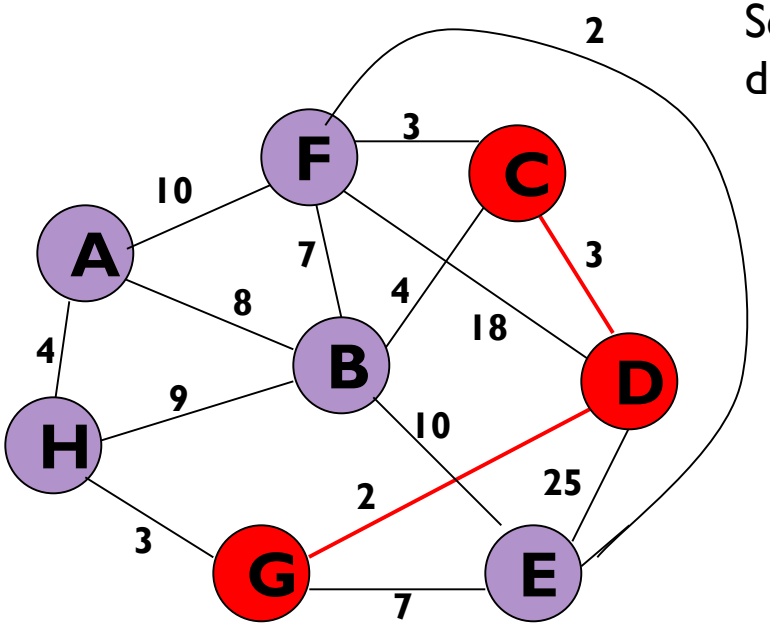
…Example:Prim's Algorithm



Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G

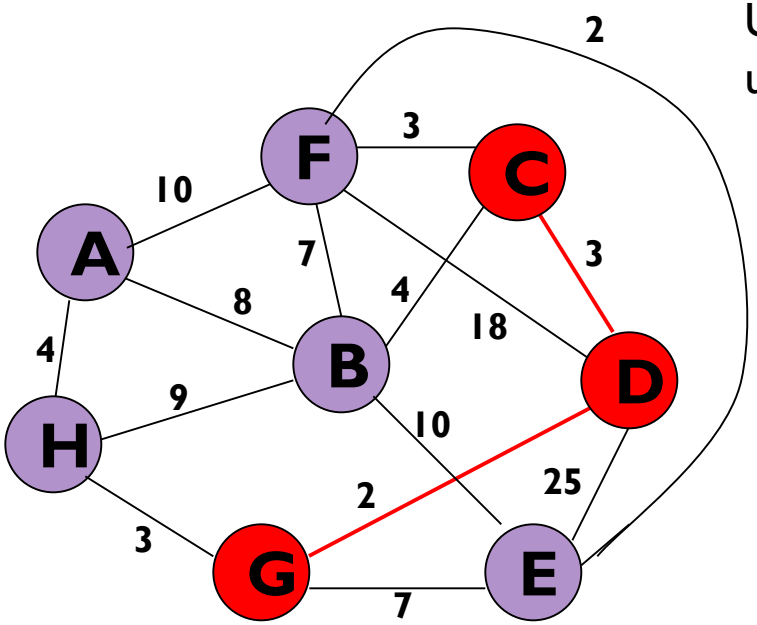
…Example:Prim's Algorithm



Select node with minimum distance

	K	d_v	p_v
A			
B			
C	T	3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G

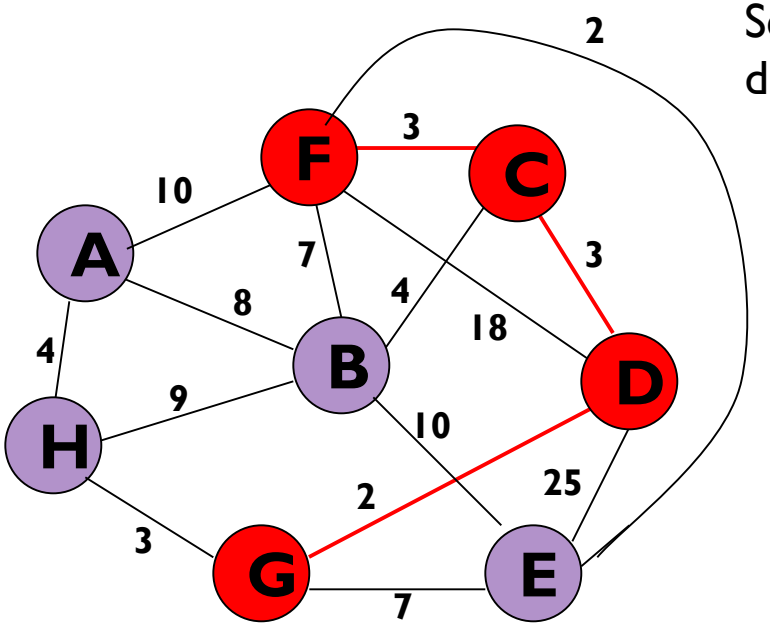
…Example:Prim's Algorithm



Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F		3	C
G	T	2	D
H		3	G

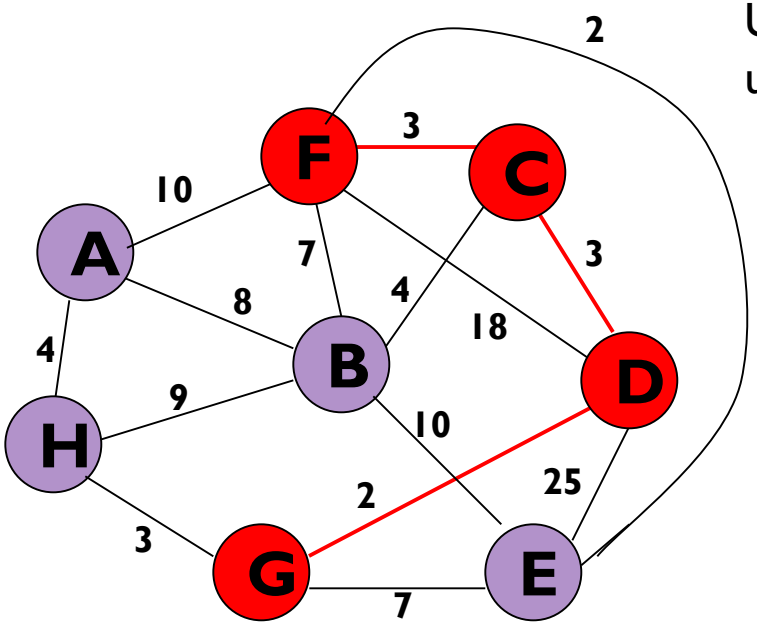
…Example:Prim's Algorithm



Select node with minimum distance

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F	T	3	C
G	T	2	D
H		3	G

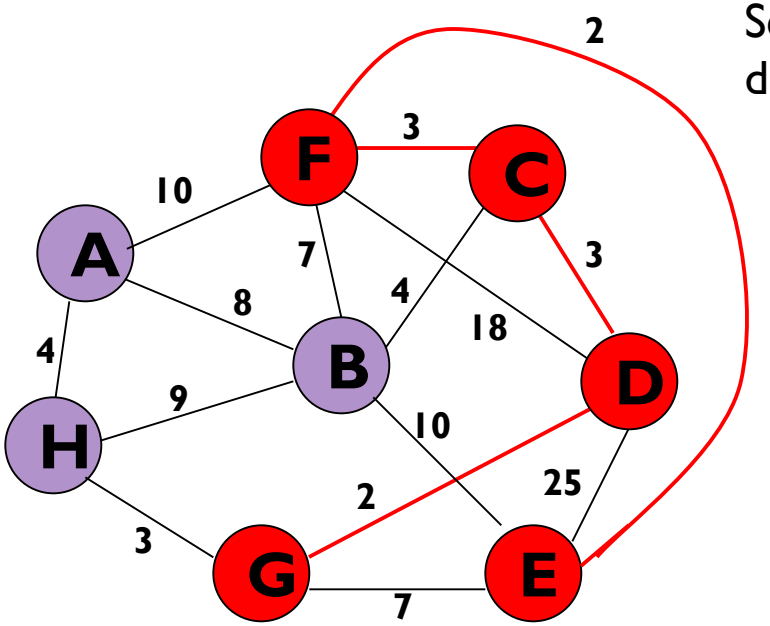
…Example:Prim's Algorithm



Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E		2	F
F	T	3	C
G	T	2	D
H		3	G

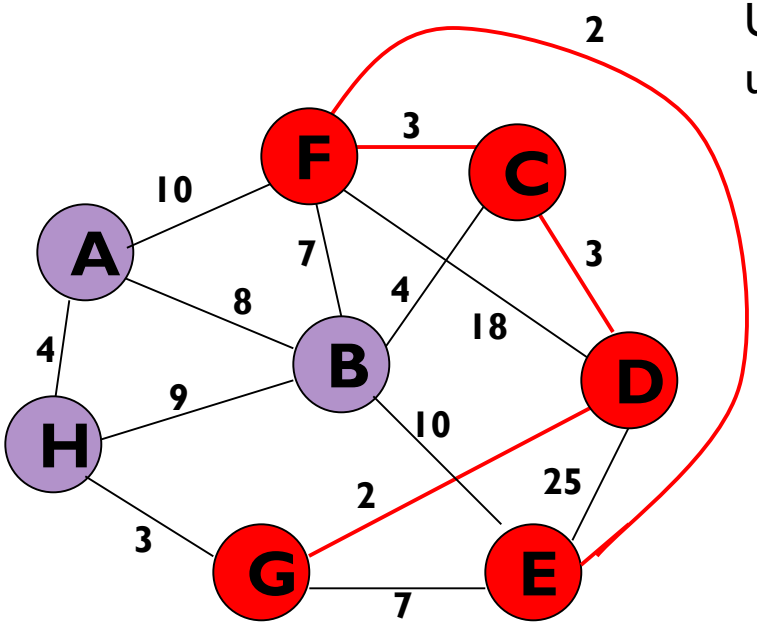
…Example:Prim's Algorithm



Select node with minimum distance

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

…Example:Prim's Algorithm

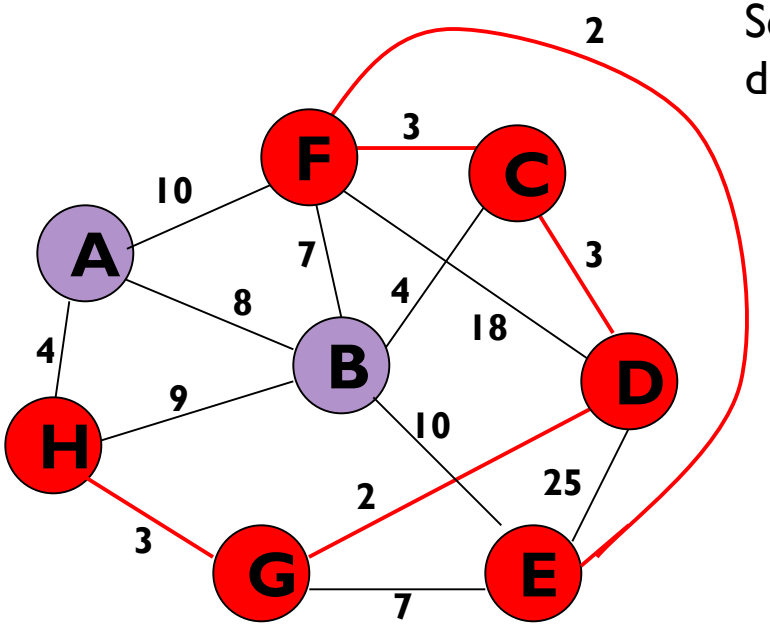


Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Table entries unchanged

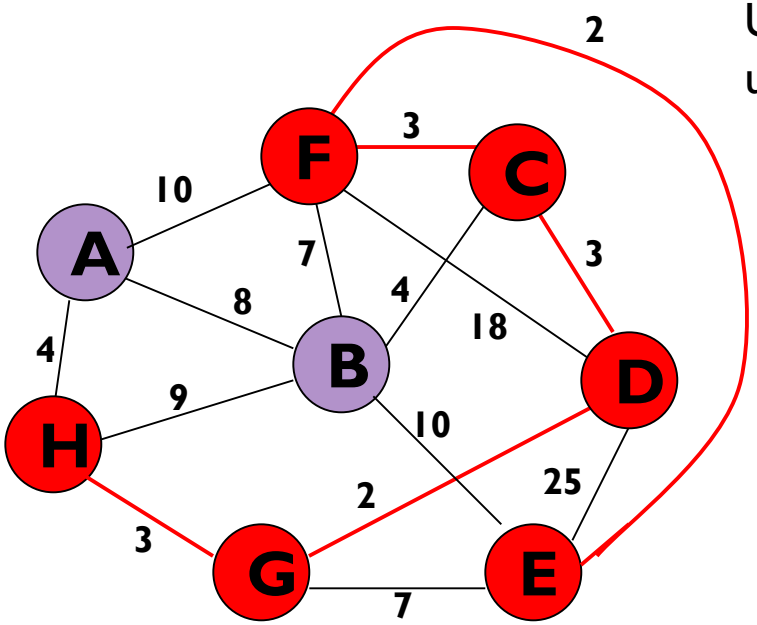
…Example:Prim's Algorithm



Select node with minimum distance

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

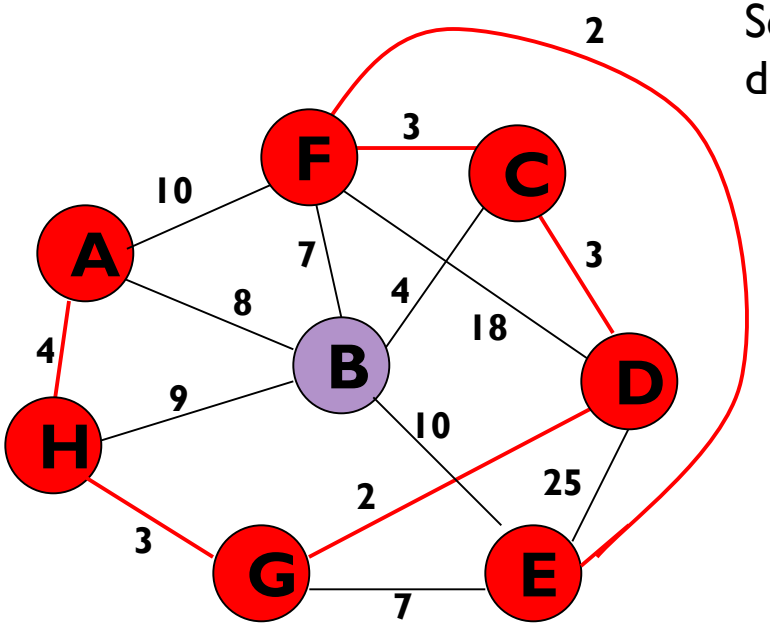
…Example:Prim's Algorithm



Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

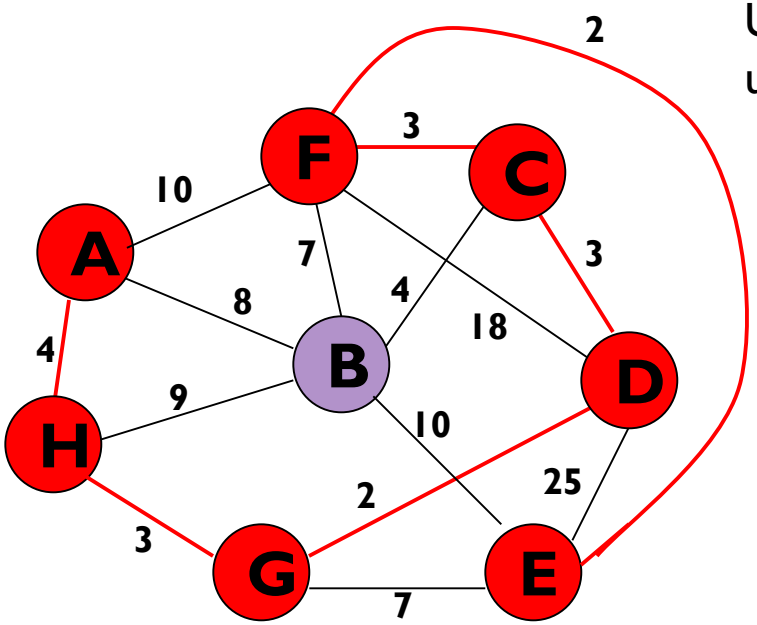
…Example:Prim's Algorithm



Select node with minimum distance

	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

…Example: Prim's Algorithm

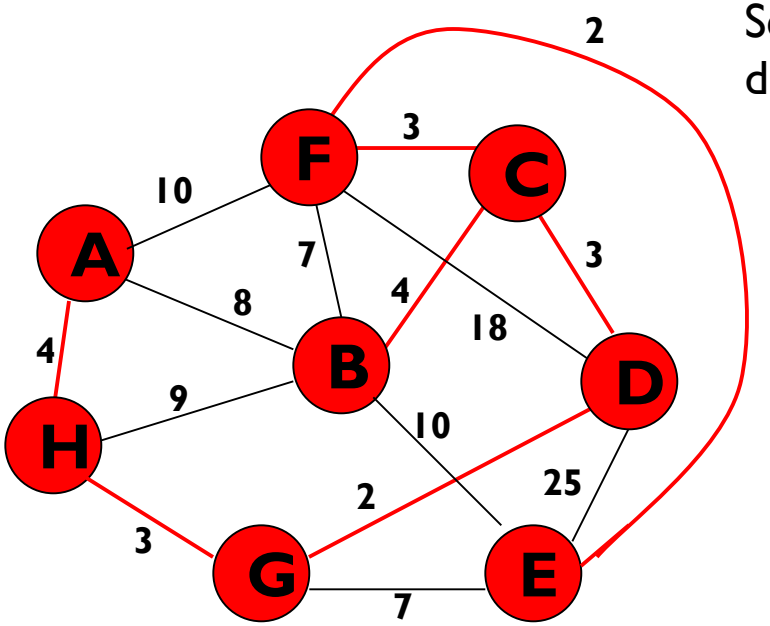


Update distances of adjacent, unselected nodes

	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Table entries unchanged

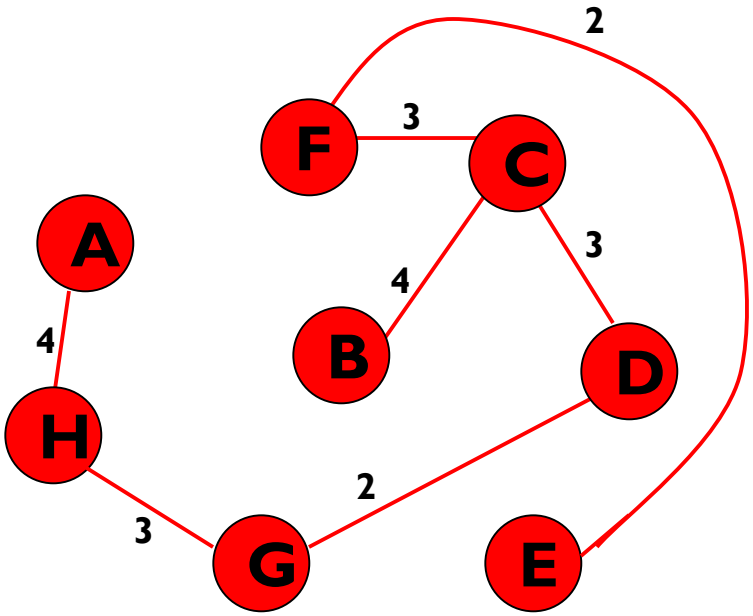
…Example:Prim's Algorithm



Select node with minimum distance

	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

…Example:Prim's Algorithm



	K	d_v	p_v	*Edge*
A	T	4	H	AH
B	T	4	C	BC
C	T	3	D	CD
D	T	0	–	
E	T	2	F	EF
F	T	3	C	FC
G	T	2	D	GD
H	T	3	G	HG

We have now all 'T' in 'K' field

Total Cost of MST = $\Sigma\, d_v = 21$

Graph Traversal

- Graph traversal is a technique used for searching a vertex in a graph.
- Graph traversal is also used to decide the order of vertices visited in the search process.
- Using graph traversal we visit all the vertices of the graph without getting into looping path.
- Two types
  - Breadth First Search (BFS)
  - Depth First Search (DFS)

BFS Vs DFS

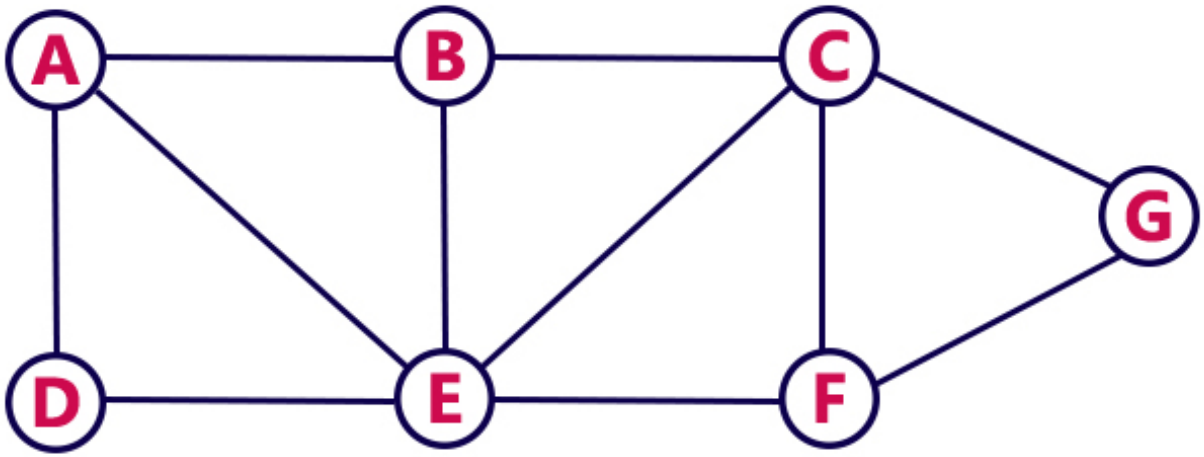
BFS	DFS
BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
BFS is better when target is closer to Source.	DFS is better when target is far from source.
As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. For one decision, we need to traverse further to augment the decision. If we reach the conclusion, we win.
Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E)

BFS

Step 1: Define a Queue of size total number of vertices in the graph.
Step 2: Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
Step 3: Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
Step 4: When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
Step 5: Repeat steps 3 and 4 until queue becomes empty.
Step 6: When queue becomes empty, then produce final spanning tree by removing unused edges from the graph.

[btechsmartclass.com]

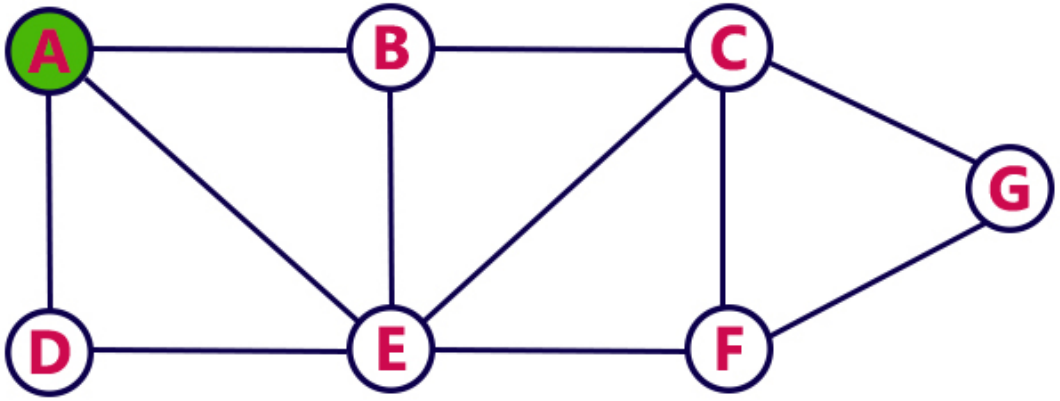
Example: BFS



…Example: BFS

Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

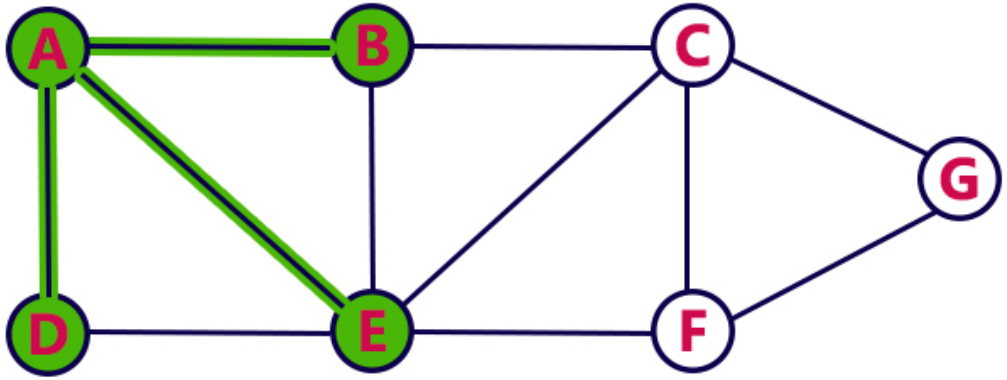


Queue

…Example: BFS

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue.



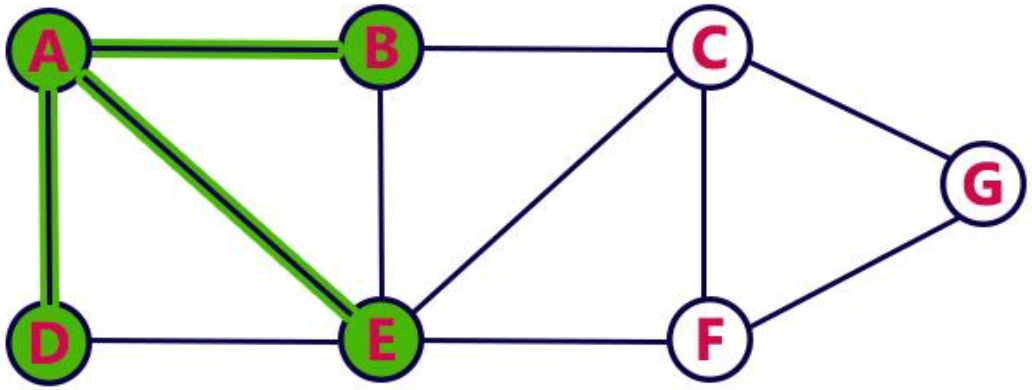
Queue

	D	E	B			

…Example: BFS

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

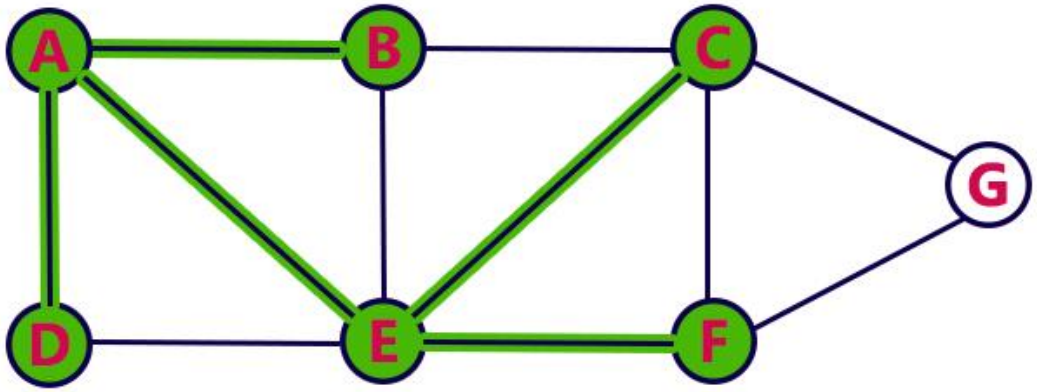


Queue

…Example: BFS

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



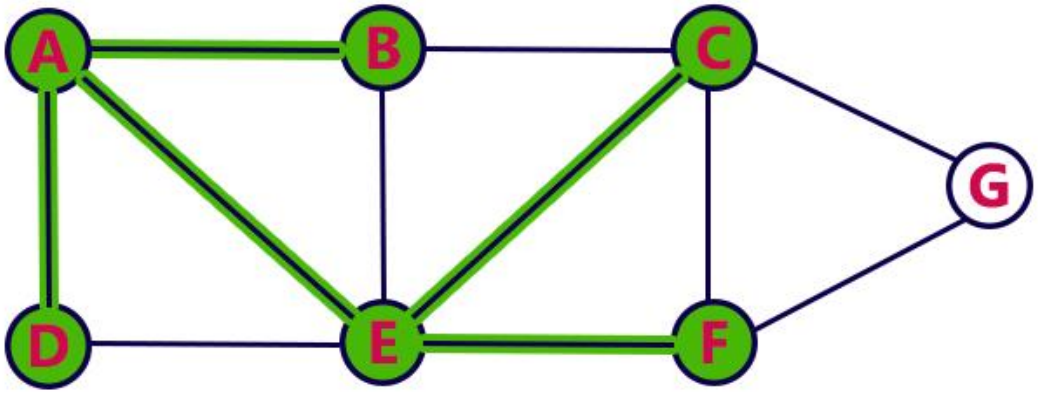
Queue

			B	C	F	

…Example: BFS

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

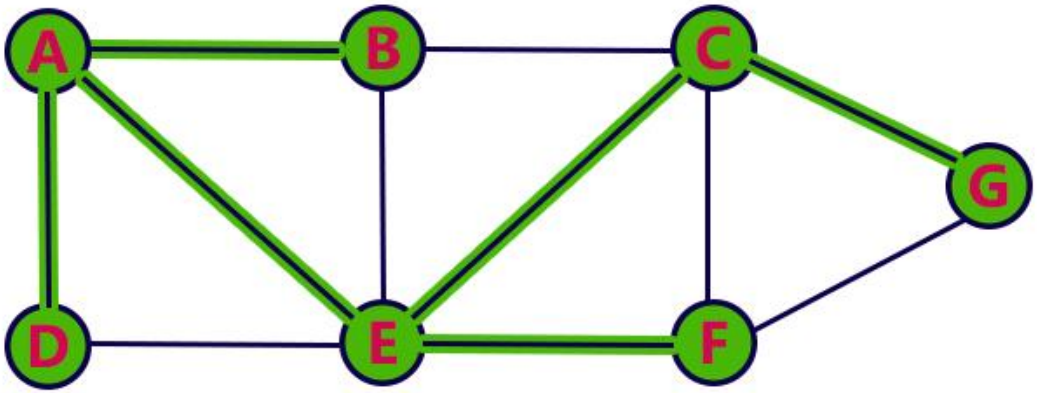


Queue

…Example: BFS

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

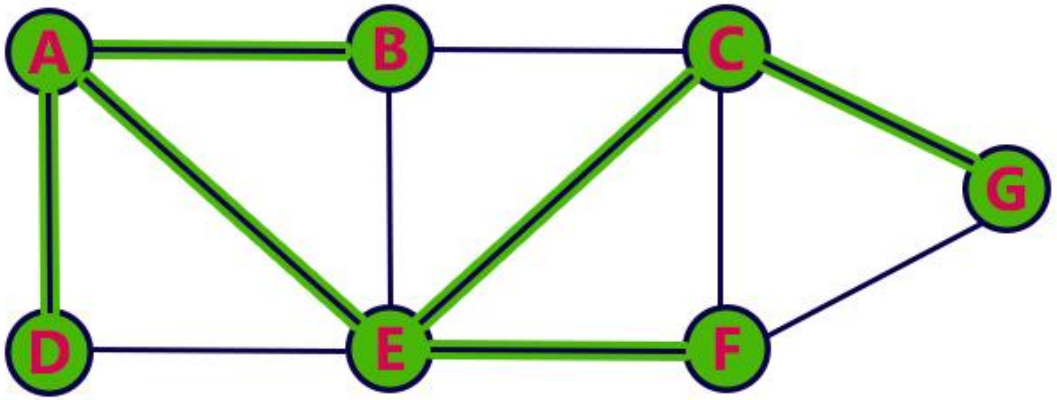


Queue

…Example: BFS

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

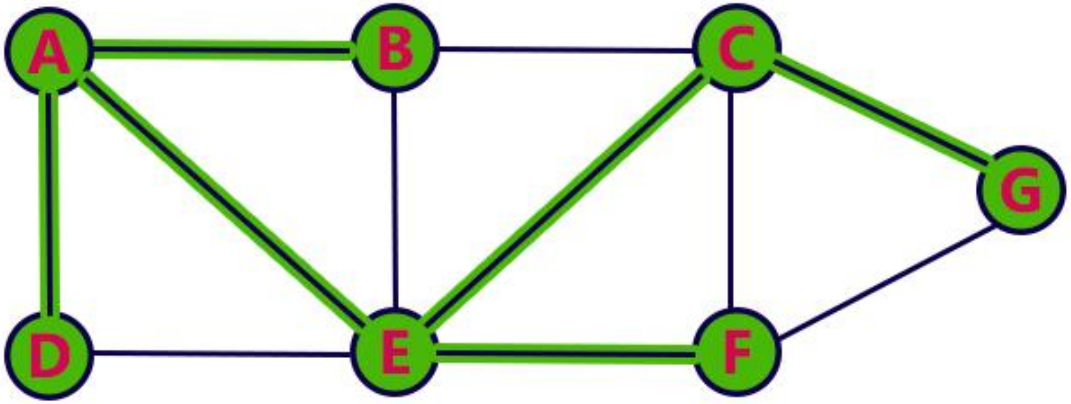


Queue

…Example: BFS

Step 8:

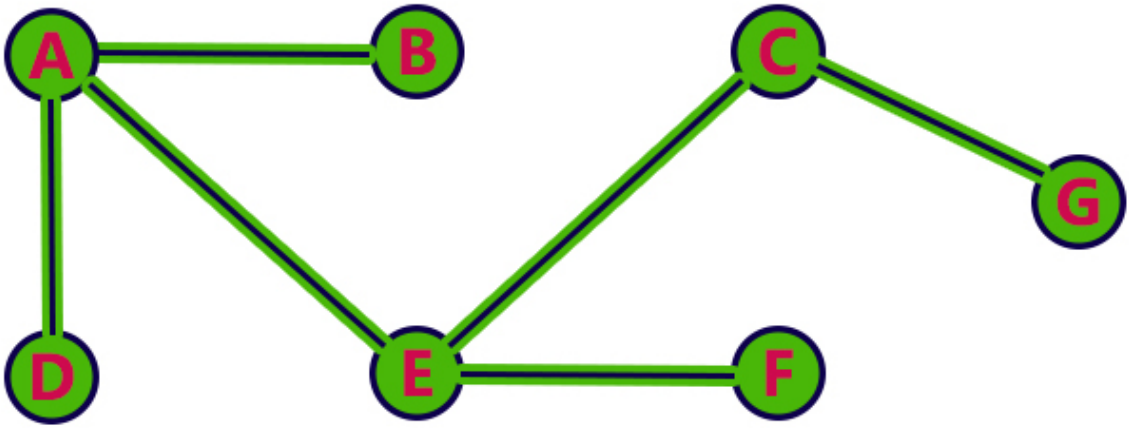
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

…Example: BFS

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...

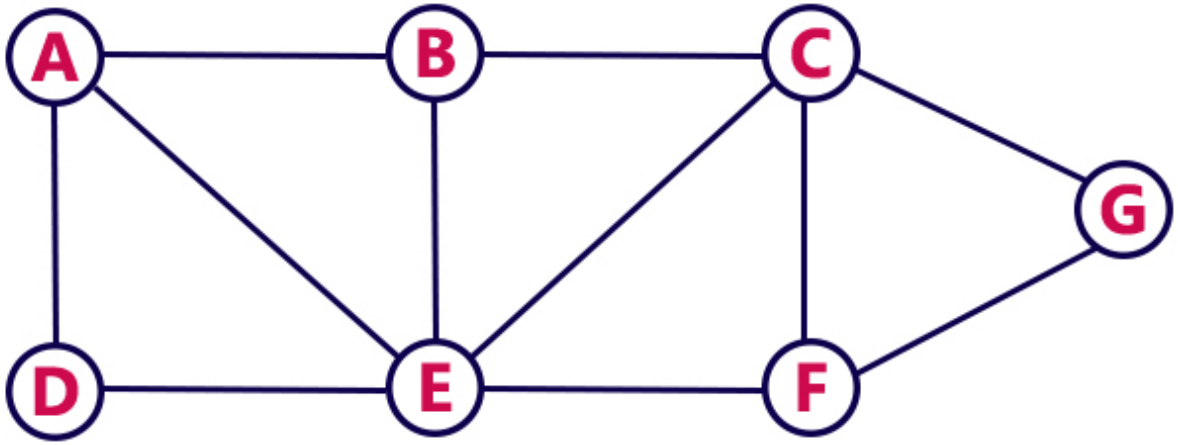


DFS

Step 1: Define a Stack of size total number of vertices in the graph.
Step 2: Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
Step 3: Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
Step 4: Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
Step 5: When there is no new vertex to visit then use **backtracking** and pop one vertex from the stack.
Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.
Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

[btechsmartclass.com]

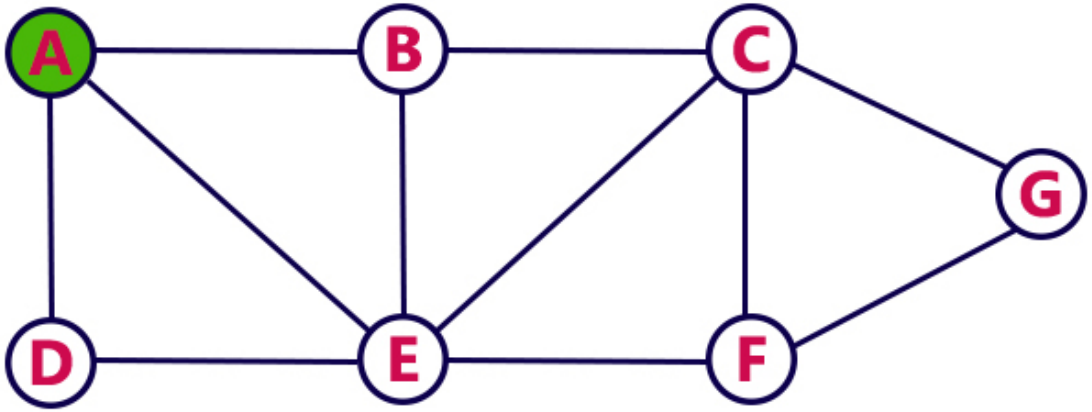
Example: DFS

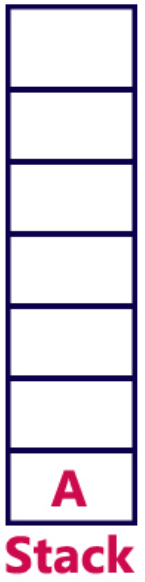


…Example: DFS

Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

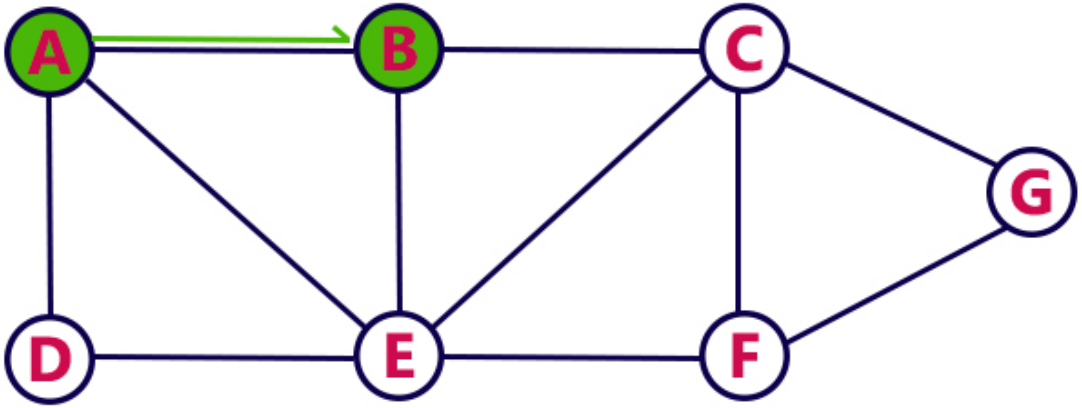


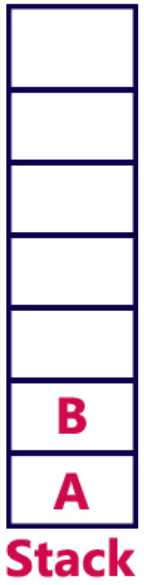


…Example: DFS

Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

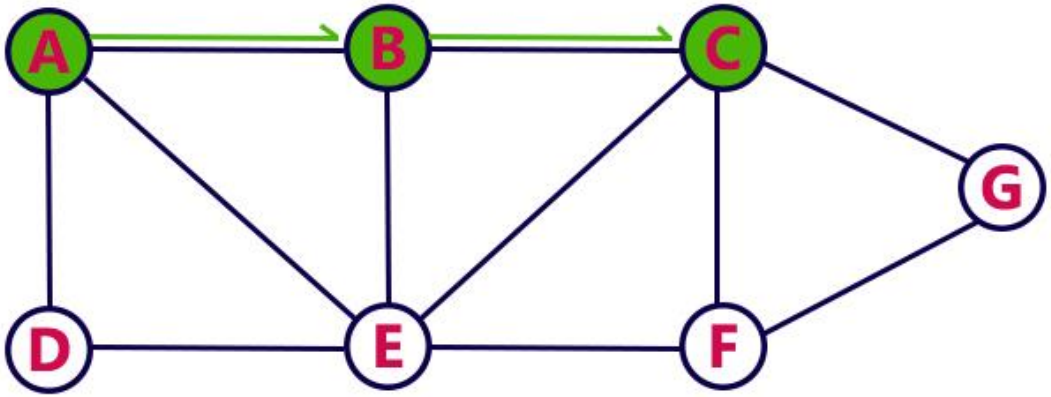


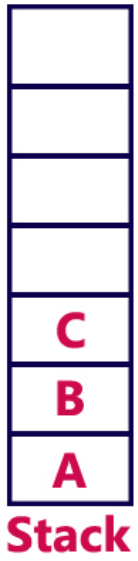


…Example: DFS

Step 3:

- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

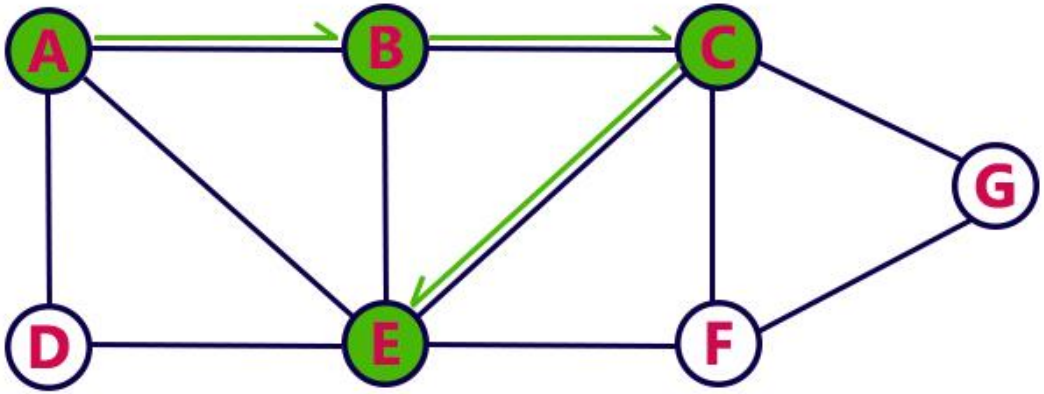




…Example: DFS

Step 4:

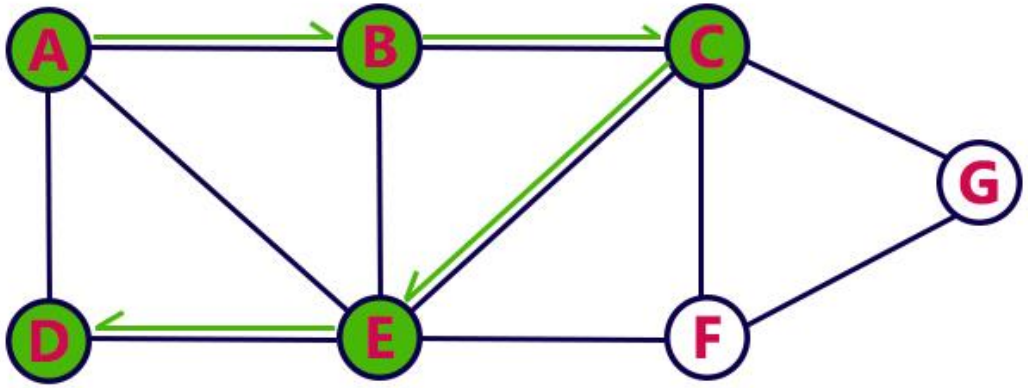
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



…Example: DFS

Step 5:

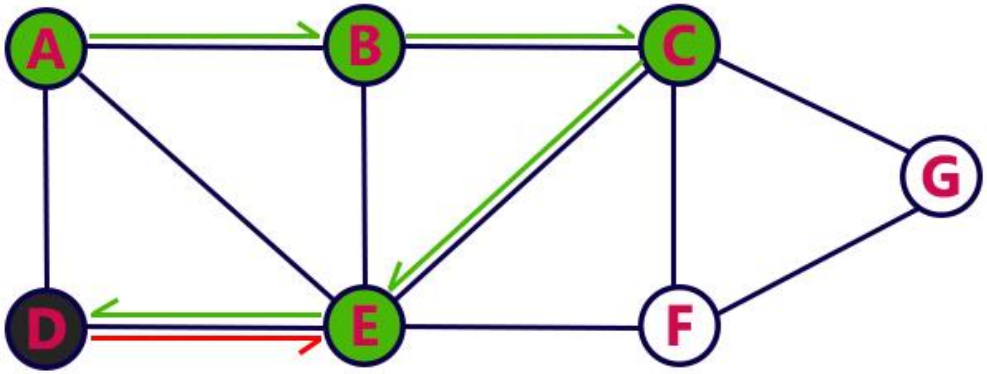
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



…Example: DFS

Step 6:

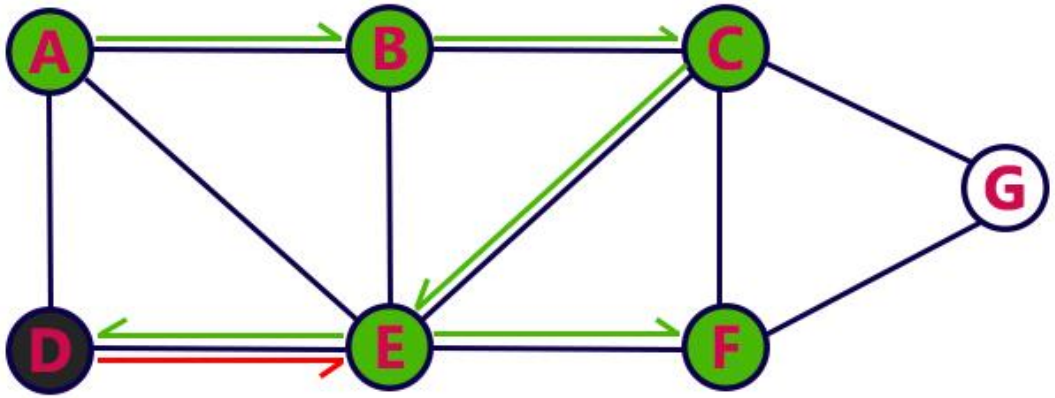
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

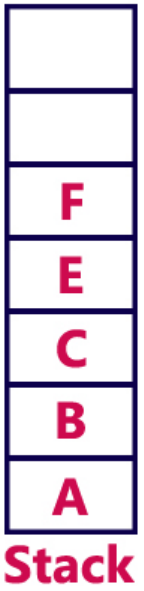


…Example: DFS

Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

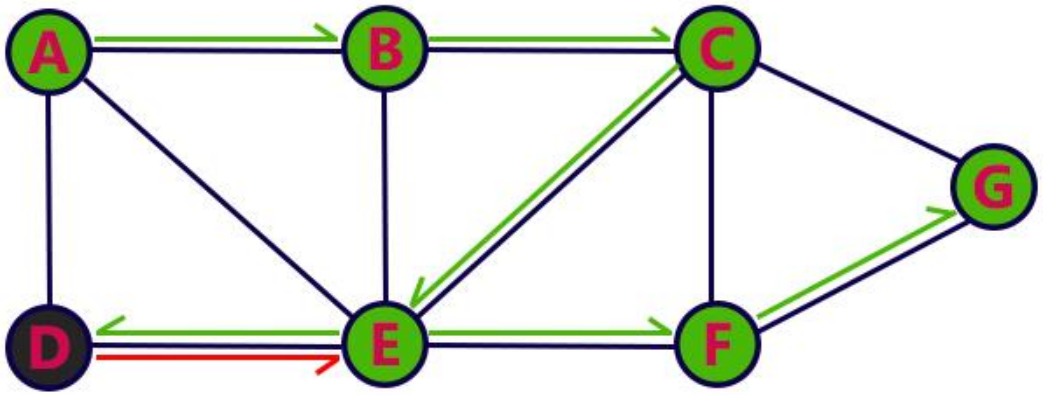




…Example: DFS

Step 8:

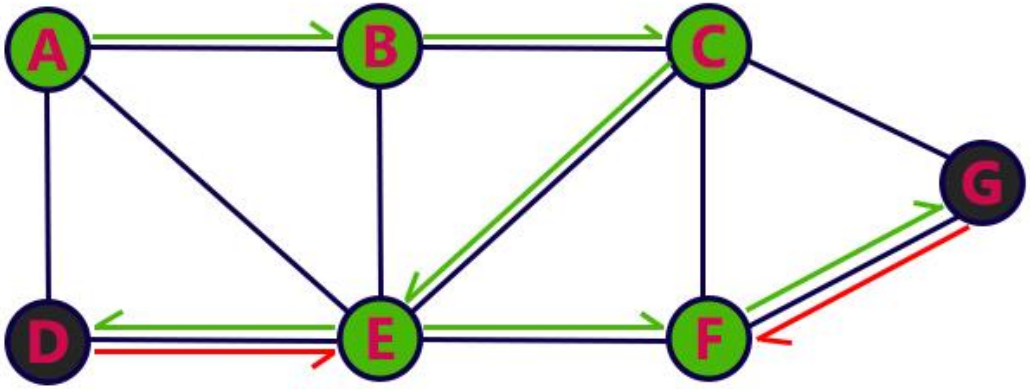
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

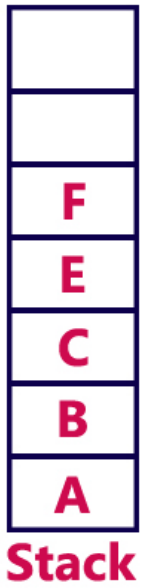


…Example: DFS

Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

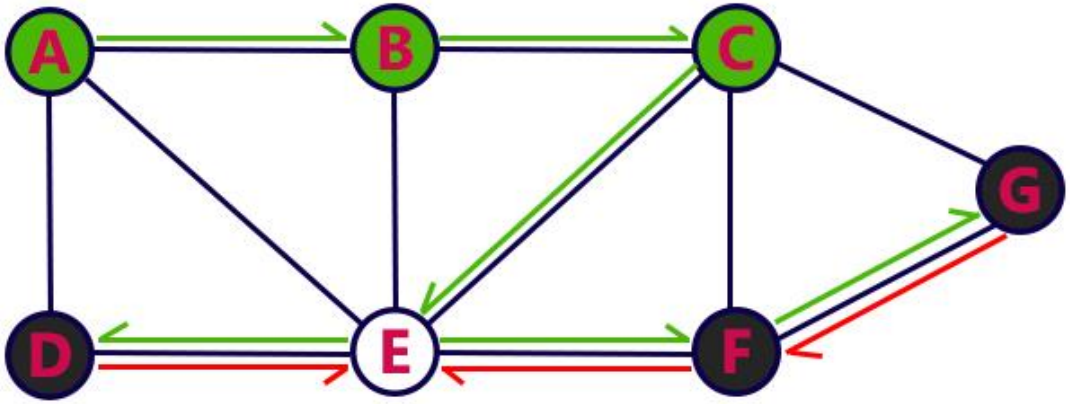


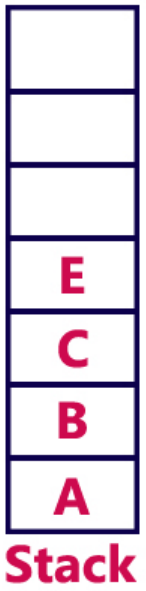


…Example: DFS

Step 10:

- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

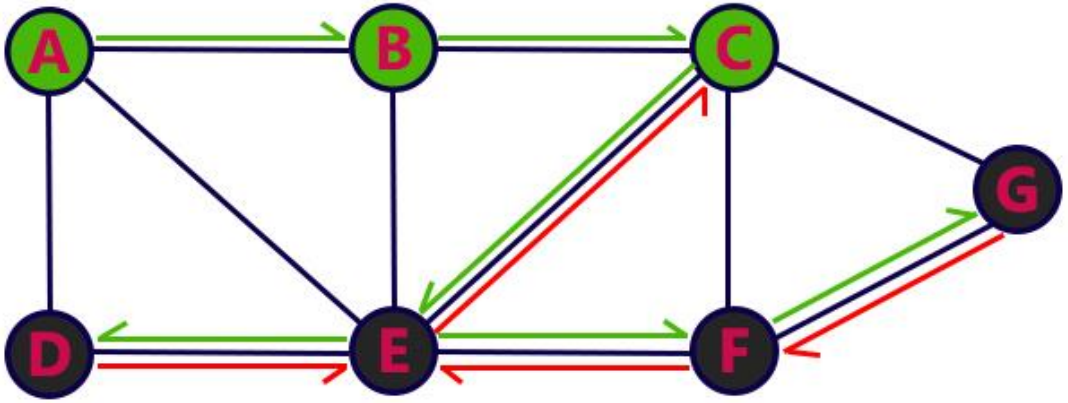


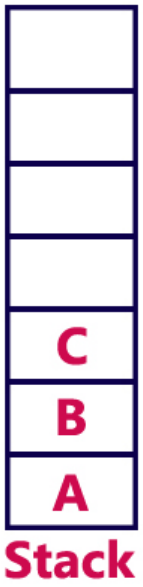


…Example: DFS

Step 11:

- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

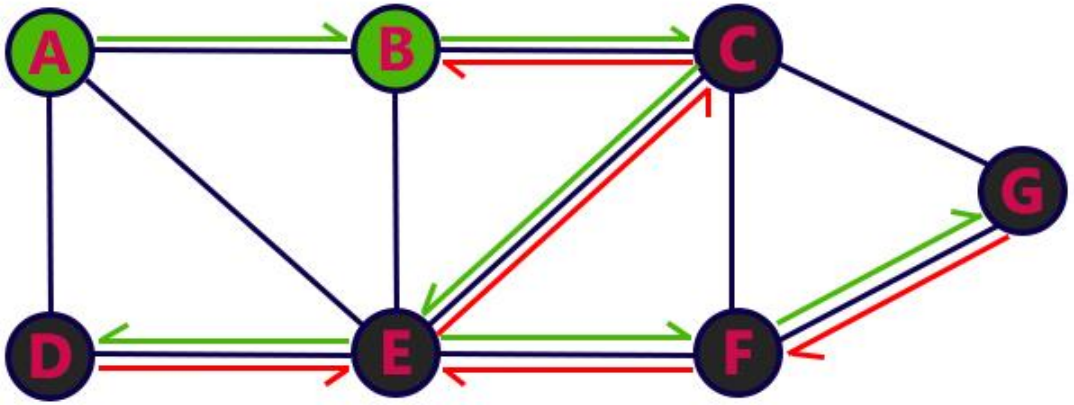




…Example: DFS

Step 12:

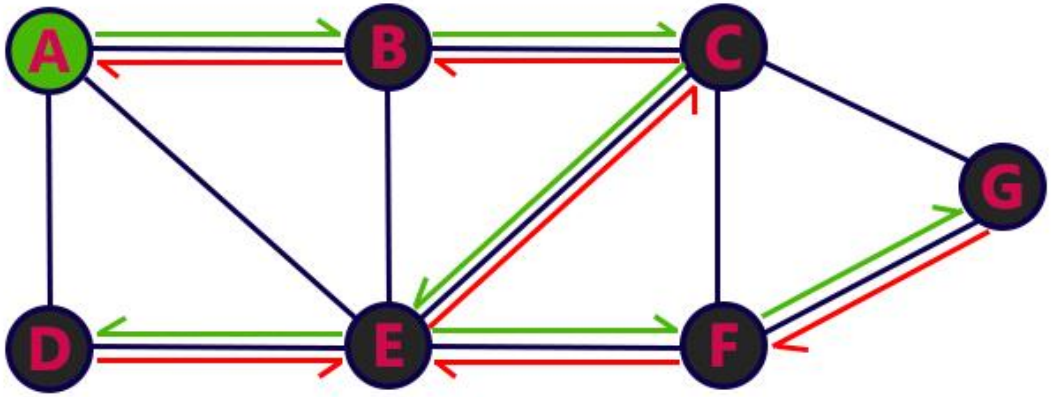
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

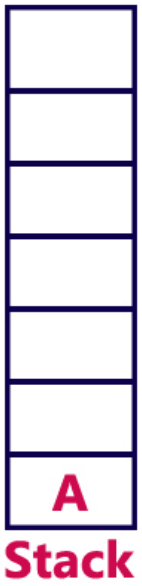


…Example: DFS

Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

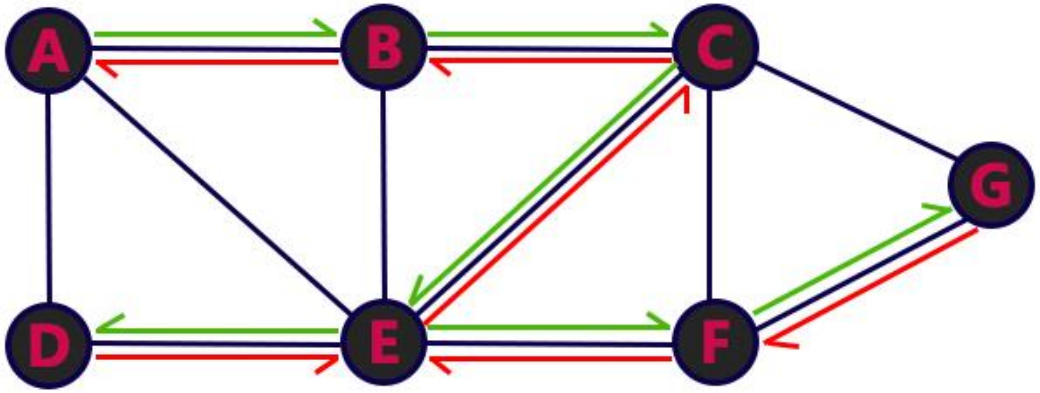


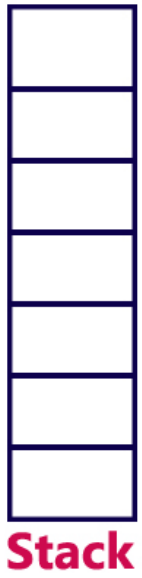


…Example: DFS

Step 14:

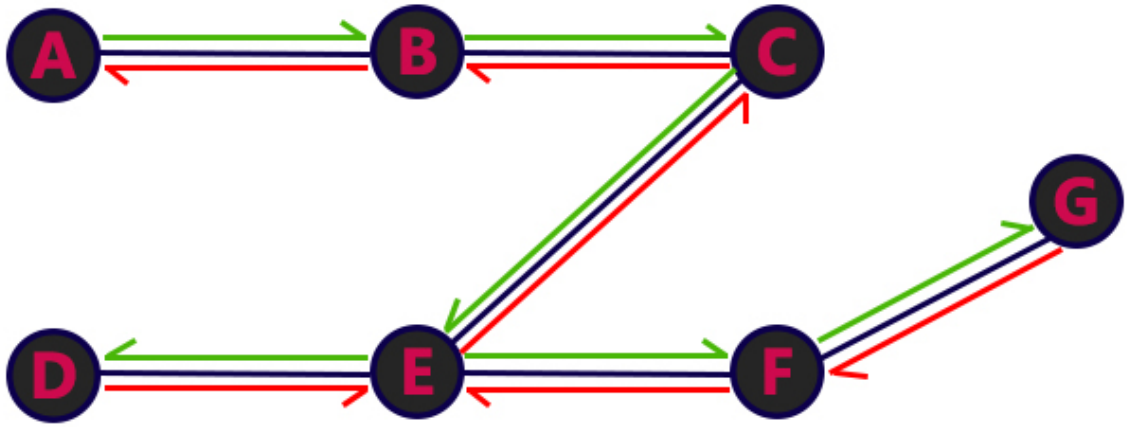
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





…Example: DFS

- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Applications: BFS and DFS

DFS

- We can detect cycles in a graph using DFS.
- We can find path between two given vertices u and v.
- We can find strongly connected components of a graph. If there is a path from each vertex to every other vertex, that is a (strongly) connected graph.

BFS

- In peer-to-peer network like bit-torrent, BFS is used to find all neighbor nodes.
- In GPS navigation system, BFS is used to find neighboring places.
- In networking, when we want to broadcast some packets, we use the BFS algorithm.

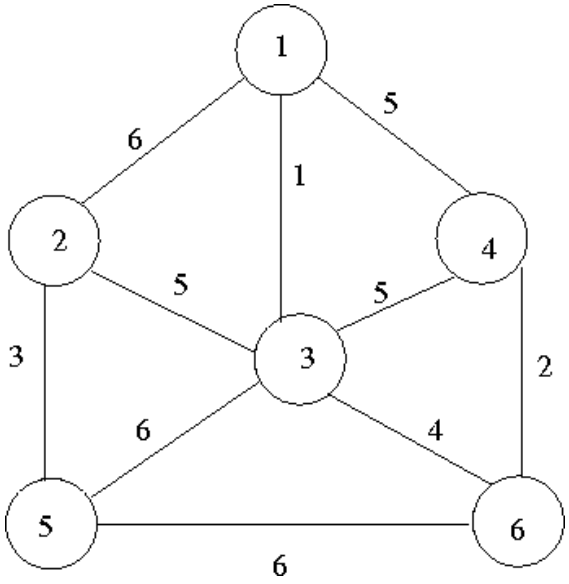
Path finding algorithm is based on BFS or DFS.

Summary

- Directed Vs Undirected Graphs
- Weighted Graphs
- Spanning Tree
- Kruskal's and Prim's Greedy algorithms for MST
- BFS and DFS

Assignments

- Find MST with (1) Kruskal's algorithm (2) Prim's algorithm



- BFS and DFS of above graph (select vertex1 as the starting point)