CSE-281: Data Structures and Algorithms

Graphs (Chapter-8)

Ref: Schaum's Outline Series, Theory and problems of Data Structures
By Seymour Lipschutz

And Online Resource
[
http://www.btechsmartclass.com/data_structures/graph-traversal-bfs.html]

Eftekhar Hossain
Assistant Professor
Dept. of ETE, CUET

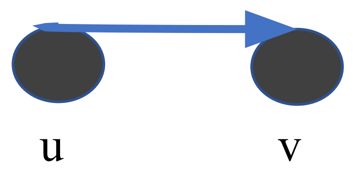
Topics to be Covered

- Graphs
- Types of Graphs
- Graph Representation
- Breadth First Search (BFS)
- Depth First Search
- Minimum Spanning Tree
- Shortest Path Algorithm

Introduction to Graphs

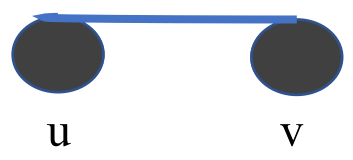
- A graph G is an ordered pair of a set V of vertices and a set E of edges.

$$G = (V, E)$$



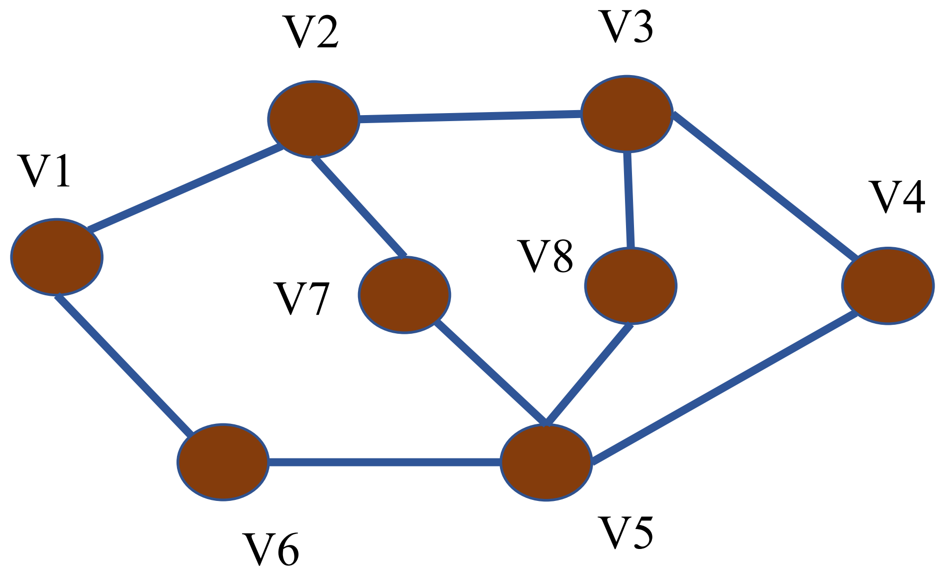
Directed

(u,v)

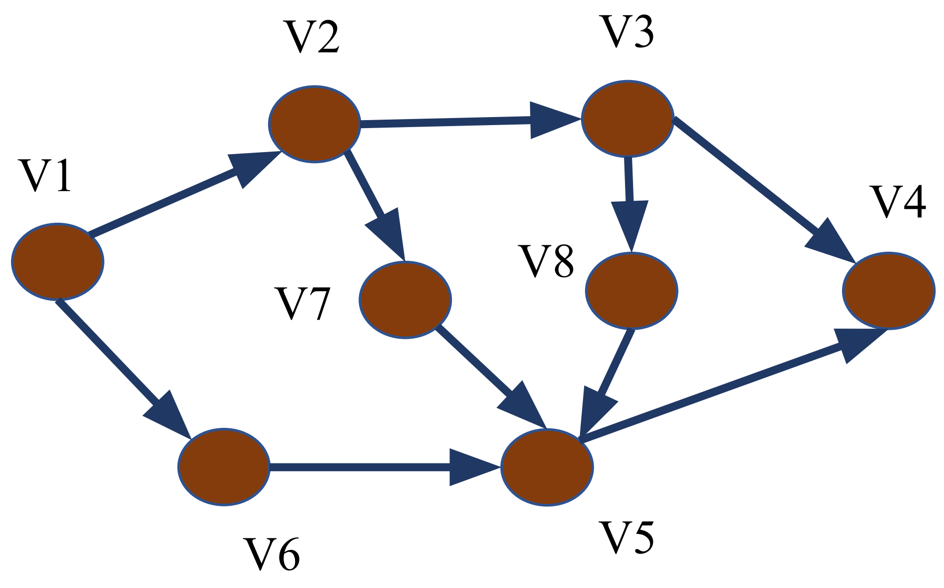


Undirected

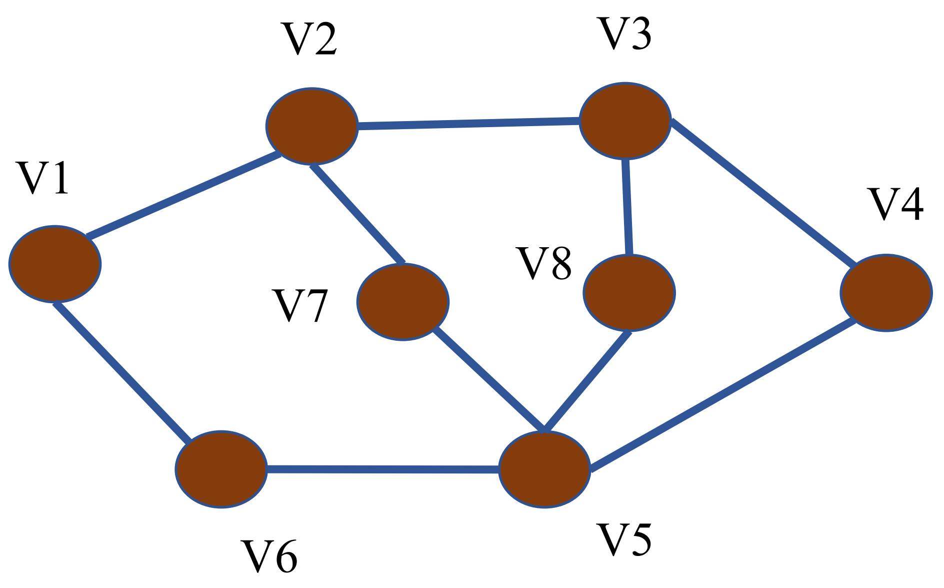
{u,v}



Directed vs Undirected

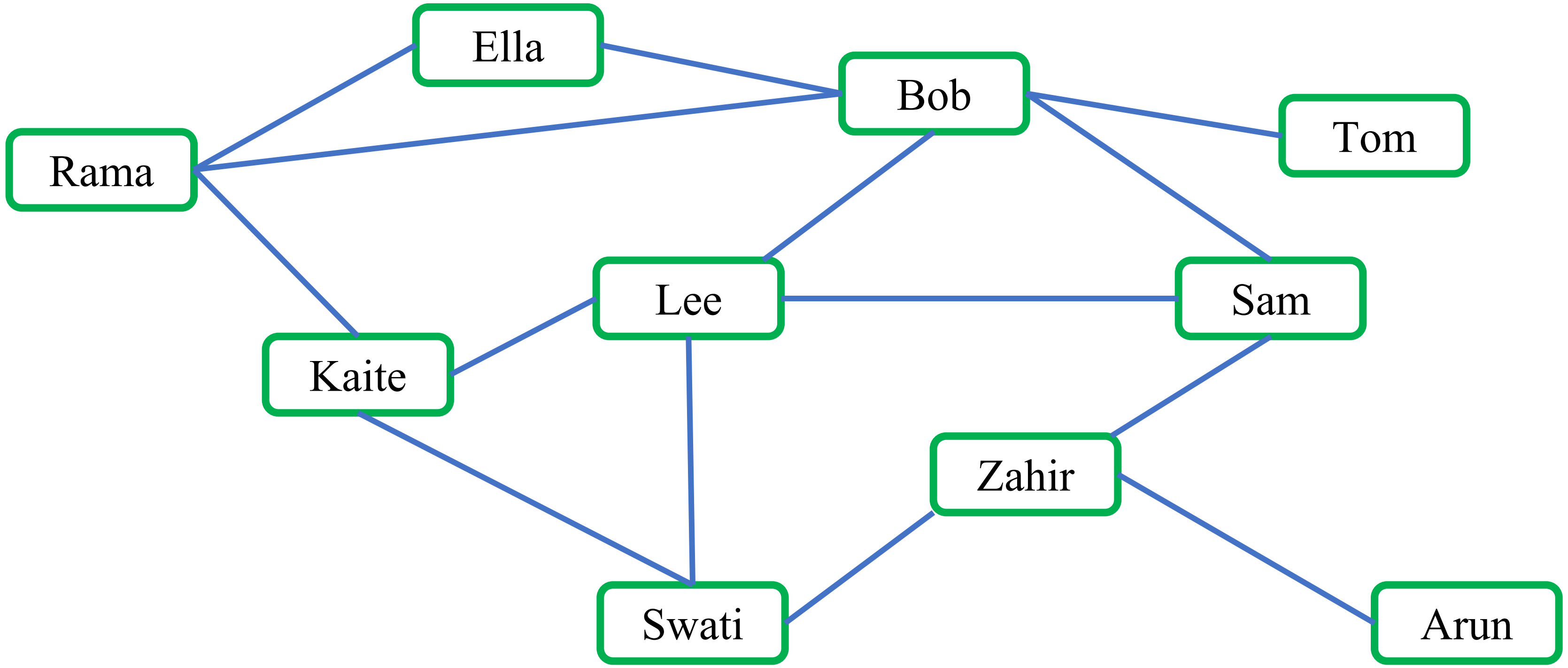


Directed



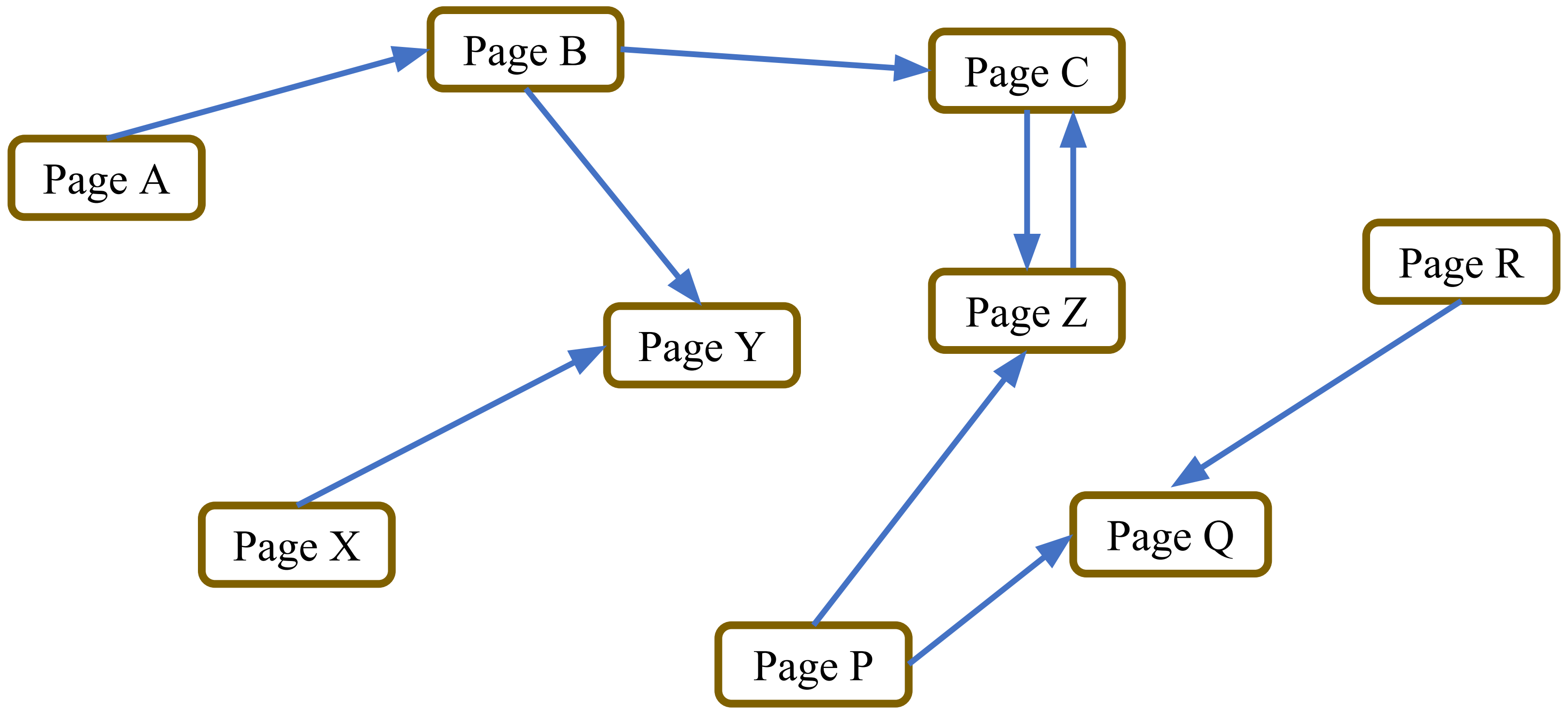
Undirected

Applications



Social Network (facebook)

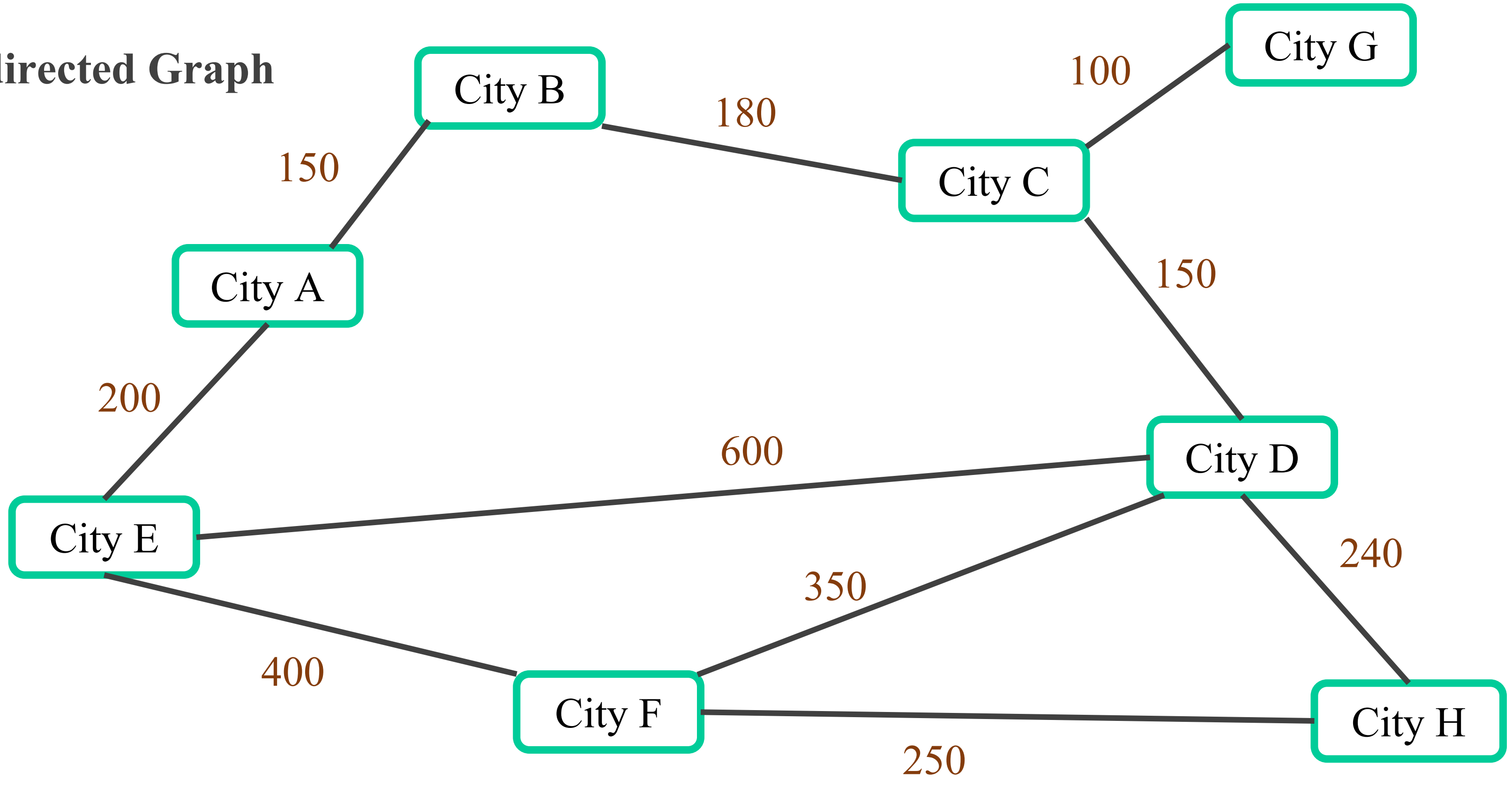
Applications



World Wide Web

Applications

Weighted Undirected Graph

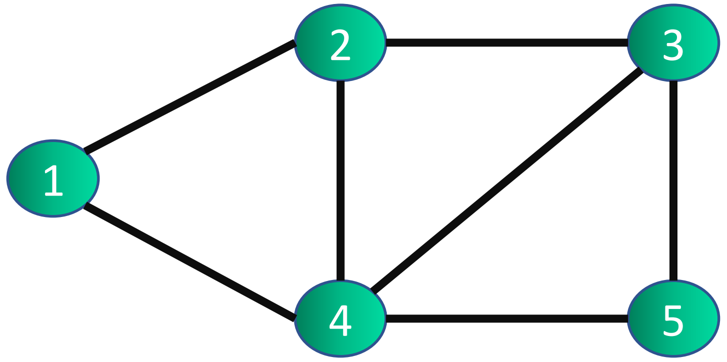


Inter City Road Network

Graph Representation

- Adjacency Matrix

n = number of vertices



It is a matrix ***A[n][n]*** where ***n*** is no. of vertices

$$\begin{cases} A[i][j] = 1, & \textit{if i and j are adjacent} \\ \quad\quad\quad\;\; 0, & \textit{Otherwise} \end{cases}$$

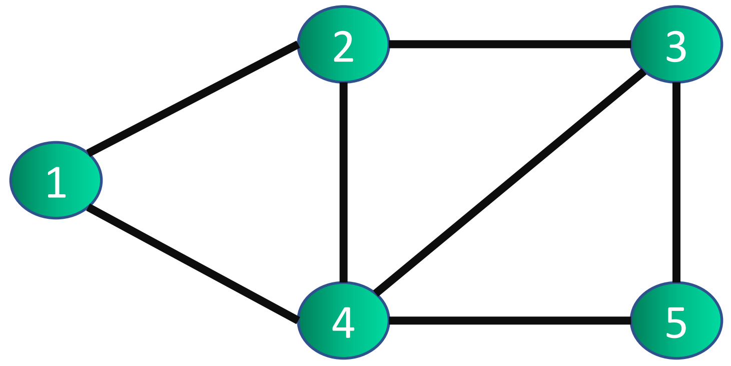
$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 & 0 \\ 3 & 0 & 1 & 0 & 1 & 1 \\ 4 & 1 & 1 & 1 & 0 & 1 \\ 5 & 0 & 0 & 1 & 1 & 0 \end{array}_{5 \times 5}$$

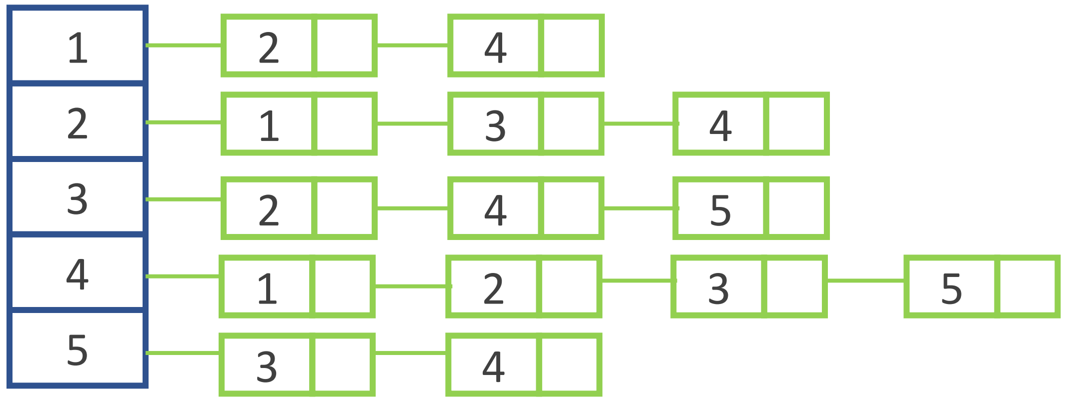
# Advantage and Disadvantage (Self)

Graph Representation

 Adjacency List

n = number of vertices





# Advantage and Disadvantage (Self)

Graph Traversal

- Graph traversal is a technique used for searching a vertex in a graph.
- The graph traversal is also used to decide the order of vertices is visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops.
- That means using graph traversal we visit all the vertices of the graph without getting into looping path.

- There are two graph traversal techniques, and they are as follows...
    - BFS (Breadth First Search)
    - DFS (Depth First Search)

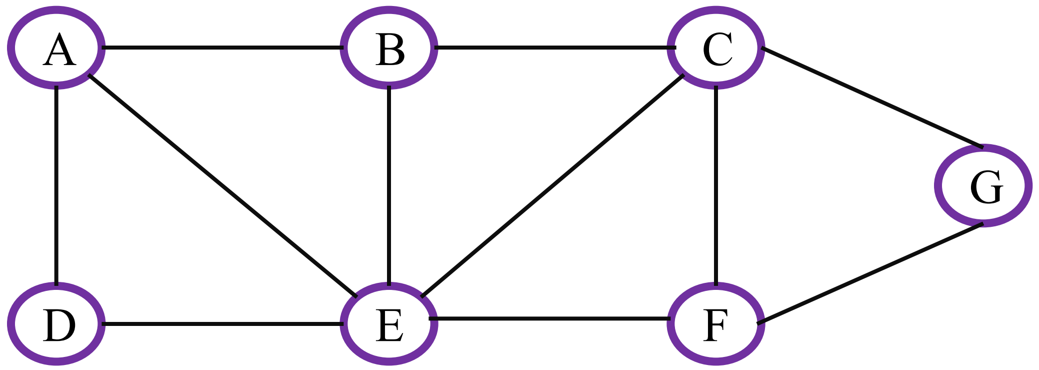
BFS

- **BFS** traversal of a graph produces a **spanning tree** as final result.
- **Spanning Tree** is a graph without loops.
- We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal.

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

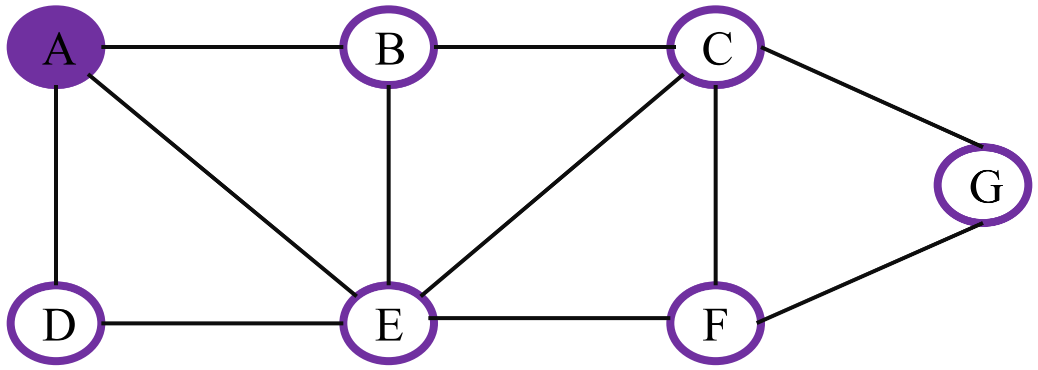
BFS (Example)



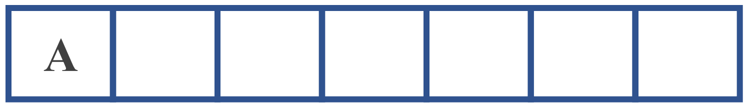
Step 1

✔ Select the vertex **A** as a Starting Point (visit **A**)

✔ Insert **A** into the queue



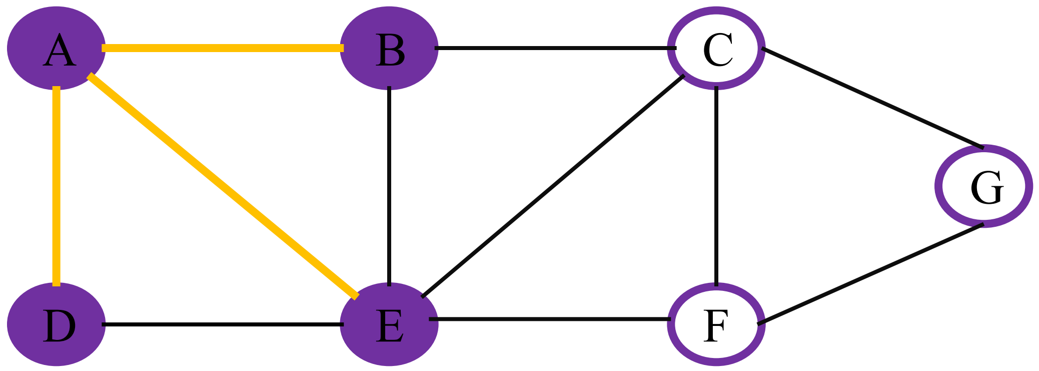
Queue



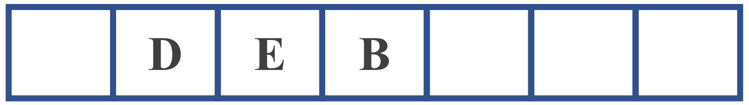
BFS (Example)

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.



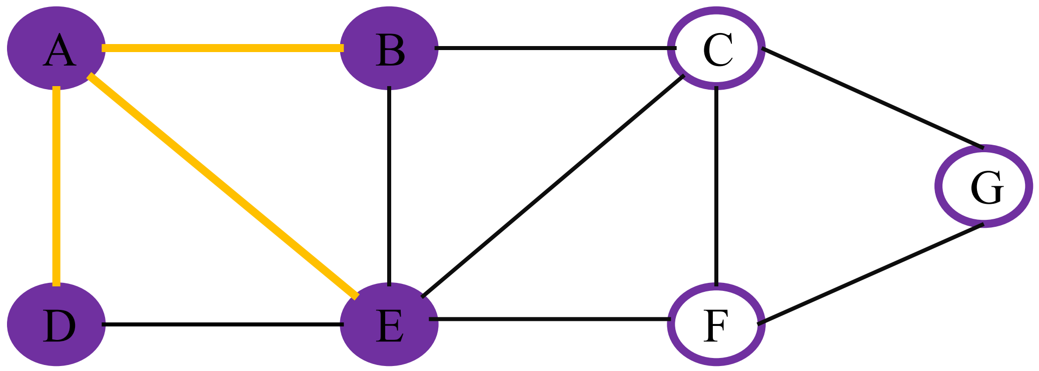
Queue



BFS (Example)

Step 3

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



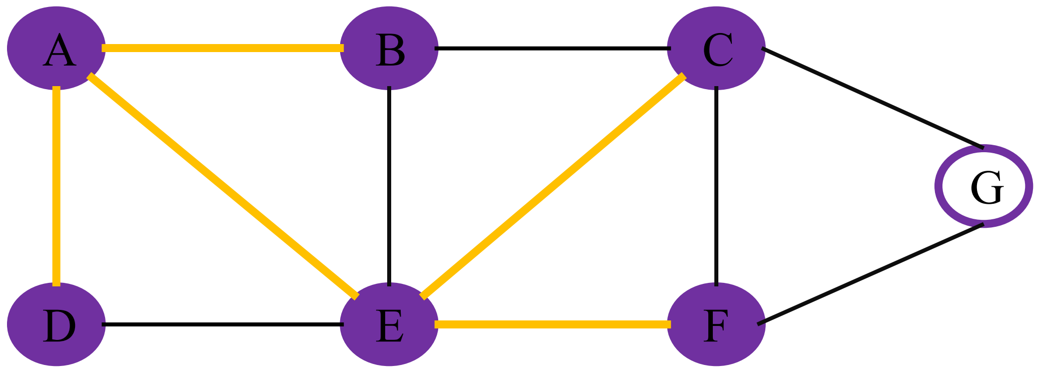
Queue

		E	**B**			

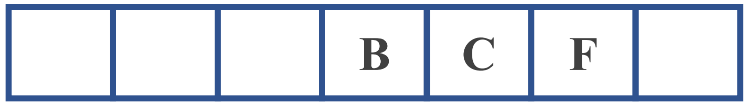
BFS (Example)

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.



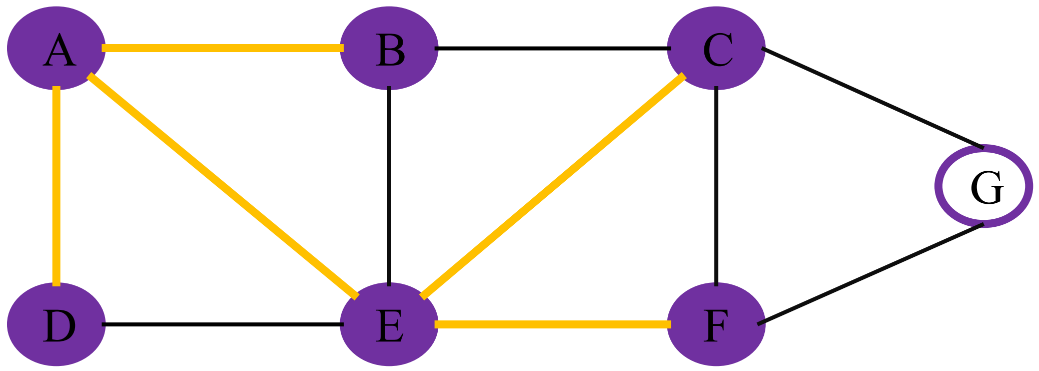
Queue



BFS (Example)

Step 5

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



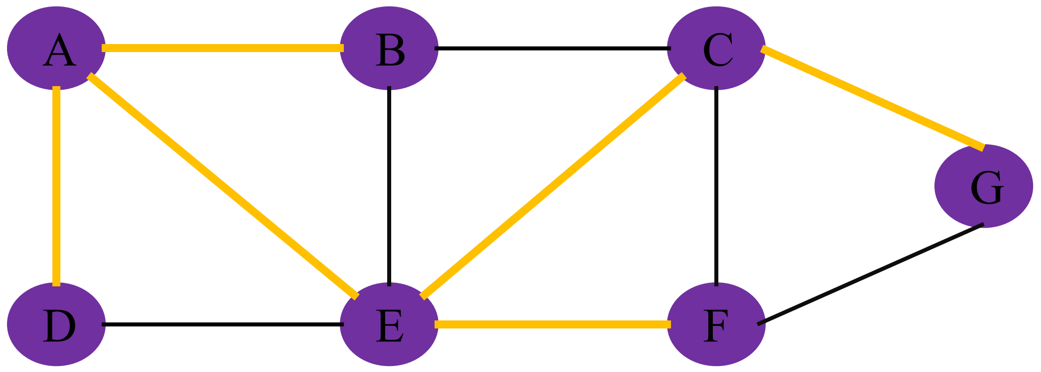
Queue

				C	F	

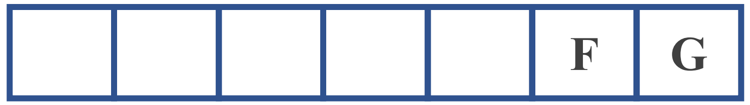
BFS (Example)

Step 6

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



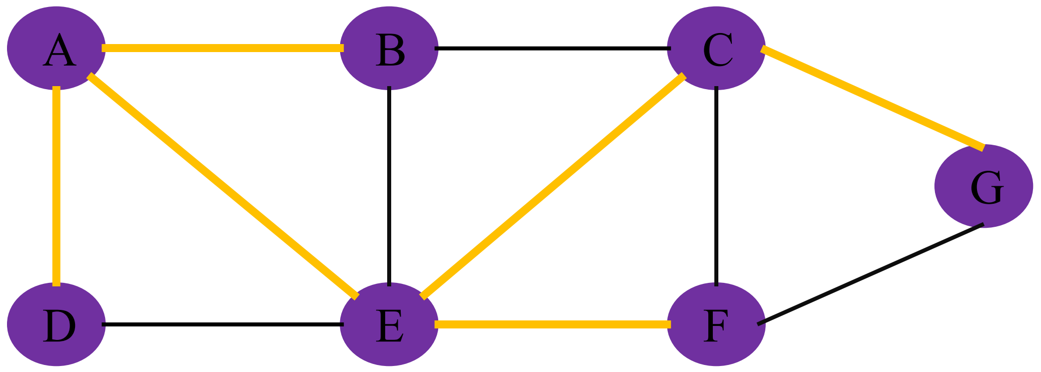
Queue



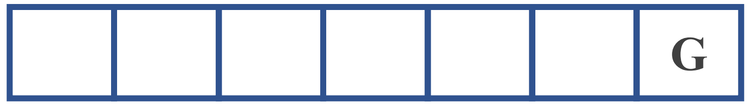
BFS (Example)

Step 7

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



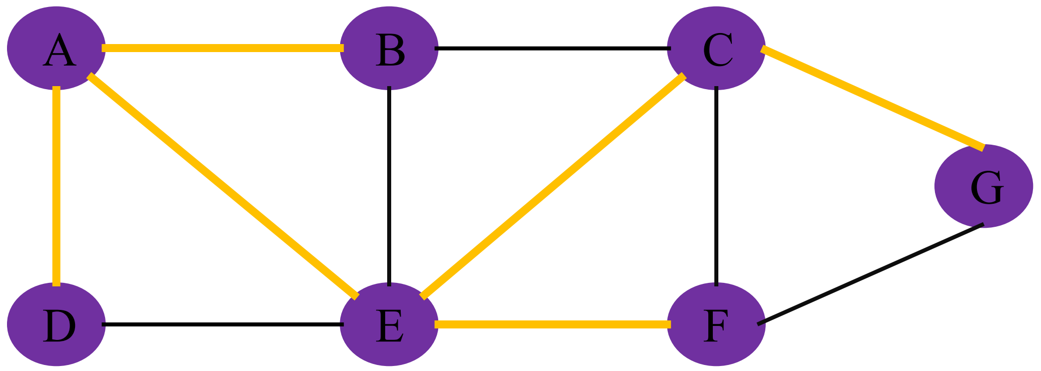
Queue



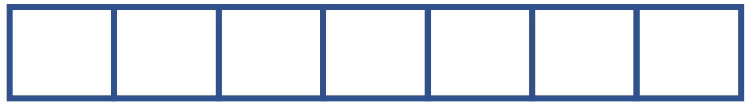
BFS (Example)

Step 8

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

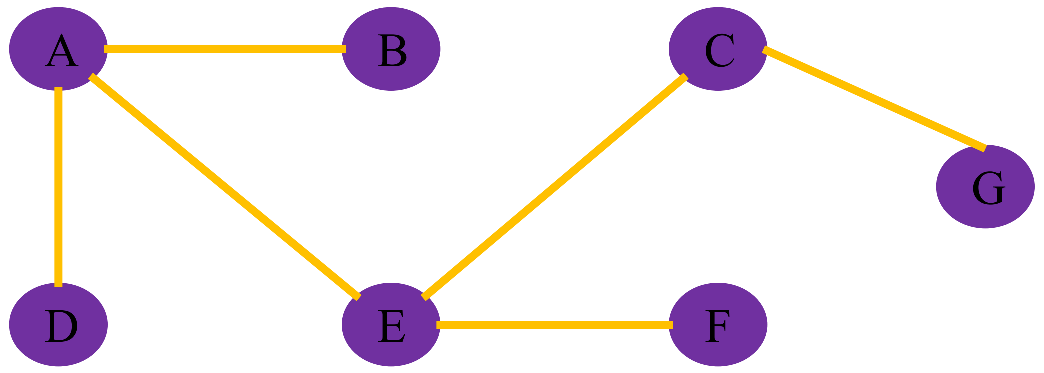


Queue



BFS (Example)

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



DFS

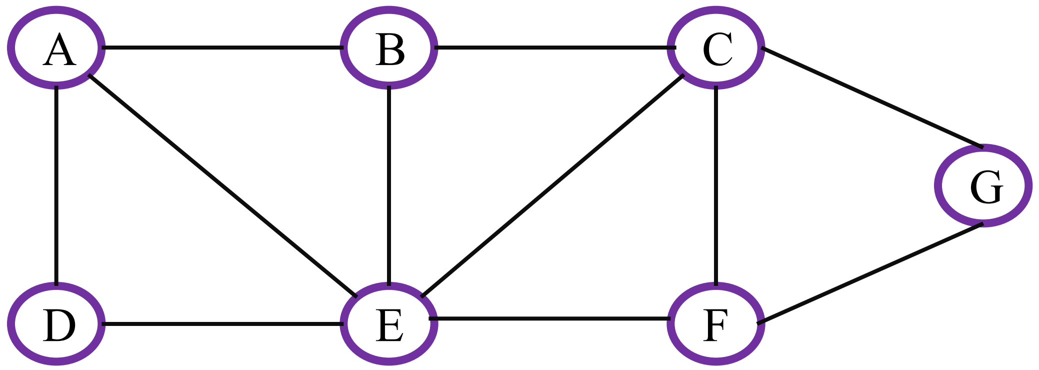
 We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal.

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 **back tracking** 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

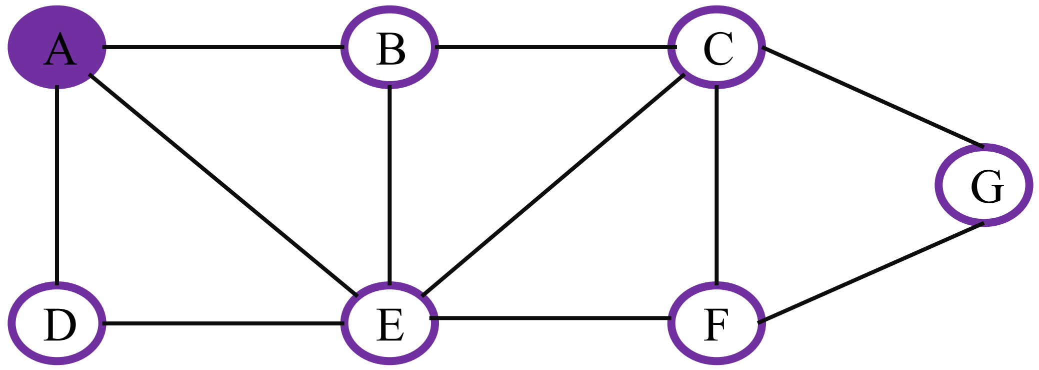
Back tracking is coming back to the vertex from which we reached the current vertex.

DFS (Example)

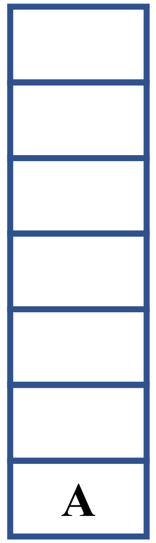


Step 1

- ✔ Select the vertex **A** as a Starting Point (visit **A**)
- ✔ Push **A** into the stack



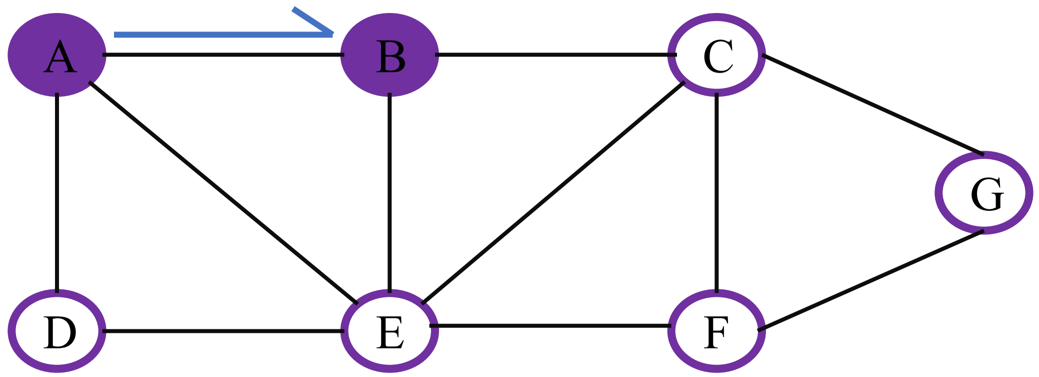
Stack



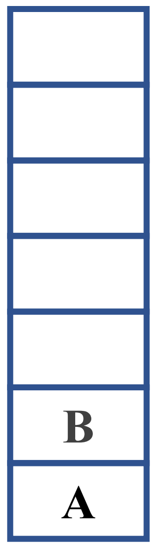
DFS (Example)

Step 2

- ✔ Visit any adjacent vertex of **A** which is not visited (**B**)
- ✔ Push **B** into the stack



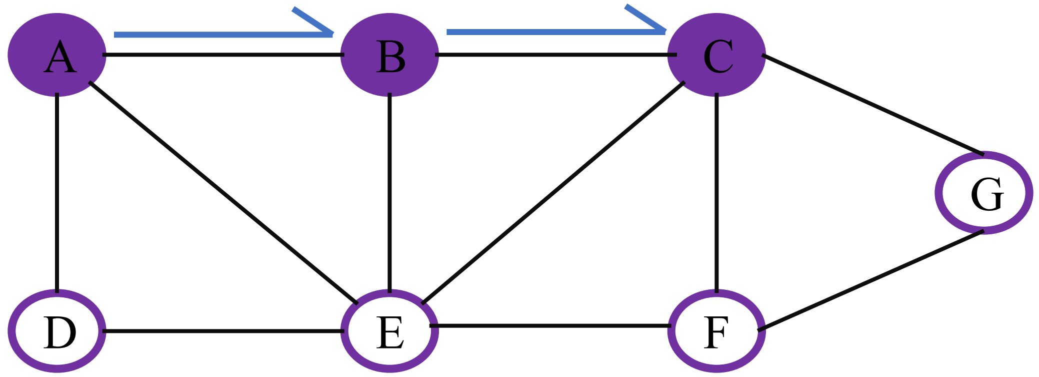
Stack



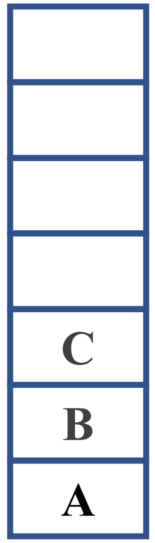
DFS (Example)

Step 3

- ✔ Visit any adjacent vertex of **B** which is not visited (**C**)
- ✔ Push **C** into the stack



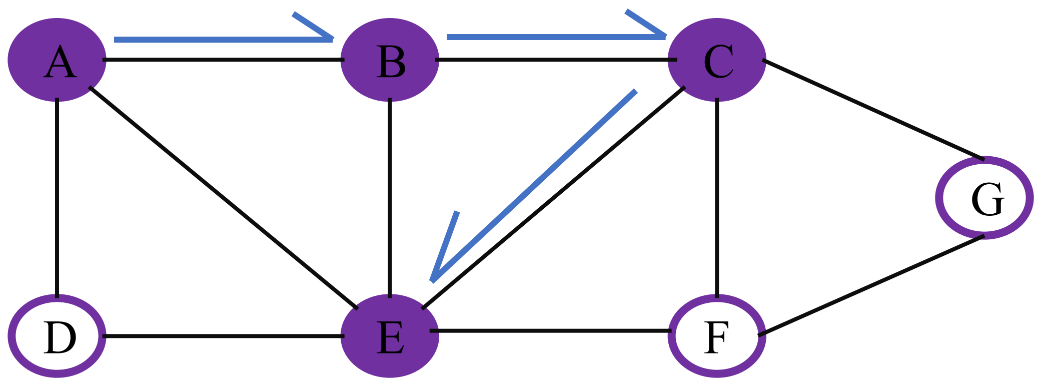
Stack

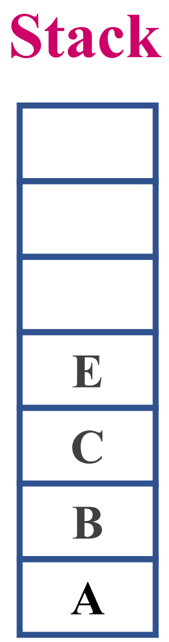


DFS (Example)

Step 4

- ✔ Visit any adjacent vertex of **C** which is not visited (**E**)
- ✔ Push **E** into the stack

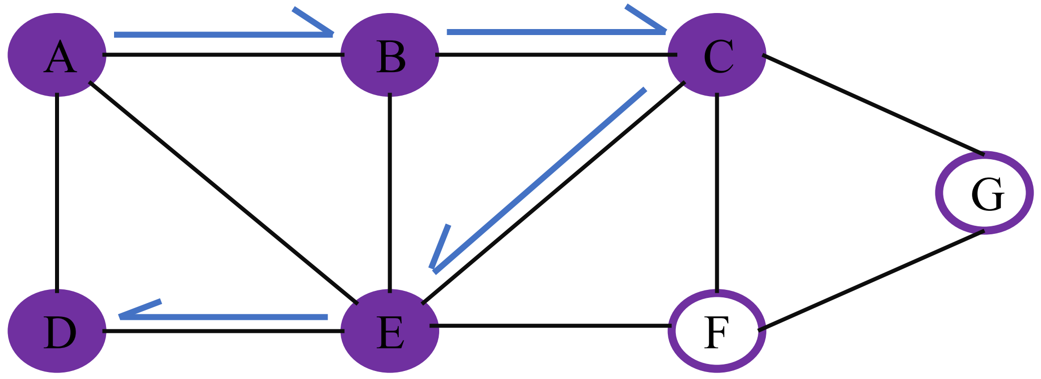


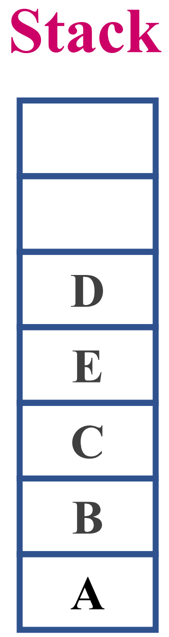


DFS (Example)

Step 5

- ✔ Visit any adjacent vertex of **E** which is not visited (**D**)
- ✔ Push **D** into the stack

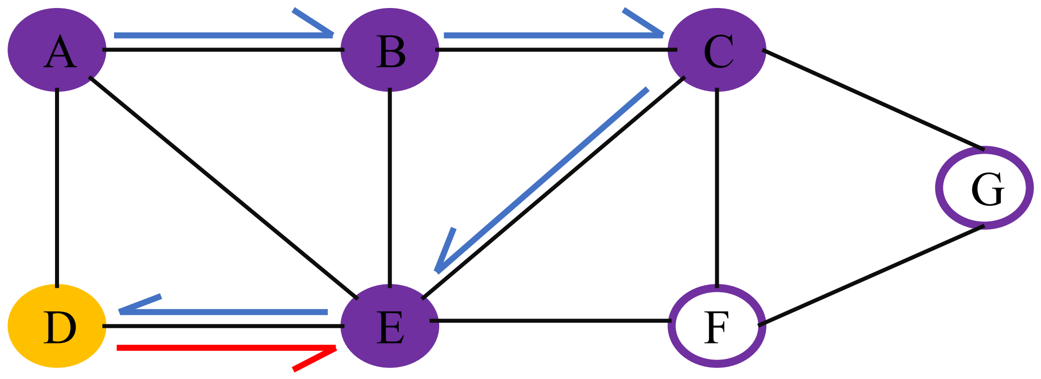


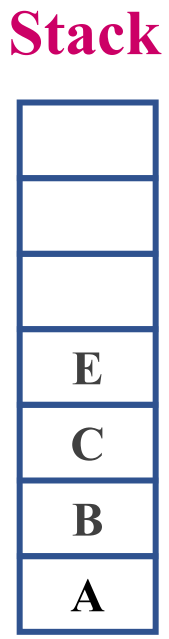


DFS (Example)

Step 6

- ✔ There is no new vertex to be visited from **D.** so use backtrack
- ✔ Pop **D** from the stack

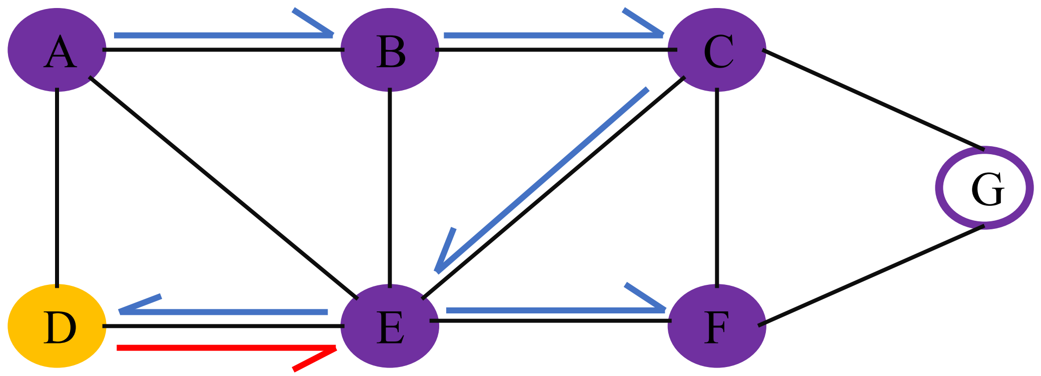


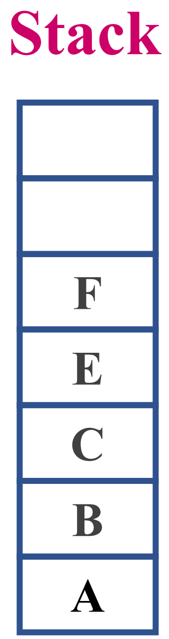


DFS (Example)

Step 7

- ✔ Visit any adjacent vertex of E which is not visited (F)
- ✔ Push **F** into the stack

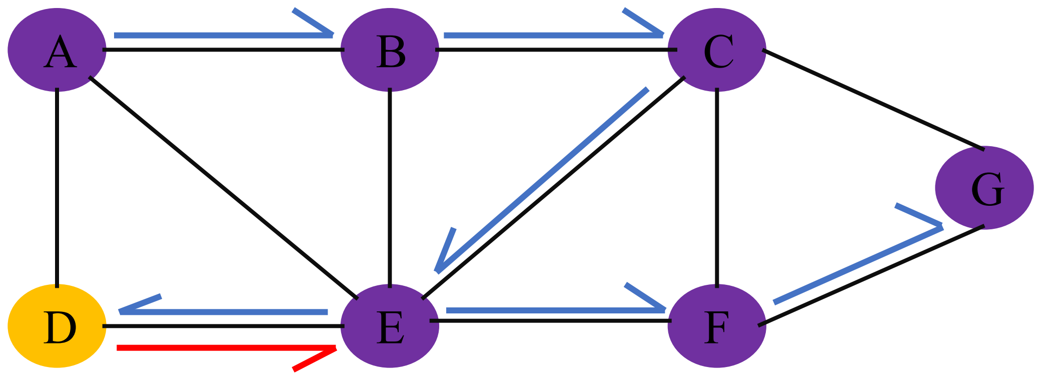


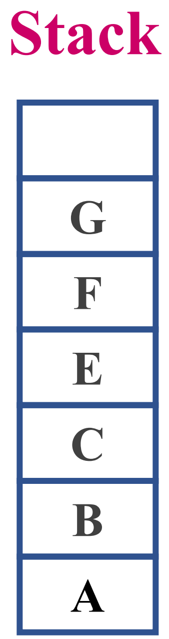


DFS (Example)

Step 8

- ✔ Visit any adjacent vertex of F which is not visited (G)
- ✔ Push **G** into the stack

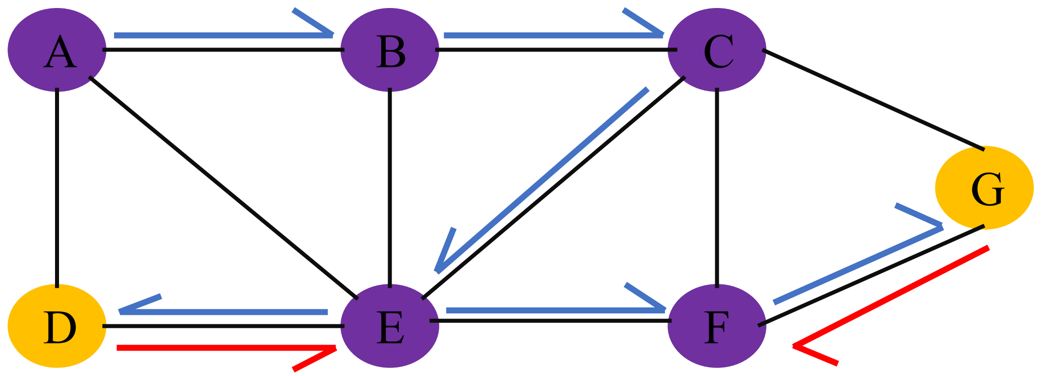




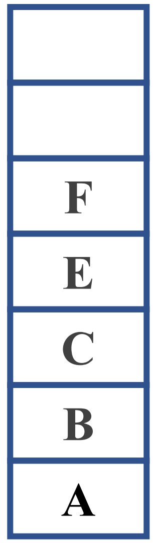
DFS (Example)

Step 9

- ✔ There is no new vertex to be visited from **G.** so use backtrack
- ✔ Pop **G** from the stack



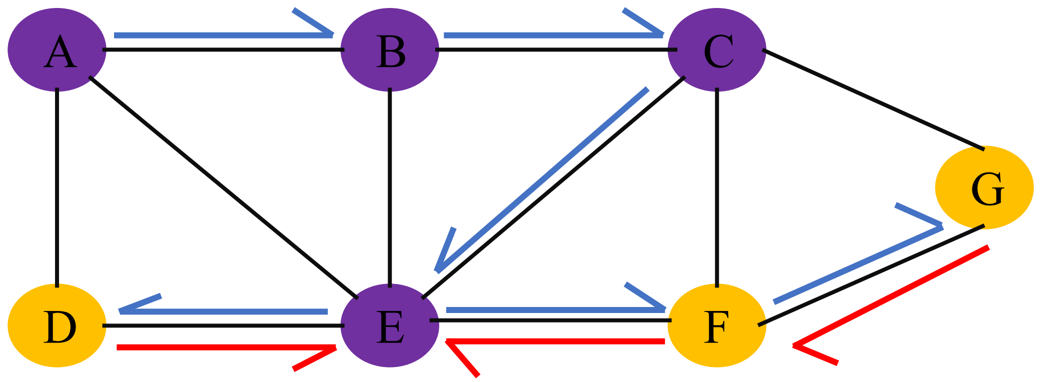
Stack



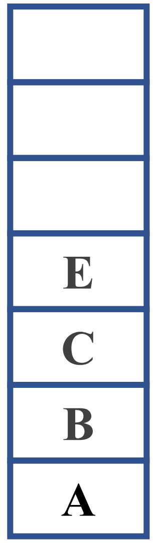
DFS (Example)

Step 10

- ✔ There is no new vertex to be visited from **F.** so use backtrack
- ✔ Pop **F** from the stack



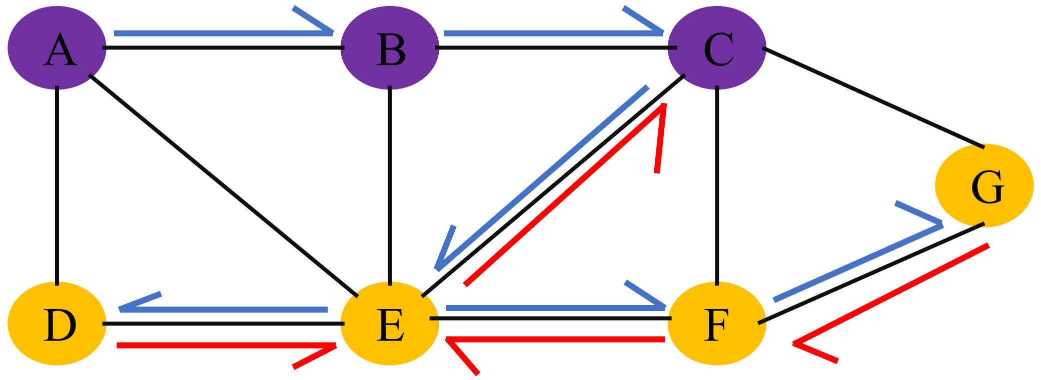
Stack



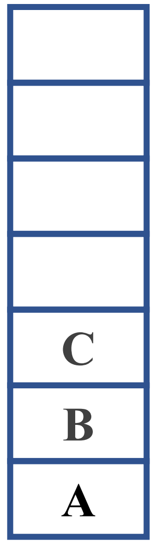
DFS (Example)

Step 11

- ✔ There is no new vertex to be visited from **E.** so use backtrack
- ✔ Pop **E** from the stack



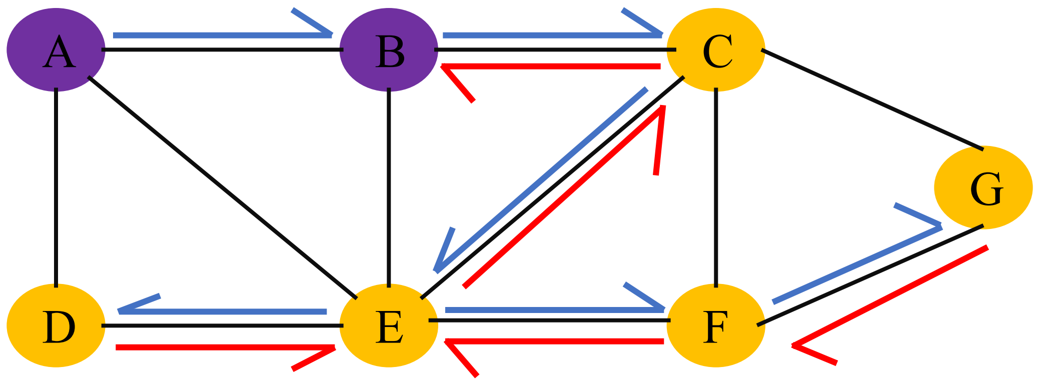
Stack

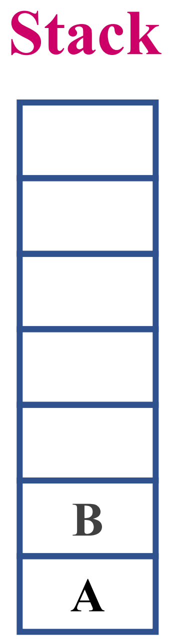


DFS (Example)

Step 12

- ✔ There is no new vertex to be visited from **C.** so use backtrack
- ✔ Pop C from the stack

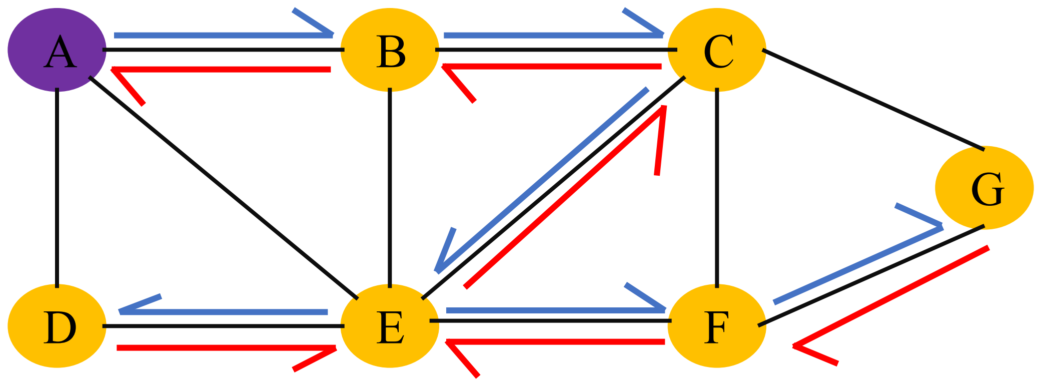


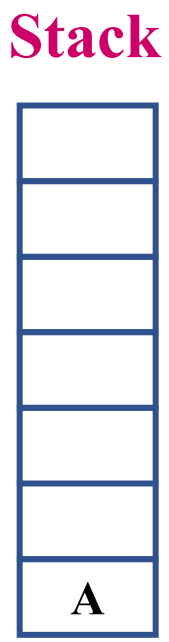


DFS (Example)

Step 13

- ✔ There is no new vertex to be visited from **B.** so use backtrack
- ✔ Pop **B** from the stack

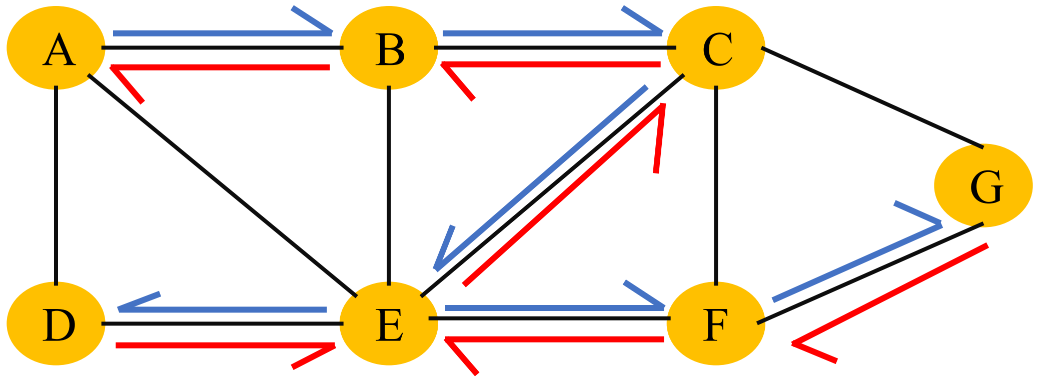




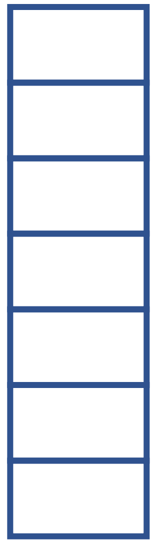
DFS (Example)

Step 14

- ✔ There is no new vertex to be visited from **A.** so use backtrack
- ✔ Pop **A** from the stack

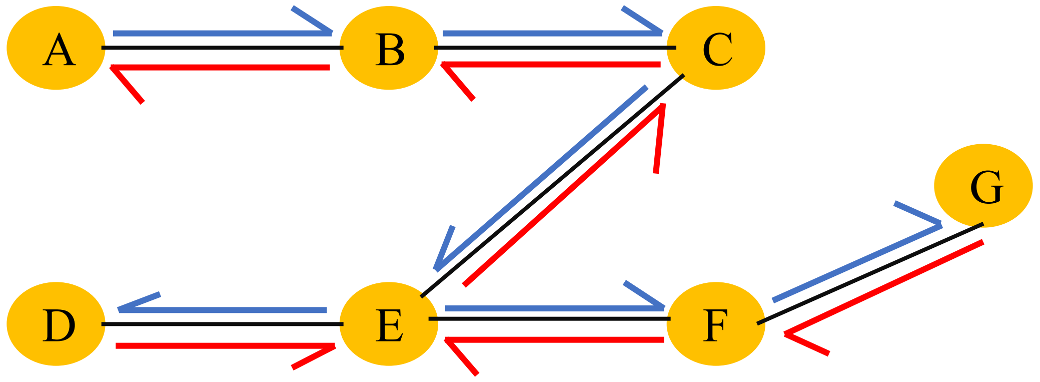


Stack



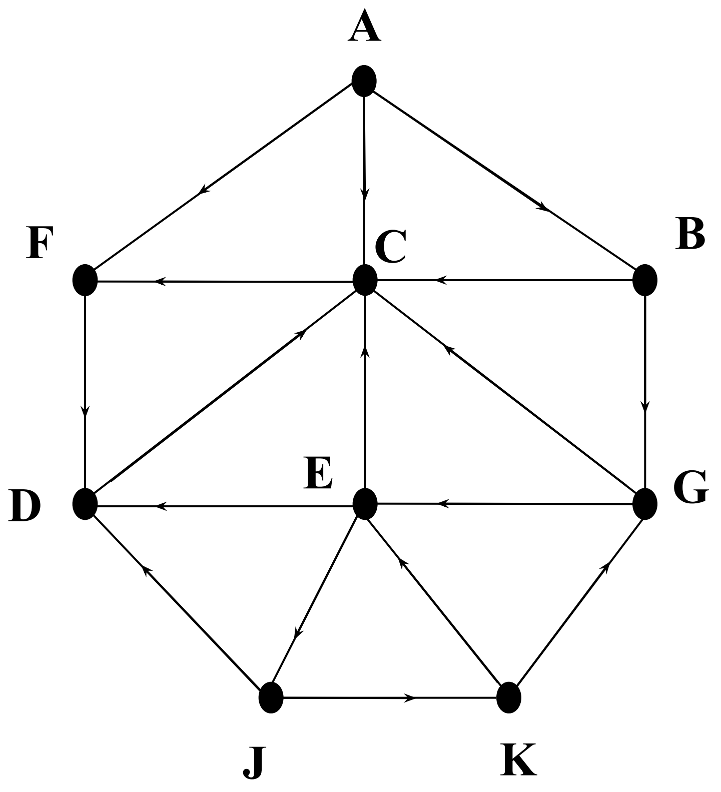
DFS (Example)

- ✔ Stack became Empty. So, stop the DFS process.
- ✔ Final result of DFS is a Spanning Tree as shown below



BFS (Example-2)

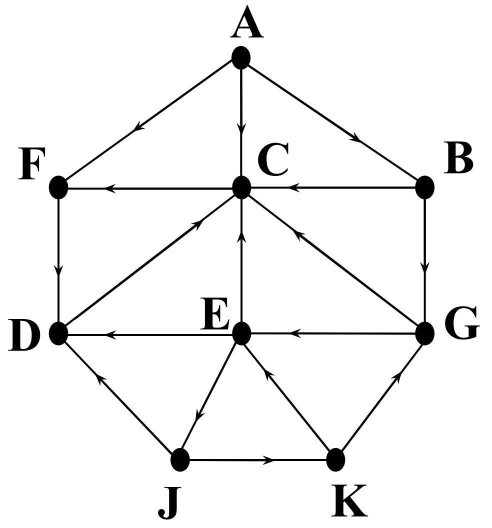
# Find the minimum path P from A to J for the Graph G



Adjacency List

A:	F, C, B
B:	C, G
C:	F
D:	C
E:	D, C, J
F:	D
G:	C,E
J:	D,K
K:	E,G

BFS (Example)

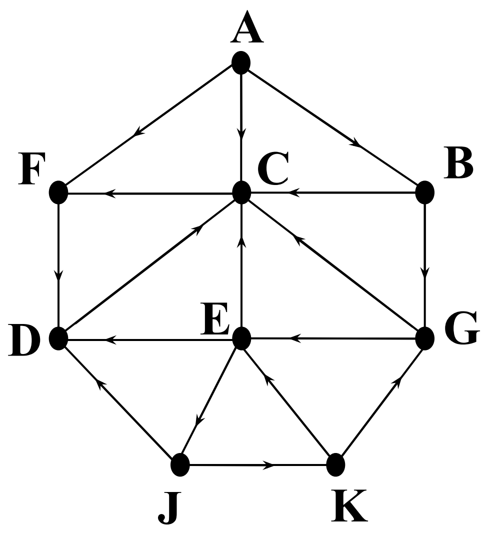


FRONT = 1

REAR = 1

QUEUE:	A
ORIG:	NULL

BFS (Example)

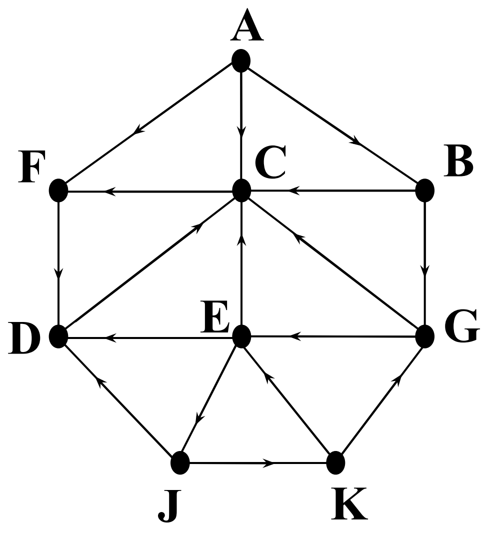


FRONT = 2

REAR = 4

QUEUE:	~~A~~	F	C	B
ORIG:	NULL	A	A	A

BFS (Example)

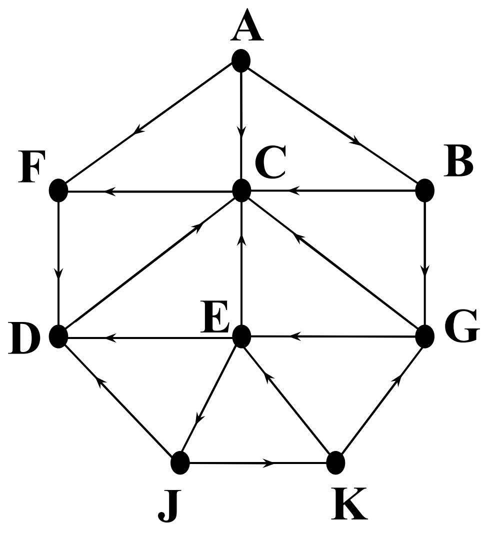


FRONT = 3

REAR = 5

QUEUE:	~~A~~	~~F~~	C	B	D
ORIG:	NULL	A	A	A	F

BFS (Example)

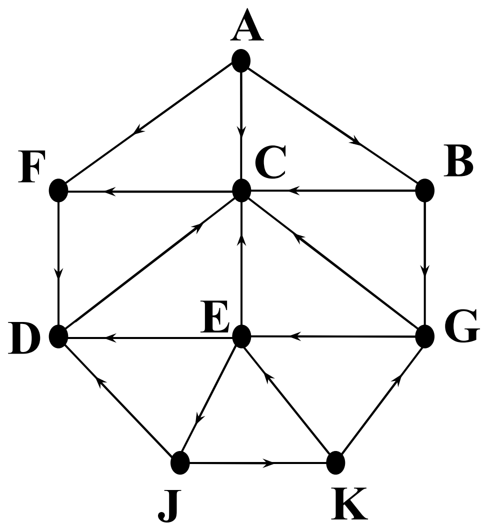


FRONT = 4

REAR = 5

QUEUE:	~~A~~	~~F~~	~~C~~	B	D
ORIG:	NULL	A	A	A	F

BFS (Example)

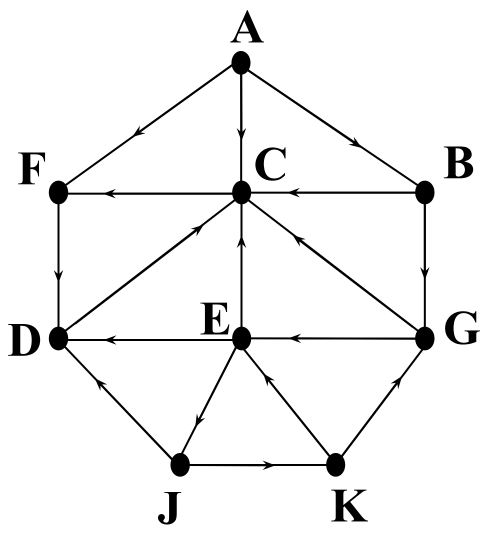


FRONT = 5

REAR = 6

QUEUE:	~~A~~	~~F~~	~~C~~	~~B~~	D	G
ORIG:	NULL	A	A	A	F	B

BFS (Example)

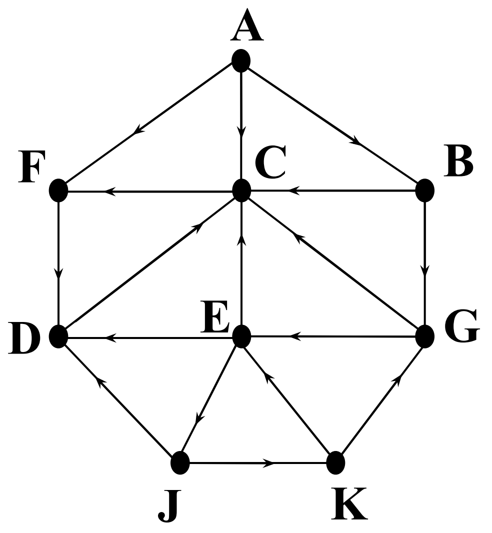


FRONT = 6

REAR = 6

QUEUE:	~~A~~	~~F~~	~~C~~	~~B~~	~~D~~	G
ORIG:	NULL	A	A	A	F	B

BFS (Example)

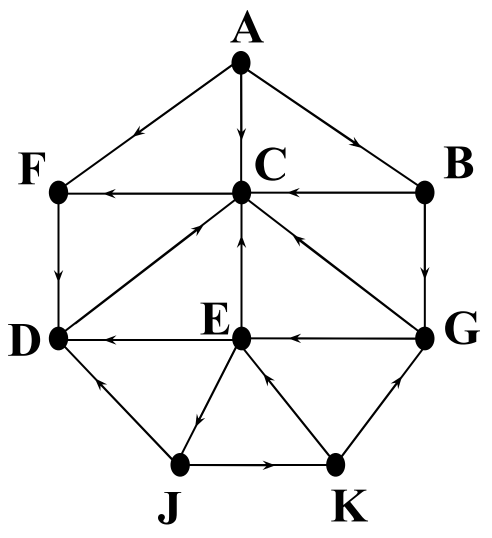


FRONT = 6

REAR = 7

QUEUE:	~~A~~	~~F~~	~~C~~	~~B~~	~~D~~	~~G~~	E
ORIG:	NULL	A	A	A	F	B	G

BFS (Example)



FRONT = 6

REAR = 8

QUEUE:	~~A~~	~~F~~	~~C~~	~~B~~	~~D~~	~~G~~	~~E~~	J
ORIG:	NULL	A	A	A	F	B	G	E

The optimum path is

$$J \longleftarrow E \longleftarrow G \longleftarrow B \longleftarrow A$$

Self

**Compare BFS and DFS

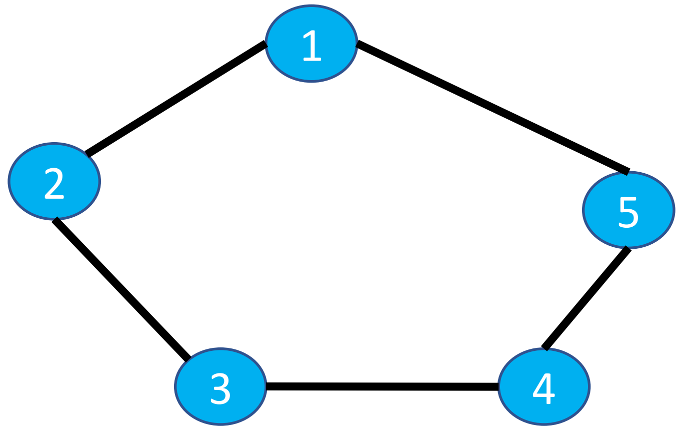
Minimum Spanning Tree

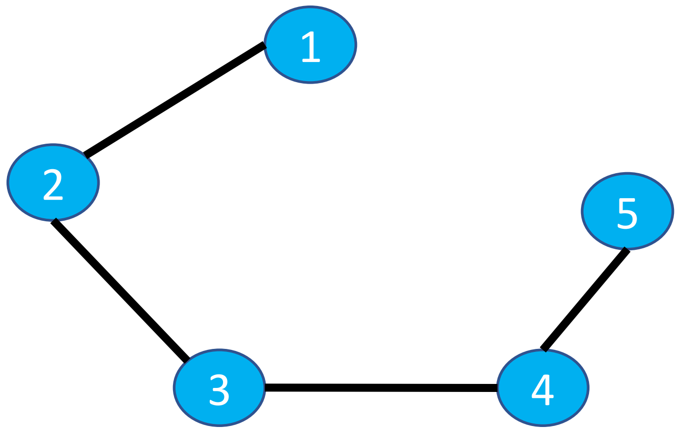
The spanning Tree of the graph, $G = (V, E)$ is

$$G'(V', E')$$

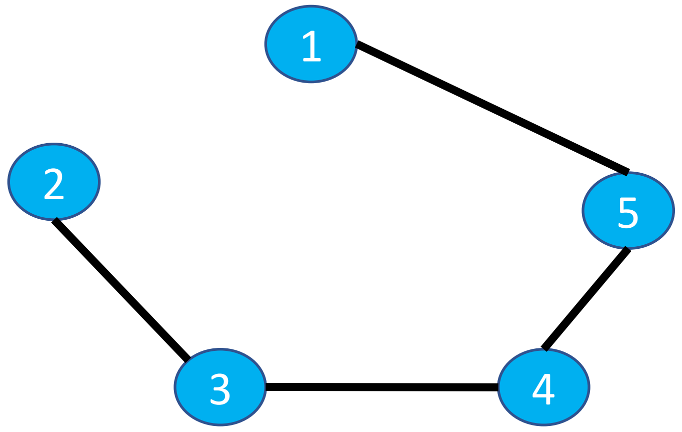
$$V' = V$$

$$E' = V-1$$

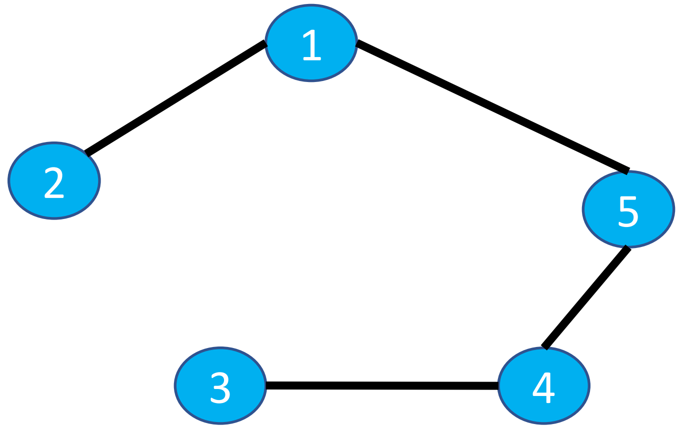




ST-1

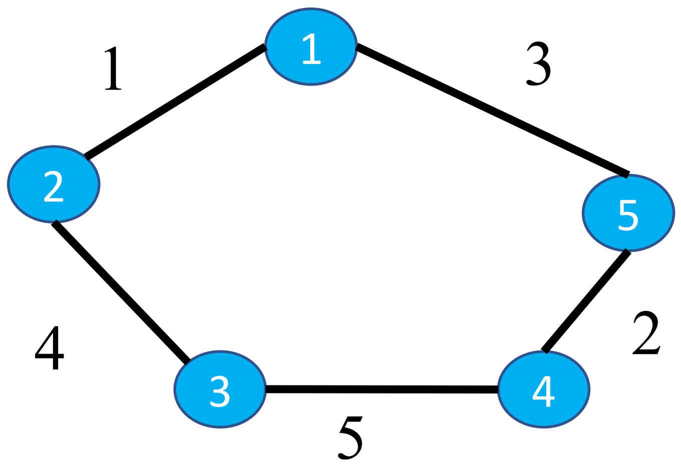


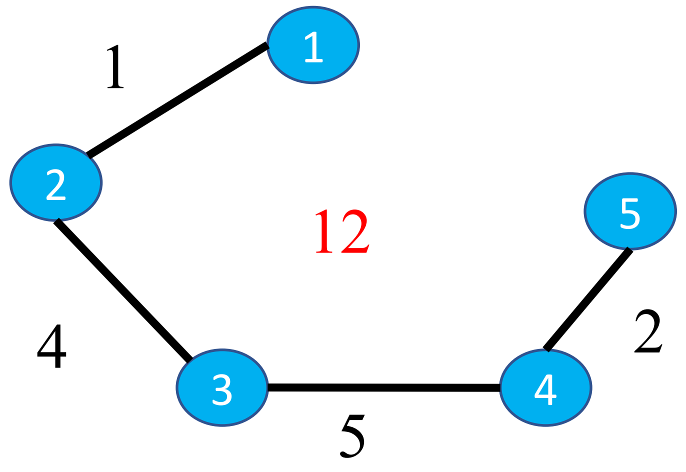
ST-2



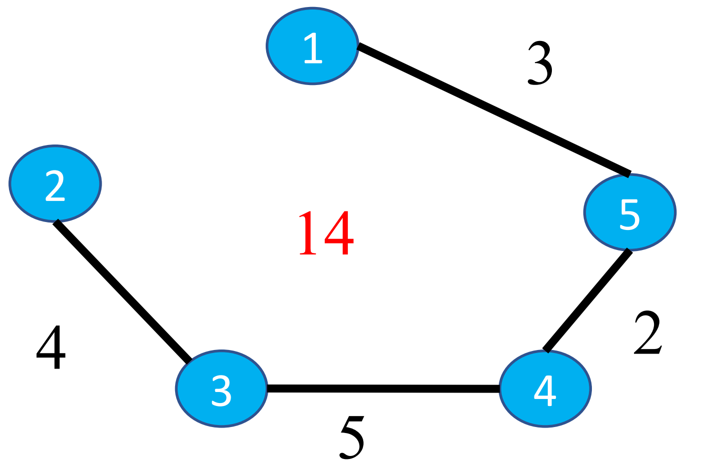
ST-3

Minimum Spanning Tree

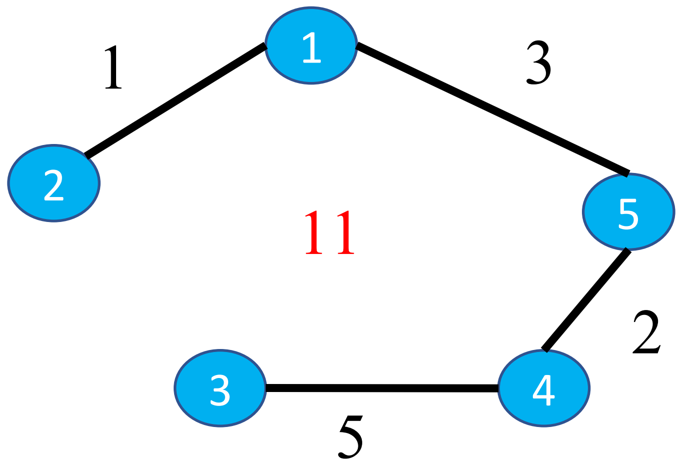




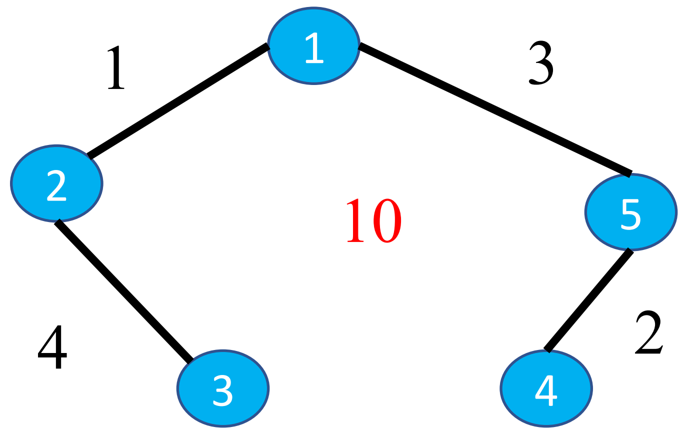
ST-1



ST-2



ST-3



MST

Minimum Spanning Tree

Self

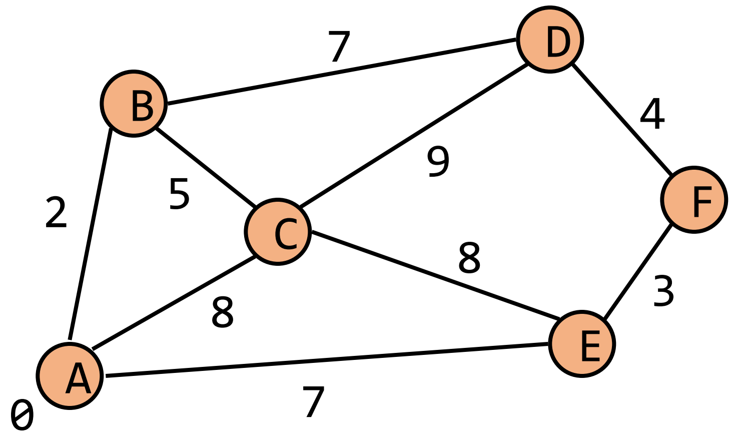
Properties of Minimum Spanning Tree

Prim's Algorithm

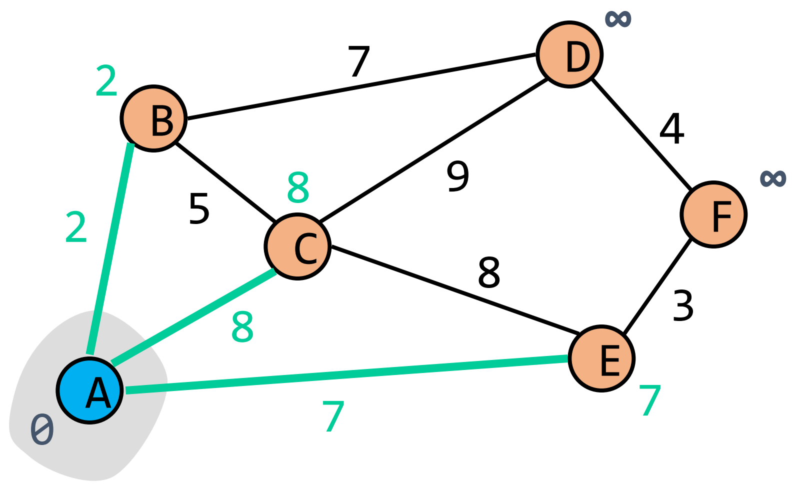
```
T = a spanning tree containing a single node s;
E = set of edges adjacent to s;
while T does not contain all the nodes {
    remove an edge (v, w) of lowest cost from E
    if (w is already in T){
      discard edge (v, w)}
    else {
        add edge (v, w) and node v to T
        add to E the edges adjacent to v
    }
}
```

- An edge of lowest cost can be found with a priority queue

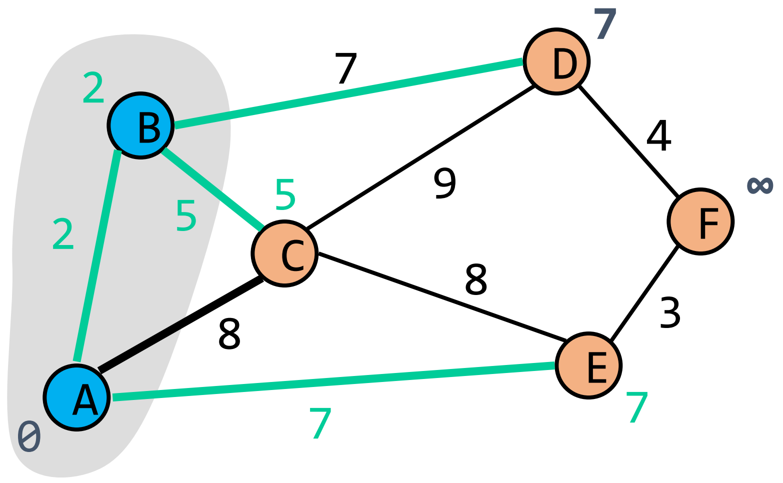
Prim's Algorithm



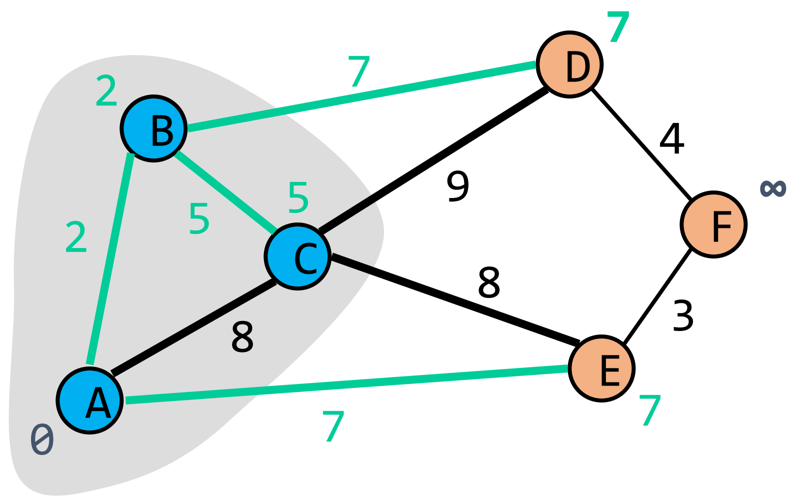
Prim's Algorithm



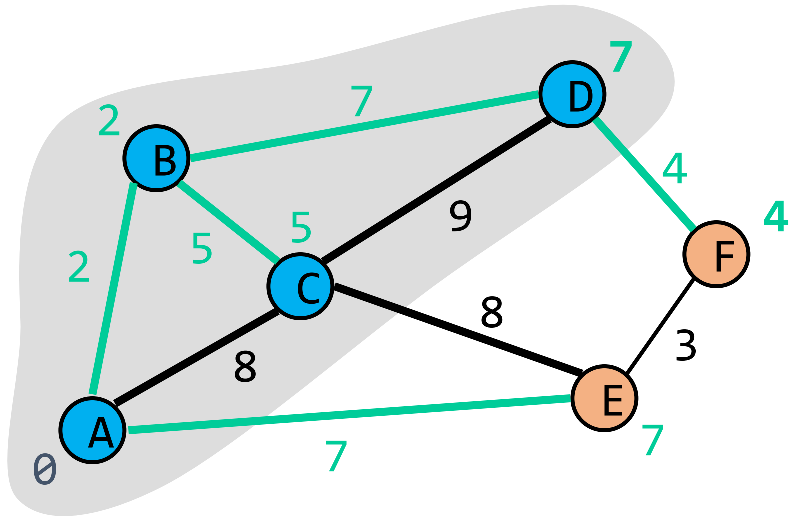
Prim's Algorithm



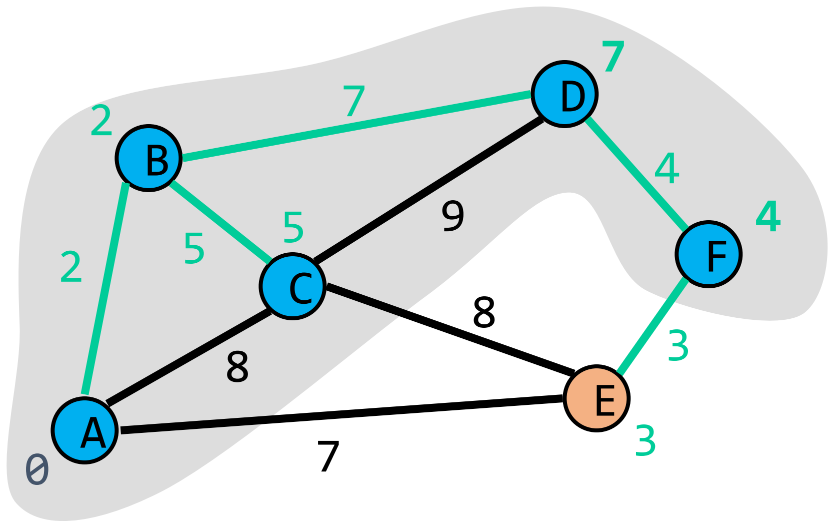
Prim's Algorithm



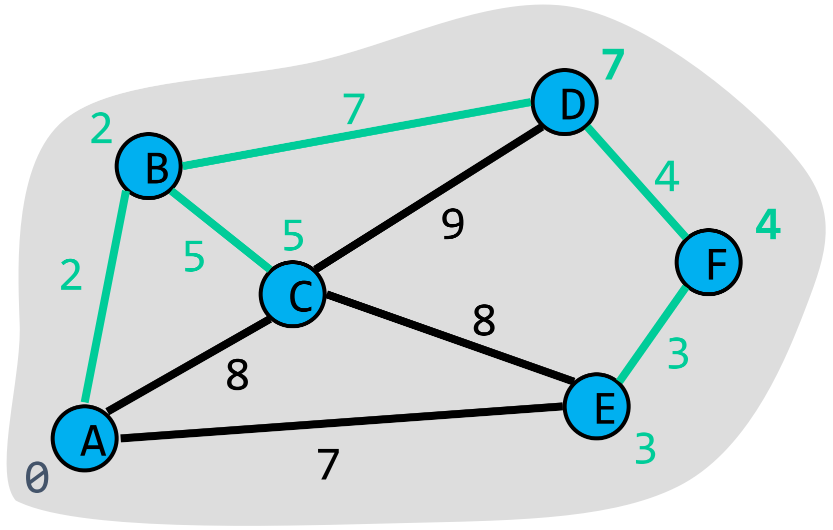
Prim’s Algorithm



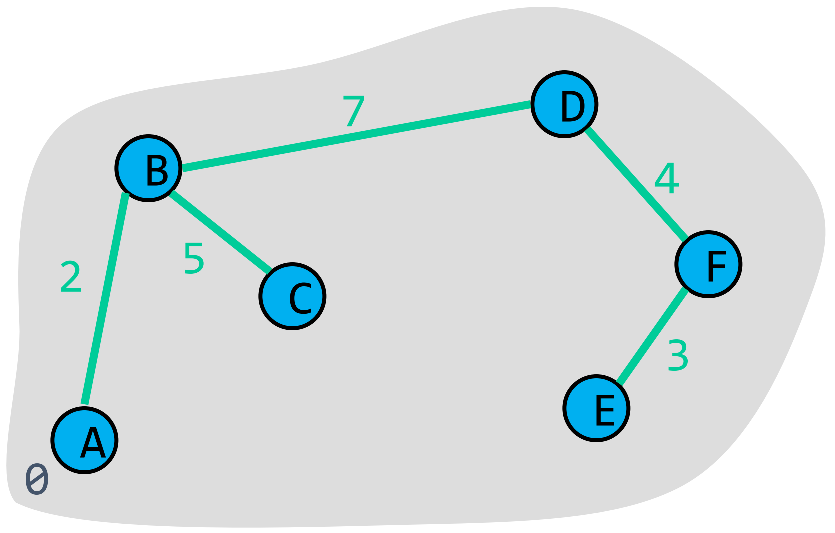
Prim's Algorithm



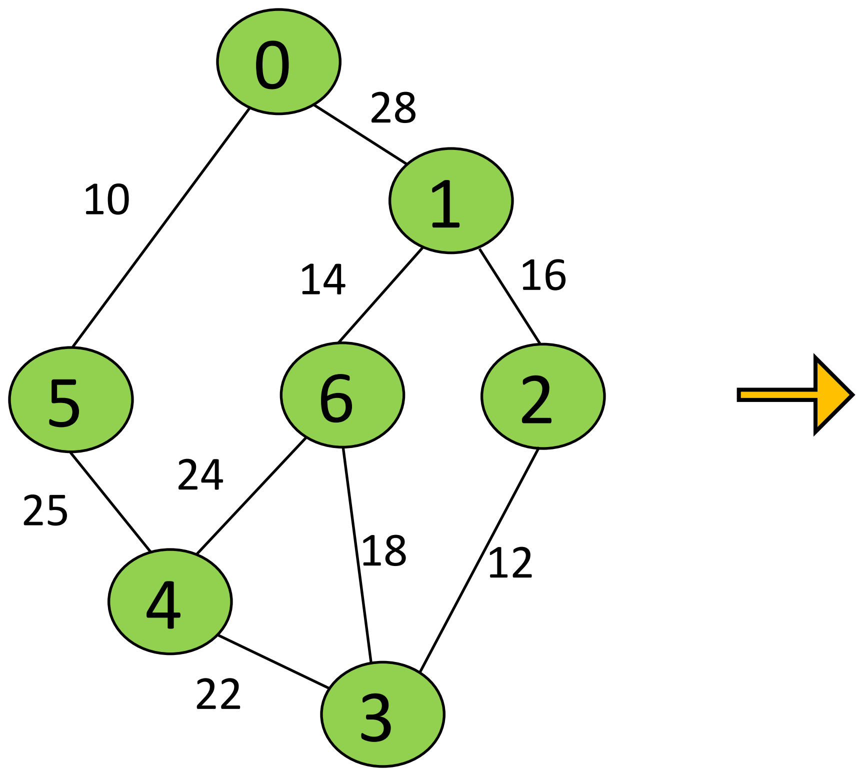
Prim's Algorithm

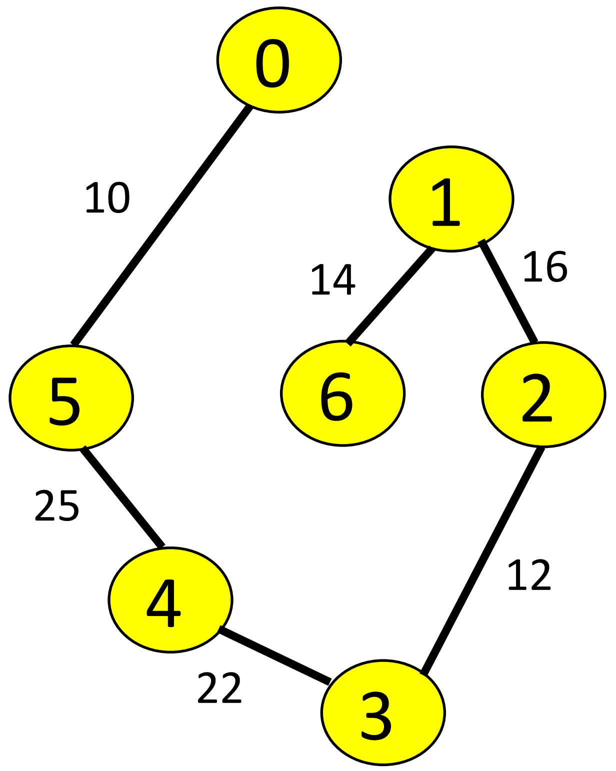


Prim's Algorithm



Example





Kruskal’s Algorithm

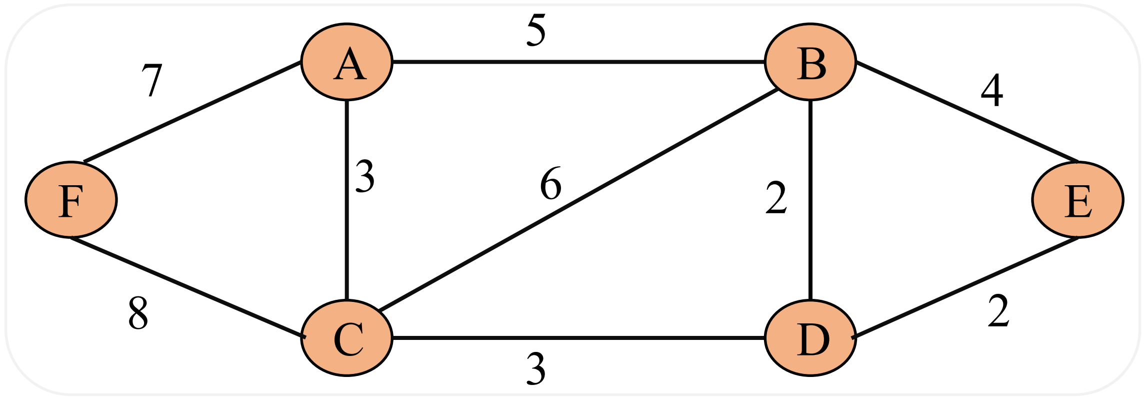
```
T= {};
while (T contains less than n-1 edges  && E is not empty) {
   choose a least cost edge (v,w) from E;
   delete (v,w) from E;
  if ((v,w) does not create a cycle in T){
      add (v,w) to T
        }
  else {
      discard (v,w)
        }
  }
if (T contains fewer than n-1 edges)
  printf(“No spanning tree\n”);
```

Kruskal's Algorithm

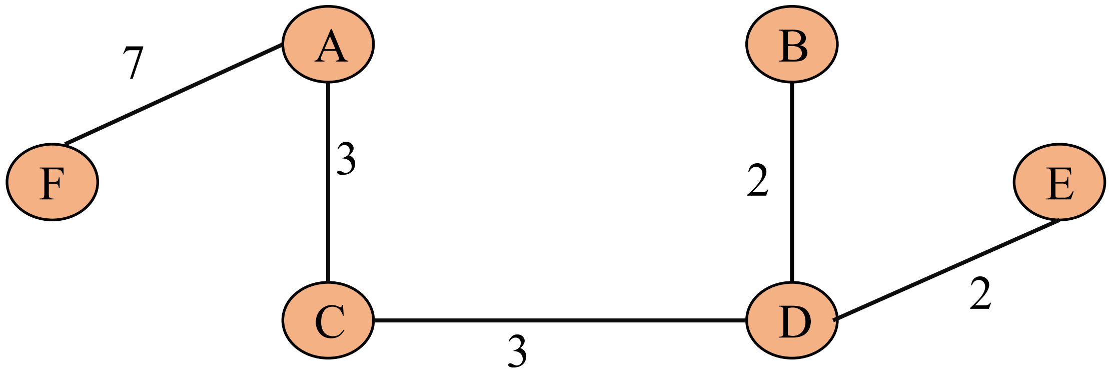
```
T = empty spanning tree;
E = set of edges;
N = number of nodes in graph;
while T has fewer than N - 1 edges {
    remove an edge (v, w) of lowest cost from E
    if adding (v, w) to T would create a cycle
        then discard (v, w)
        else add (v, w) to T
}
```

- Finding an edge of lowest cost can be done just by sorting the edges

Example

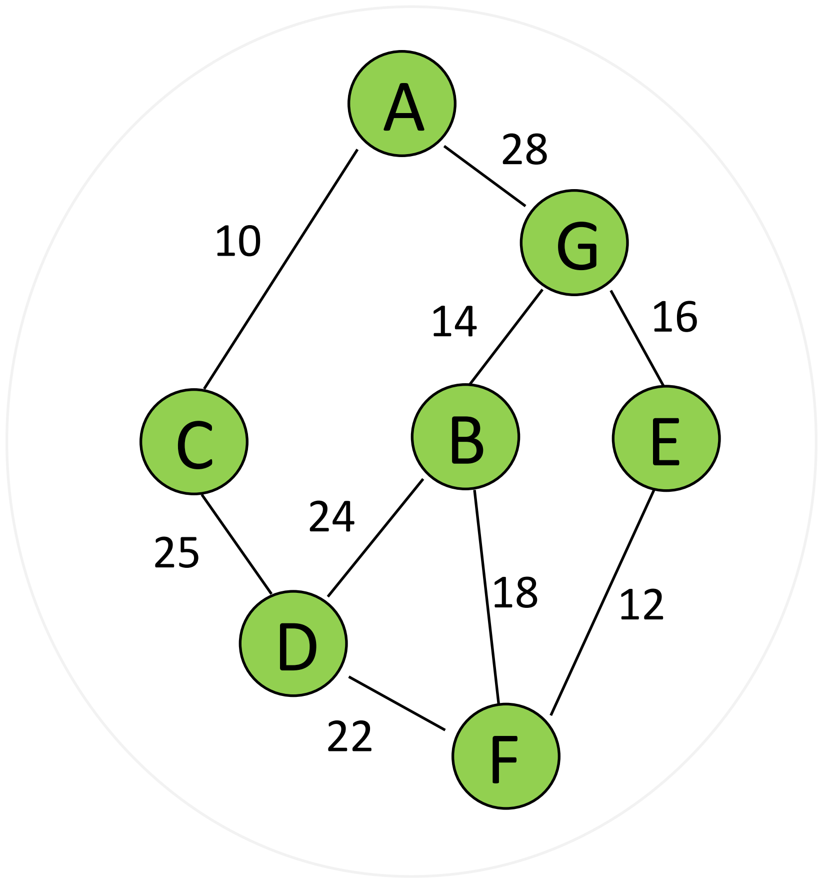


BD □ 2
DE □ 2
CD □ 3
CA □ 3
✖ BE □ 4
✖ AB □ 5
✖ CB □ 6
FA □ 7
✖ FC □ 8



Minimum Spanning Tree

Exercise



Time Complexity

- Let v be number of vertices and e the number of edges of a given graph.
- **Kruskal's algorithm:** $O(e \log e)$
- **Prim's algorithm:** $O(e \log v)$

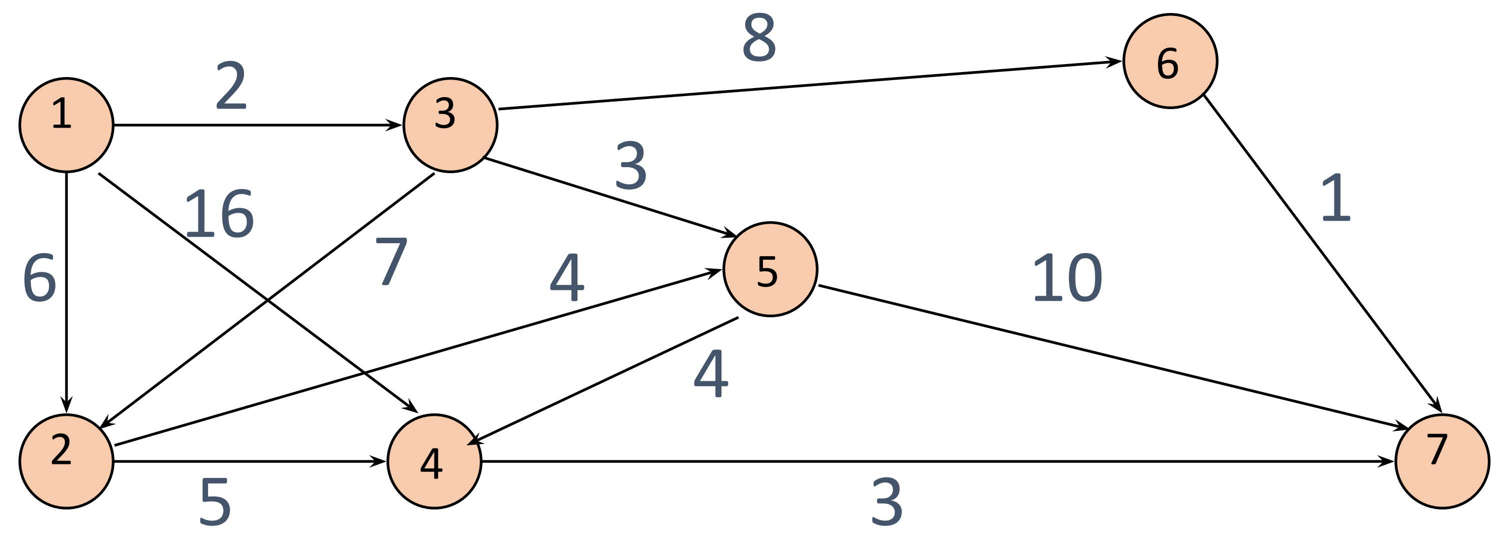
Shortest Path Problem

- Path length is sum of weights of edges on path.
- The vertex at which the path begins is the **source vertex**.
- The vertex at which the path ends is the **destination vertex**.

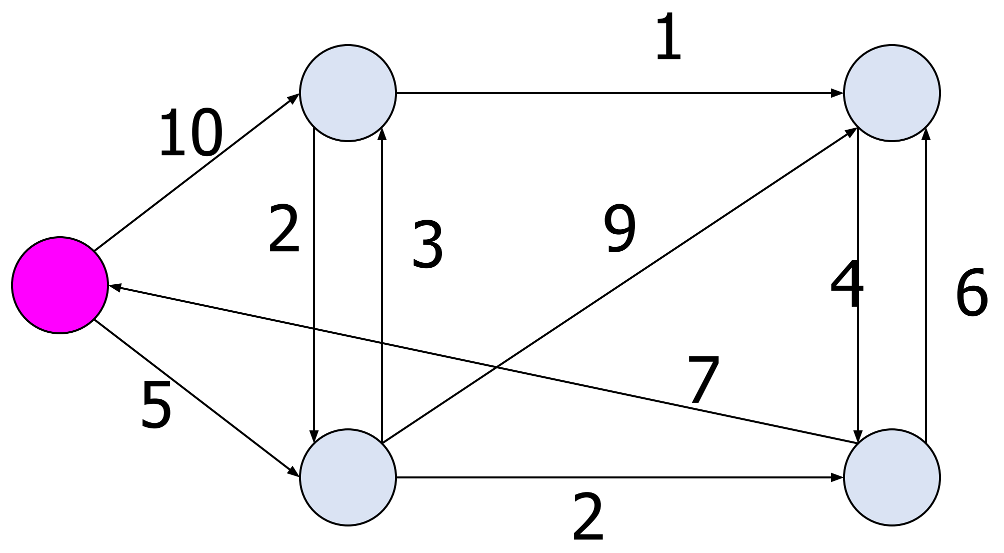
□ Types:

- Single source single destination.
- Single source all destinations.
- All pairs (every vertex is a source and destination).

Shortest Path Problem

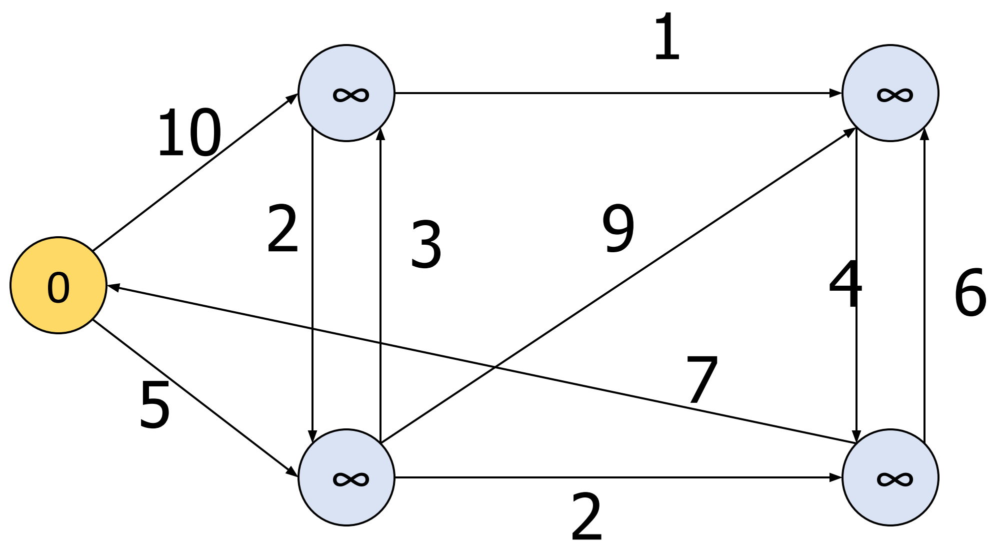


Dijkstra's Algorithm - Example

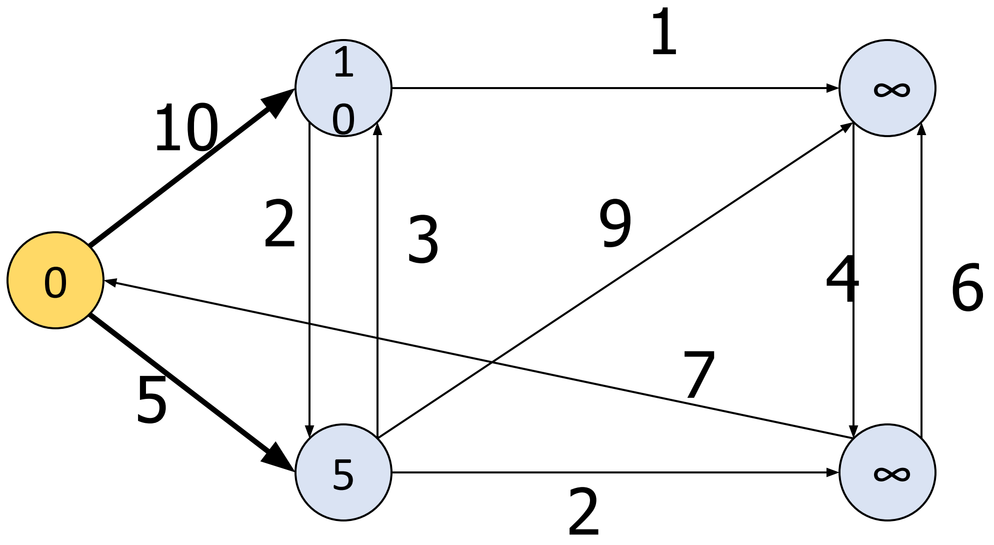


source
destination
🟦 the shortest path between the source and destination
🟦 a subpath which is also the shortest path between its source and destination

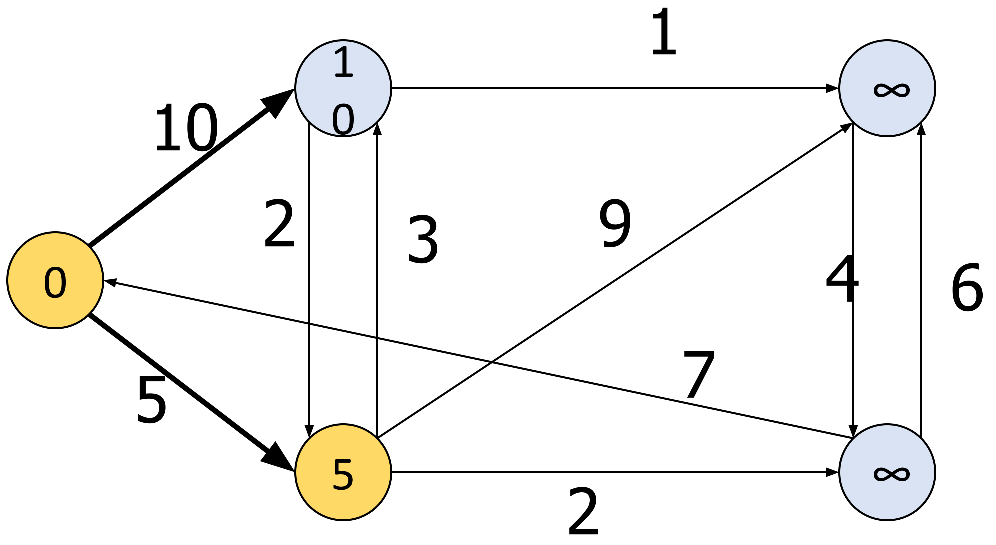
Dijkstra's Algorithm - Example



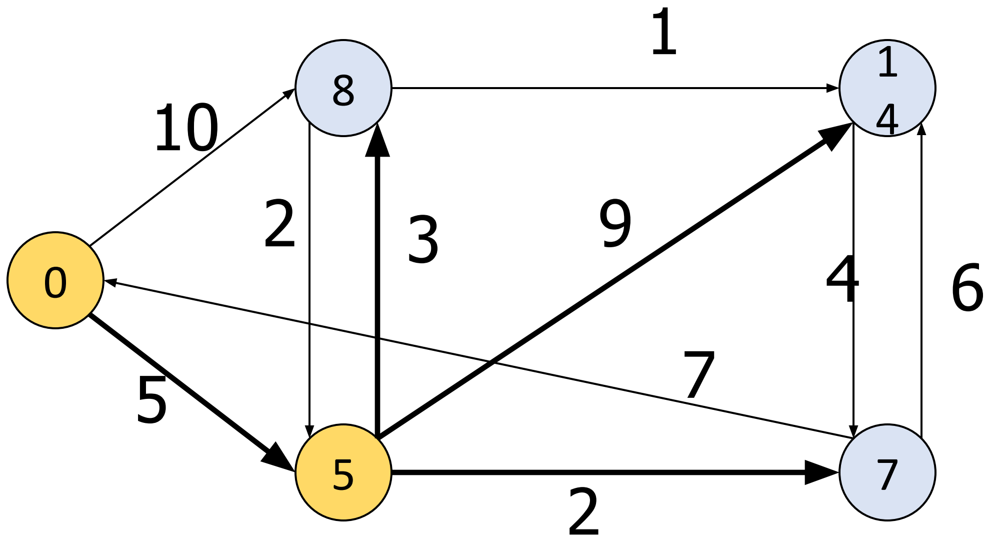
Dijkstra's Algorithm - Example



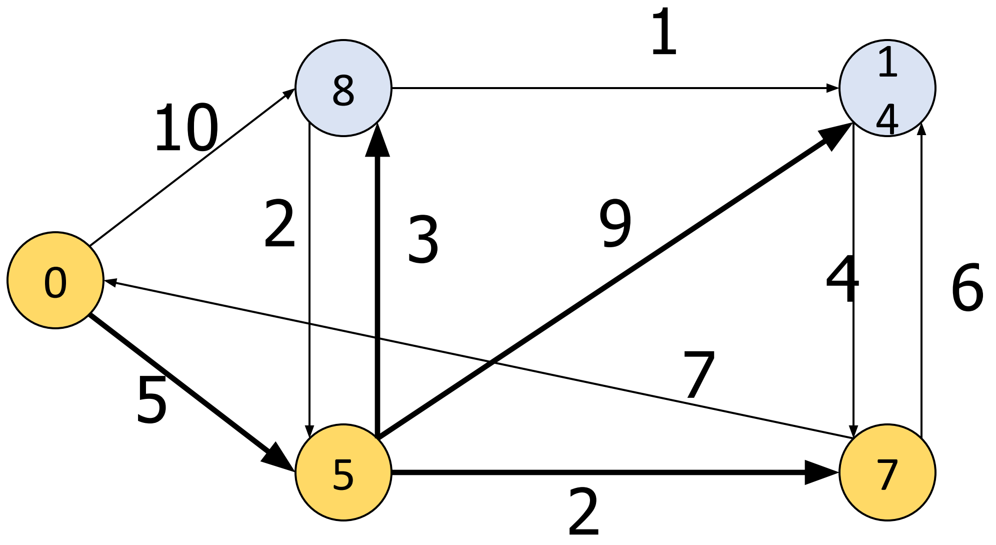
Dijkstra’s Algorithm - Example



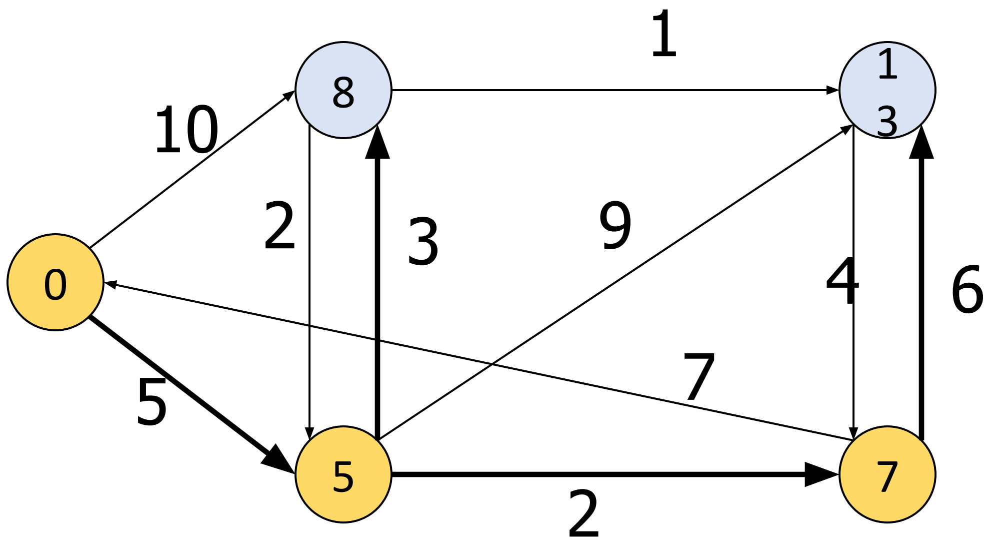
Dijkstra’s Algorithm - Example



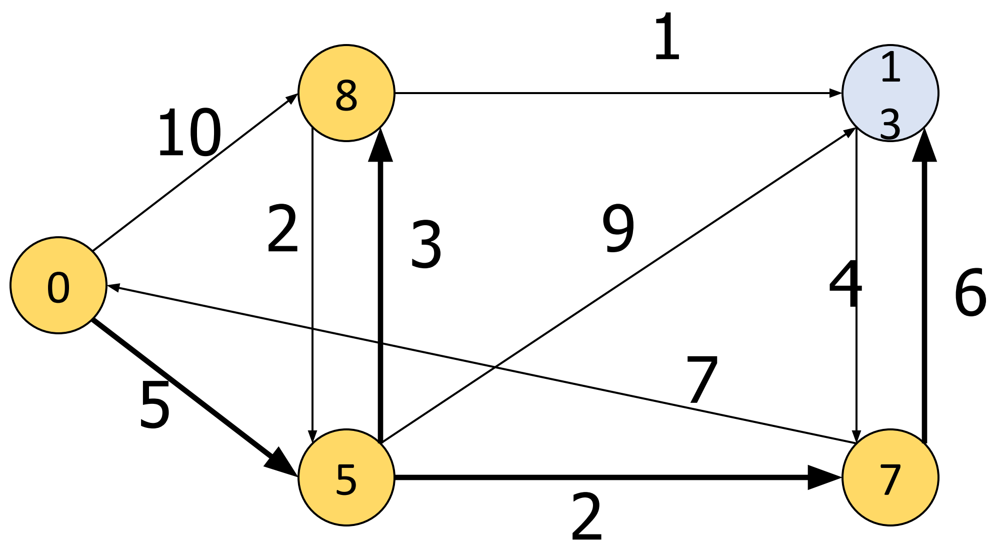
Dijkstra's Algorithm - Example



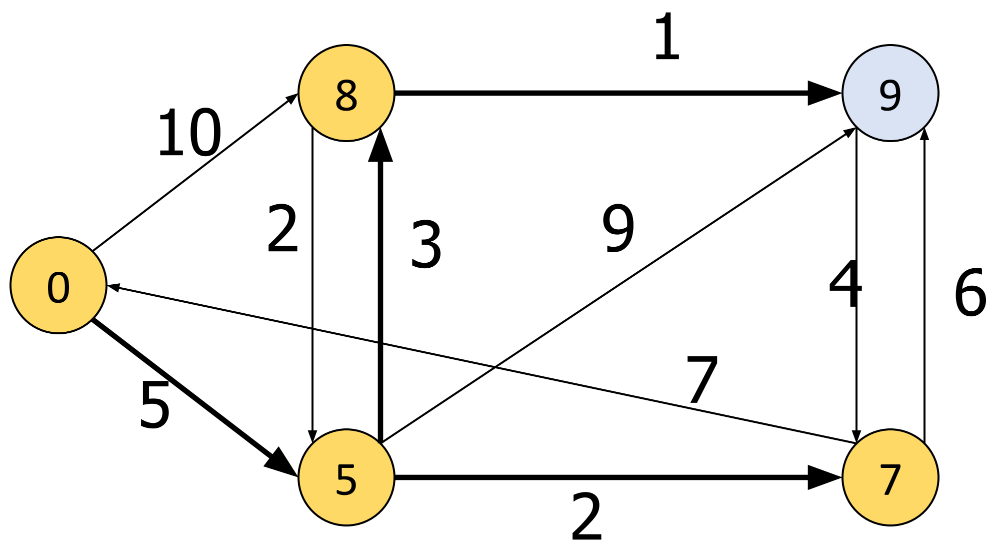
Dijkstra's Algorithm - Example



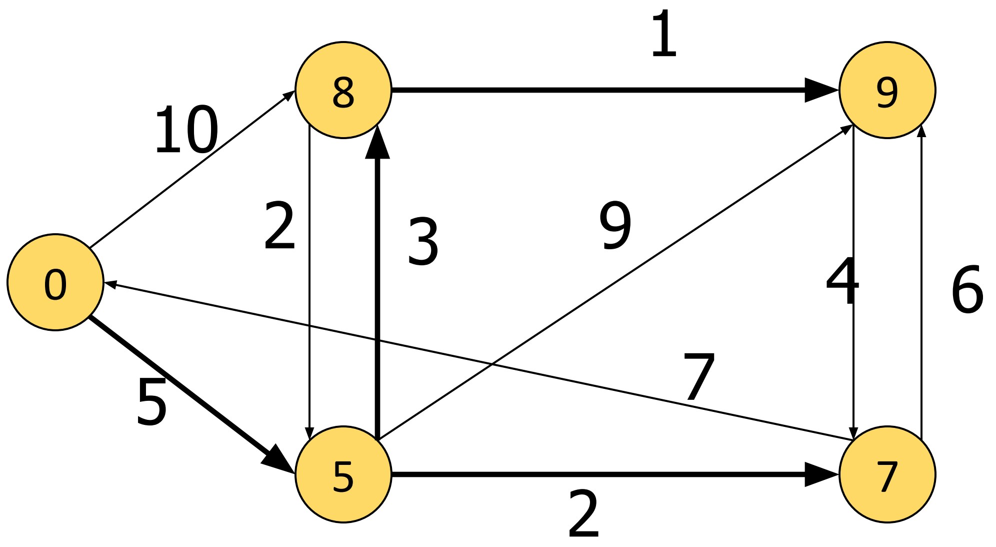
Dijkstra's Algorithm - Example



Dijkstra's Algorithm - Example

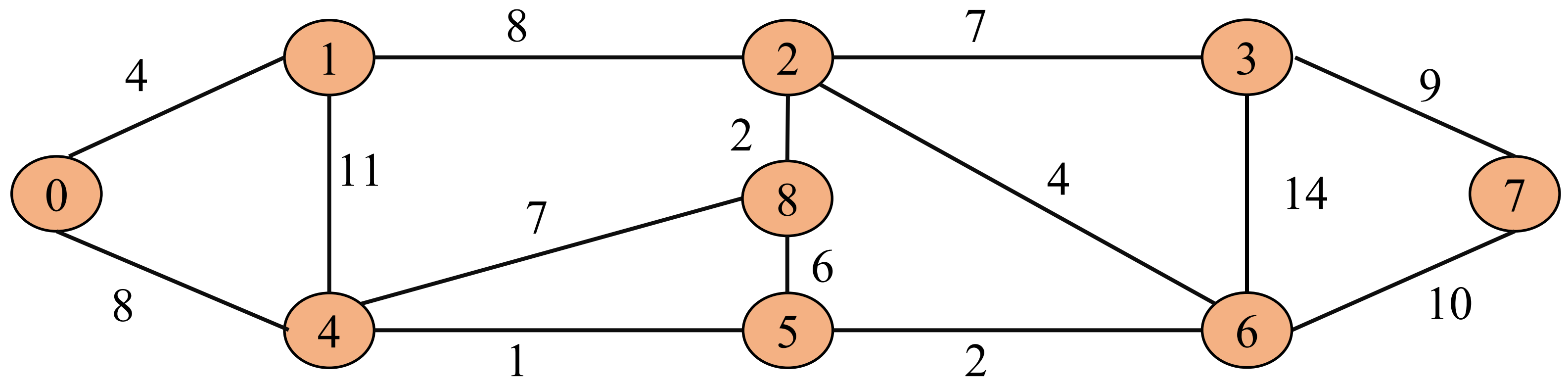


Dijkstra’s Algorithm - Example



Terminate single source all destinations greedy algorithm as soon as shortest path to desired vertex has been generated.

Dijkstra’s Algorithm - Exercise



Thank You