Semester III,
B. E. Computer Science & Engineering
(Artificial Intelligence and Machine Learning)
Course Code : CAT201
Course : Data Structure
L: 3Hrs, T: 1Hr, P: 0Hr, Per Week Total Credits : 04

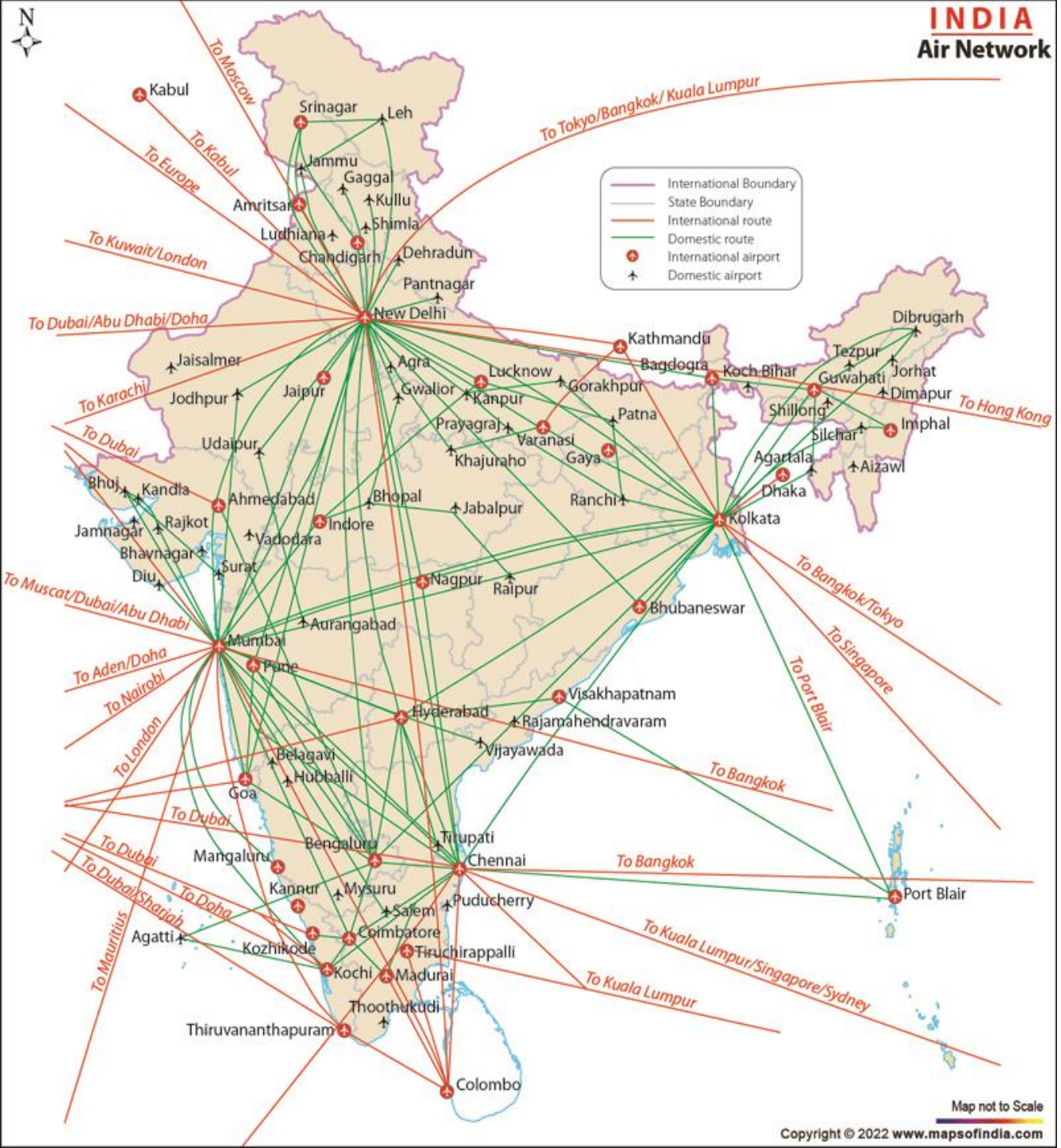
Course Objectives

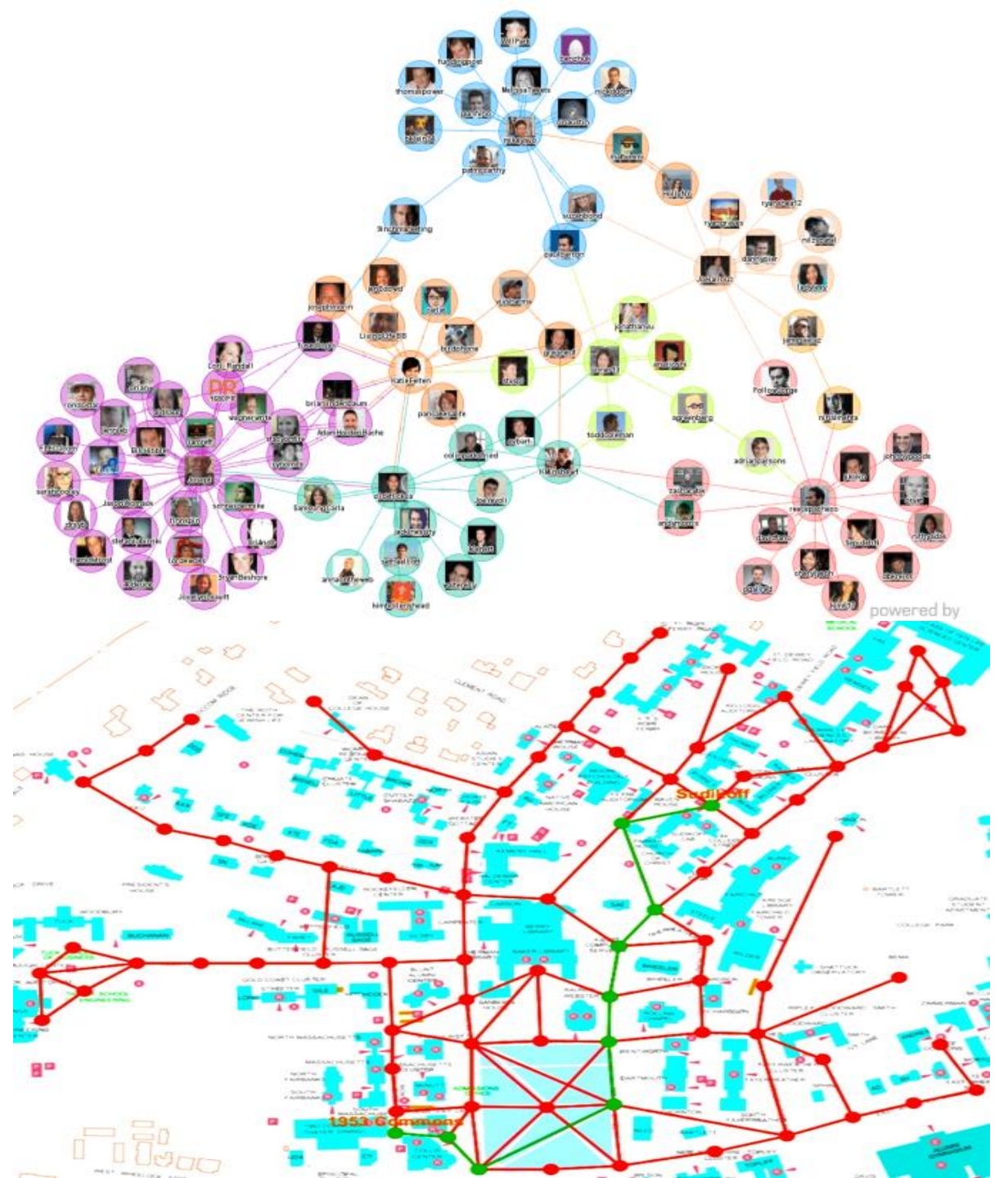
1. To impart to students the basic concepts of data structures and algorithms.
2. To familiarize students on different searching and sorting techniques.
3. To prepare students to use linear (stacks, queues, linked lists) and non-linear (trees, graphs) data structures.
4. To enable students to devise algorithms for solving real-world problems.

Course Outcomes

On completion of the course the student will be able to

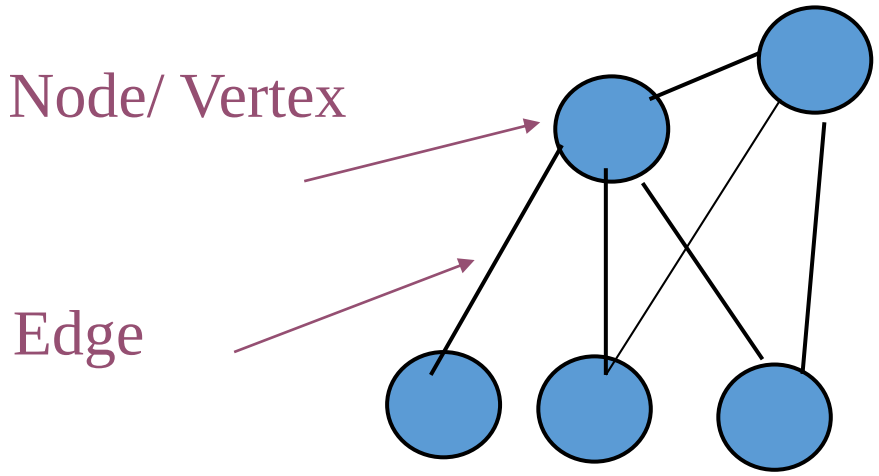
1. Recognize different ADTs and their operations and specify their complexities.
2. Design and realize linear data structures (stacks, queues, linked lists) and analyze their computation complexity.
3. Devise different sorting (comparison based, divide-and-conquer, distributive, and tree- based) and searching (linear, binary) methods and analyze their time and space requirements.
4. Design traversal and path finding algorithms for Trees and Graphs.

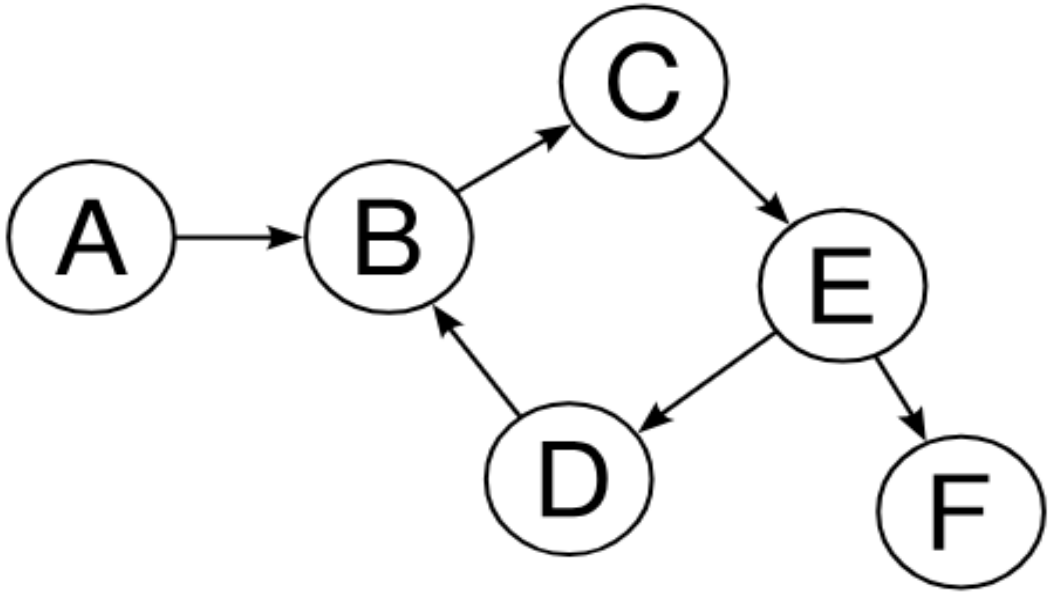

INDIA
Air Network
N
International Boundary
State Boundary
International route
Domestic route
International airport
Domestic airport
Kabul
Srinagar
Leh
Jammu
Gagga
Kullu
Amritsar
Shimla
Ludhiana
Chandigarh
Dehradun
Pantnagar
New Delhi
Jaisalmer
Jodhpur
Jaipur
Agra
Gwalior
Lucknow
Kanpur
Gorakhpur
Udaipur
Prayagraj
Varanasi
Khajuraho
Patna
Gaya
Ranchi
Bhuj
Kandla
Ahmedabad
Bhopal
Jabalpur
Jamnagar
Rajkot
Indore
Vadodara
Bhavnagar
Surat
Diu
Nagpur
Raipur
Aurangabad
Mumbai
Pune
Goa
Belagavi
Hubballi
Hyderabad
Visakhapatnam
Rajamahendravaram
Vijayawada
Bengaluru
Tirupati
Mangaluru
Chennai
Kannur
Mysuru
Salem
Puducherry
Coimbatore
Agatti
Kozhikode
Tiruchirappalli
Kochi
Madurai
Thoothukudi
Thiruvananthapuram
Colombo
Kolkata
Bhubaneswar
Port Blair
Kathmandu
Bagdogra
Koch Bihar
Dibrugarh
Tezpur
Jorhat
Guwahati
Dimapur
Shillong
Imphal
Silchar
Agartala
Aizawl
Dhaka
To Moscow
To Kabul
To Europe
To Tokyo/Bangkok/ Kuala Lumpur
To Kuwait/London
To Dubai/Abu Dhabi/Doha
To Karachi
To Hong Kong
To Dubai
To Muscat/Dubai/Abu Dhabi
To Aden/Doha
To Nairobi
To London
To Dubai
To Dubai
To Dubai/Sharjah
To Doha
To Mauritius
To Bangkok/Tokyo
To Singapore
To Port Blair
To Bangkok
To Bangkok
To Kuala Lumpur/Singapore/Sydney
To Kuala Lumpur
Map not to Scale
Copyright © 2022 www.mapsofindia.com


powered by

Graph Definition

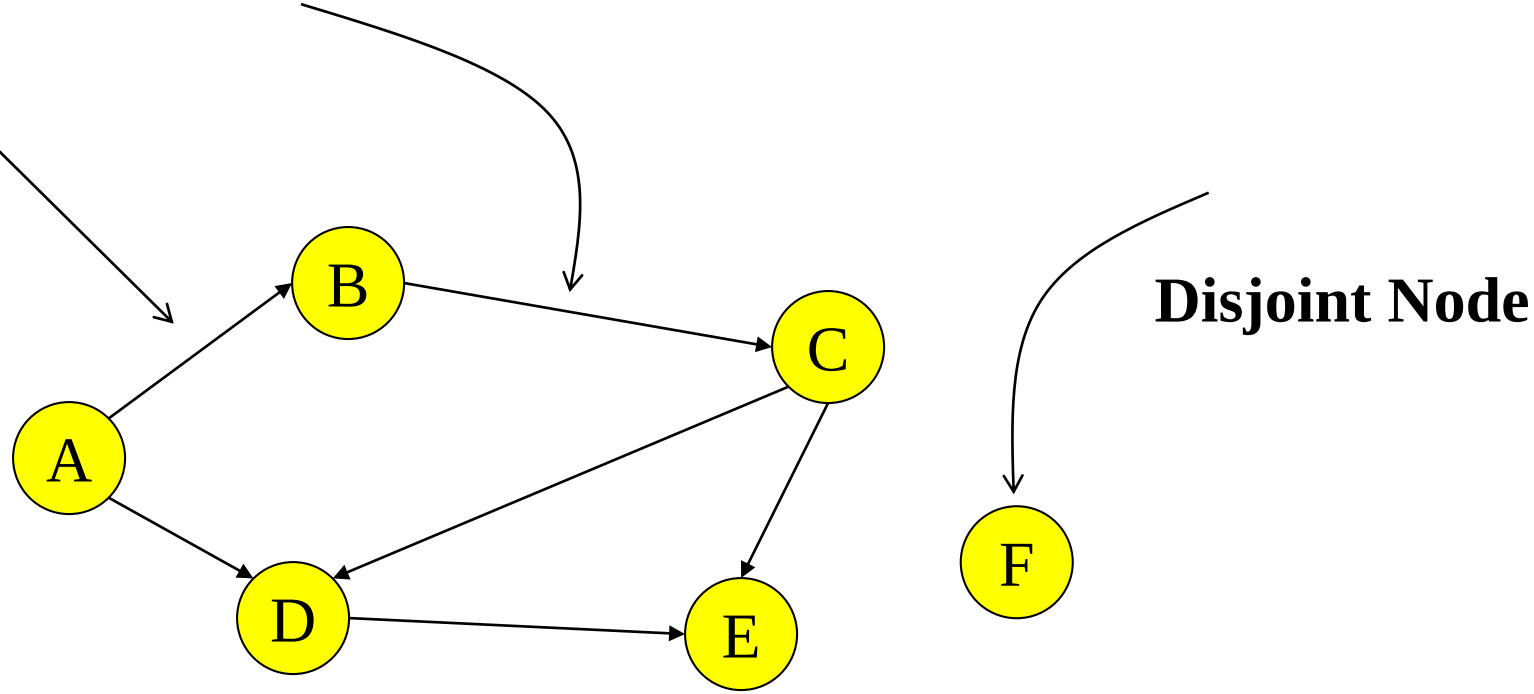
- A graph is simply a collection of nodes plus edges
  - Linked lists, trees, and heaps are all special cases of graphs
- The nodes are known as vertices (node = “vertex”)
- Formal Definition: A graph *G* is a pair (*V*, *E*) where
  - *V* is a set of vertices or nodes
  - *E* is a set of edges that connect vertices





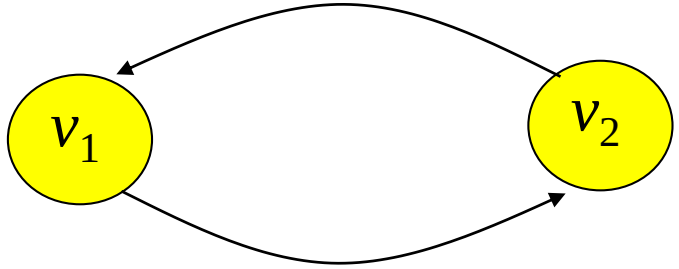
Graph Example

- Here is a directed graph $G = (V, E)$
  - Each edge is a pair (v_1, v_2), where v_1, v_2 are vertices in V
  - V = {A, B, C, D, E, F}
  - E = {(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)}

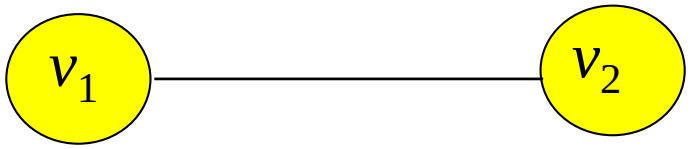


Directed vs Undirected Graphs

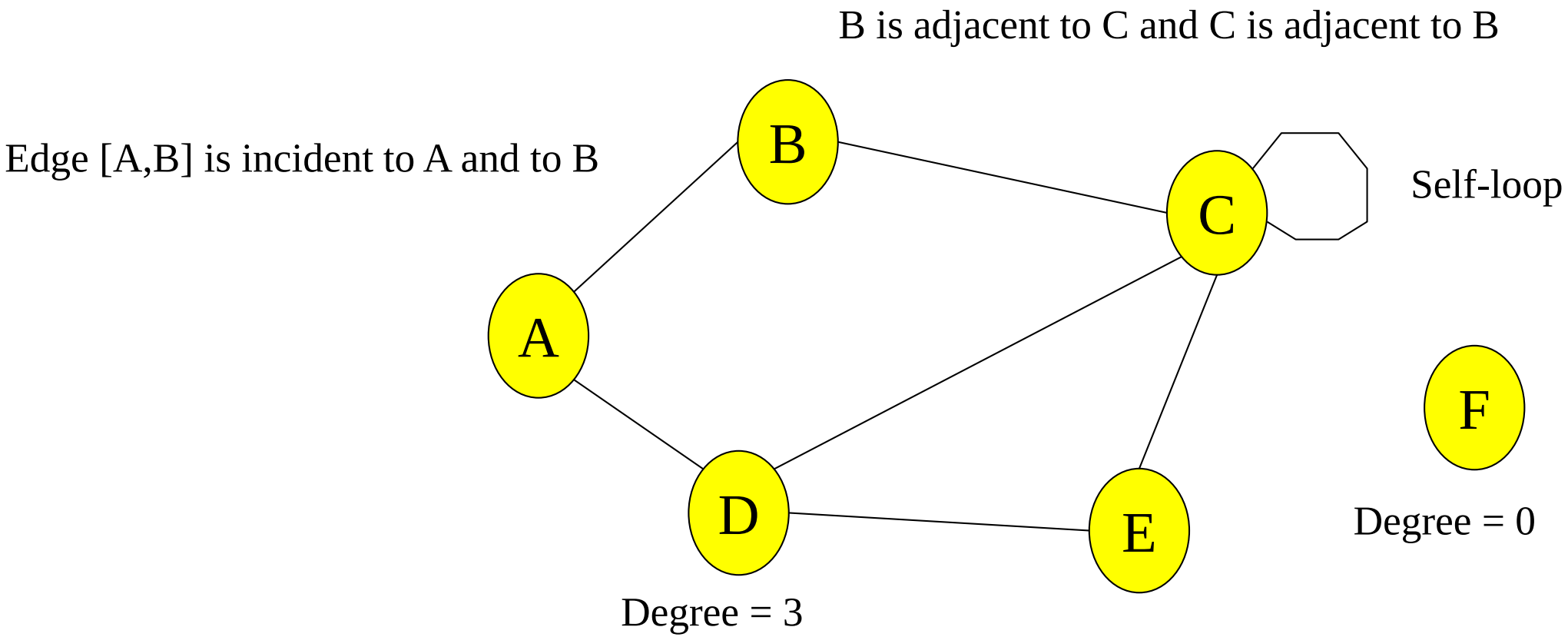
- If the order of edge pairs (v_1, v_2) matters, the graph is directed (also called a digraph): $(v_1, v_2) \neq (v_2, v_1)$



- If the order of edge pairs (v_1, v_2) does not matter, the graph is called an undirected graph: in this case, $(v_1, v_2) = (v_2, v_1)$



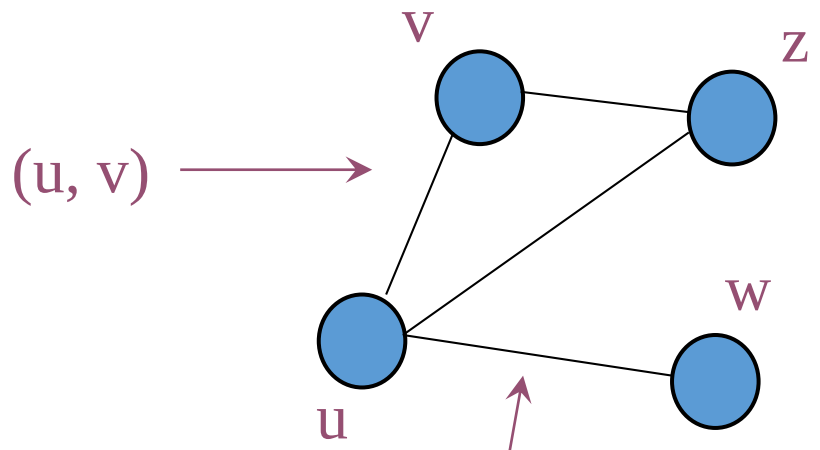
Undirected Graph Terminology



- **Degree**: The total number of edges connected to a vertex is said to be the degree of that vertex.

Directed Graph Terminology

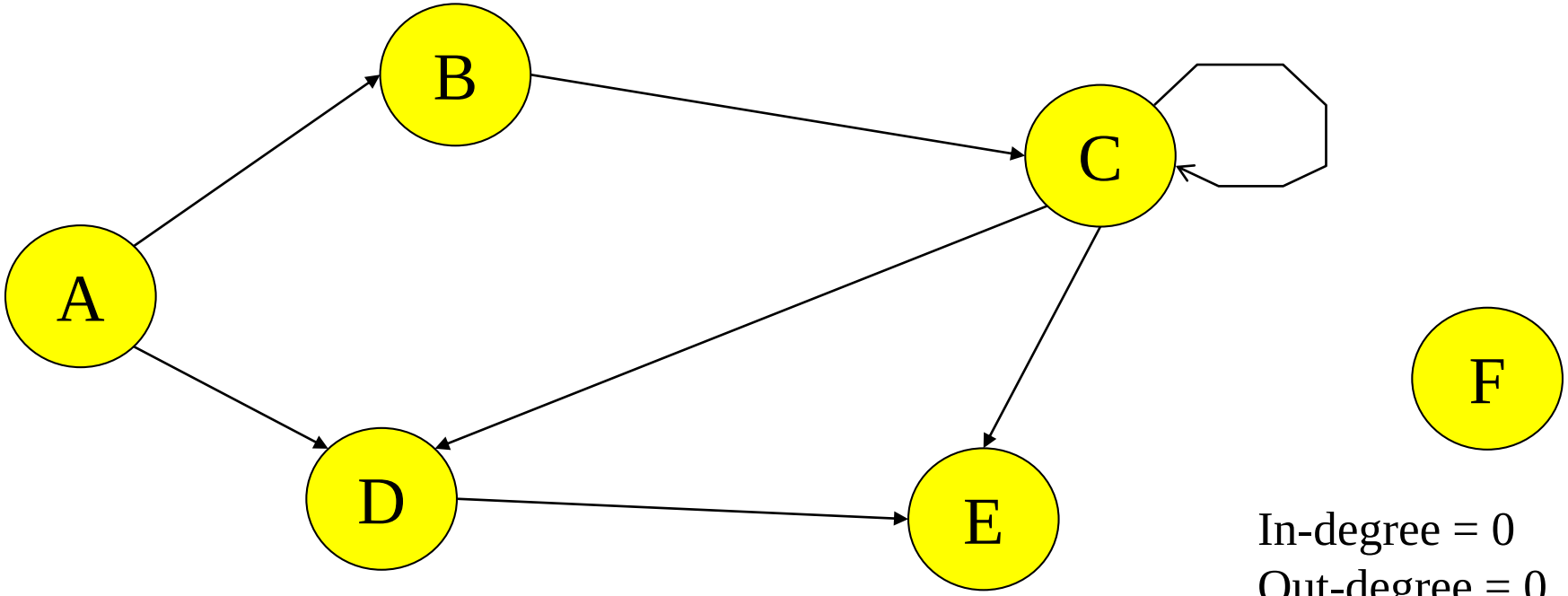
- Vertex u is adjacent *to* vertex v in a directed graph G if (u,v) is an edge in G
  - vertex u is the initial vertex of (u,v)
- Vertex v is adjacent *from* vertex u
  - vertex v is the terminal (or end) vertex of (u,v)
- Degree
  - in-degree is the number of edges with the vertex as the terminal vertex
  - out-degree is the number of edges with the vertex as the initial vertex



u and v are adjacent
v and w are not adjacent

Directed Terminology

B adjacent to C and C adjacent from B



In-degree = 2
Out-degree = 1

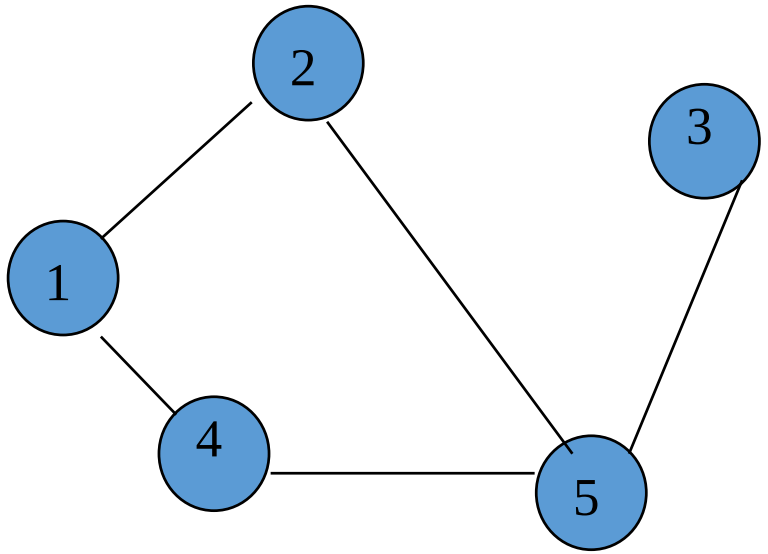
In-degree = 0
Out-degree = 0

Graph Representations

- Space and time are analyzed in terms of:
  - Number of vertices = $|V|$ and
  - Number of edges = $|E|$
- There are at least two ways of representing graphs:
  - The *adjacency matrix* representation
  - The *adjacency list* representation

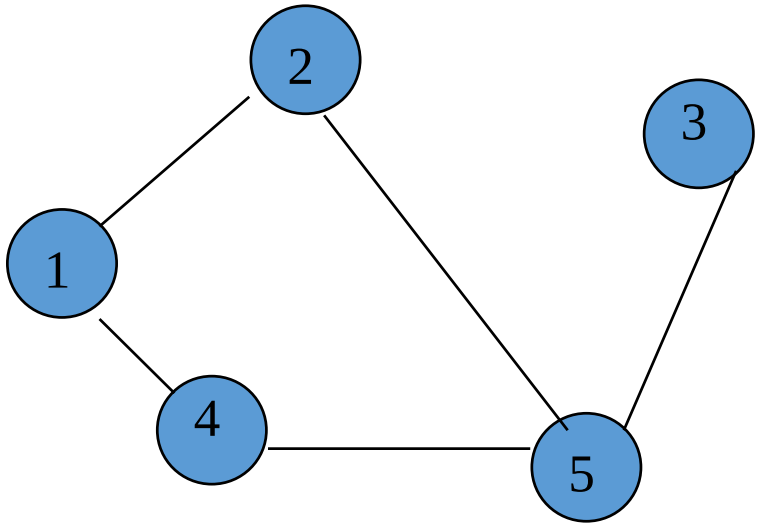
Adjacency Matrix

- 0/1 n x n matrix, where n = # of vertices
- A(i,j) = 1 iff (i,j) is an edge



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

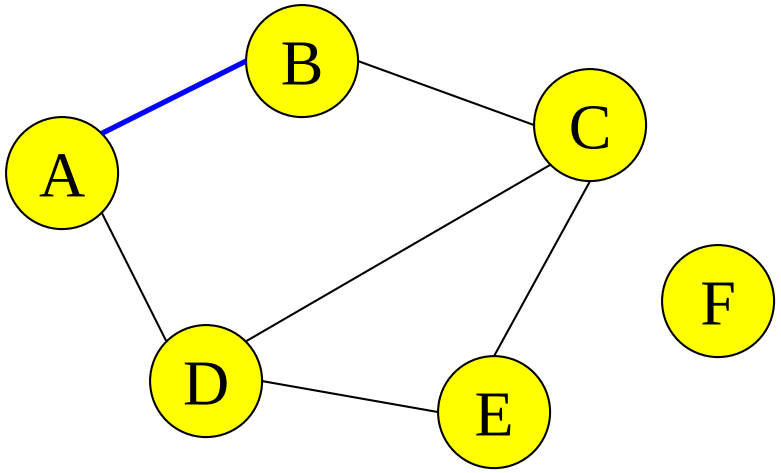
Adjacency Matrix Properties



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

- Diagonal entries are zero.
- Adjacency matrix of an undirected graph is symmetric.
  - A(i,j) = A(j,i) for all i and j.

Adjacency Matrix

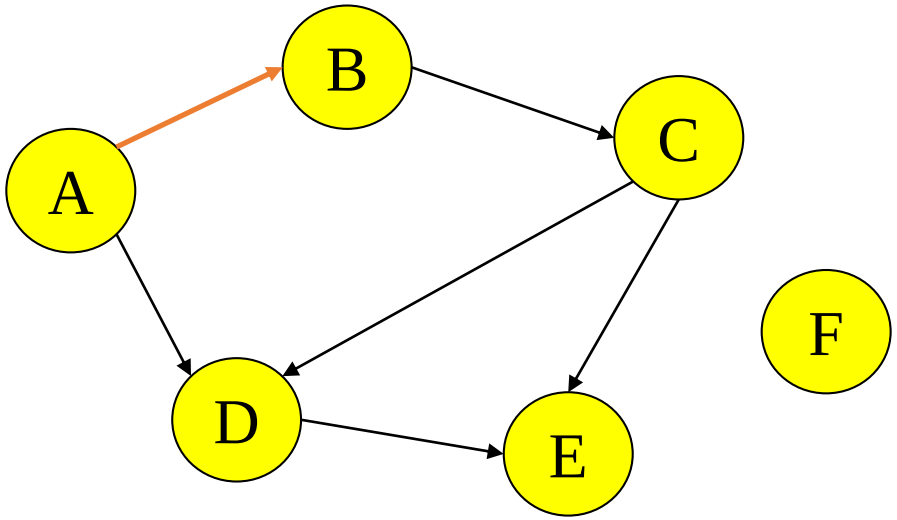


$$M(v, w) = \begin{cases} 1 \text{ if } [v, w] \text{ is in E} \\ 0 \text{ otherwise} \end{cases}$$

	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	0	0	0
C	0	1	0	1	1	0
D	1	0	1	0	1	0
E	0	0	1	1	0	0
F	0	0	0	0	0	0

Space = $|V|^2$

Adjacency Matrix for a Digraph

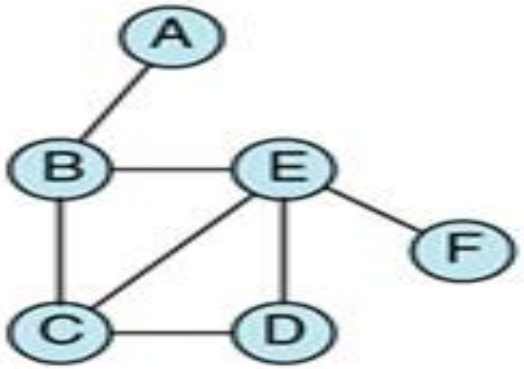


$$M(v, w) = \begin{cases} 1 \text{ if } (v, w) \text{ is in E} \\ 0 \text{ otherwise} \end{cases}$$

	A	B	C	D	E	F
A	0	1	0	1	0	0
B	0	0	1	0	0	0
C	0	0	0	1	1	0
D	0	0	0	0	1	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

Space = $|V|^2$

Adjacency Matrix Example



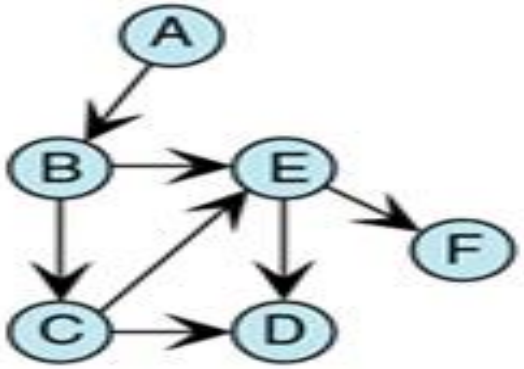
A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency matrix

(a) Adjacency matrix for non-directed graph



A
B
C
D
E
F

Vertex vector

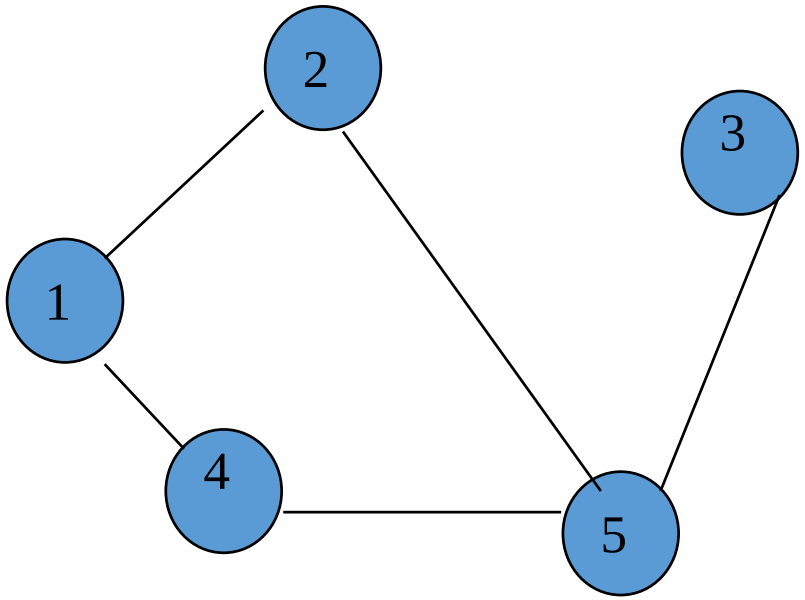
	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Adjacency matrix

(a) Adjacency matrix for directed graph

Adjacency Lists

- Adjacency list for vertex i is a linear list of vertices adjacent from vertex i.
- An array of n adjacency lists.



aList[1] = (2,4)

aList[2] = (1,5)

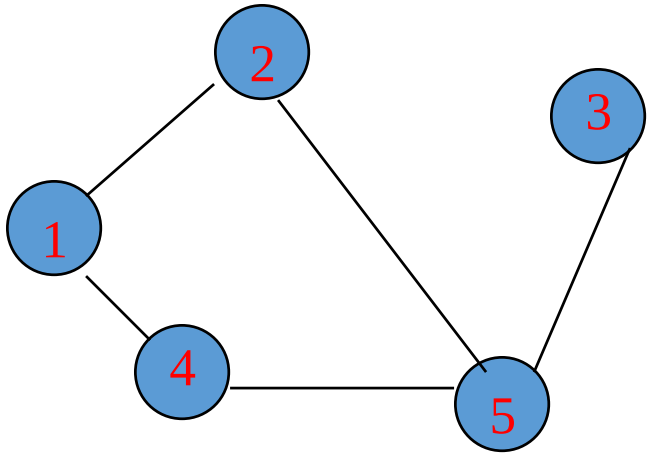
aList[3] = (5)

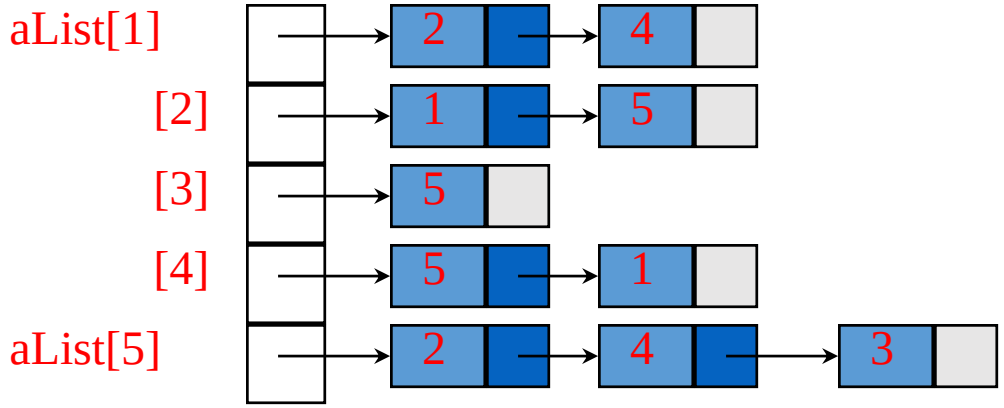
aList[4] = (5,1)

aList[5] = (2,4,3)

Linked Adjacency Lists

- Each adjacency list is a chain.





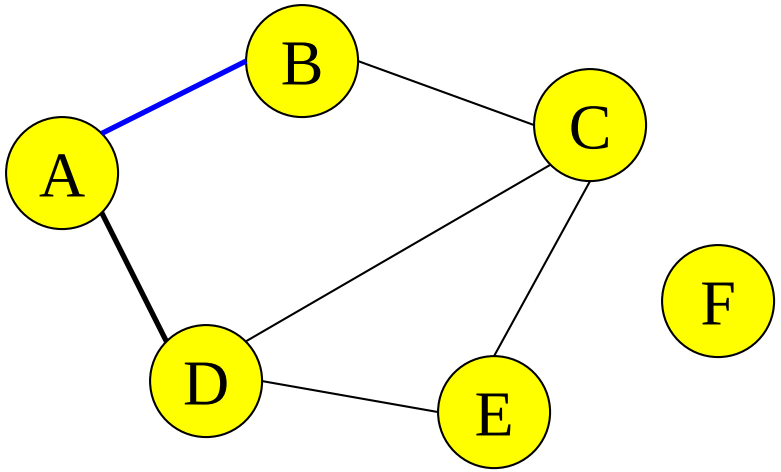
Array Length = n

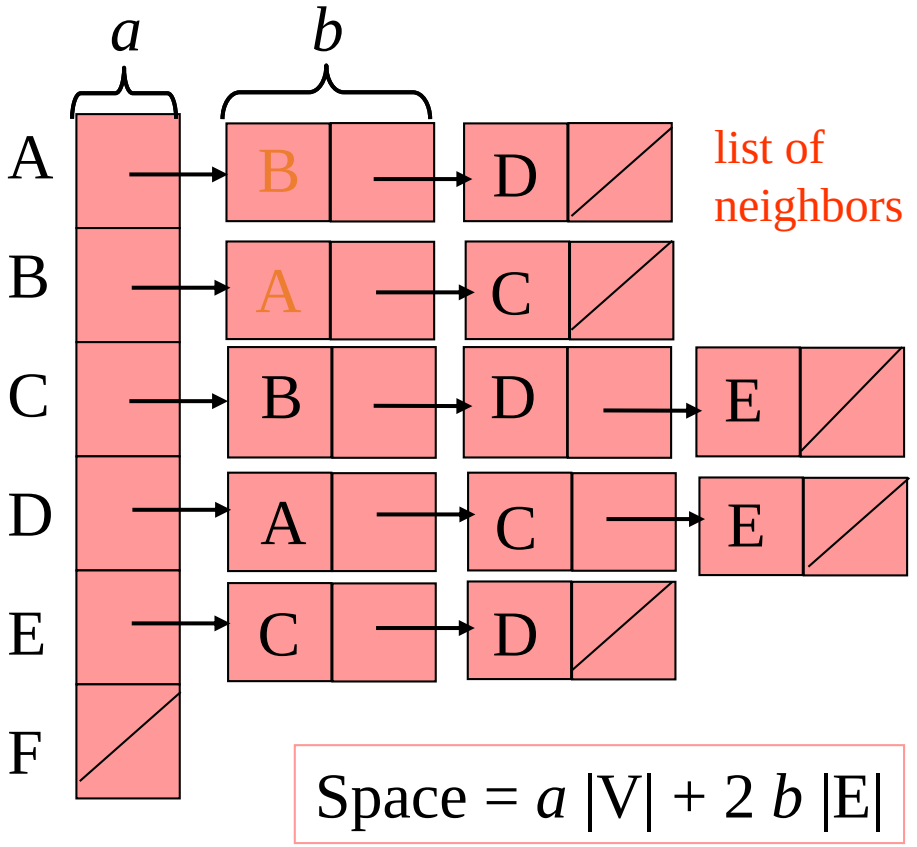
# of chain nodes = 2e (undirected graph)

# of chain nodes = e (digraph)

Adjacency List

For each v in V, $L(v)$ = list of w such that $[v, w]$ is in E

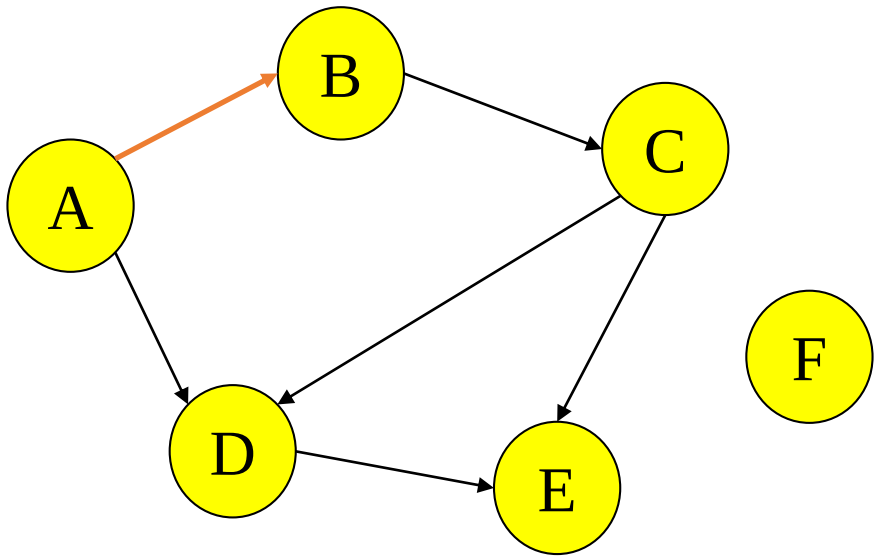


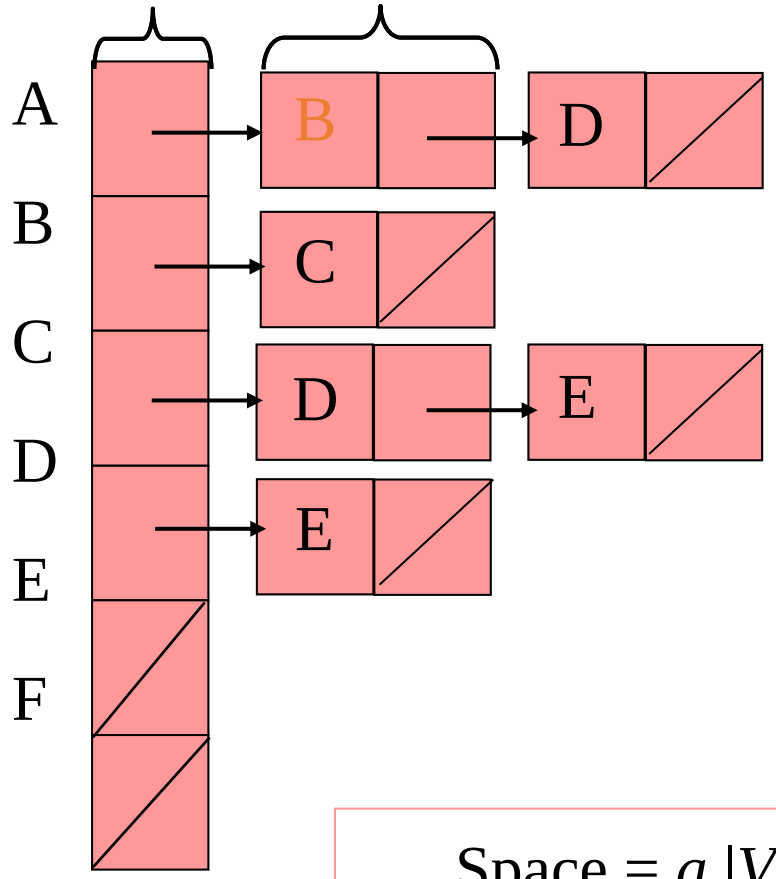


Space = a |V| + 2 b |E|

Adjacency List for a Digraph

For each v in V, $L(v)$ = list of w such that (v, w) is in E





A is a source
E is a sink
F is disconnected from the rest

Space = $a\,|V| + b\,|E|$

Basic Terminology in a Graph

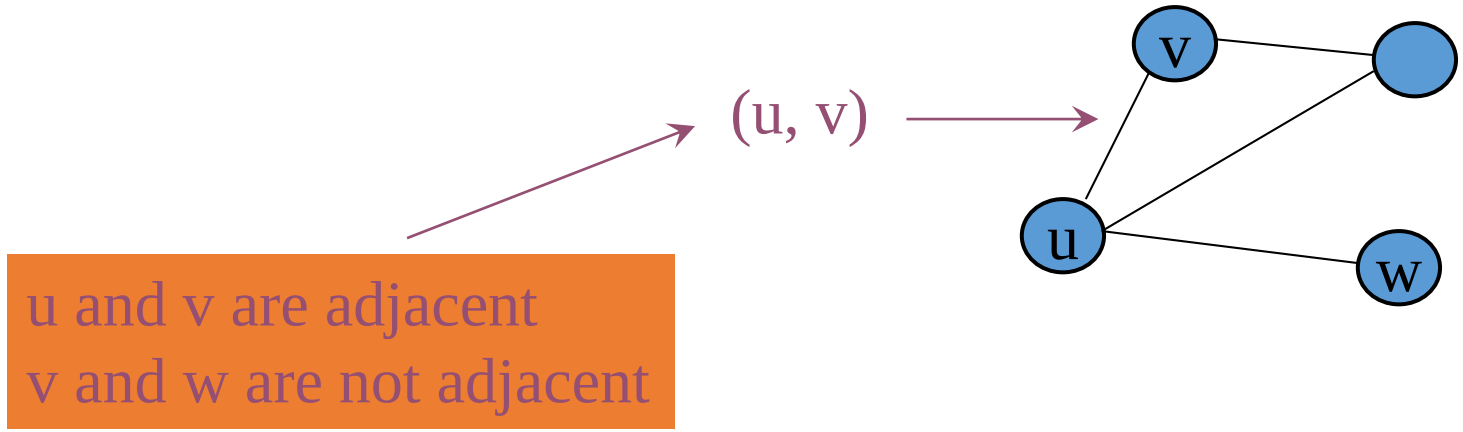
- **Vertex**: An individual data element of a graph is called Vertex.
- **Edge**: An edge is a connecting link between two vertices. An Edge is also known as Arc.
- **Mixed Graph**: A graph with undirected and directed edges is said to be a mixed graph.
- **Origin**: If an edge is directed, its first endpoint is said to be the origin of it.
- **Destination**: If an edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of the edge.
- **Adjacency**: Two node or vertices are adjacent if they are connected through an edge.
- **Path**: The Path represents a sequence of edges between the two vertices.
- **Degree**: The total number of edges connected to a vertex is said to be the degree of that vertex.
- **In-Degree**: In-degree of a vertex is the number of edges which are coming into the vertex.
- **Out-Degree**: Out-degree of a vertex is the number of edges which are going out from the vertex.

Basic Terminology in a Graph

- **Simple Graph**: A graph is said to be simple if there are no parallel and self-loop edges.
- **Directed acyclic graph (DAG)**: A directed acyclic graph (DAG) is a graph that is directed and without cycles connecting the other edges. This means that it is impossible to traverse the entire graph starting at one edge.
- **Weighted Graph**: A weighted graph is a graph in which a number (known as the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem.
- **Complete Graph**: A complete graph is a graph in which each pair of vertices is joined by an edge. A complete graph contains all possible edges.
- **Connected graph**: A connected graph is an undirected graph in which every unordered pair of vertices in the graph is connected. Otherwise, it is called a disconnected graph.
- **Minimum Spanning Tree (MST)**: A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted (un)directed graph that connects all the vertices, without any cycles and with the minimum possible total edge weight.

Adjacent

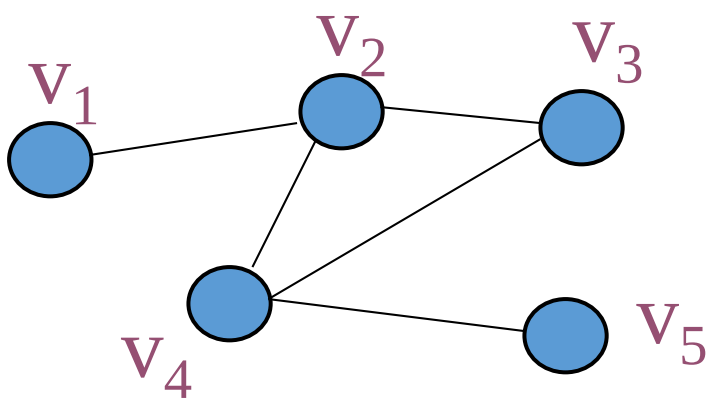
- Two nodes u and v are said to be adjacent if (u, v) $\in$ E



Path and simple path

- A path from v_1 to v_k is a sequence of nodes v_1, v_2, …, v_k that are connected by edges (v_1, v_2), (v_2, v_3), …, (v_{k-1}, v_k)
- A path is called a simple path if every node appears at most once.

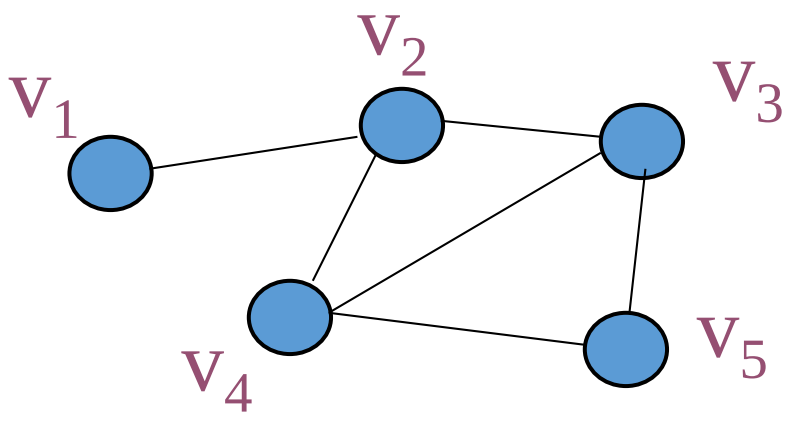
- v_2, v_3, v_4, v_2, v_1 is a path
- v_2, v_3, v_4, v_5 is a path, also it is a simple path



Cycle and simple cycle

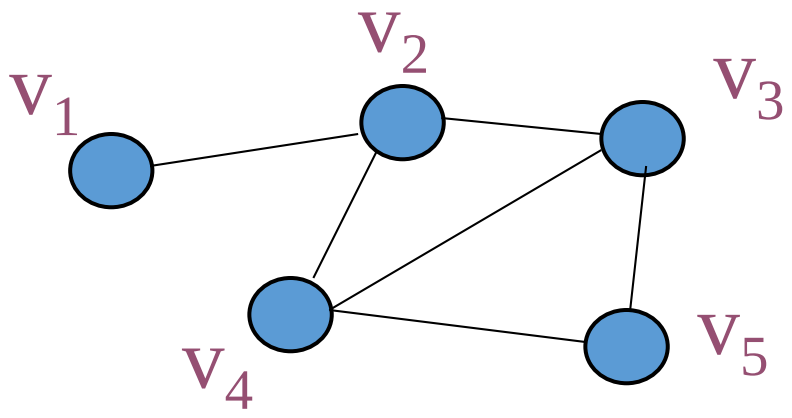
- A cycle is a path that begins and ends at the same node
- A simple cycle is a cycle if every node appears at most once, except for the first and the last nodes

- $v_2, v_3, v_4, v_5, v_3, v_2$ is a cycle
- v_2, v_3, v_4, v_2 is a cycle, it is also a simple cycle



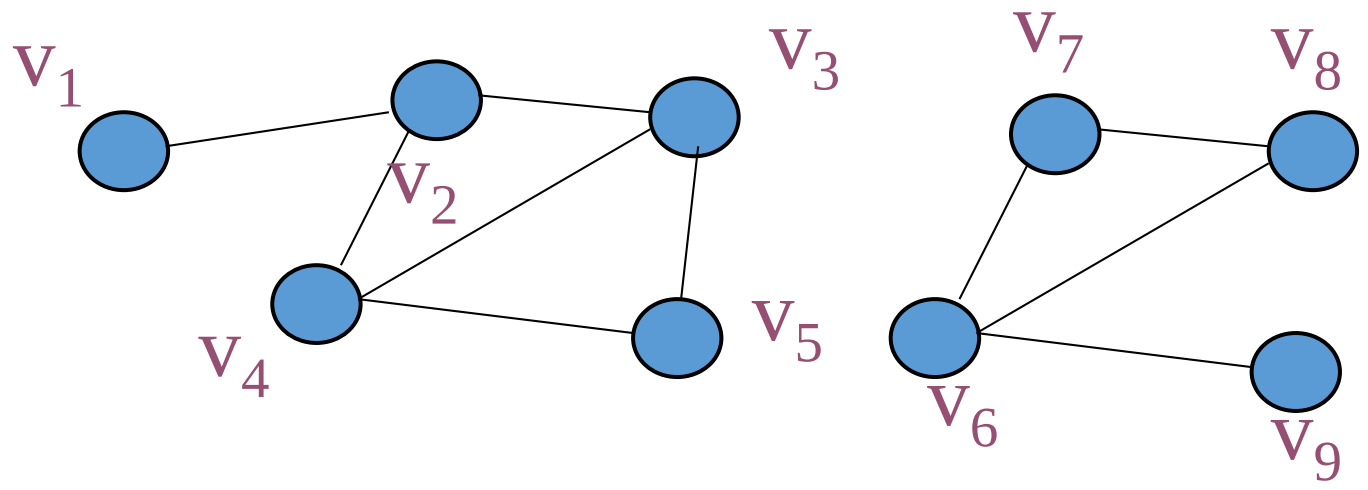
Connected graph

- A graph G is connected if there exists path between every pair of distinct nodes; otherwise, it is disconnected



This is a connected graph because there exists path between every pair of nodes

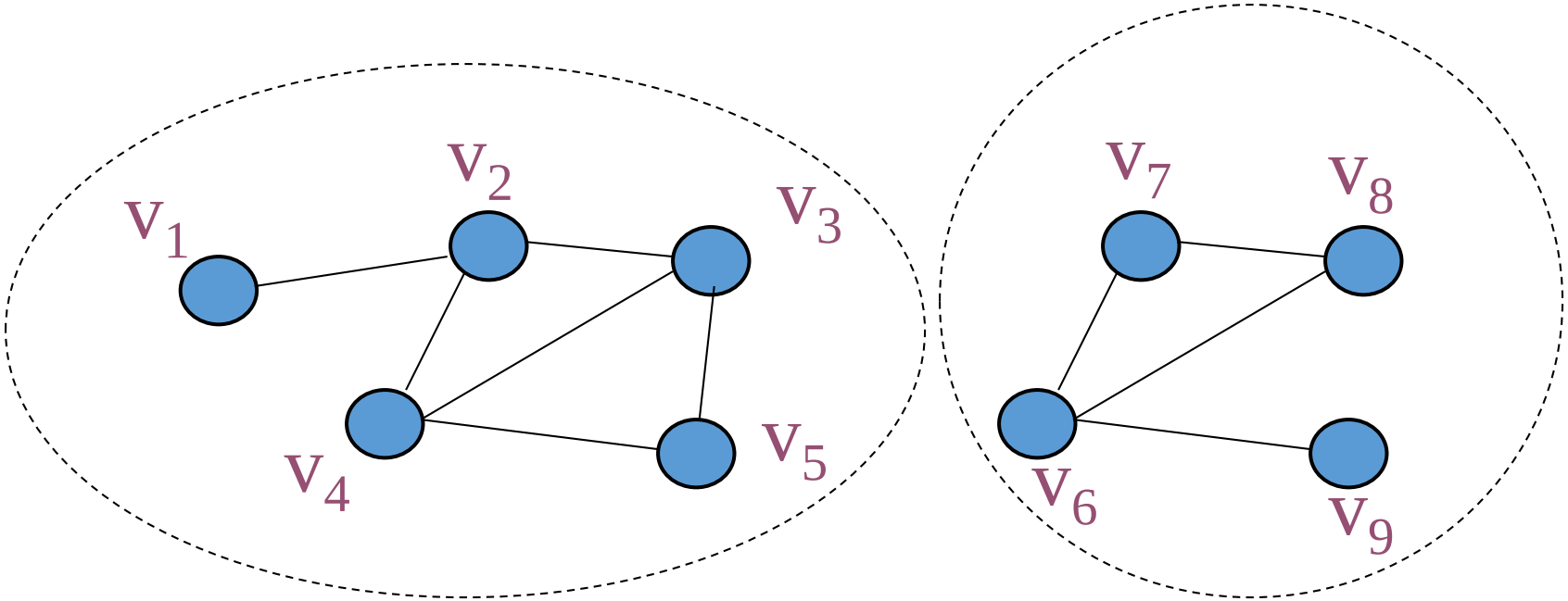
Example of disconnected graph



This is a disconnected graph because there does not exist path between some pair of nodes, says, v_1 and v_7

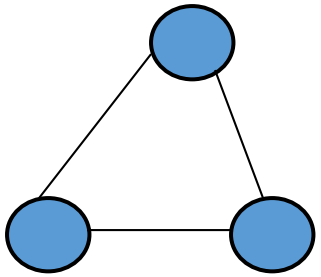
Connected component

- If a graph is disconnect, it can be partitioned into a number of graphs such that each of them is connected. Each such graph is called a connected component.

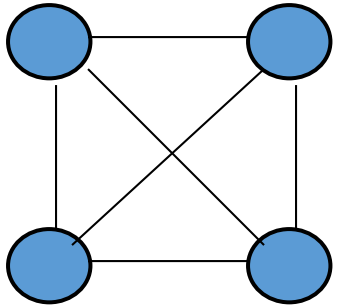


Complete graph

- A graph is complete if each pair of distinct nodes has an edge



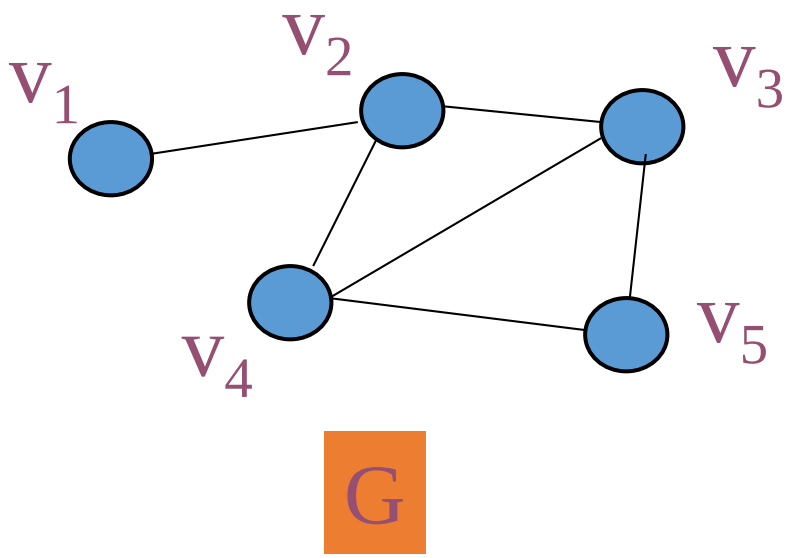
Complete graph with 3 nodes

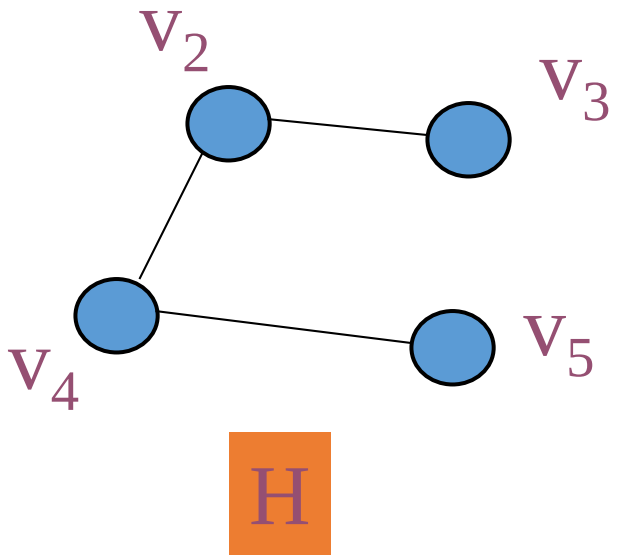


Complete graph with 4 nodes

Subgraph

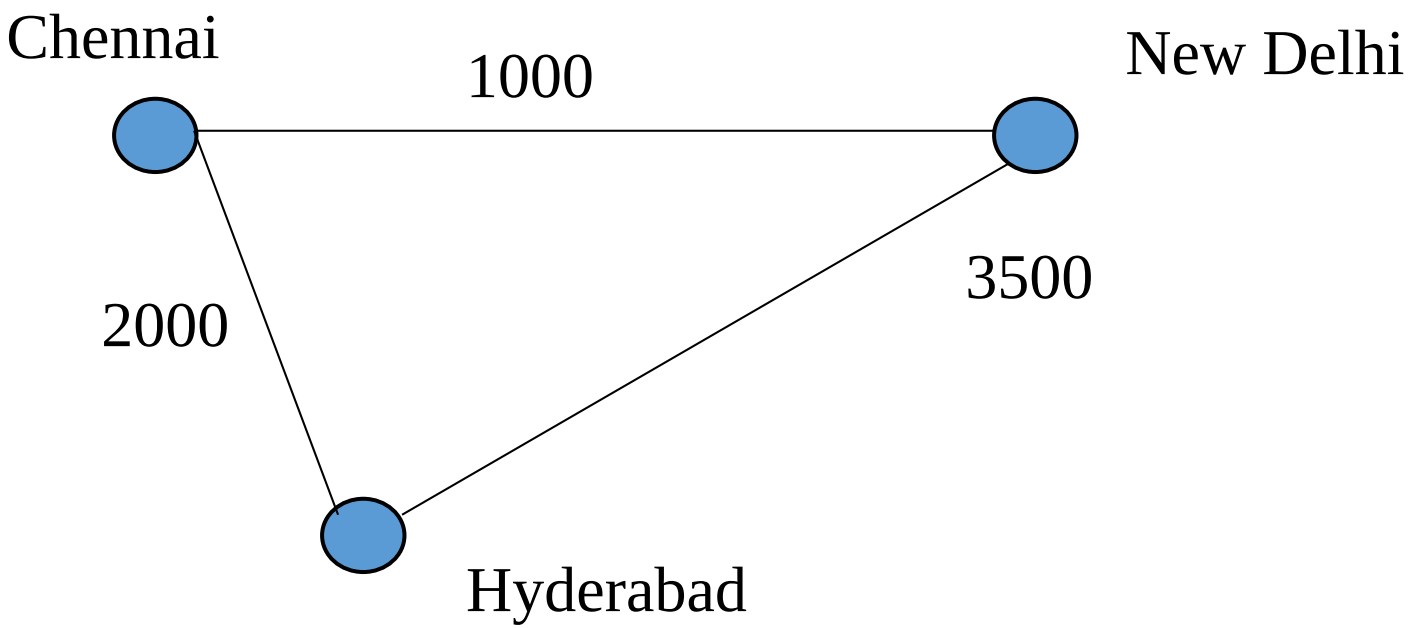
- A subgraph of a graph G =(V, E) is a graph H = (U, F) such that $U \subseteq V$ and $F \subseteq E$.





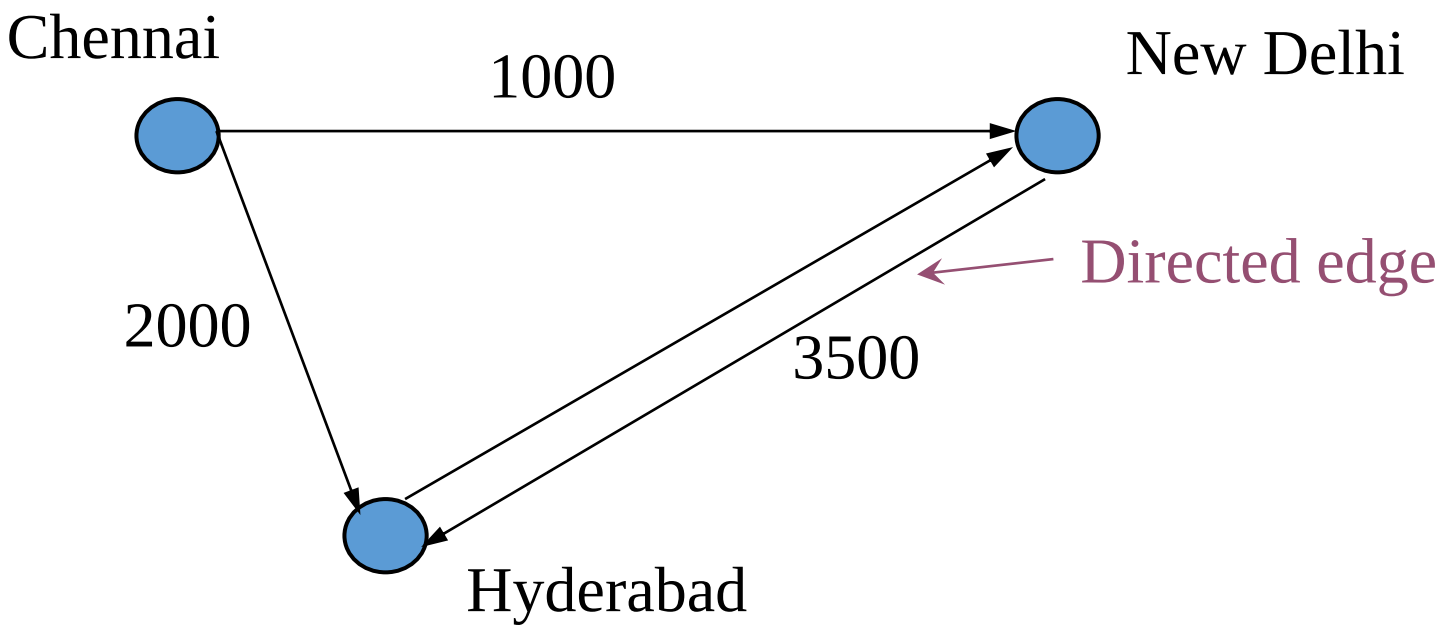
Weighted graph

- If each edge in G is assigned a weight, it is called a weighted graph

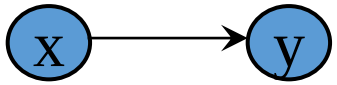


Directed graph (digraph)

- All previous graphs are undirected graph
- If each edge in E has a direction, it is called a directed edge
- A directed graph is a graph where every edges is a directed edge



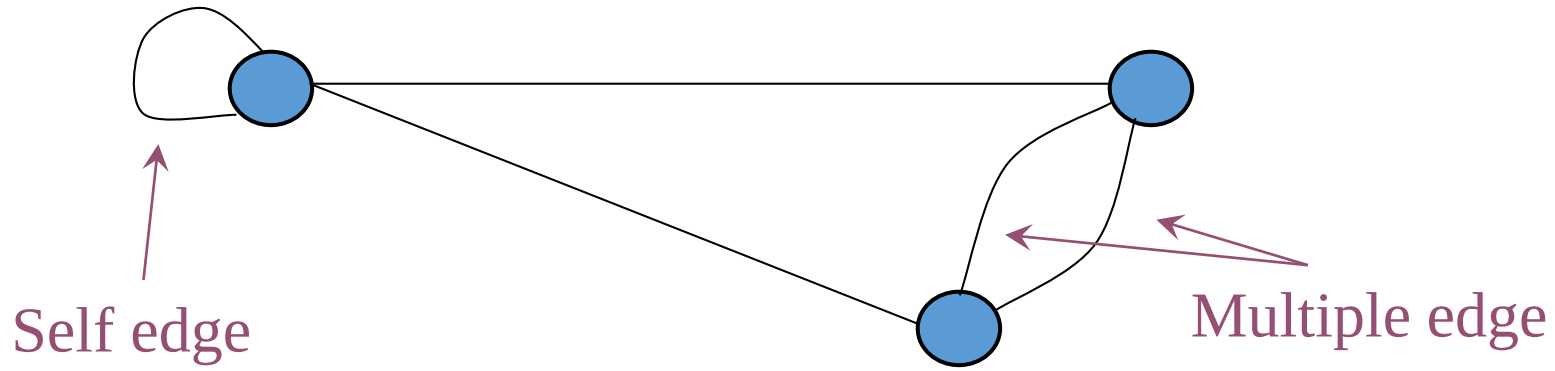
More on directed graph



- If (x, y) is a directed edge, we say
  - y is adjacent to x
  - y is successor of x
  - x is predecessor of y
- In a directed graph, directed path, directed cycle can be defined similarly

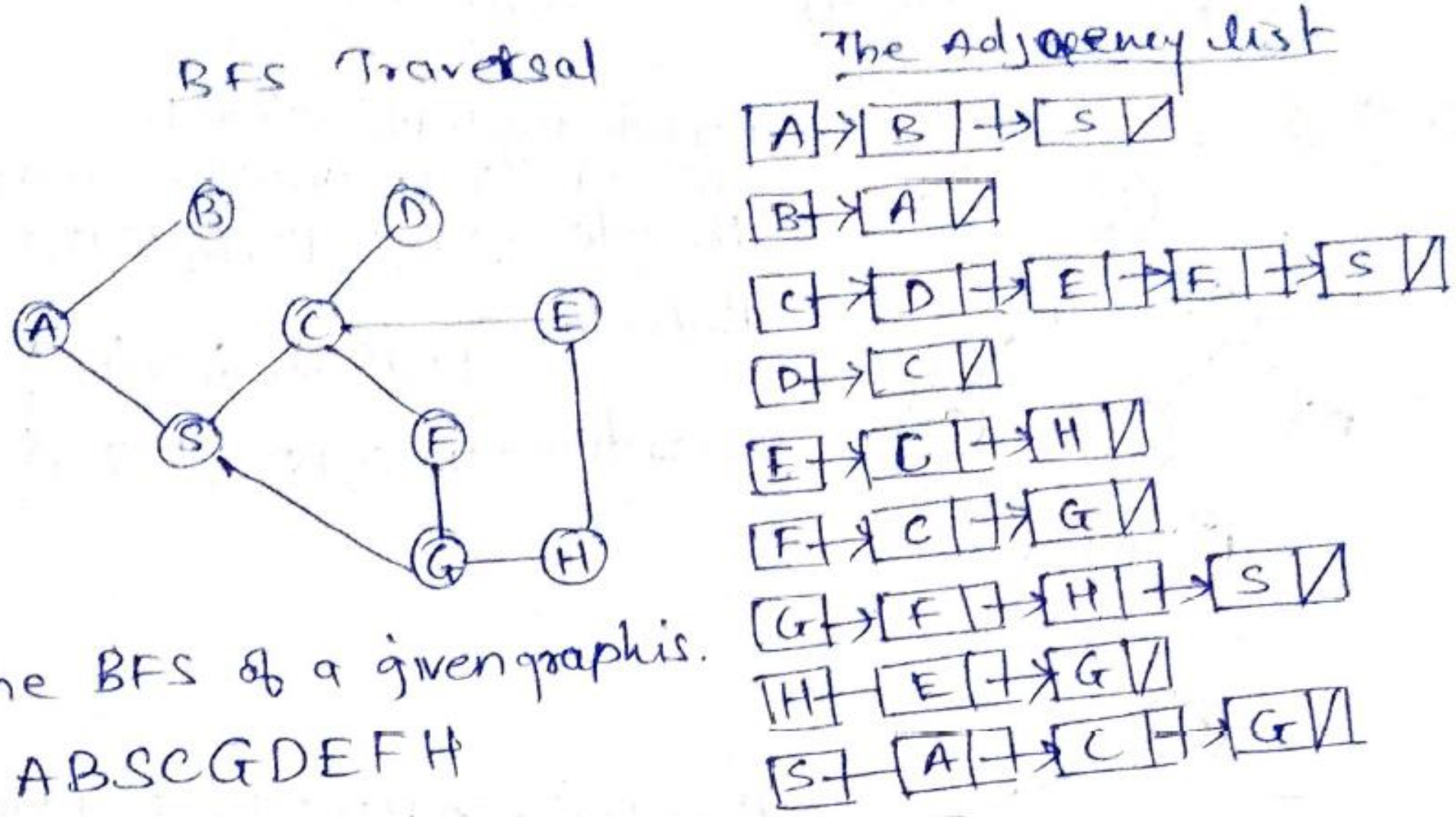
Multigraph

- A graph cannot have duplicate edges.
- Multigraph allows multiple edges and self edge (or loop).



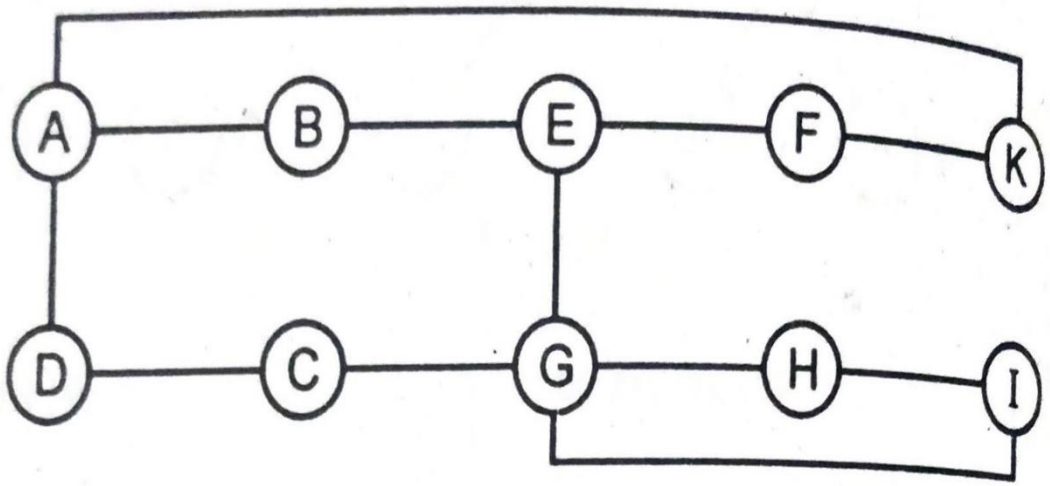
Graph Traversing Techniques

Please refer the class notebook for a more detailed explanation of the process and solutions to the other problems.



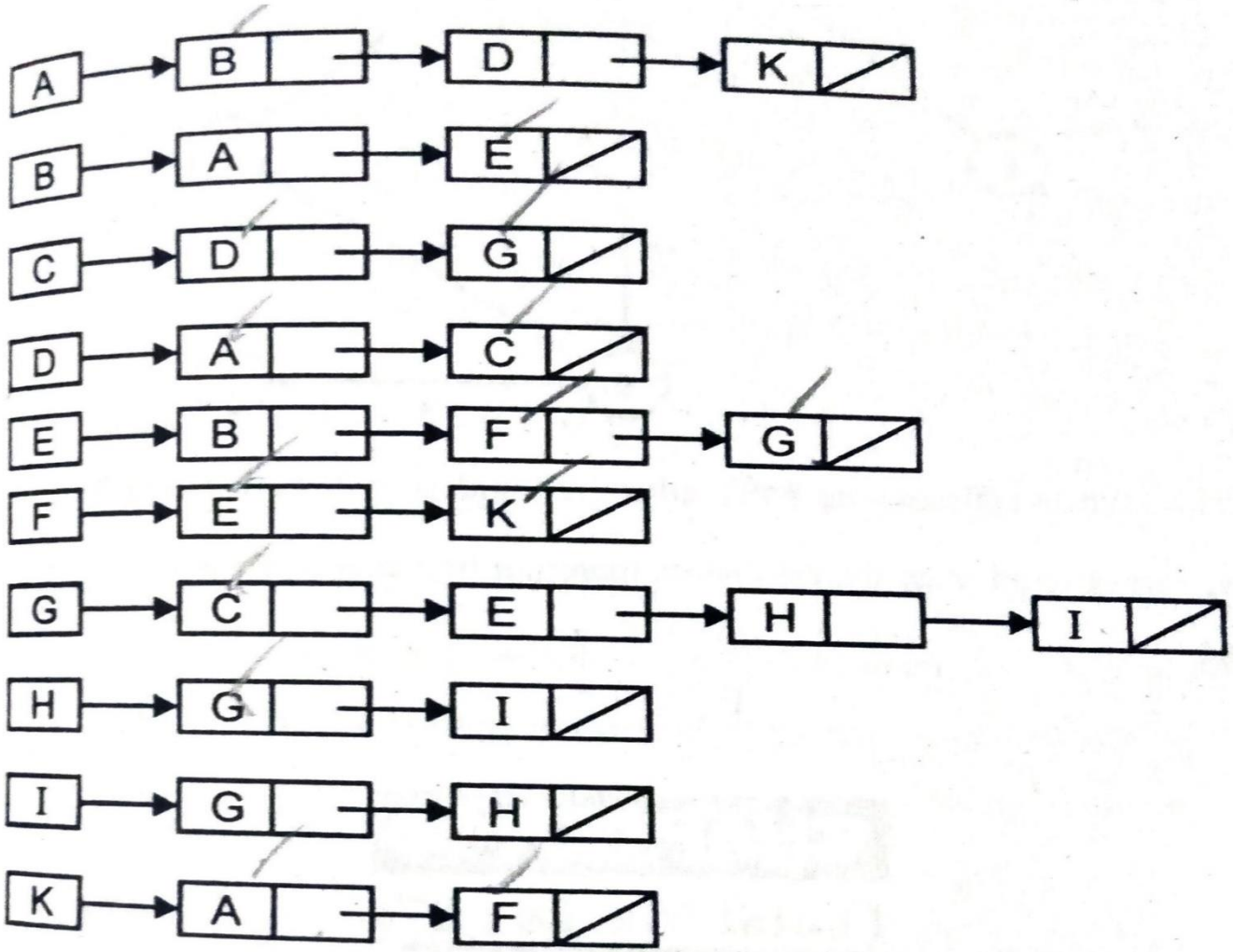
Graph Traversing Techniques

Find out the DFS treversal of the following graph starting at Node A



Please refer the class notebook for a more detailed explanation of the process and solutions to the other problems.

Starting node is A For a given graph, adjacency list is



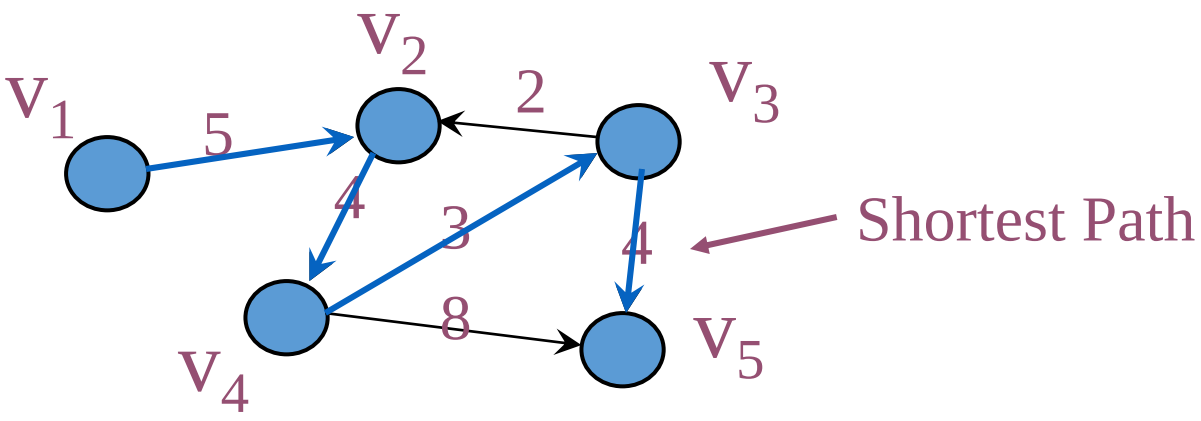
The DFS for a given graph is ABEFKGCDHI.

Shortest path

- Consider a weighted directed graph
  - Each node x represents a city x
  - Each edge (x, y) has a number which represent the cost of traveling from city x to city y
- Problem: find the minimum cost to travel from city x to city y
- Solution: find the shortest path from x to y

Formal definition of shortest path

- Given a weighted directed graph G.
- Let P be a path of G from x to y.
- cost(P) = $\sum_{e \in P}$weight(e)
- The shortest path is a path P which minimizes cost(P)



Dijkstra's Algorithm

- Classic algorithm for solving shortest path in weighted graphs (with *only positive* edge weights)
- Similar to breadth-first search, but uses a priority queue instead of a FIFO queue:
  - Always select (expand) the vertex that has a lowest-cost path to the start vertex
  - a kind of "greedy" algorithm
- Correctly handles the case where the lowest-cost (shortest) path to a vertex is not the one with fewest edges

- Consider a graph G, each edge (u, v) has a weight w(u, v) > 0.
- Suppose we want to find the shortest path starting from v_1 to any node v_i
- Let VS be a subset of nodes in G
- Let cost[v_i] be the weight of the shortest path from v_1 to v_i that passes through nodes in VS only.

Dijkstra’s Algorithm

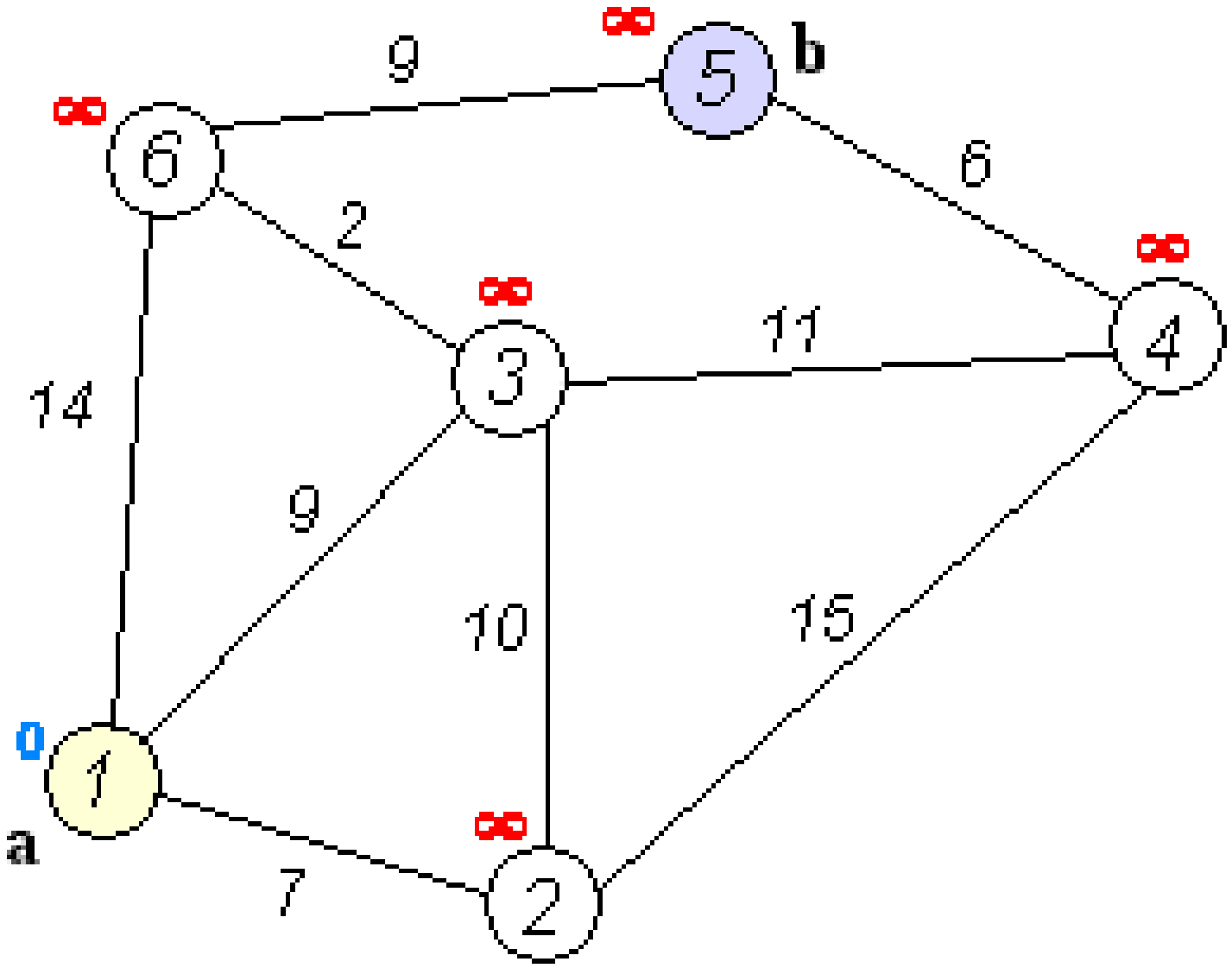
- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
- The process continues until all the nodes in the graph have been added to the path.
- This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Dijkstra’s Algorithm

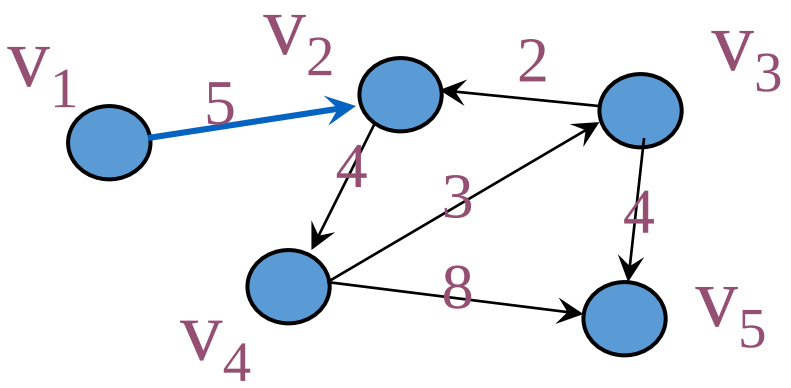
The following is the step that we will follow to implement Dijkstra's Algorithm:

- **Step 1:** First, we will mark the source node with a current distance of 0 and set the rest of the nodes to INFINITY.
- **Step 2:** We will then set the unvisited node with the smallest current distance as the current node, suppose X.
- **Step 3:** For each neighbor N of the current node X: We will then add the current distance of X with the weight of the edge joining X-N. If it is smaller than the current distance of N, set it as the new current distance of N.
- **Step 4:** We will then mark the current node X as visited.
- **Step 5:** We will repeat the process from '**Step 2**' if there is any node unvisited left in the graph.

Example for Dijkstra's Algorithm Simulation

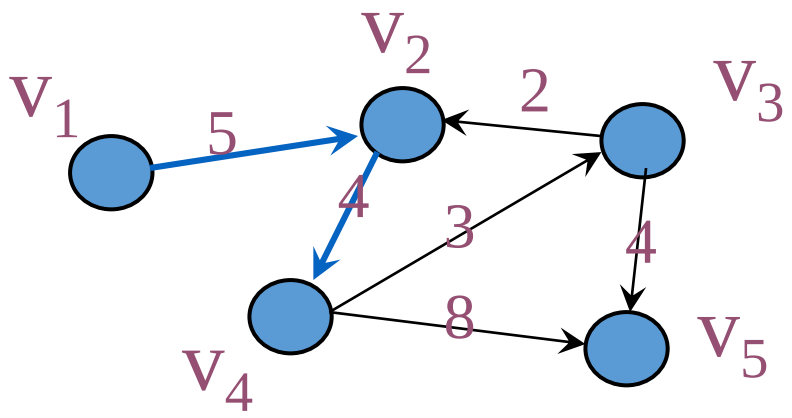


Example for Dijkstra's Algorithm



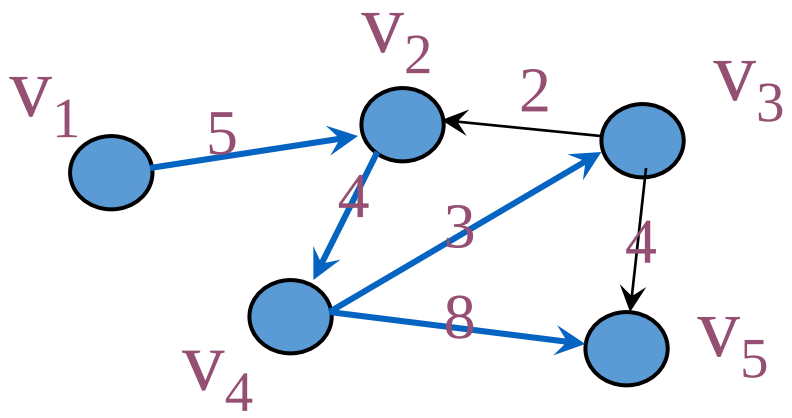
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		**[v_1]**	**0**	**5**	∞	∞	∞

Example for Dijkstra's Algorithm



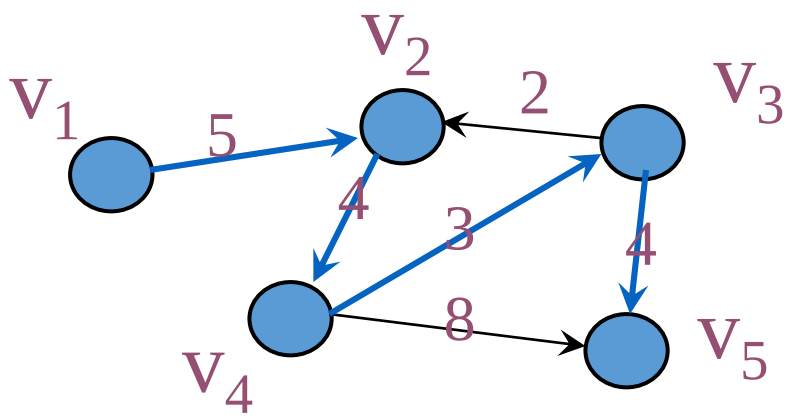
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		**[v_1]**	**0**	**5**	∞	∞	∞
2	**v_2**	**[v_1, v_2]**	**0**	**5**	∞	**9**	∞

Example for Dijkstra's Algorithm



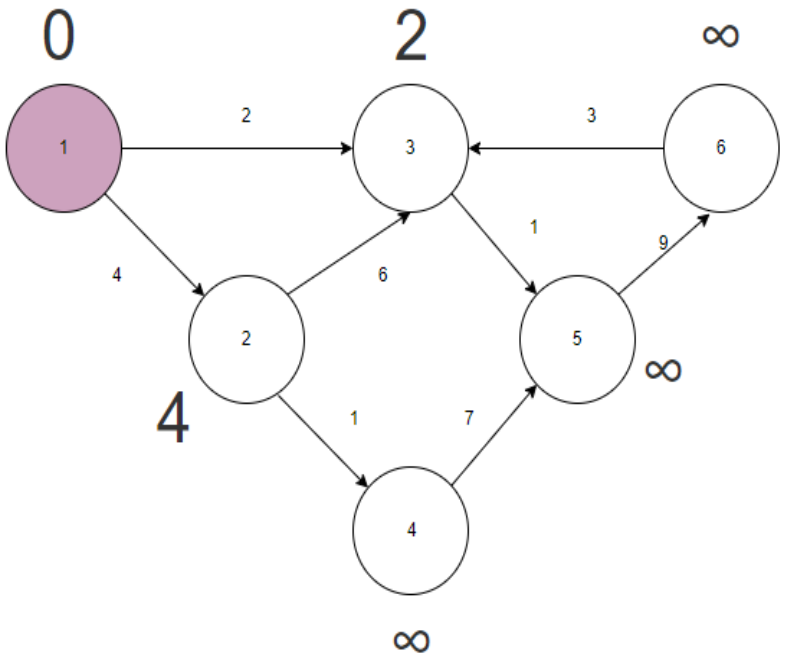
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞
3	v_4	[v_1, v_2, v_4]	0	5	12	9	17

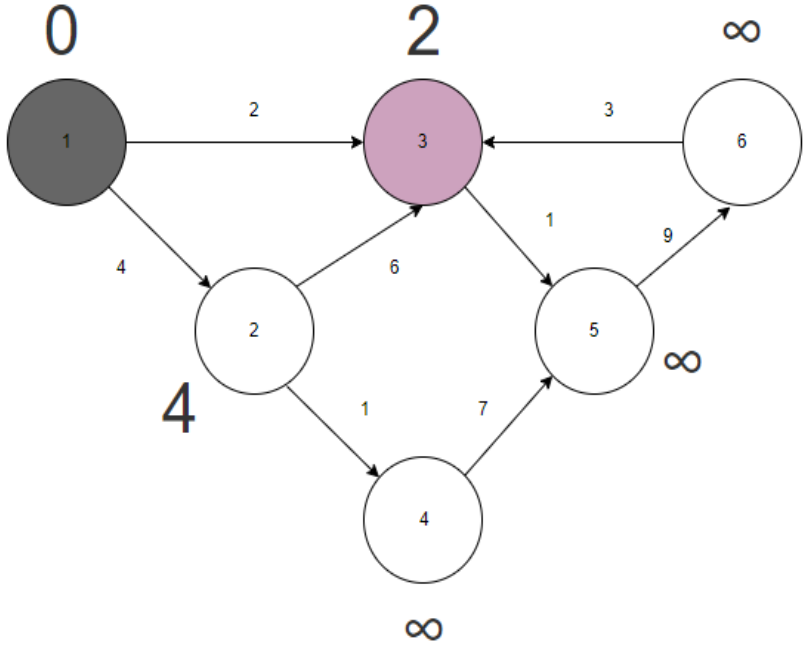
Example for Dijkstra's Algorithm

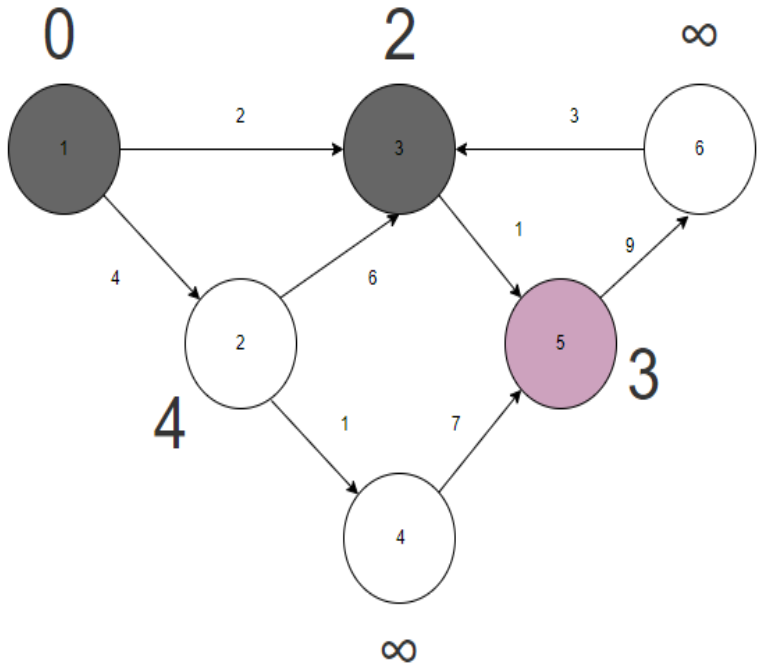


	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		**[v_1]**	**0**	**5**	∞	∞	∞
2	**v_2**	**[v_1, v_2]**	**0**	**5**	∞	**9**	∞
3	**v_4**	**[v_1, v_2, v_4]**	**0**	**5**	**12**	**9**	**17**
4	**v_3**	**[v_1, v_2, v_4, v_3]**	**0**	**5**	**12**	**9**	**16**
5	**v_5**	**[v_1, v_2, v_4, v_3, v_5]**	**0**	**5**	**12**	**9**	**16**

Example for Dijkstra's Algorithm

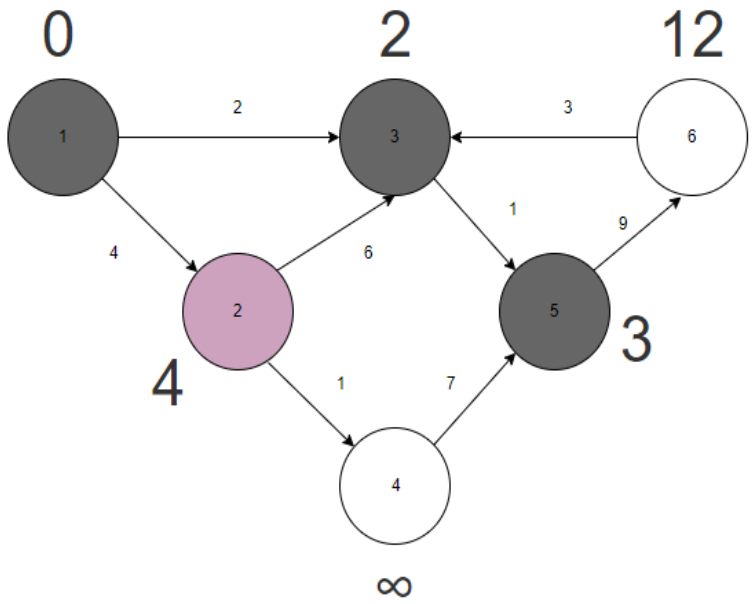


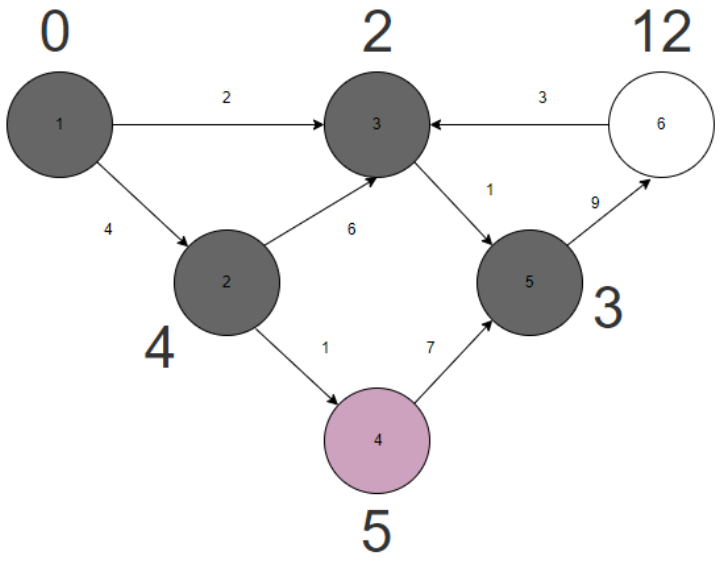


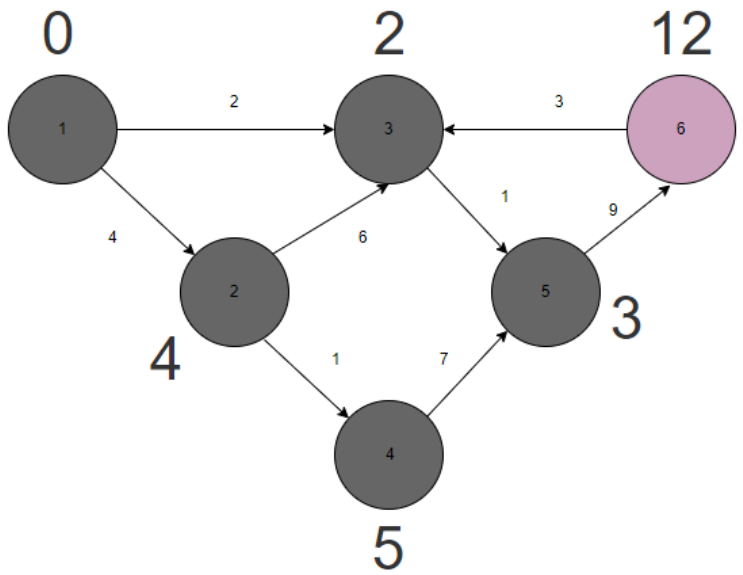


	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]	cost[v_6]
1		[v_1]	0	4	2	∞	∞	∞
2	v_3	[v_1, v_3]	0	4	2	∞	3	∞
3								
4								
5								
6								

Example for Dijkstra's Algorithm

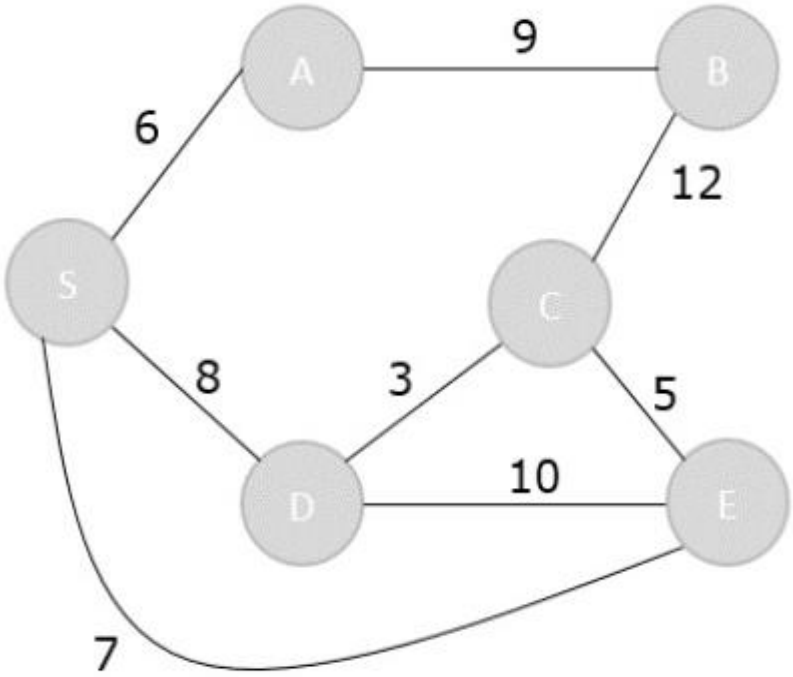






	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]	cost[v_6]
1		[v_1]	0	4	2	∞	∞	∞
2	v_3	[v_1, v_3]	0	4	2	∞	3	∞
3	v_5	[v_1, v_3, v_5]	0	4	2	∞	3	12
4	v_2	[v_1, v_3, v_5, v_2]	0	4	2	5	3	12
5	v_4	[v_1, v_3, v_5, v_2, v_4]	0	4	2	5	3	12
6	v_6	[v_1, v_3, v_5, v_2, v_4]	0	4	2	5	3	12

Example for Dijkstra's Algorithm



	v	VS	cost[S]	cost[A]	cost[B]	cost[C]	cost[D]	cost[E]
1		[S]	0	∞	∞	∞	∞	∞
2	A	[S, A]	0	6	∞	∞	8	7
3	E	[S, A, E]	0	6	15	∞	8	7
4	D	[S, A, E, D]	0	6	15	12	8	7
5	C	[S, A, E, D, C]	0	6	15	11	8	7
6	B	[S, A, E, D, C, B]	0	6	15	11	8	7

Dijkstra's Algorithm

Algorithm shortestPath()

n = number of nodes in the graph;

for i = 1 to n

** cost[v_i] = w(v_1, v_i);**

VS = { v_1 };

for step = 2 to n {

** find the smallest cost[v_i] s.t. v_i is not in VS;**

** include v_i to VS;**

** for (all nodes v_j not in VS) {**

**   if (cost[v_j] > cost[v_i] + w(v_i, v_j))**

**   cost[v_j] = cost[v_i] + w(v_i, v_j);**

** }**

}

Dijkstra's Algorithm

```
function Dijkstra(Graph, source):
      dist[source]  = 0                  // Distance from source to source is set to 0
      for each vertex v in Graph           // Initializations
         if v ≠ source
            dist[v]  = infinity          // Unknown distance function from source to each node set to infinity
         add v to Q                      // All nodes initially in Q

    while Q is not empty                 // The main loop
       v = vertex in Q with min dist[v]  // In the first run-through, this vertex is the source node
       remove v from Q

       for each neighbor u of v          // where neighbour u has not yet been removed from Q.
          alt = dist[v] + length(v, u)
          if alt < dist[u]              // A shorter path to u has been found
             dist[u]  = alt            // Update distance of u  // In general time complexity is O(V + E(log V)
   return dist[]                            // The time complexity is O(V²) being V the number of vertexes.
 end function                               //And the space complexity O(V).
```

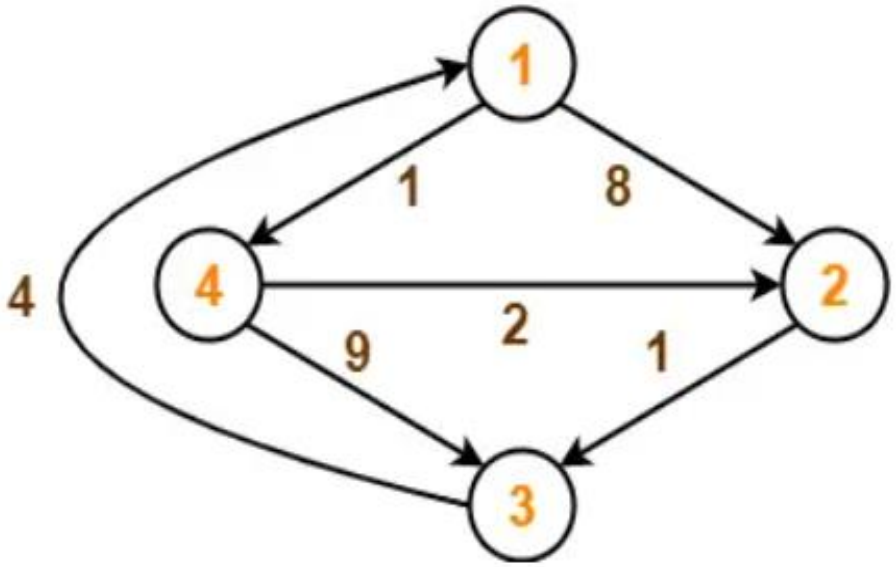
Floyd Warshall Algorithm

- **Floyd Warshall is a dynamic programming algorithm used to solve all pair shortest path problems.**
- **Dynamic Programming is an approach used in data structure and algorithms for solving problems efficiently, this approach is known for providing the most optimized result.**
- **Dynamic programming solves a class of problems using overlapping sub problems, memorization, and optimal substructure property.**
- **Floyd Warshall algorithm is the shortest path algorithm, similar to Dijkstra's algorithm and Bellman ford's algorithm.**
- **The only difference is Dijkstra's and Bellman Ford's algorithms are single source shortest path algorithms, whereas Floyd Warshall is all pair shortest path algorithm, it can compute the shortest path between every pair of vertices in the graph.**
- **Floyd Warshall Algorithm is unique compared to other algorithms because it can handle negative edge weights.**
- **The algorithm can also work with graphs that have cycles.**
- **Furthermore, it is an efficient algorithm that can solve problems involving a large number of nodes in the graph.**

Floyd Warshall Algorithm

1. Initialize a distance matrix D wherein D[i][j] represents the shortest distance between vertex i and vertex j.
2. Set the diagonal entries of the matrix to 0, & all other entries to infinity (∞).
3. For every area (u, v) inside the graph, replace the gap matrix to mirror the weight of the brink: D[u][v] = weight(u,v).
4. For every vertex in the graph, all pairs of vertices (i,j) and check if the path from i to j through k is shorter than the current best path.
   i. If it is, update the matrix: D[i][j] = min(D[i][j], D[i][k] D[k][j]).
5. After all iterations, the matrix D will contain the shortest course distances between all pairs of vertices.

Floyd Warshall Algorithm Example



$D_0 =$

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	∞	0	∞
4	∞	2	9	0

Step-01: Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.

In the given graph, there are neither self edges nor parallel edges.

Step-02: Write the initial distance matrix.

It represents the distance between every pair of vertices in the form of given weights.

For diagonal elements (representing self-loops), distance value = 0.

For vertices having a direct edge between them, distance value = weight of that edge.

For vertices having no direct edge between them, distance value = ∞.

Floyd Warshall Algorithm Example

$D_1 =$

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	9	0

$D_2 =$

	1	2	3	4
1	0	8	9	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	3	0

$D_3 =$

	1	2	3	4
1	0	8	9	1
2	5	0	1	6
3	4	12	0	5
4	7	2	3	0

Step 3: Minimize the shortest paths between any 2 pairs in the previous operation.

Step 4: For any 2 vertices (i,j), one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be:

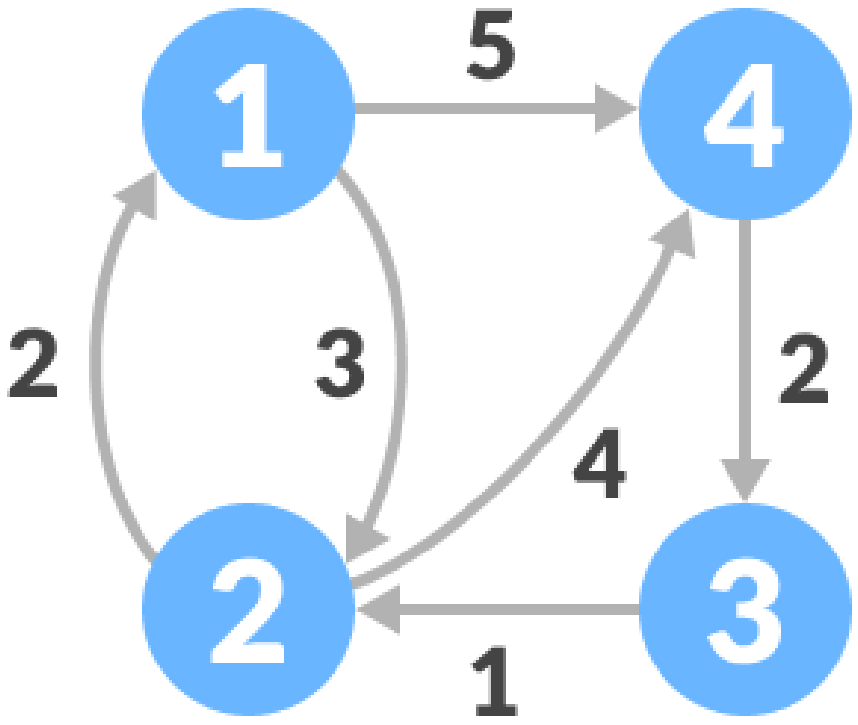
min (dist[i][k] + dist[k][j], dist[i][j]).

$D_4 =$

	1	2	3	4
1	0	3	4	1
2	5	0	1	6
3	4	7	0	5
4	7	2	3	0

dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k,j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

Floyd Warshall Algorithm Example



$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Floyd Warshall Algorithm Example

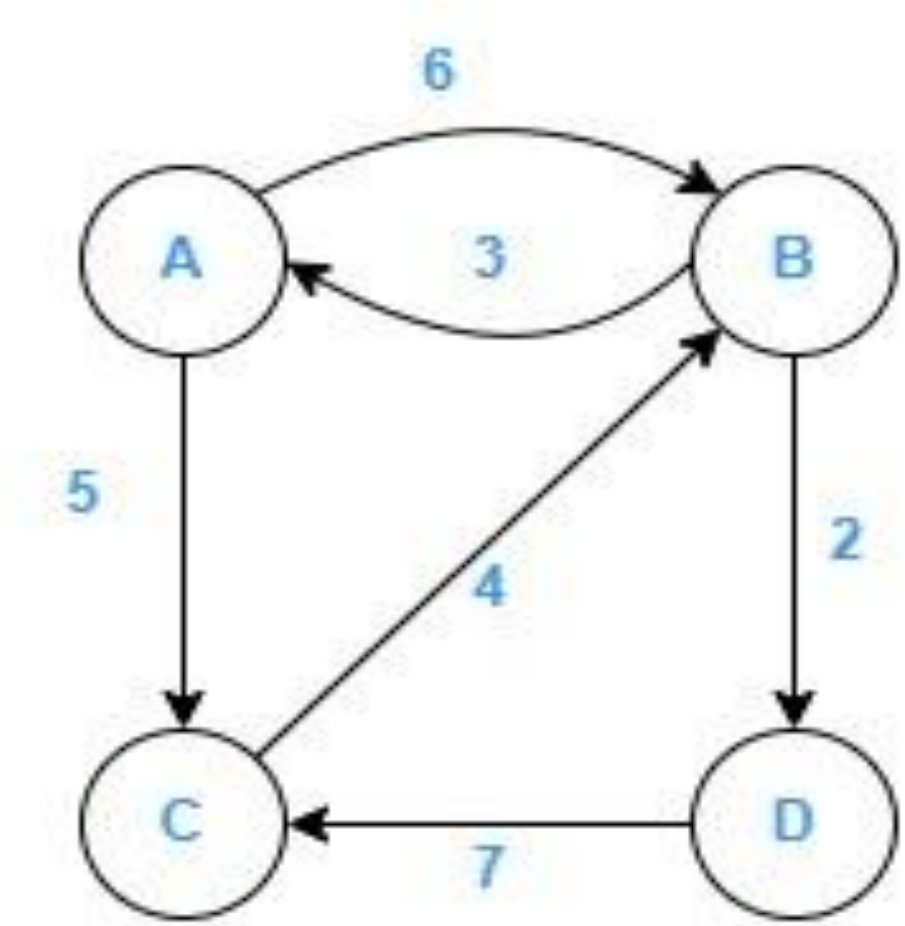
$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Floyd Warshall Algorithm Example



	A	B	C	D
A	**0**	**6**	**5**	∞
B	**3**	**0**	∞	**2**
C	∞	**4**	**0**	∞
D	∞	∞	**7**	**0**

Floyd Warshall Algorithm Example

	A	B	C	D
A	**0**	**6**	**5**	∞
B	**3**			
C	∞			
D	∞			

	A	B	C	D
A	**0**	**6**	**5**	∞
B	**3**	**0**	**8**	**2**
C	∞	**4**	**0**	∞
D	∞	∞	**7**	**0**

Selecting 1st row and 1st column and update the matrix

	A	B	C	D
A		**6**		
B	**3**	**0**	**8**	**2**
C		**4**		
D		∞		

	A	B	C	D
A	**0**	**6**	**5**	**8**
B	**3**	**0**	**8**	**2**
C	**7**	**4**	**0**	**6**
D	∞	∞	**7**	**0**

Selecting 2nd row and 2nd column and update the matrix

Floyd Warshall Algorithm Example

	A	B	C	D
A			5	
B			8	
C	7	4	0	6
D			7	

	A	B	C	D
A	0	6	5	8
B	3	0	8	2
C	7	4	0	6
D	14	11	7	0

Selecting 3rd row and 3rd column and update the matrix

	A	B	C	D
A				8
B				2
C				6
D	14	11	7	0

	A	B	C	D
A	0	6	5	8
B	3	0	8	2
C	7	4	0	6
D	14	11	7	0

Selecting 4th row and 4th column and update the matrix

Floyd Warshall Algorithm Example

```
int n = 4; // size of the adjaceny matrix

void fillDistanceMatrix(int A[n][n], int D[n][n]) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         if (i == j)
            D[i][j] = 0;
         else if (A[i][j] == 0)
            D[i][j] = INF;
         else
            D[i][j] = A[i][j];
      }
   }
}
```

```
void floydWarshall(int A[n][n], int D[n][n]) {
   fillDistanceMatrix(A, D);

   for (int k = 0; k < n; k++) {
      for (int i = 0; i < n; i++) {
         for (int j = 0; j < n; j++) {
            if (D[i][k] < INF && D[k][j] < INF)
               if (D[i][k] + D[k][j] < D[i][j])
                  D[i][j] = D[i][k] + D[k][j];
         }
      }
   }
}
```

Floyd Warshall Algorithm Example

```
int main() {
    int A[4][4] = {{0, 5, INF, 10},  {INF, 0, 3, INF},  {INF, INF, 0, 1}, {INF, INF, INF, 0}};
    int D[4][4];

    floydWarshall(A, D);

    printf("Shortest distances between all pairs of vertices:\n");
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if (D[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", D[i][j]);
        }
        printf("\n");
    }
   return 0;
}
```

Floyd Warshall Algorithm: **Application**

- **GPS navigation systems/ Google Maps**
- **Network routing protocols/Routing data packets**
- **Flight scheduling systems**
- **Traffic flow analysis**
- **Social Network Analysis**
- **Calculate the inversion of the real matrix**
- **Calculating transitive closure of directed graphs**
- **To check whether an undirected graph is bipartite**
- **To find the shortest path in a directed graph**
- **To compare two graphs to find the similarities between them.**
- **To find a maximum bandwidth path in network systems.**

Floyd Warshall Algorithm

Advantages:

- The Floyd Warshall Algorithm is easy to understand and implement.
- It can handle negative edge weights and graphs with cycles.
- It can solve problems involving a large number of nodes in the graph.

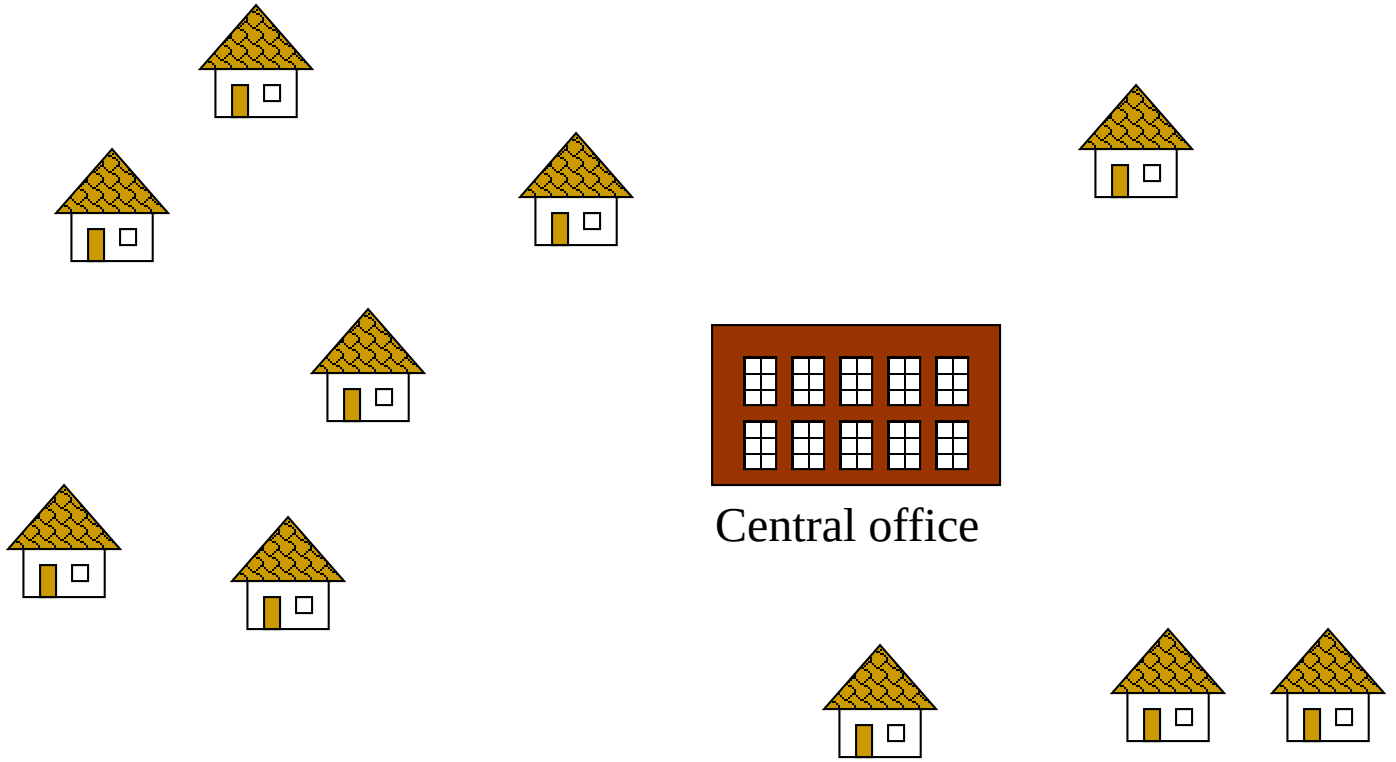
Disadvantages:

- The algorithm has a high time complexity of $O(V^3)$.
- The algorithm requires a large amount of memory to store the distance matrix. The space complexity is O (n^2).
- The algorithm does not work well for very large graphs.

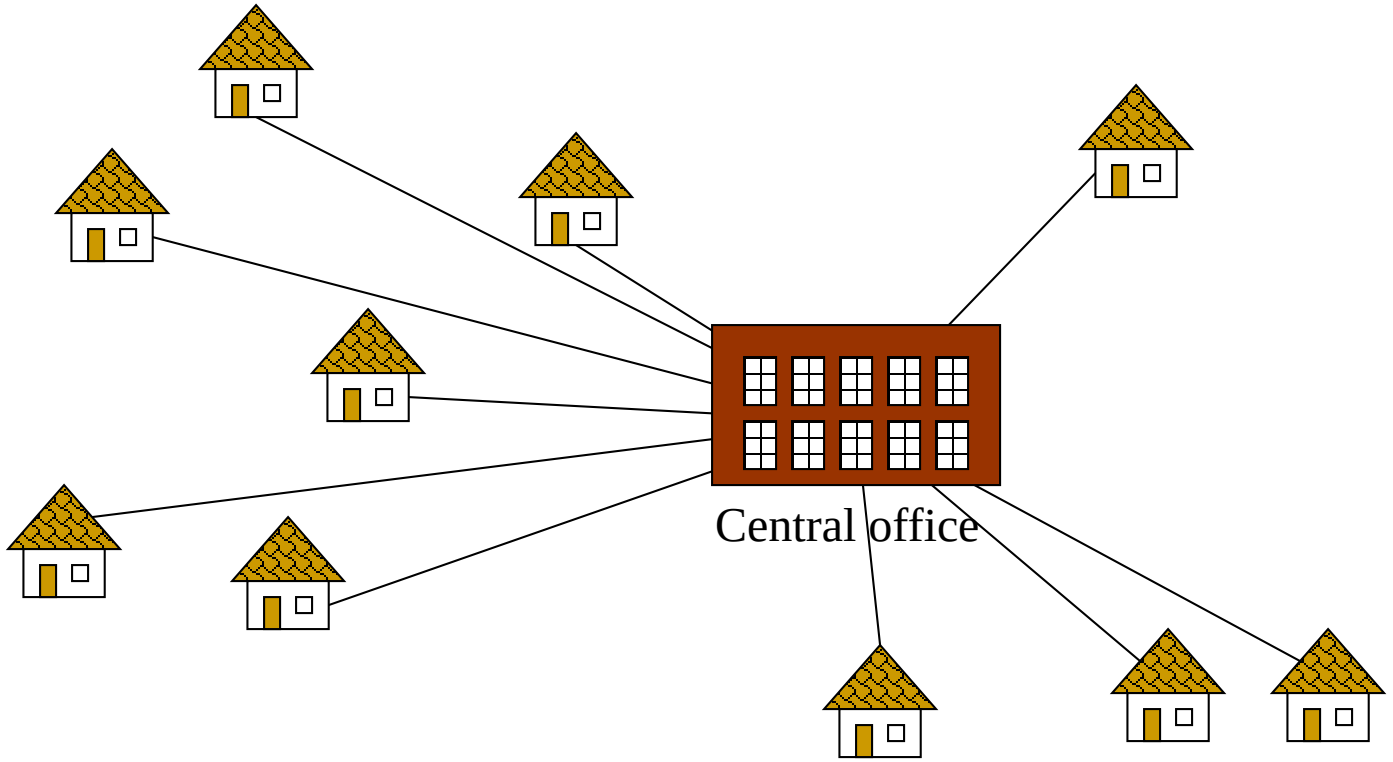
Minimum Cost Spanning Trees (MCST/MST)

Prims Algorithm
Kruskals Algorithm

Problem: Laying Telephone Wire

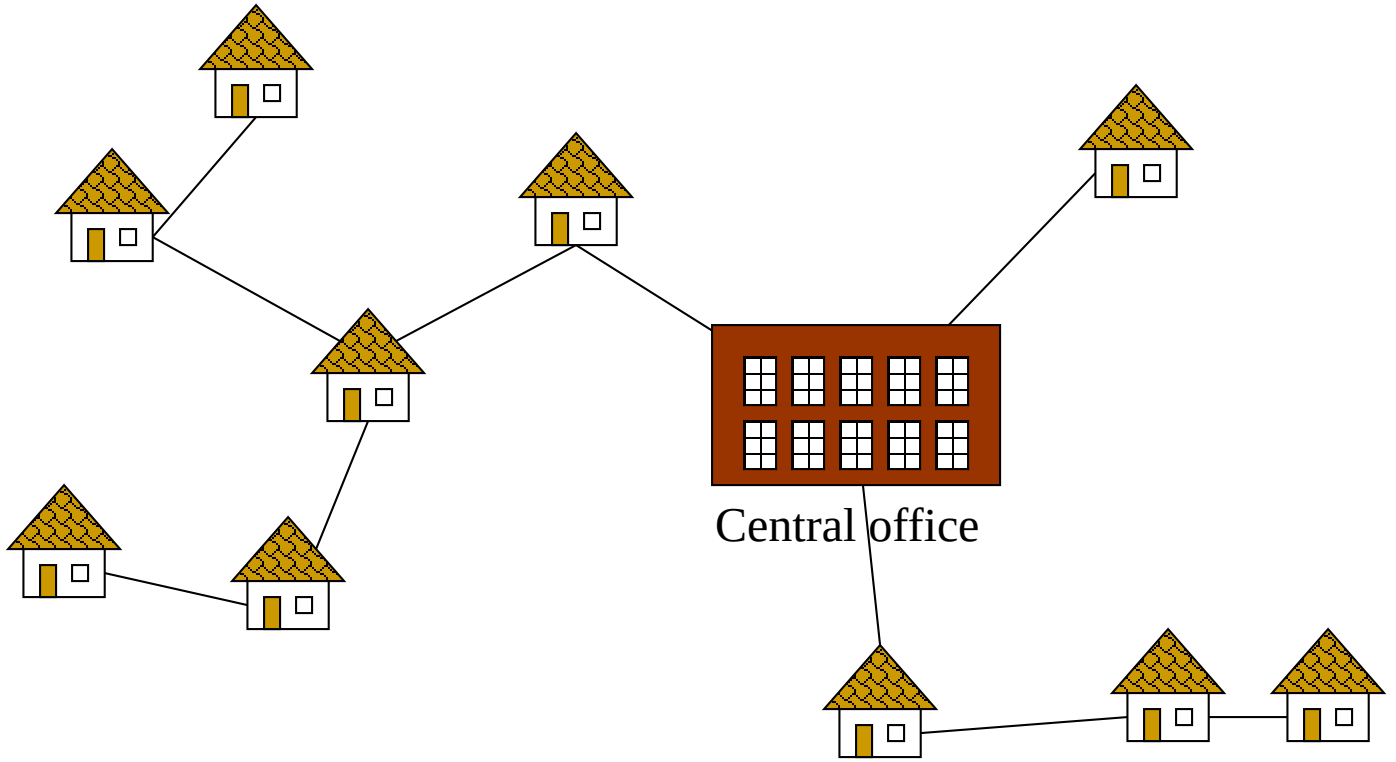


Wiring: Naïve Approach



Expensive!

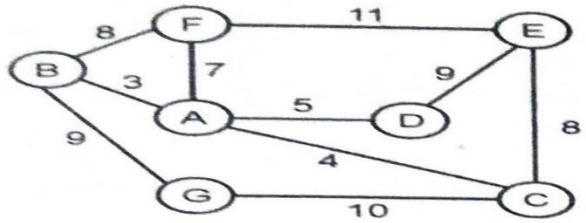
Wiring: Better Approach



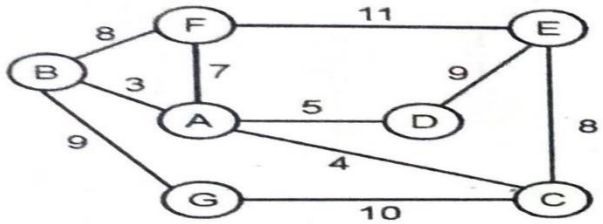
Minimize the total length of wire connecting the customers

Kruskal's Algorithm

Find the minimal spanning tree for the following graph:



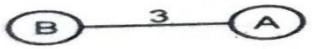
Ans. Step I : The graph contains 10 edges.



Step II : The edges after sorting edge weight,

	Edge weight
1	AB = 3
2	AC = 4
3	AD = 5
4	AF = 7
5	BF = 8
6	CE = 8
7	BG = 9
8	DE = 10
9	CG = 10
10	EF = 11

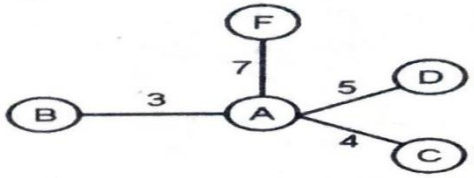
(i) Select edge AB.



(ii) Select edge AC

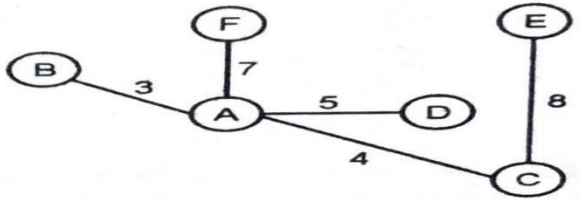
(iii) Select edge AD.

(iv) Select edge AF

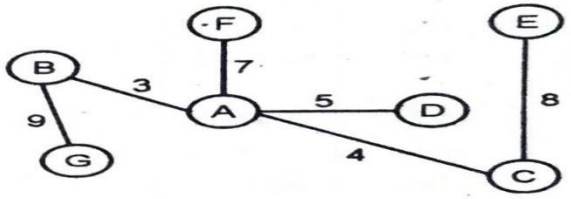


(v) Reject Edge BF since it create a cycle.

(vi) Select Edge CE



(vii) Select edge BG.

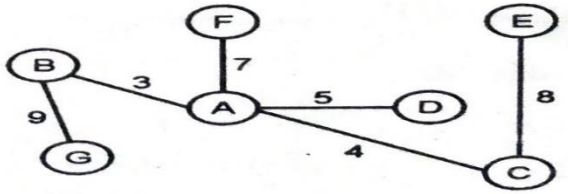


(viii) Reject edge DE Since it create a cycle.

(ix) Reject Edge CG Since it creates a cycle.

(x) Reject edge EF since it also create a cycle.

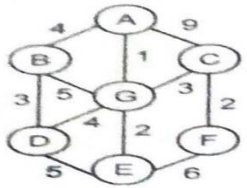
Step IV : So the minimum spanning Tree is



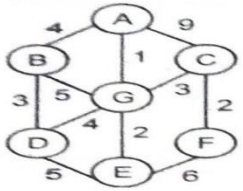
And the minimum spanning tree cost

3 + 4 + 5 + 7 + 8 + 9 = 36

Kruskal's Algorithm



Ans. Step I : Given graph is

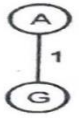


Step II : The given graph is having 11 edges. After the sorting on weights the edges are.

Edge	weight
(1)	AG = 1
(2)	EG = 2
(3)	CF = 2
(4)	BD = 3
(5)	CG = 3
(6)	DG = 4
(7)	AB = 4
(8)	BG = 5
(9)	DE = 5
(10)	EF = 6
(11)	AC = 9

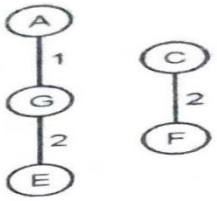
Step III :

(i) Select edge AG.

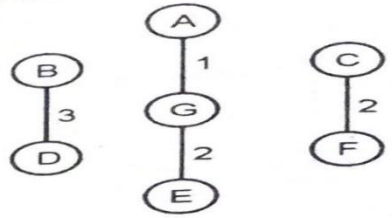


(ii) Select edge EG.

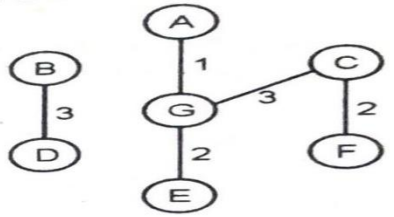
(iii) Select edge CF



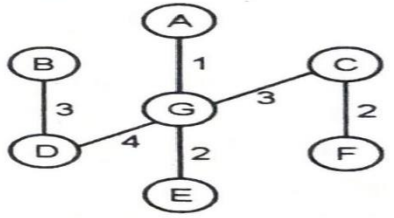
(iv) Select edge BD.



(v) Select edge CG.

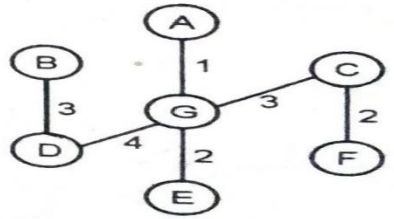


(vi) Select edge DG.



(vii) Discard all other edges since all they are forming the cycle in the graph.

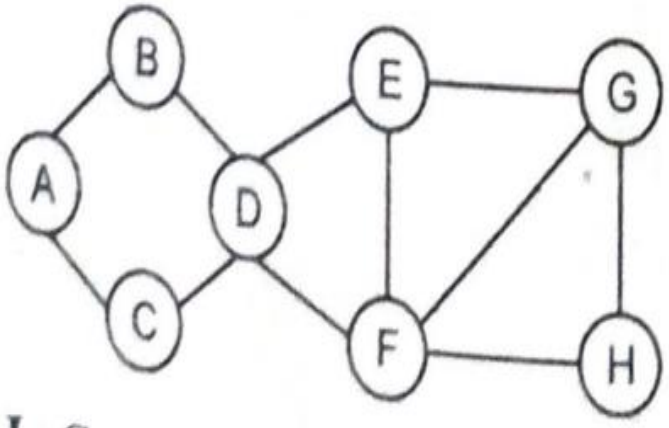
Step IV : So the minimum spanning tree is



And the minimum spanning tree cost is
1 + 2 + 2 + 3 + 3 + 4 = 15

Kruskal's Algorithm

How many minimum spanning trees does the following graph have? Draw them.



Step I : Consider given graph with weights :

Step II : The given graph is having 11 edges. After sorting them, we have

	Edges	weight
(1)	CD	1
(2)	EF	1
(3)	FH	1
(4)	AB	2
(5)	BD	2
(6)	DE	2
(7)	EG	2
(8)	FG	2
(9)	AC	3
(10)	GH	3
(11)	DF	4

Step III : The above graph is having two (2) minimum spanning trees with the cost of 11.

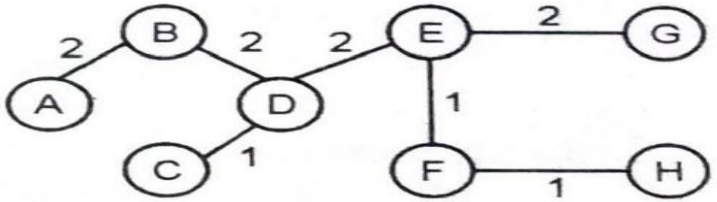


Fig.1

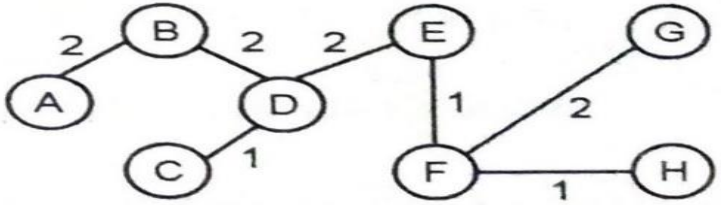
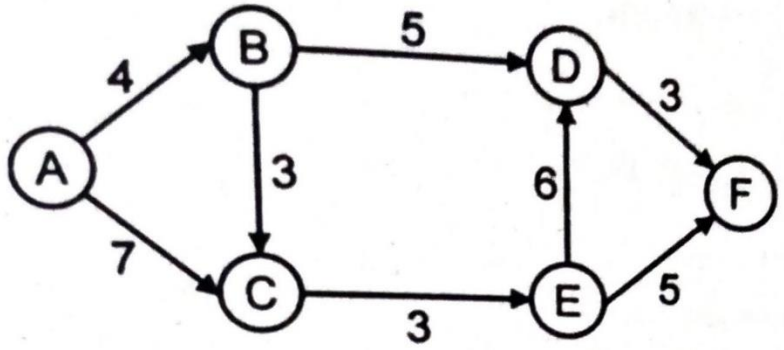


Fig.2

Prim's Algorithm

Find the minimum spanning tree for the following graph.



Minimum spanning tree using PRIMS algorithm :

Set of vertices in MST	Edge belongings to partial MST	Edge selected	Partial MST
A	AB = 4 AC = 7	AB = 4	Fig.(a)
AB	AC = 7 BC = 3 BD = 5	BC = 3	Fig.(b)
ABC	AC = 7 BD = 5 CE = 3	CE = 3	Fig.(c)
ABCE	AC = 7 BD = 5 EF = 5 ED = 6	BD = 5	Fig.(d)
ABCDE	AC = 7 EF = 5 ED = 6 DF = 3	DF = 3	Fig.(e)

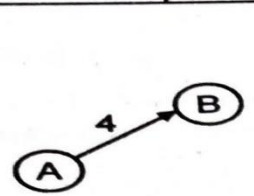


Fig.(a)

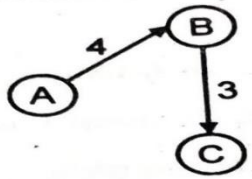


Fig.(b)

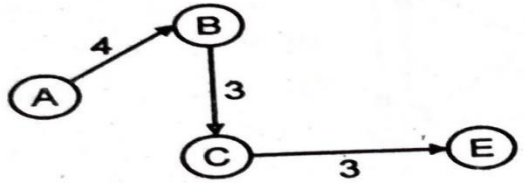


Fig.(c)

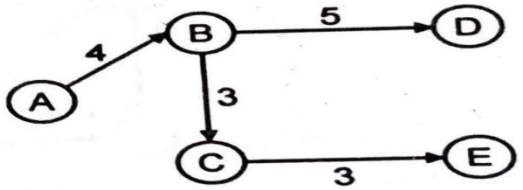


Fig.(d)

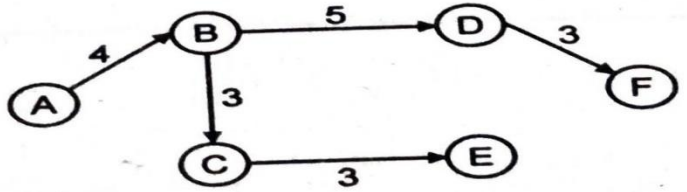
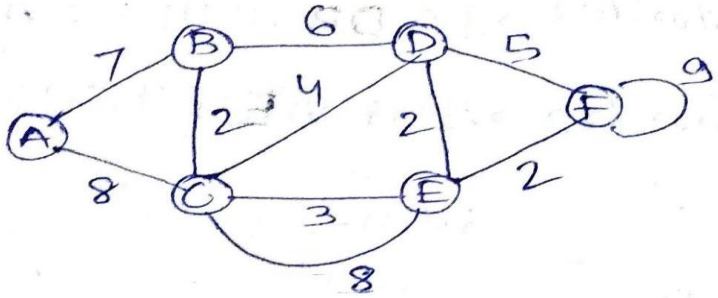


Fig.(e)

And the cost of MST using PRIMS algorithm is $4 + 5 + 3 + 3 + 3 = 18$.

Prim's Algorithm

PRIM'S ALGORITHM (MST)

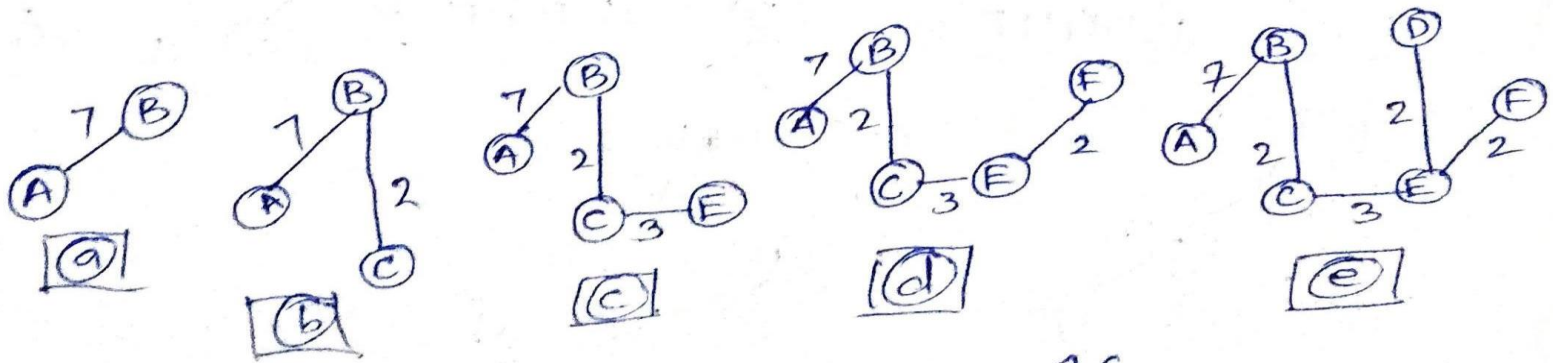


STEP-1

Set of vertices in MST	Edge belongs to Partial MST	Edge selected	Partial MST
A	AB = 7 Ae = 8	AB	(a)
B	Ae = 8 BC = 2 BD = 6	BC	(b)
C	Ae = 8, BD = 6 CE = 3, CF = 8 CD = 4	CE	(c)
E	Ae = 8, BD = 6 CE = 8, CD = 4 ED = 2, EF = 2	EF	(d)
F	Ae = 8, BD = 6 CE = 8, CD = 4 ED = 2, FF = 9 DF = 5	ED	(e)

After diagram (MST) e even if we select any Edge, then it will create the loop.

STEP-2

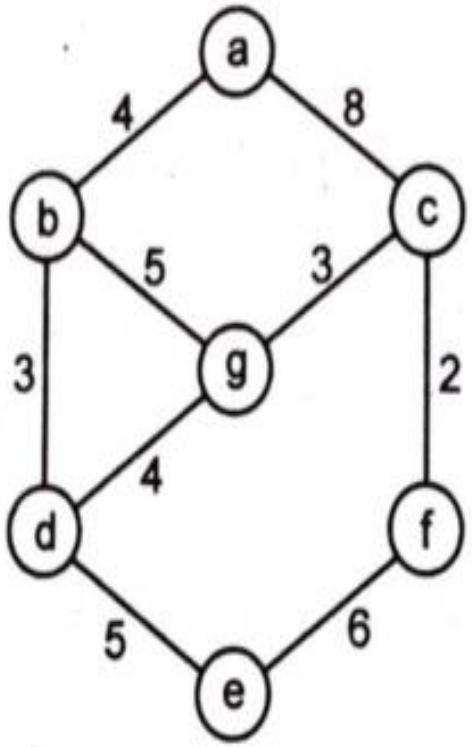


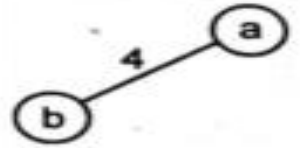
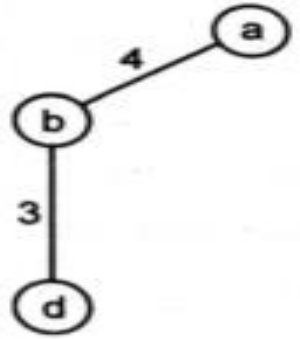
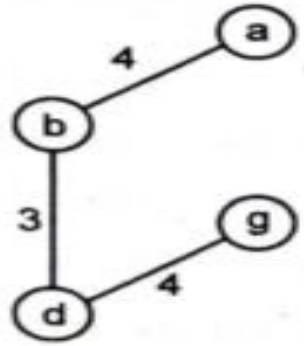
STEP-3

Cost of MST = 7 + 2 + 3 + 2 + 2 = 16

Prim's Algorithm

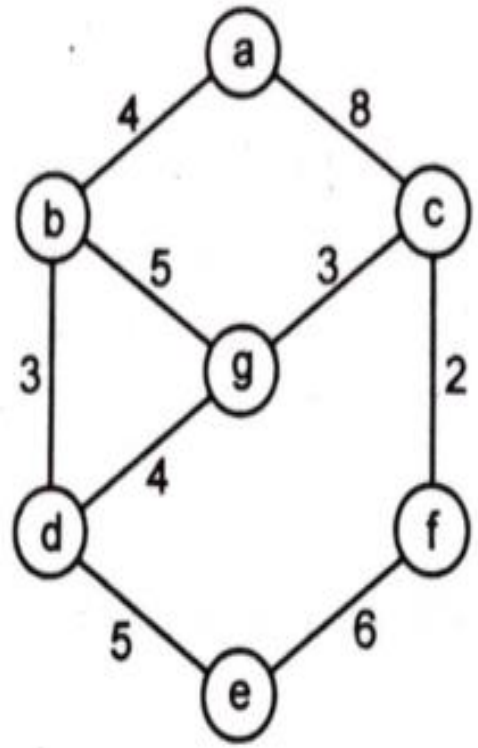
Construct the Minimum Spanning Tree (MST) of following graph using Prim's algorithm :

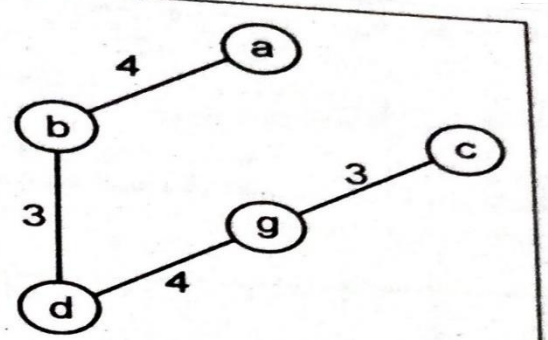
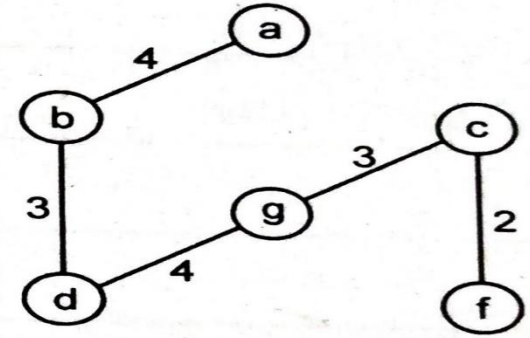
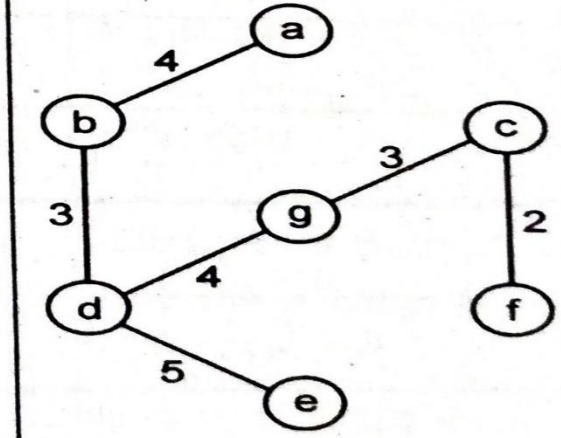


Set of vertices in MST	Edge belonging to parital MST	Edge selected	Partial MST
a	ab = 4, ac = 8	ab	
ab	ac = 8, bd = 3, bg = 5	bd	
abd	ac = 8, bg = 5, dg = 4, de = 5	dg	

Prim's Algorithm

Construct the Minimum Spanning Tree (MST) of following graph using Prim's algorithm :



abdg	ac = 8, bg = 5, de = 5, gc = 3	gc	
abcdg	ac = 8, bg = 5, de = 5, cf = 2	cf	
abcdfg	ac = 8, bg = 5, de = 5, fe = 6	de	

Cost of MST = 4 + 3 + 4 + 3 + 2 + 5 = 21

Running time of Prim's algorithm

Initialization of priority queue (array): $O(|V|)$

Update loop: $|V|$ calls

- Choosing vertex with minimum cost edge: $O(|V|)$
- Updating distance values of unconnected vertices: each edge is considered only **once** during entire execution, for a **total** of $O(|E|)$ updates

Overall cost:

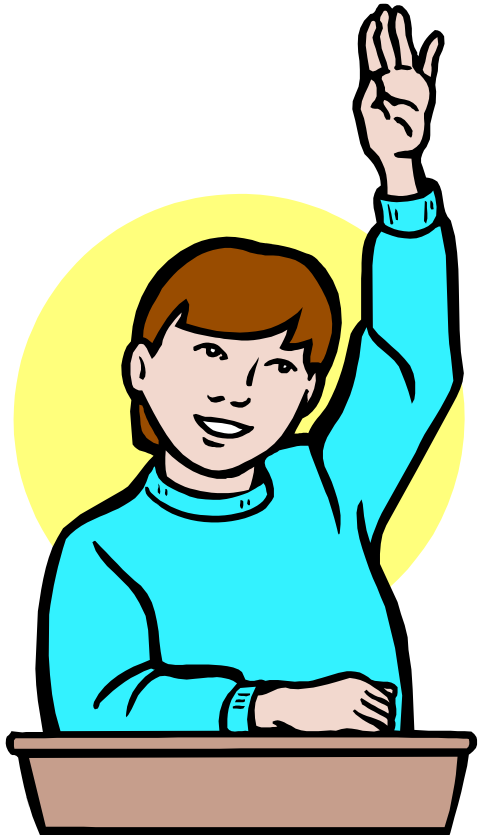
$$O(|E| + |V|^2)$$

Summary

- Graphs can be used to represent many real-life problems.
- There are numerous important graph algorithms.
- We have studied some basic concepts and algorithms.
  - Graph Traversal
  - Spanning Tree
  - Minimum Spanning Tree
  - Shortest Path

Question?

- A good question deserve a good grade…



Dr. Amit Pimpalkar

+91-988-171-3450

pimpalkarap@rknec.edu