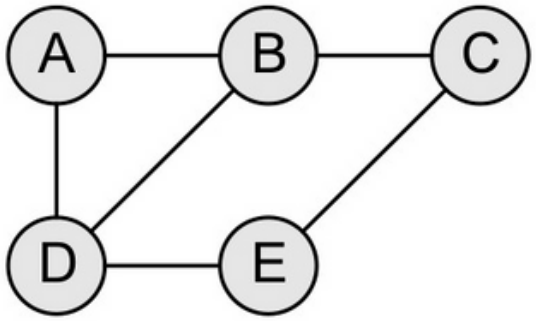
Data Structures and Algorithms

GRAPHS

Abhinav Kesarwani
B.Tech Student | Artificial Intelligence & Data Science
Gati Shakti Vishwavidyalaya (Ministry of Railways, Govt. of India)
Linkedin

Introduction

- A graph is an abstract data structure that is used to implement the graph concept from mathematics.
- A graph is basically, a collection of vertices (also called nodes) and edges that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of a having a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can be represented



Why Graphs are useful?

Graphs are widely used to model any situation where entities or things are related to each other in pairs; for example, the following information can be represented by graphs:

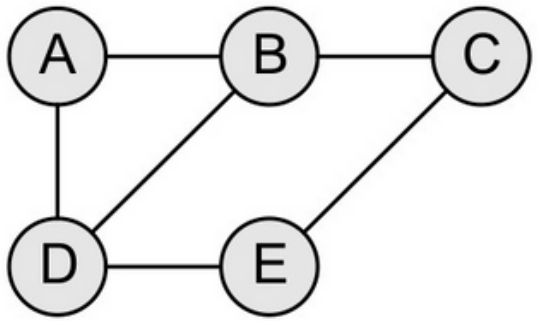
- ***Social Networks*** in which various connections between the accounts can be shown.
- ***Transportation networks*** in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.
- In ***World Wide Web***, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u.
- In ***Operating System***, we come across the Resource Allocation Graph where each process and resources are considered to be vertices.

Definition of Graphs

- A **graph G** is defined as an **ordered set (V, E)**, where **V(G)** represents the set of vertices and **E(G)** represents the edges that connect these vertices.

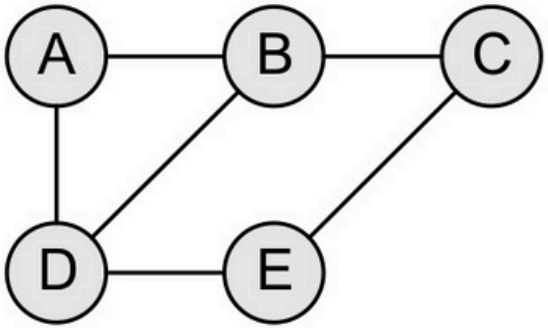
- This graph have V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}.

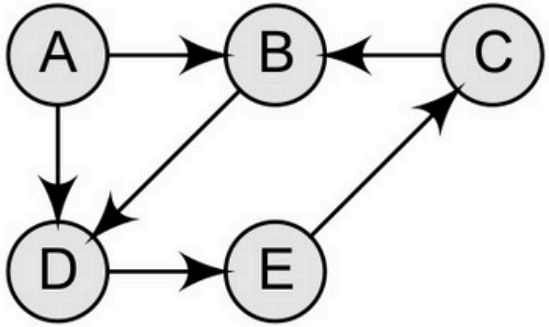
- There are five vertices or nodes and six edges in the graph.



Types of Graphs

- In an **undirected graph**, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. In a
- **directed graph**, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

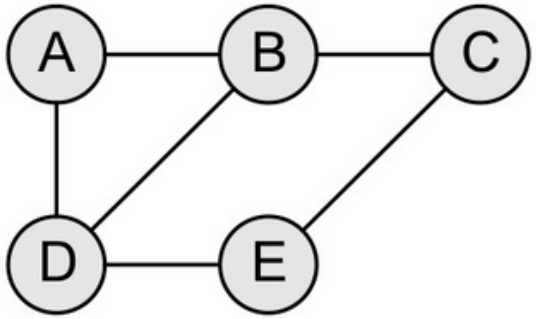




Graph Terminology

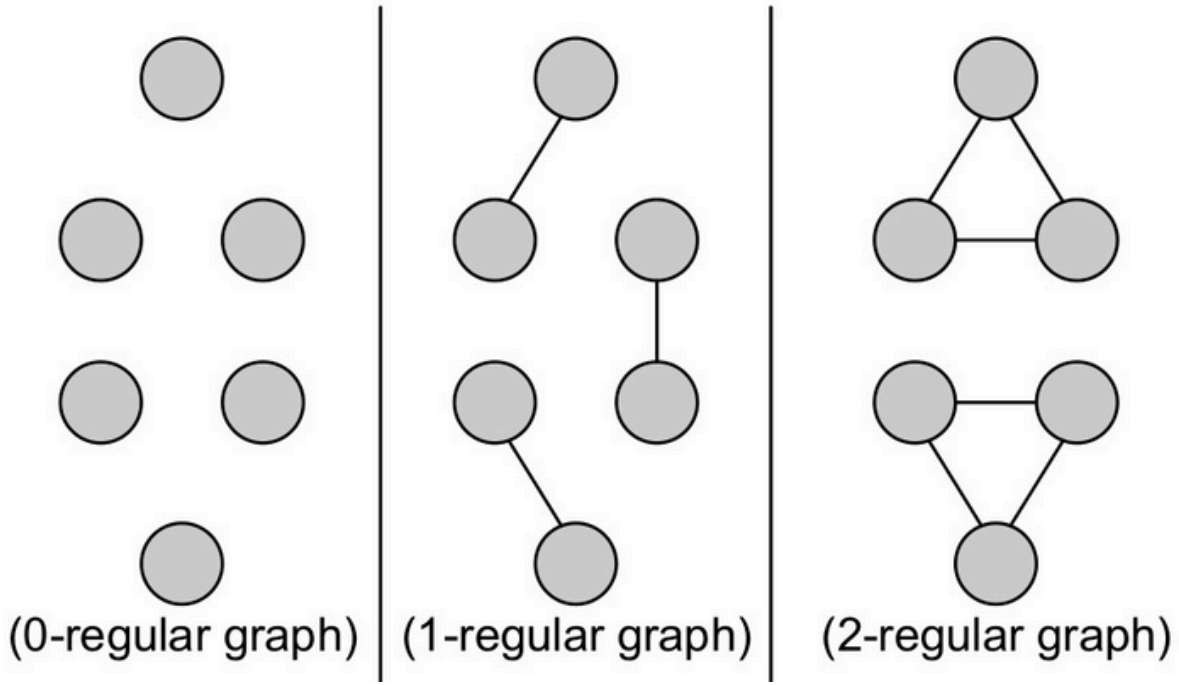
- **Adjacent nodes or neighbours:** For every edge, e = (u, v) that connects nodes u and v, the nodes u and v are the end-points and are said to be the adjacent nodes or neighbours.
- **Degree of a node:** Degree of a node u, deg(u), is the total number of edges containing the node u.

If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.



Graph Terminology

- **Regular graph:** It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree k is called a k–regular graph or a regular graph of degree k.



Graph Terminology

- **Path:** A path P written as P = *{v0, v1, v2, …, vn}*, of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, *u = v0, v = vn* and *vi–1* is adjacent to vi for *i* = 1, 2, 3, …, n.

- **Cycle:** A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).

- **Connected graph:** A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has **n(n–1)/2 edges,** where n is the number of nodes in G.

Graph Terminology

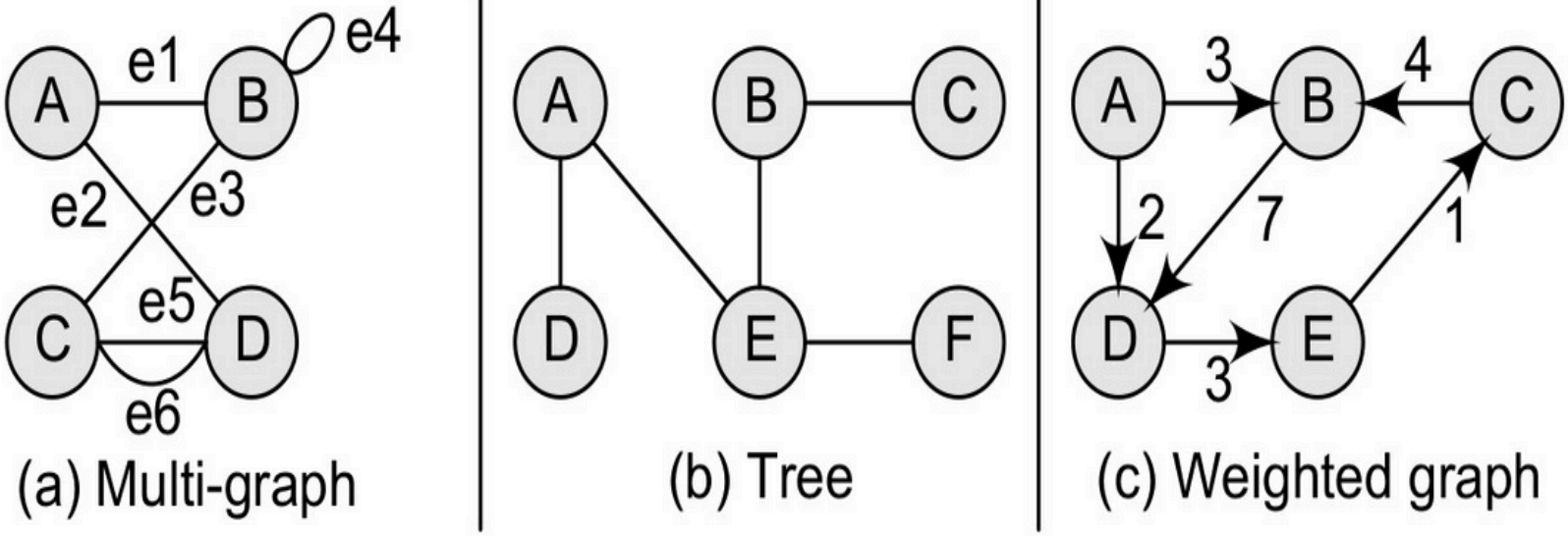
Labelled graph or weighted graph: A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by **w(e)** is a positive value which indicates the cost of traversing the edge.

Multiple edges: Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

Loop: An edge that has identical end-points is called a loop. That is, e = (u, u).

Graph Terminology

- **Multi-graph:** A graph with multiple edges and/or loops is called a multi-graph.
- **Size of a graph:** The size of a graph is the total number of edges in it.



Directed Graph

- A directed graph G, also known as a **digraph**, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G.

  For an edge (u, v),

  - The edge begins at u and terminates at v.
  - u is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e.
  - u is the predecessor of v. Correspondingly, v is the successor of u.
  - Nodes u and v are adjacent to each other.

Terminology of a Directed Graph

Out-degree of a node: The out-degree of a node u, written as outdeg(u), is the number of edges that originate at u.
In-degree of a node: The in-degree of a node u, written as indeg(u), is the number of edges that terminate at u.
Degree of a node: The degree of a node, written as deg(u), is equal to the sum of in-degree and out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg(u).
Isolated vertex: A vertex with degree zero. Such a vertex is not an end-point of any edge.
Pendant vertex (also known as leaf vertex): A vertex with degree one.

Terminology of a Directed Graph

Cut vertex: A vertex which when deleted would disconnect the remaining graph.
Source: A node u is known as a source if it has a positive out-degree but a zero in-degree.
Sink: A node u is known as a sink if it has a positive in-degree but a zero out-degree.
Reachability: A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.
Strongly connected directed graph: A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.

Terminology of a Directed Graph

Weakly connected digraph: A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

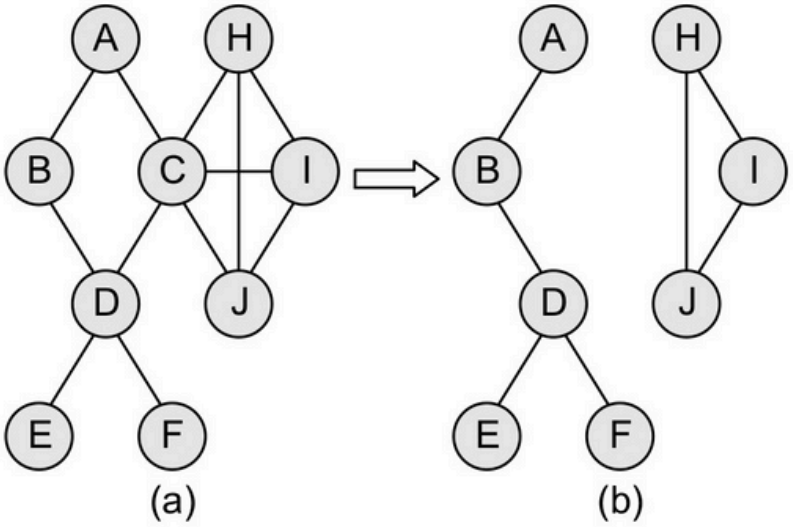
Parallel/Multiple edges: Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

Simple directed graph: A directed graph G is said to be a simple directed graph **if and only if it has no parallel edges**. However, a simple directed graph may contain cycles with an exception that it cannot have more than one loop at a given node.

Bi-Connected Components

A vertex **v** of G iscalled an **articulationpoint**, if removing **v**along withthe edges incident on v, results in agraph that has at least two connected components.

A bi-connected graph is defined as a connected graph that has **no articulation vertices**. That is, a bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected.

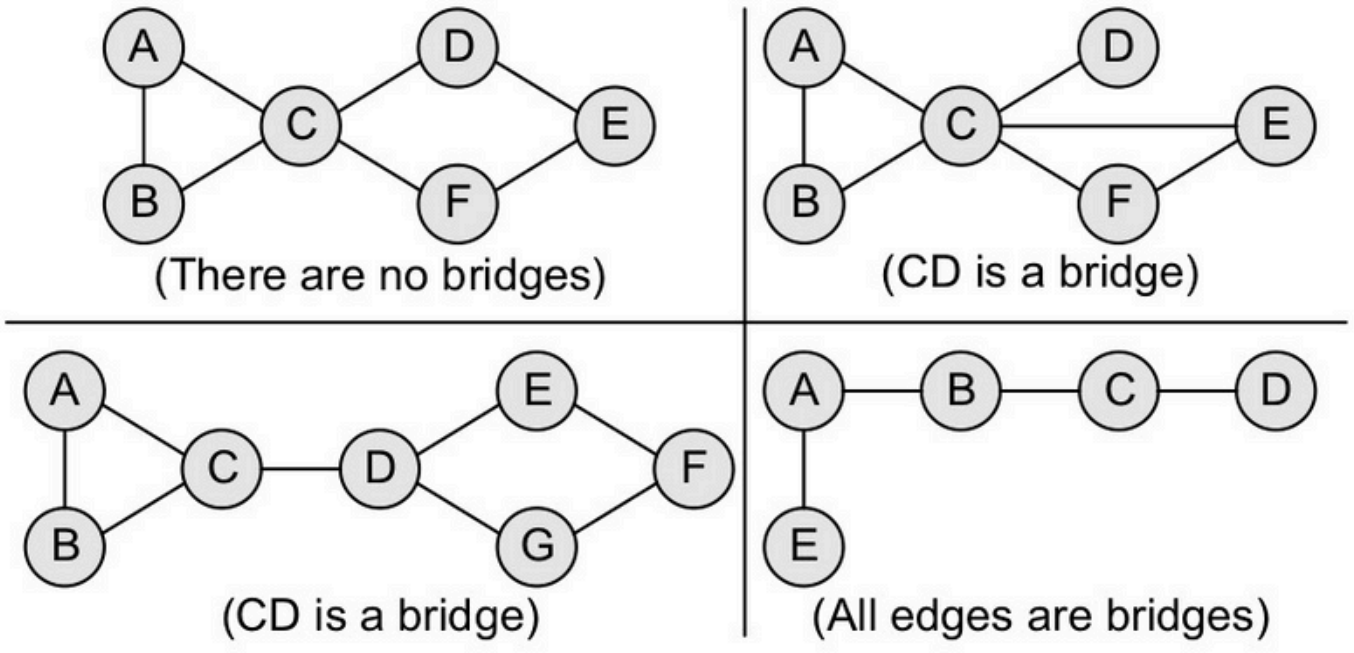


Bi-Connected Components

By definition:

A bi-connected undirected graph is a connected graph that cannot be broken into disconnected pieces by deletingany single vertex.

An edge in a graph is called a bridge if removing that edge results in a disconnected graph.



Representation of Graphs

There are three common ways of storing graphs in the computer's memory:

- **Sequential** representation by using an adjacency matrix.
- **Linked** representation by using an adjacency list that stores the neighbours of a node using a linked list.
- **Adjacency multi-list** which is an extension of linked representation.

Adjacency Matrix Representation

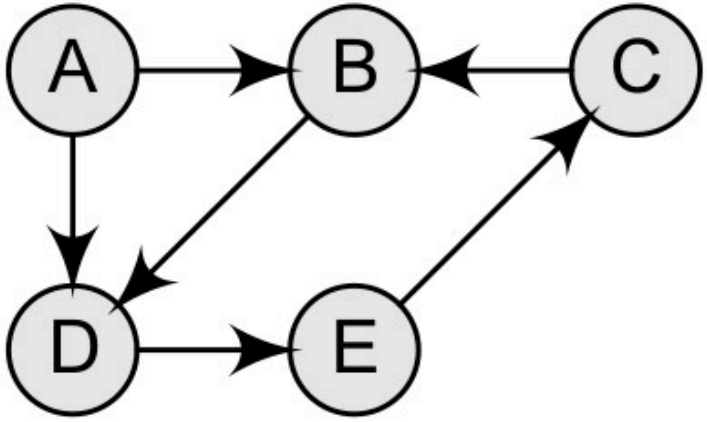
An adjacency matrix is used to represent which nodes are adjacent to one another. By definition:

- Two nodes are said to be adjacent if there is an edge connecting them.
- In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by **traversing one edge**.
- For any graph G having **n** nodes, the adjacency matrix will have the dimension of **n x n**.
- In an adjacency matrix, the rows and columns are labelled by graph vertices.

$$a_{ij} \begin{cases} 1 & \text{[if } v_i \text{ is adjacent to } v_j\text{, that is there is an edge } (v_i, v_j)\text{]A} \\ 0 & \text{[otherwise]} \end{cases}$$

Adjacency Matrix Representation

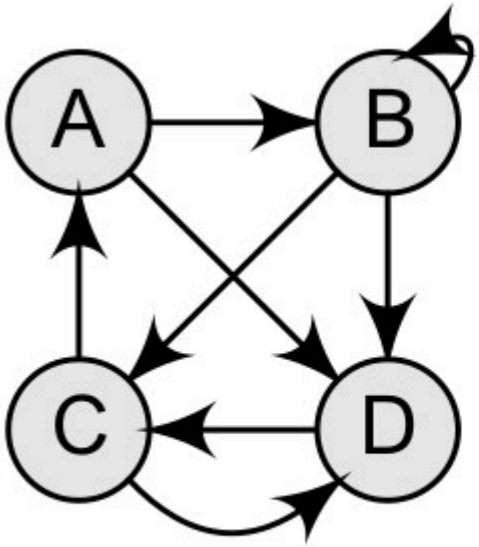
- Sincean adjacency matrix contains only 0s and 1s,it is called a bit matrix or a Boolean matrix.
- The entries in the matrix depend on the ordering of the no desin G.
- A change in the order of nodes will result in a different.



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

(a) Directed graph

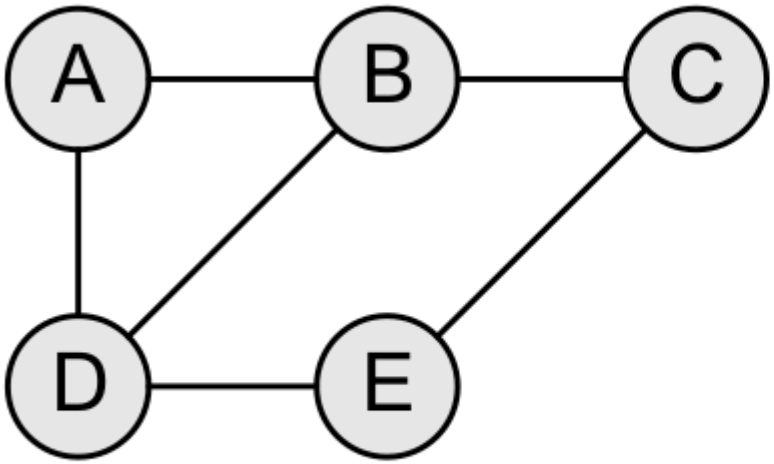
Adjacency Matrix Representation



	A	B	C	D
A	0	1	0	1
B	0	1	1	1
C	1	0	0	1
D	0	0	1	0

(b) Directed graph with loop

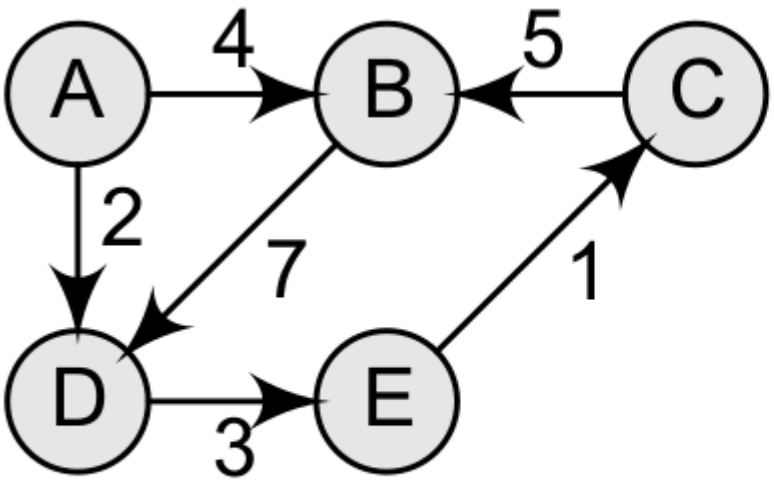
Adjacency Matrix Representation



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

(c) Undirected graph

Adjacency Matrix Representation



	A	B	C	D	E
A	0	4	0	2	0
B	0	0	0	7	0
C	0	5	0	0	0
D	0	0	0	0	3
E	0	0	1	0	0

(d) Weighted graph

Adjacency List Representation

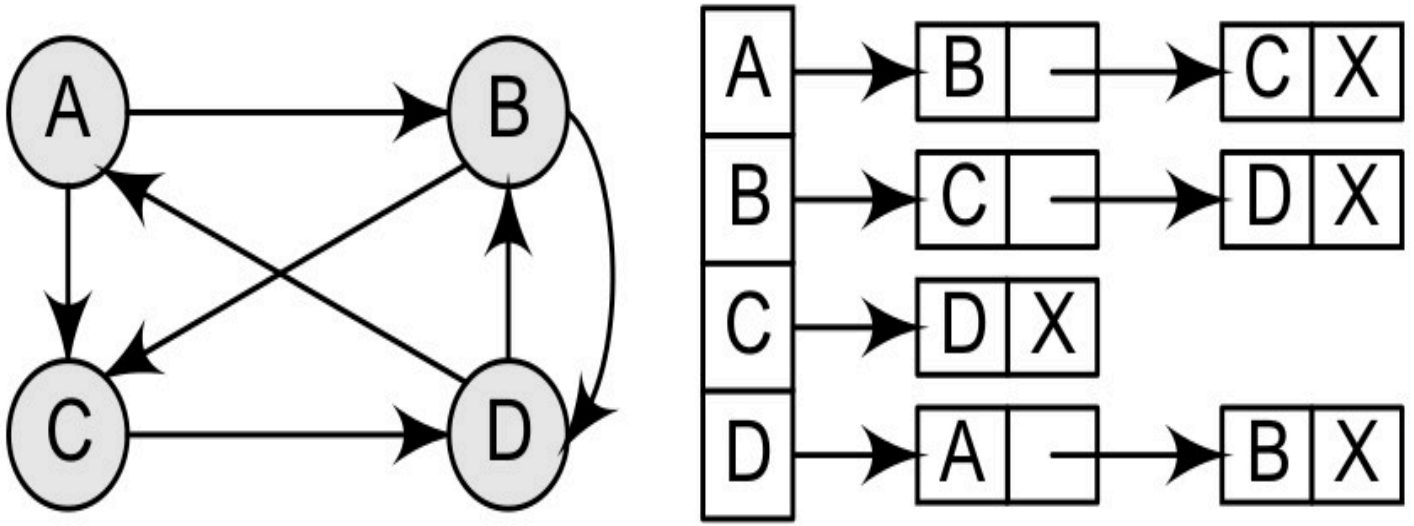
- An adjacency list is another way in which graphs can be represented.
- This structure consists of a list of all nodes in **G**. Every nodeisin turn linked to its own list that contains the names of all other nodes that are adjacent to it.
- The key advantages of using an adjacency list are:

1) It is easy to follow and clearly shows the adjacent nodes of a particular node.

2) It is often used for storing graphs that have a small-to-moderate number of edges. That is, preferred for sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.

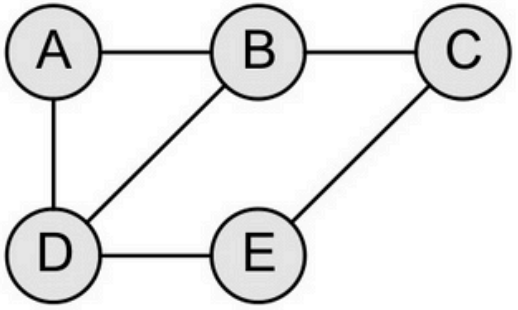
3) Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

Adjacency List Representation

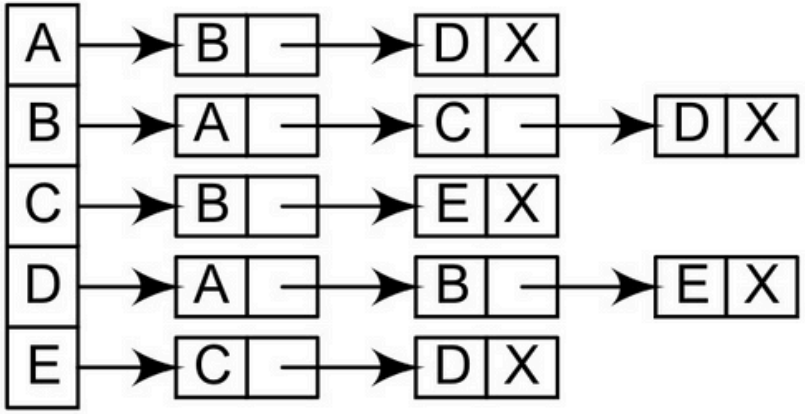


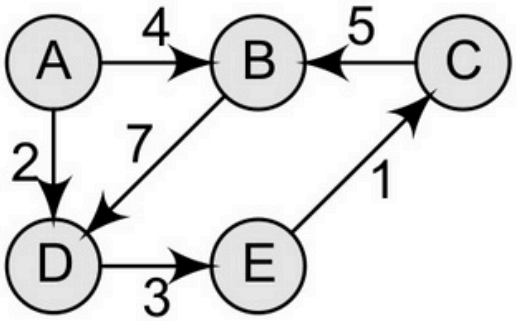
Graph G and its adjacency list

Adjacency List Representation

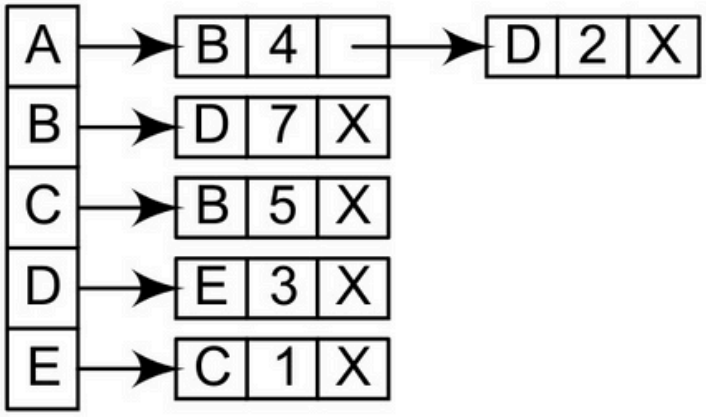


(Undirected graph)





(Weighted graph)



Graph Traversal Algorithms

- By traversing a graph, we mean the method of examining the nodes and edges of the graph.There are two standard methods of graph traversal which are:

1. Breadth-first search
2. Depth-first search

- While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack.
- But both these algorithms make use of a variable STATUS. During the execution of the algorithm, every node in the graph will have the variable STATUS set to 1, 2 or 3, depending on its current state.

Graph Traversal Algorithms

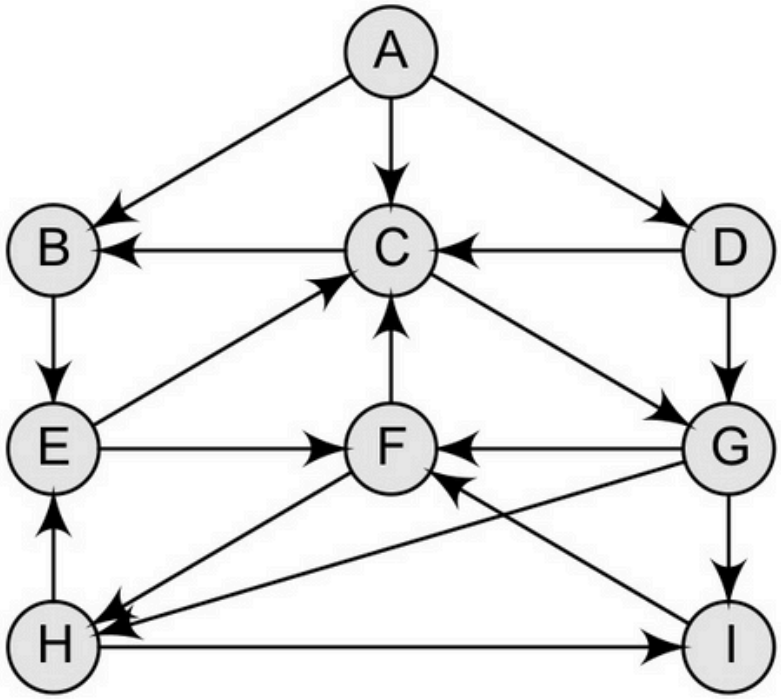
Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

Breadth-First Search Algorithm

- Breadth-first search(BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes.
- Then for each of those nearest nodes, we explorestheirunexplored neighbour nodes, and so on, until it finds the goal.
- We start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth.

Breadth-First Search Algorithm

Consider the graph G. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Breadth-First Search Algorithm

- The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered.

- During the execution of the algorithm, weuse two arrays:

QUEUE and ORIG.

- While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge.

- Initially, FRONT = REAR = –1. The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0	QUEUE = A
REAR = 0	ORIG = \0

Breadth-First Search Algorithm

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG = \0	A	A	A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG = \0	A	A	A	B

Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Breadth-First Search Algorithm

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Breadth-First Search Algorithm

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG = \0	A	A	A	B	C	E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG = \0	A	A	A	B	C	E	G	G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as A –> C –> G –> I.

Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Breadth-First Search Algorithm

```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
        (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Breadth-First Search Algorithm

Space complexity:

- In the breadth-first search algorithm,all the nodes at a particular level must be saved until their child nodes in the next level have been generated.

- If the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as O (| E | + | V |), where | E | is the total number of edges in G and | V | is the number of nodes or vertices.

Time complexity:

- Time complexity can be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.

Breadth-First Search Algorithm

Applications ofBreadth-First Search Algorithm:

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

Depth-first Search Algorithm

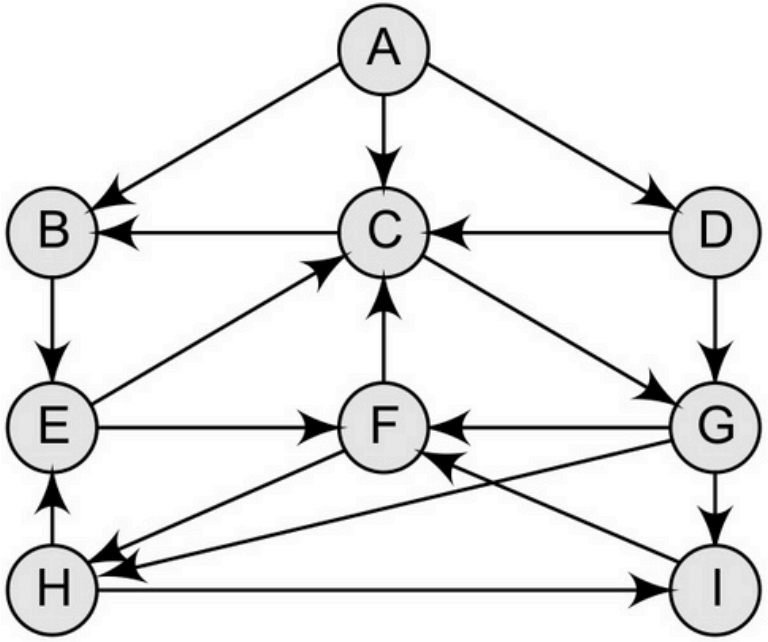
- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.
- When a dead-end is reached, the algorithm backtracks, returningto the most recent node that has not been completely explored.
- Depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on.
- During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

Depth-first Search Algorithm

- The algorithm proceeds like this until we reach a dead-end (end of path P).
- On reaching the dead-end, **we backtrack to find another path P'. The algorithm terminates when backtracking leads back to the starting node A.**
- In this algorithm, **edges that lead to a new vertex are called discovery edges** and edges that lead to an already visited vertex are called back edges.
- This algorithm is **similar to the in-order traversal** of a binary tree. Its implementation is similar to that of the breadth-first search algorithm but here we use a stack instead of a queue.
- Again, we use a variable STATUS to represent the current state of the node.

Depth-first Search Algorithm

Consider the graph G. The adjacency list of G is also given.Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H.



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Depth-first Search Algorithm

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H

STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I

STACK: E, F

Adjacency lists
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Depth-first Search Algorithm

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Depth-first Search Algorithm

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B | STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E | STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Depth-first Search Algorithm

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Depth-first Search Algorithm

Time complexity:

The time complexity of a depth-first search is proportional to number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V|+|E|)).

Depth-first Search Algorithm

Applications of Depth-First SearchAlgorithm

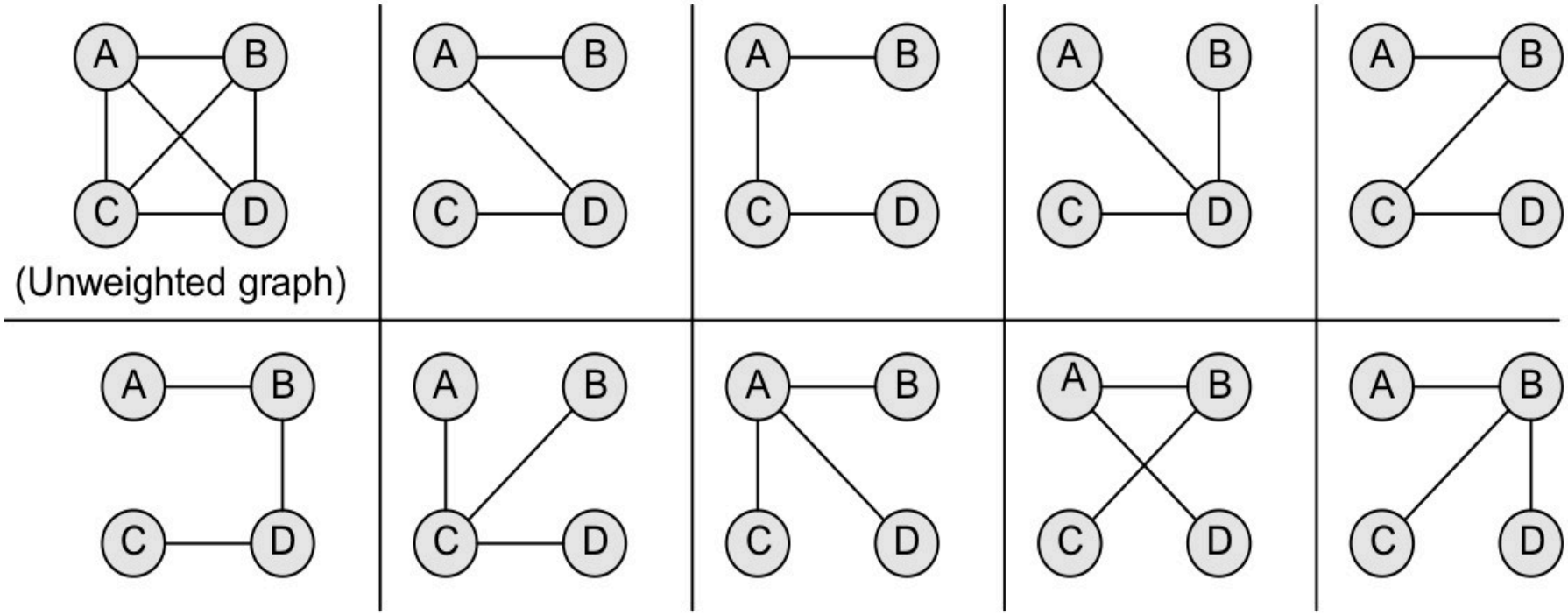
- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

Minimum Spanning Tree

- A spanning tree of a connected,undirected graph G is a sub-graph of G **which is a tree thatconnects all the vertices together**.
- A graph G can have many different spanning trees.
- A minimum spanning tree (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree.
- In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

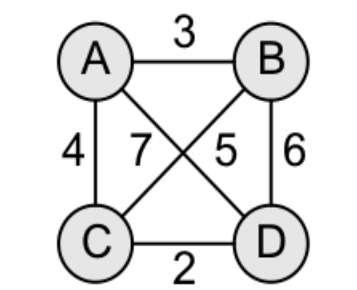
Minimum Spanning Tree

Consider an unweighted graph G, we can draw many distinct spanning trees.Eight of them are given here. For an unweighted graph, every spanning tree is a minimums panning tree.

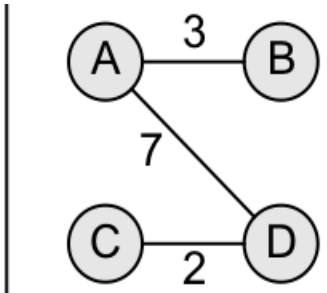


Minimum Spanning Tree

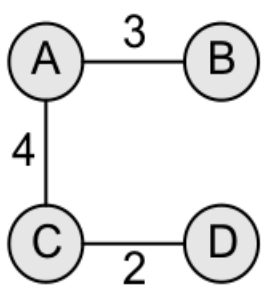
Consider an weighted graph G, MST is only one.



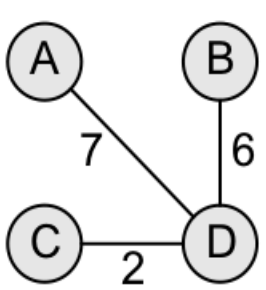
(Weighted graph)



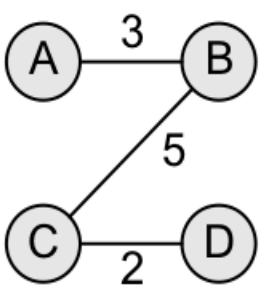
(Total cost = 12)



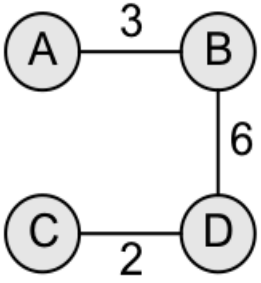
(Total cost = 9)



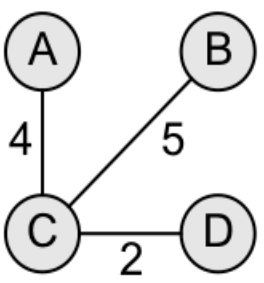
(Total cost = 15)



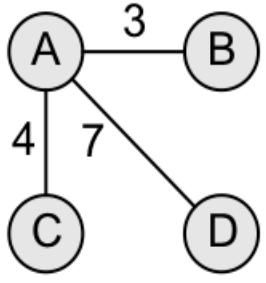
(Total cost = 10)



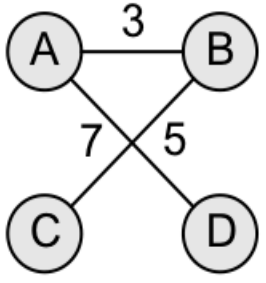
(Total cost = 11)



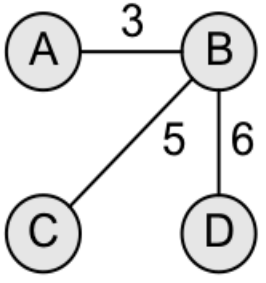
(Total cost = 11)



(Total cost = 14)



(Total cost = 15)



(Total cost = 14)

Applications of MSTs

- MSTs are widely used fordesigningnetworks.
- MSTs are used to find airline routes.
- MSTs are also used to find the cheapest way to connect terminals, such as cities, electronic components or computers via roads, airlines, railways, wires or telephone lines.
- MSTs are applied in routing algorithms for finding the most efficient path.

Prim’s Algorithm

Prim’s algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph.

For this, the algorithm maintains three sets of vertices which can be given as below:

Tree vertices: Vertices that are a part of the minimum spanning tree T.

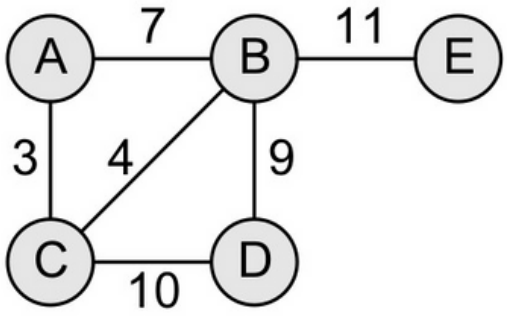
Fringe vertices: Vertices that are currently not a part of T, but are adjacent to some tree vertex.

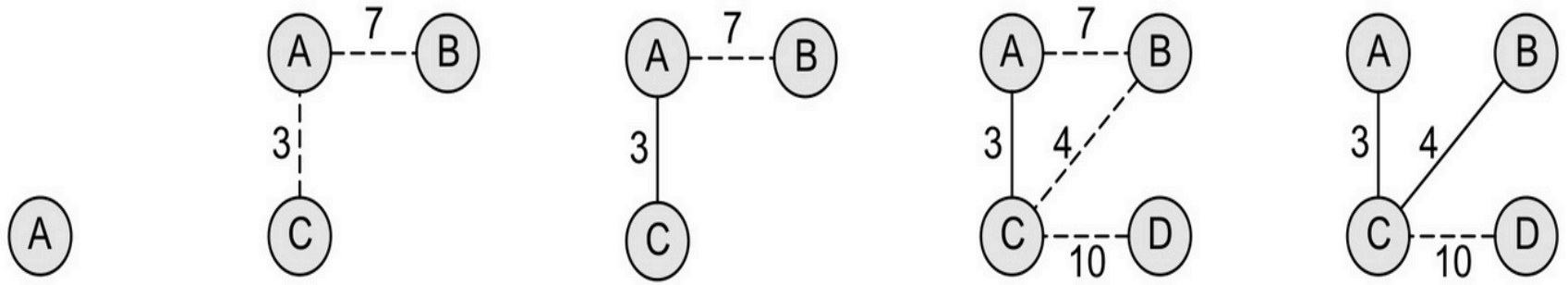
Unseen vertices: Vertices that are neither tree vertices nor fringe vertices fall under this category.

Prim's Algorithm

```
Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
Step 3:     Select an edge e connecting the tree vertex and
            fringe vertex that has minimum weight
Step 4:     Add the selected edge and the vertex to the
            minimum spanning tree T
        [END OF LOOP]
Step 5: EXIT
```

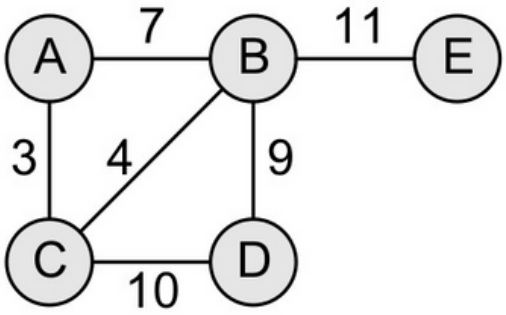
Prim's Algorithm

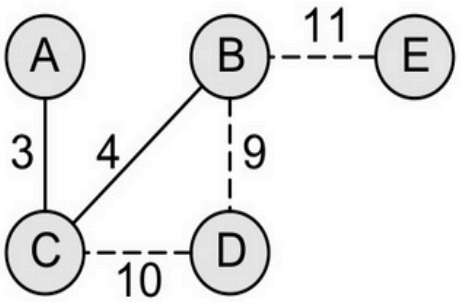




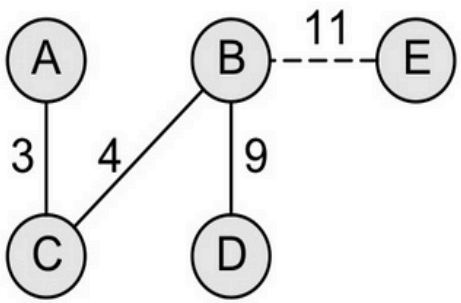
Step 1 Step 2 Step 3 Step 4 Step 5

Prim's Algorithm

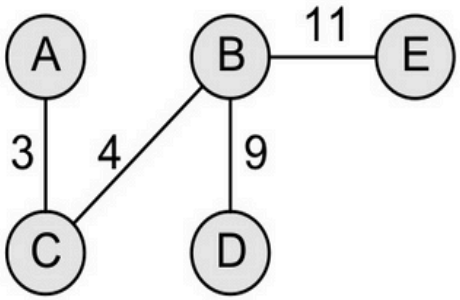




Step 6



Step 7



Step 8

Kruskal’s Algorithm

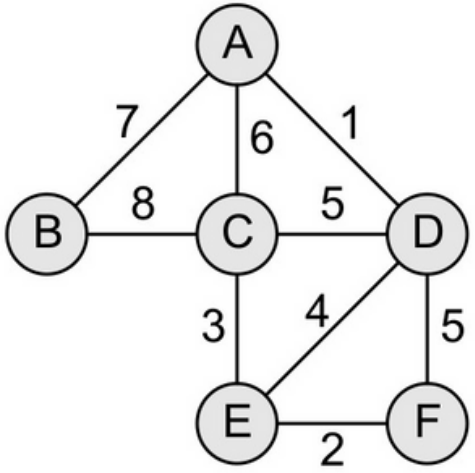
- Kruskal’s algorithm is used to find the minimum spanning tree for a connected weighted graph.

- Kruskal’s algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum.

Kruskal's Algorithm

```
Step 1: Create a forest in such a way that each graph is a separate
        tree.
Step 2: Create a priority queue Q that contains all the edges of the
        graph.
Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY
Step 4:      Remove an edge from Q
Step 5: IF the edge obtained in Step 4 connects two different trees,
        then Add it to the forest (for combining two trees into one
        tree).
        ELSE
             Discard the edge
Step 6: END
```

Kruskal's Algorithm

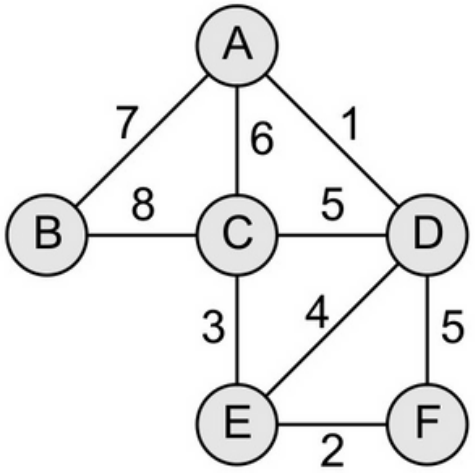


Initially, we have F = {{A}, {B}, {C}, {D}, {E}, {F}}

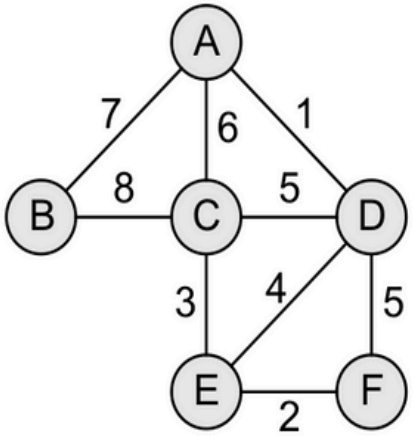
MST = {}

Q = {(A, D), (E, F), (C, E), (E, D), (C, D), (D, F),
(A, C), (A, B), (B, C)}

Kruskal's Algorithm



Step 1: Remove the edge `(A, D)` from `Q` and make the following changes:



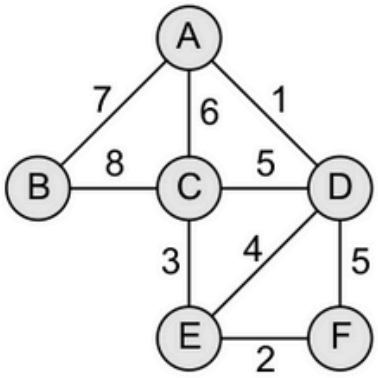
F = {{A, D}, {B}, {C}, {E}, {F}}

MST = {A, D}

Q = {(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

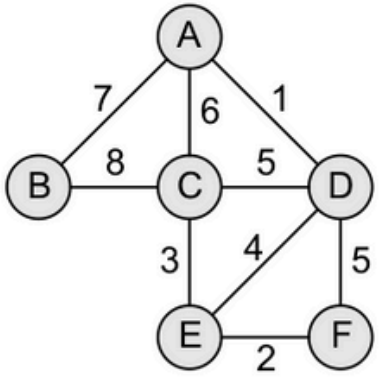
Kruskal's Algorithm

Step 2: Remove the edge `(E, F)` from `Q` and make the following changes:



F = {{A, D}, {B}, {C}, {E, F}}
MST = {(A, D), (E, F)}
Q = {(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

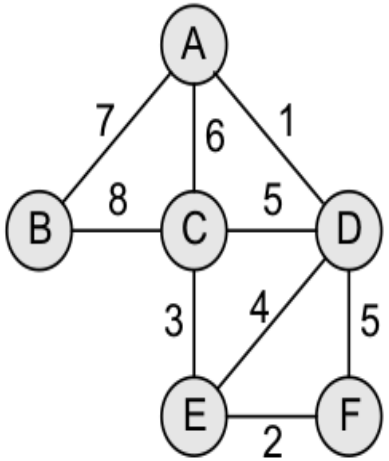
Step 3: Remove the edge `(C, E)` from `Q` and make the following changes:



F = {{A, D}, {B}, {C, E, F}}
MST = {(A, D), (C, E), (E, F)}
Q = {(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Kruskal's Algorithm

Step 4: Remove the edge (E, D) from Q and make the following changes:



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

MST = {(A, D), (C, E), (E, F), (E, D)}

Q = {(C, D), (D, F), (A, C), (A, B), (B, C)}

Kruskal's Algorithm

Step 5: Remove the edge (C, D) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST. Therefore,

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(D, F), (A, C), (A, B), (B, C)\}$$

Step 6: Remove the edge (D, F) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

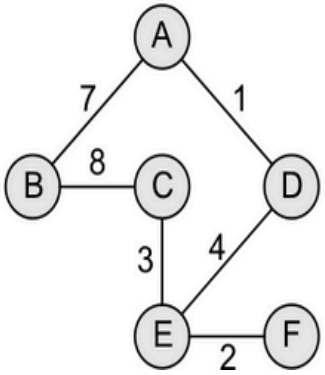
$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(A, C), (A, B), (B, C)\}$$

Kruskal's Algorithm

Step 7: Remove the edge (A, C) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(A, B), (B, C)\}$$

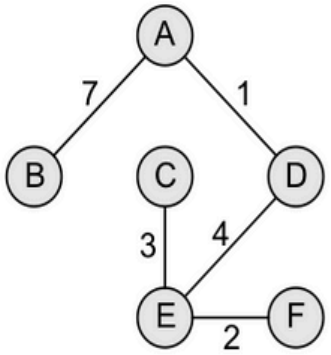
Step 8: Remove the edge (A, B) from Q and make the following changes:



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {(B, C)}

Kruskal's Algorithm

Step 9: The algorithm continues until Q is empty. Since the entire forest has become one t
all the remaining edges will simply be discarded. The resultant MS can be given as shown bel



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {}

Reference

This presentation is based on and adapted from lecture materials by Dr.Shweta Saharan

- Visualizations and animations from🔗 Visualgo – An interactive visualizer for data structures and algorithms
- Content organized and customized by Abhinav Kesarwani, B.Tech AI&DS Student, GSV

Thank You