Graphs

By Asst Prof Christopher Uz

Course: DSAA

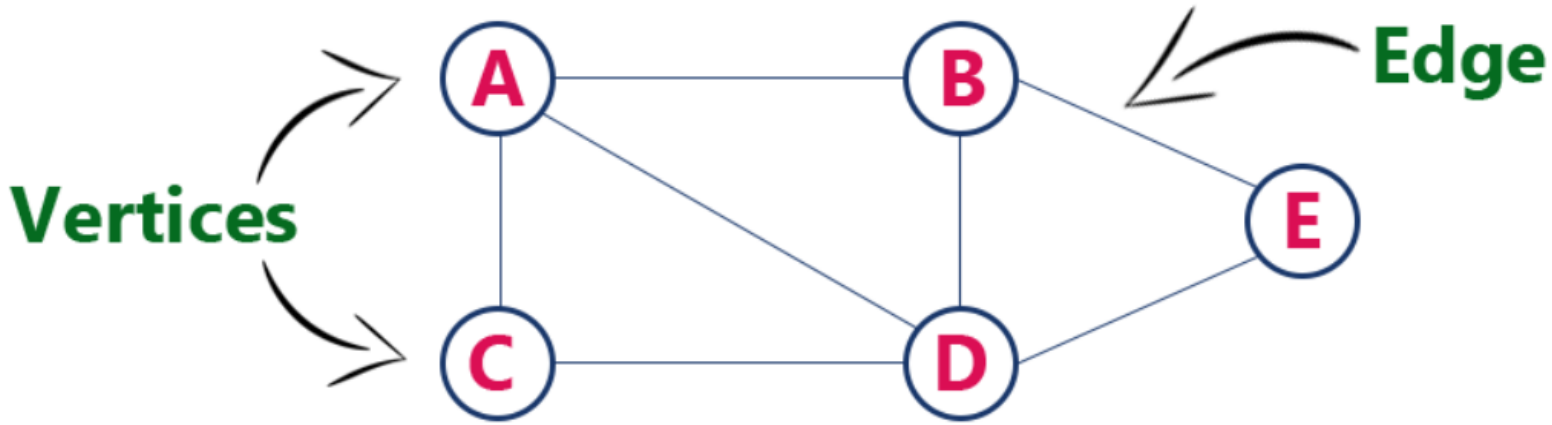
Pillai College of Engineering

Introduction to Graphs

- A graph can be defined as *group of vertices and edges* that are used to connect these vertices
- A graph can be seen as a cyclic tree, where the vertices (nodes) maintain any complex relationship among them instead of having parent child relationship
- E.g., On Facebook
  - Everything is a node. That includes User, Photo, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node
  - Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship
  - All of facebook is a collection of these nodes and edges because facebook uses a graph data structure to store its data

Introduction to Graphs…

- A graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**
- The graph below has 5 vertices and 7 edges.
- This graph G can be defined as G = (V , E) where
- V = {A,B,C,D,E}
- E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology

- **Adjacent Nodes:** If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbors or adjacent nodes
- **Degree of the Node:** A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as *isolated node*
- **Path:** A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U
- **Closed Path:** A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$
- **Simple Path:** If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path

Graph Terminology…

- **Cycle:** A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices

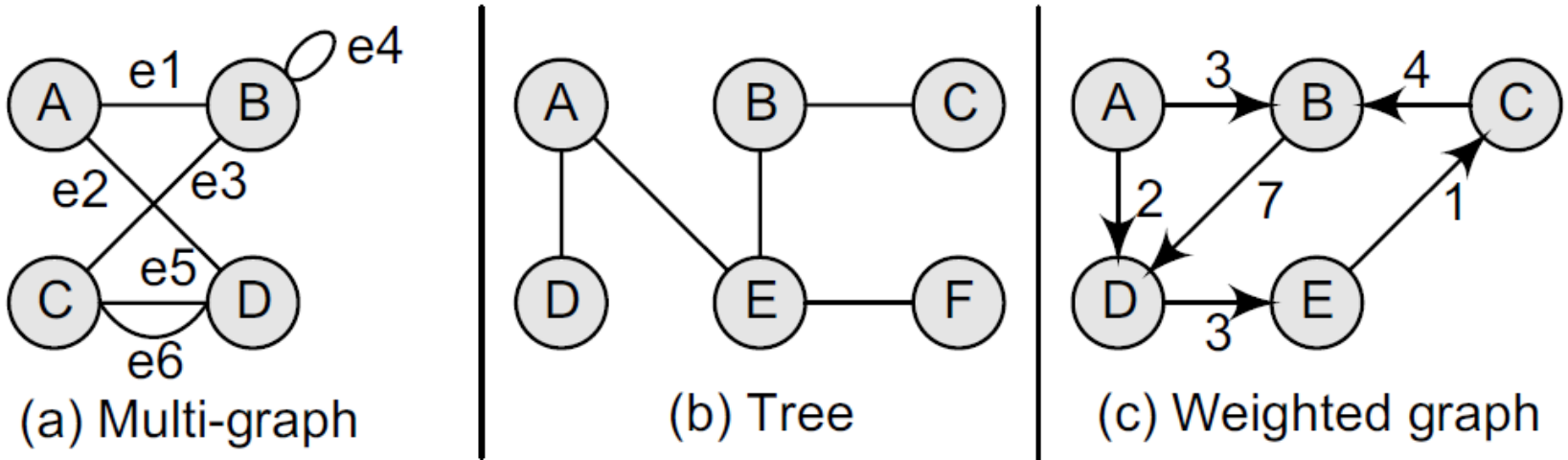
- **Connected Graph:** A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph. A connected graph that does not have any cycle is called a tree

- **Complete Graph:** A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph

- **Weighted Graph:** In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive(+) value indicating the cost of traversing the edge

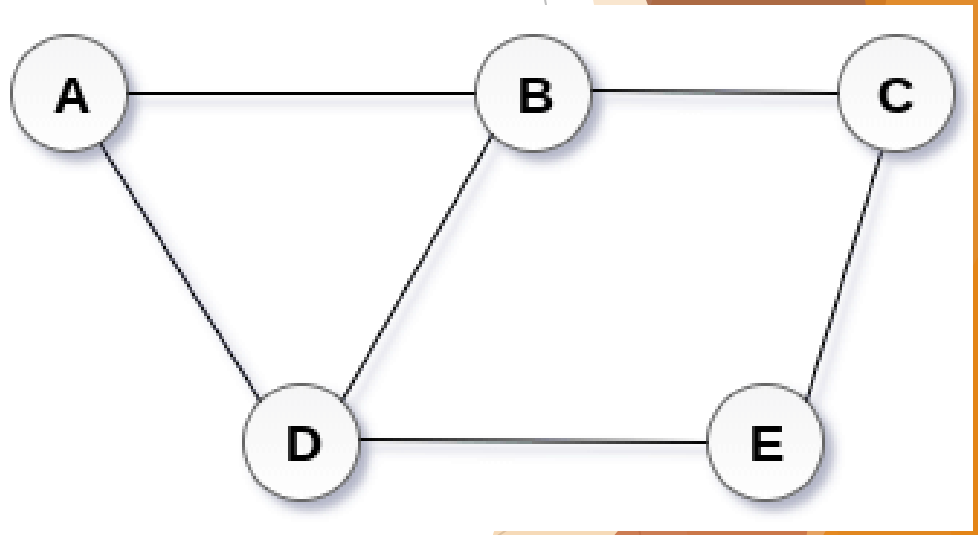
Graph Terminology…

- **Loop:** An edge that is associated with the similar end points can be called as Loop
- **Edge:** An edge is a connecting link between two vertices. The 3 types of edges are *Undirected Edge*, *Directed Edge* and *Weighted Edge*
- **Multi-graph:** A graph with *multiple edges* and/or loops is called a multi-graph



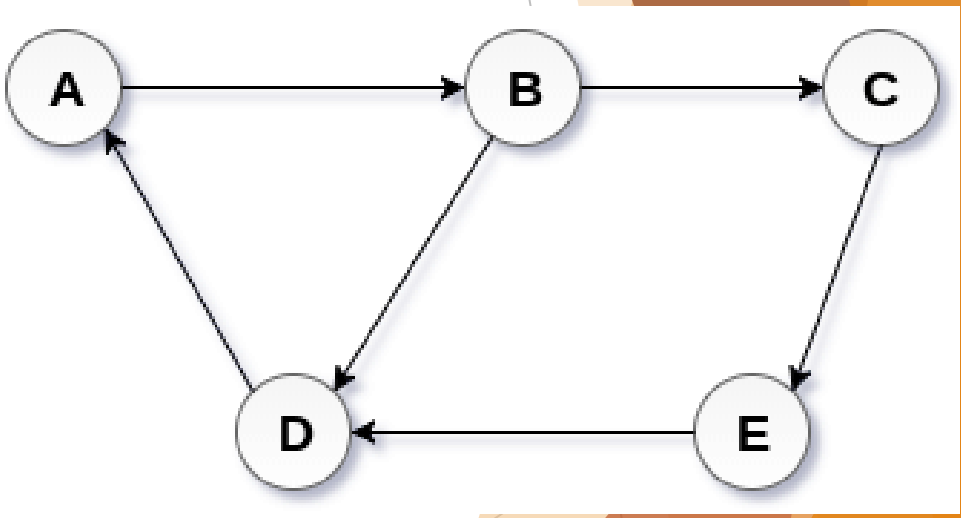
Undirected Graph

- A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge
- The order of the two connected vertices is unimportant
- If an edge exists between vertex A and B, then the vertices can be traversed from B to A as well as A to B

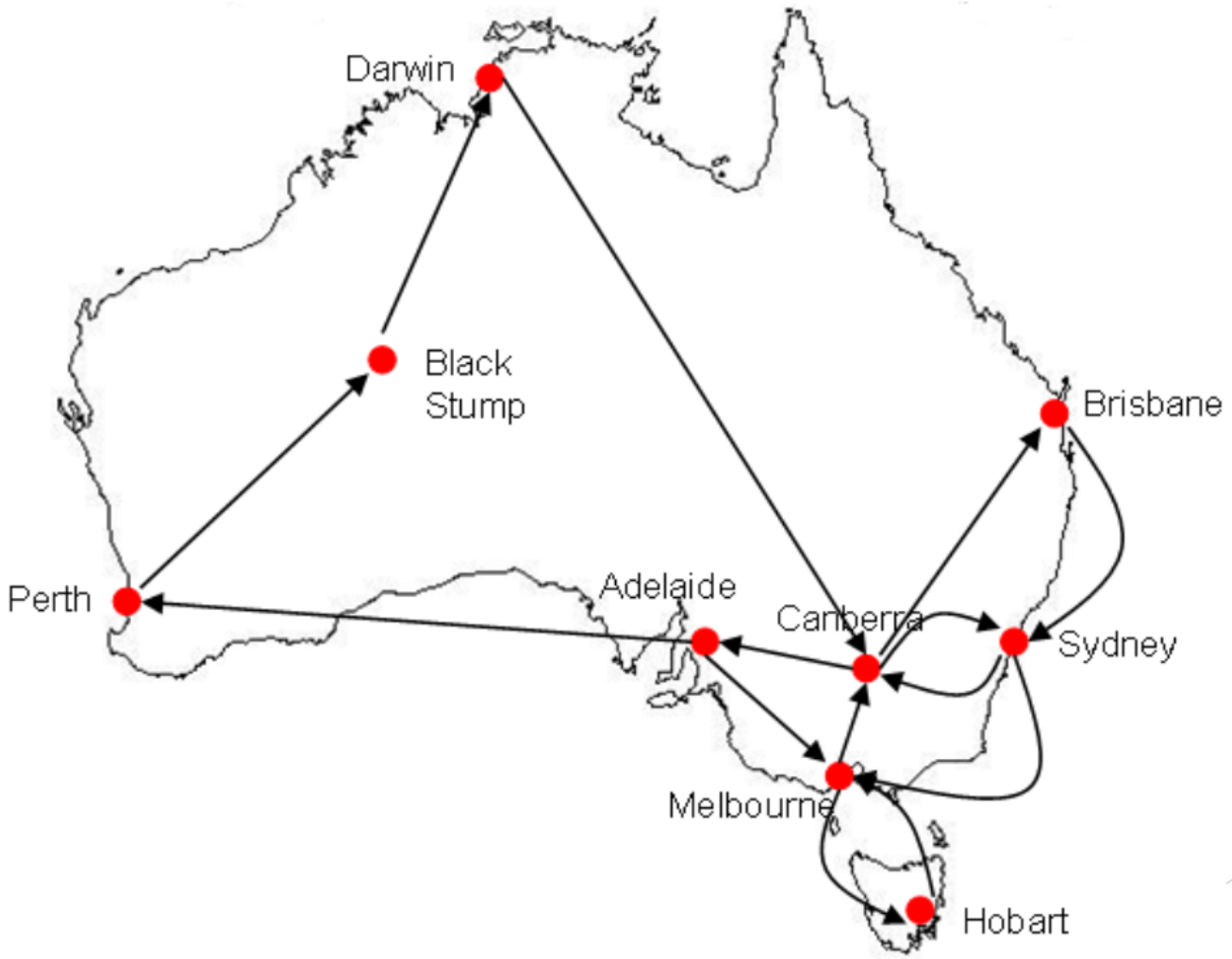


Directed Graph(Digraph)

- A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge
- Each edge is associated with two vertices, called its **source** and **target** vertices
- The **order** of the two connected vertices is important



Directed Graph: Airline Routing Example



Graph Representation

- Different kinds of graphs require different kinds of implementations, but the fundamental concepts of all graph implementations are similar

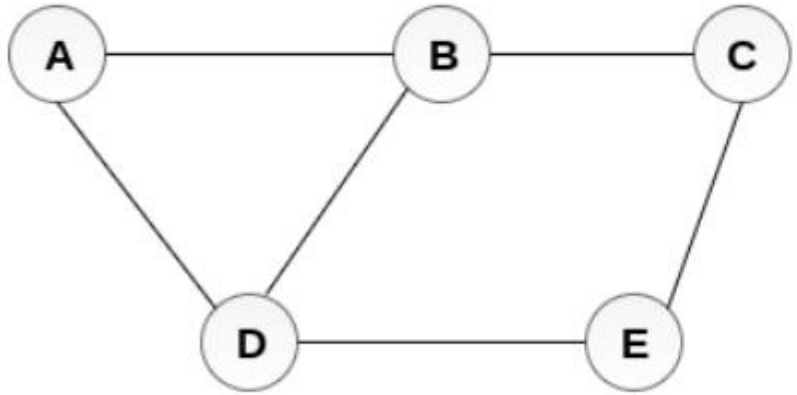
- There are two ways to store a graph:
  - Adjacency Matrix
  - Adjacency List

Adjacency Matrix

- In adjacency matrix, the rows and columns are represented by the graph vertices
- A graph having n vertices, will have a dimension **n x n**
- This matrix is filled with either 1 or 0
- Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex

Adjacency Matrix for Undirected graph

- An entry M_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between V_i and V_j



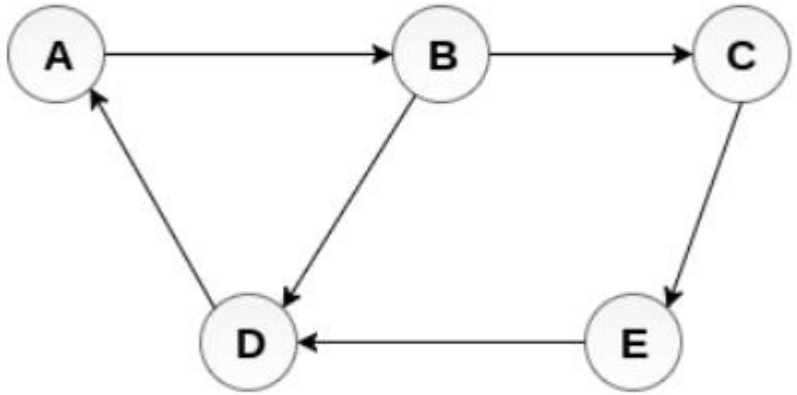
Undirected Graph

	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

Adjacency Matrix

Adjacency Matrix for Directed graph

- The adjacency matrices for the directed and undirected graph are different
- In directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j



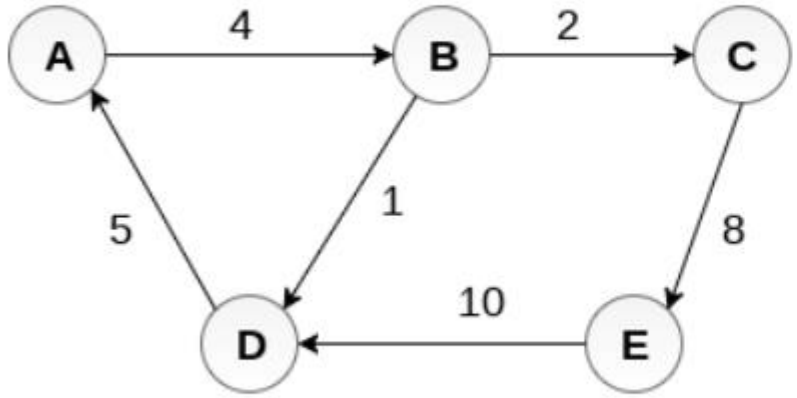
Directed Graph

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

Adjacency Matrix

Adjacency Matrix for Weighted graph

- Representation of weighted directed graph is different
- Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges



Weighted Directed Graph

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

Adjacency Matrix

Pros of Adjacency Matrix

- The basic operations like adding an edge, removing an edge, and checking whether there is an edge from vertex i to vertex j are extremely time efficient, constant time operations

- If the graph is dense and the number of edges is large, an adjacency matrix should be the first choice. Even if the graph and the adjacency matrix is sparse, we can represent it using data structures for sparse matrices

- The biggest advantage, however, comes from the use of matrices. The recent advances in hardware enable us to perform even expensive matrix operations on the GPU

Cons of Adjacency Matrix

- The *VxV* space requirement of the adjacency matrix makes it a memory hog. Graphs out in the wild usually don't have too many connections and this is the major reason why adjacency lists are the better choice for most tasks

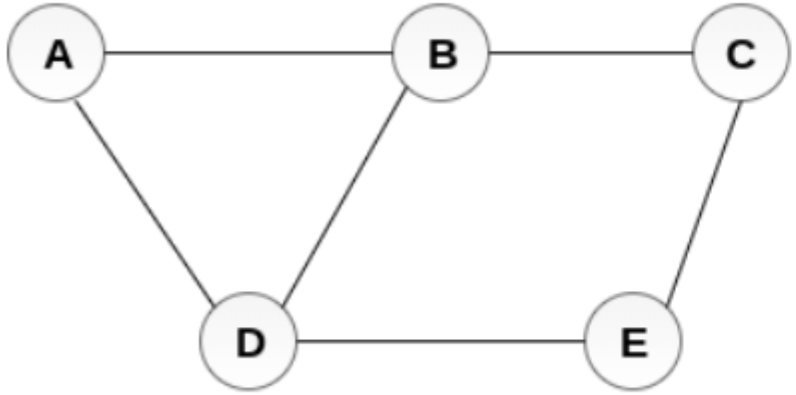
- While basic operations are easy, operations like *inEdges* and *outEdges* are expensive when using the adjacency matrix representation

Adjacency List

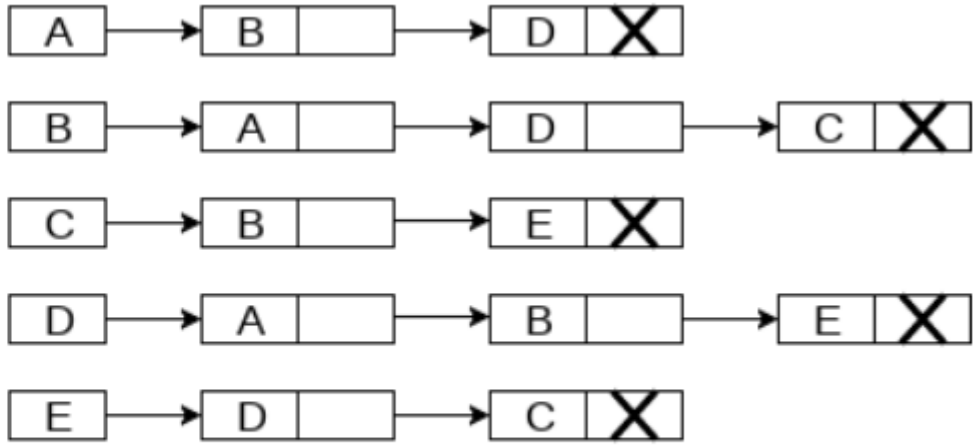
- An adjacency list represents a graph as an array of linked lists
- The *index of the array represents a vertex* and each element in its linked list represents the *other vertices that form an edge with the vertex*
- i.e., every vertex of a graph contains list of its adjacent vertices
- An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node
- If all the adjacent nodes are traversed, then store the NULL in the pointer field of last node of the list

Adjacency List for Undirected graph

- The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph



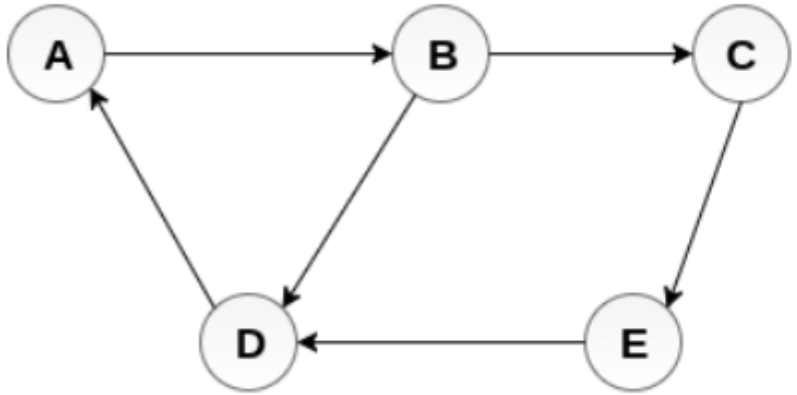
Undirected Graph



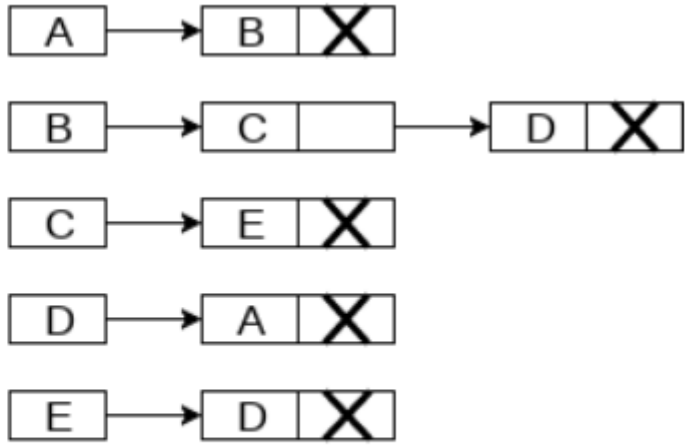
Adjacency List

Adjacency List for Directed graph

- In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph



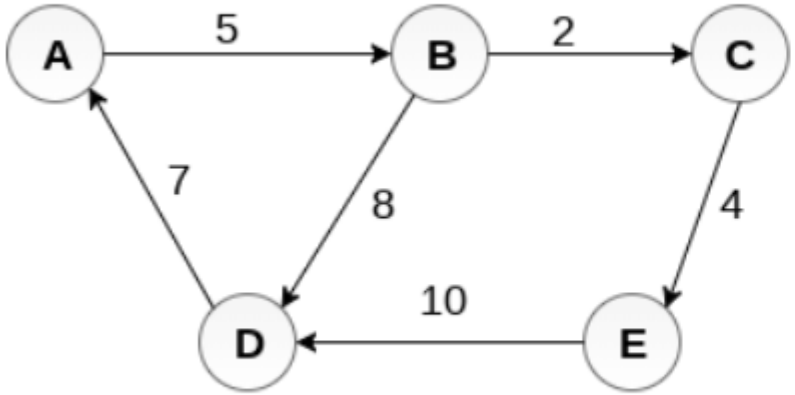
Directed Graph



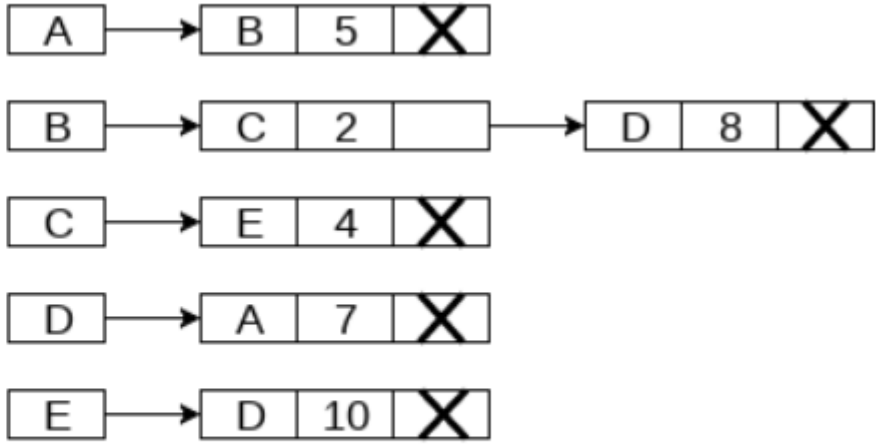
Adjacency List

Adjacency List for Weighted graph

- In the case of weighted directed graph, each node contains an extra field that is called the weight of the node



Weighted Directed Graph



Adjacency List

Pros of Adjacency List

- An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a sparse graph with millions of vertices and edges, this can mean a lot of saved space

- It also helps to find all the vertices adjacent to a vertex easily

Cons of Adjacency List

- Finding the adjacent list is not quicker than the adjacency matrix because all the connected nodes must be first explored to find them

Which representation is best Adjacency matrix or Adjacency list?

- If the space is available, then an adjacency matrix is easier to implement and is generally easier to use than adjacency lists
- An adjacency list is efficient in terms of storage because we only need to store the values for the edges
- For a graph with millions of vertices, this can mean a lot of saved space
- In general, your choice of representations should be based on your expectations as to which operations are most frequent
- If each vertex has only a few edges (sparse graph), then an adjacency matrix is mostly wasted space filled with the value **0**

Graph Traversal

Introduction to Graph Traversal

- Graph traversal is a technique used for a searching vertex in a graph
- The graph traversal is also used *to decide the order of vertices is visited* in the search process
- A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path
- The two popular graph traversal techniques are
  - **DFS** (Depth First Search)
  - **BFS** (Breadth First Search)

Introduction to Graph Traversal…

- Breadth First Search uses a queue as an auxiliary data structure to store nodes for further processing while 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 or 2, depending on its current state
- The table below shows the value of *STATUS* and its significance

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

DFS (Depth First Search)

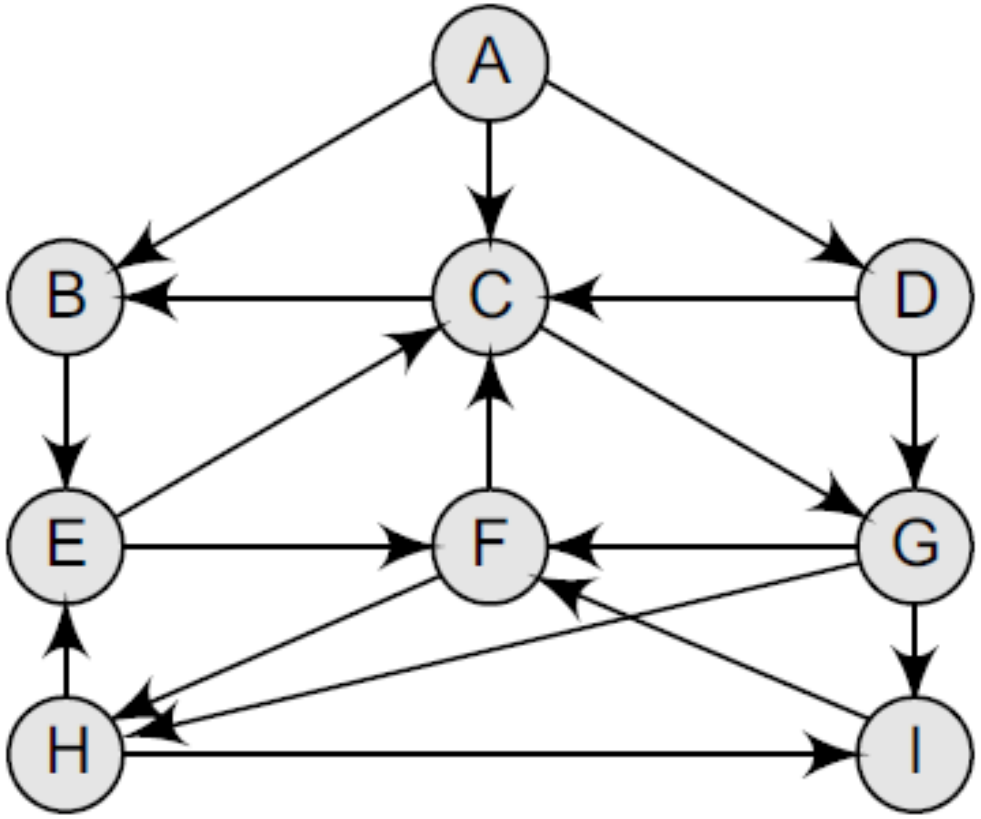
- Depth First Search (DFS) algorithm starts with the initial node of the graph G, and then goes deeper and deeper until we find the goal node or the node which has no children
- The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored
- The data structure which is being used in DFS is **stack**
- In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges

DFS (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

DFS Sample Graph

- Suppose we want to print all the nodes that can be reached from the 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

DFS Sample Graph…

STACK: H

PRINT: H

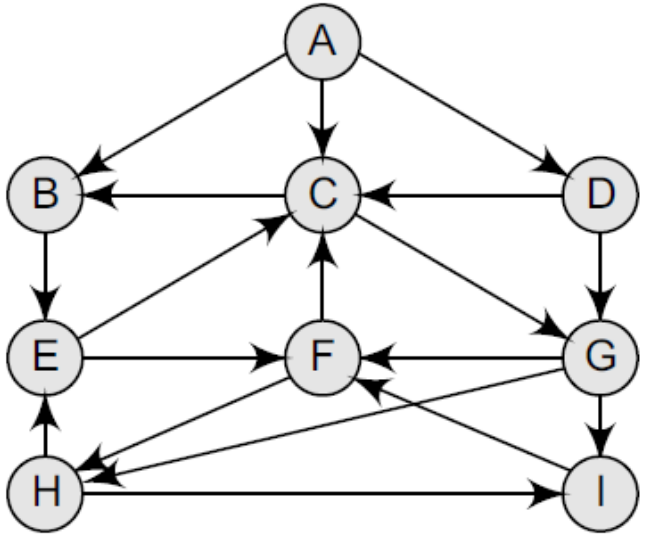
STACK: E, I

PRINT: I

STACK: E, F

PRINT: F

STACK: E, C

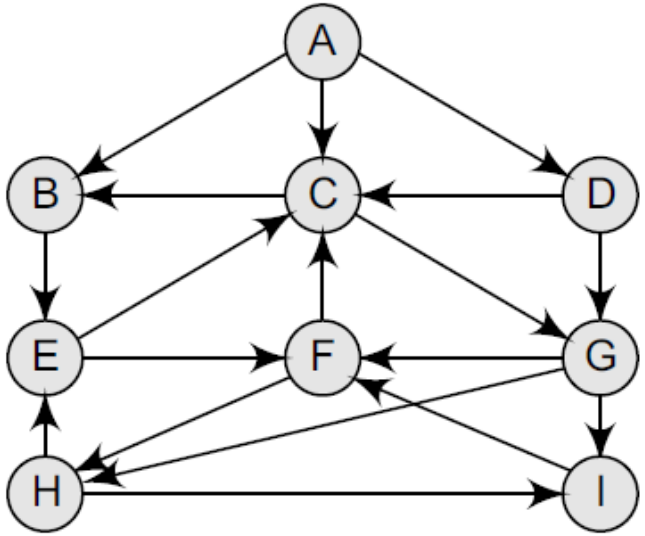


Adjacency lists

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

DFS Sample Graph…

PRINT: C	STACK: E, B, G
PRINT: G	STACK: E, B
PRINT: B	STACK: E
PRINT: E	STACK:



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

H, I, F, C, G, B, E are the nodes which are reachable from the node H

Features of DFS Algorithm

- **Space complexity** The space complexity of a depth-first search is lower than that of a breadth first search

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

- **Completeness** Depth-first search is said to be a complete algorithm. If there is a solution, depth first search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge

Applications of DFS Algorithm

- 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
- *NOTE:* Spanning tree is a tree that connects all the vertices of a graph with the minimum possible number of edges

BFS (Breadth First Search)

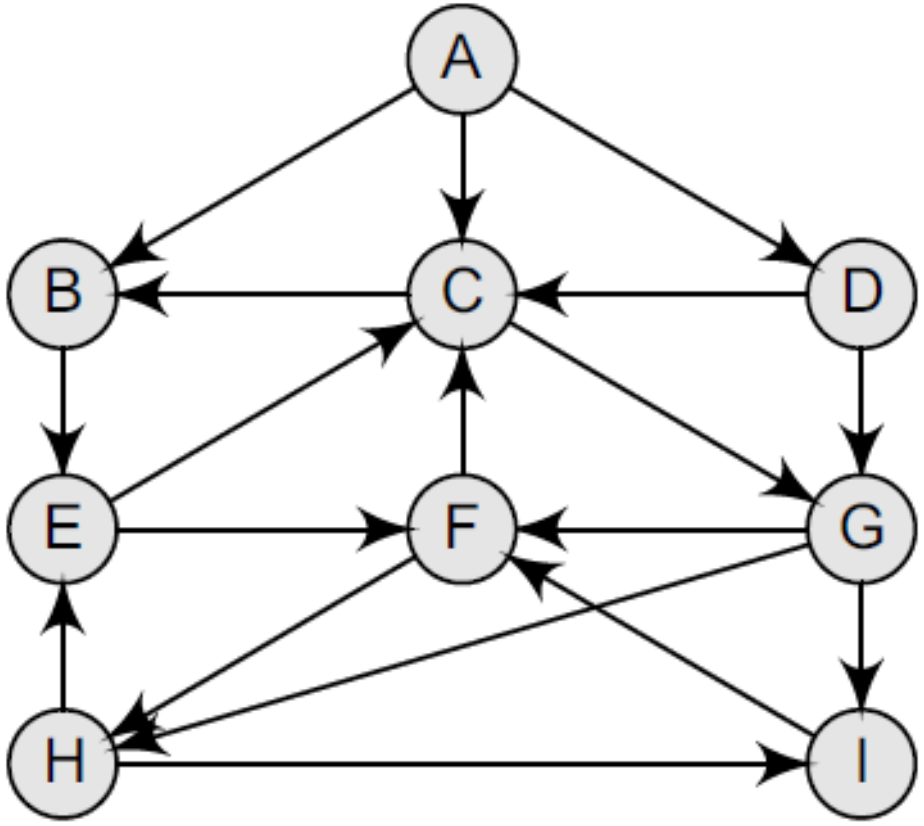
- Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes
- Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal
- The algorithm starts with examining the node A and all of its neighbours
- In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps
- The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice

BFS (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

BFS Sample Graph

- Assume that G represents the daily flights between different cities, and we want to fly from city A to I with minimum stops



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

BFS Sample Graph…

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

- We use 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
- Add A to QUEUE and add NULL to ORIG

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

- 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

BFS Sample Graph…

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

- Dequeue a node 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

- Dequeue a node and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G

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

BFS Sample Graph…

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

- 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

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

- 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

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

BFS Sample Graph…

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

- 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

- 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

Features of BFS Algorithm

- **Space complexity** 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)
- **Time complexity** The time complexity can also be expressed as O(E + V), since every vertex and every edge will be explored in the worst case
- **Completeness** Breadth-first search is said to be a complete algorithm. If there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge
- **Optimality** Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node

Applications of BFS Algorithm

- 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

Applications of Graphs

- In Computer science graphs are used to represent the flow of computation
- Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge
- In Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph
- Dijkstra's Algorithm, Minimum Spanning Tree: Prim's Algorithm, Kruskal's Algorithm are also some of the applications of graphs that will be covered in later slides

Minimum Spanning Tree (MST)

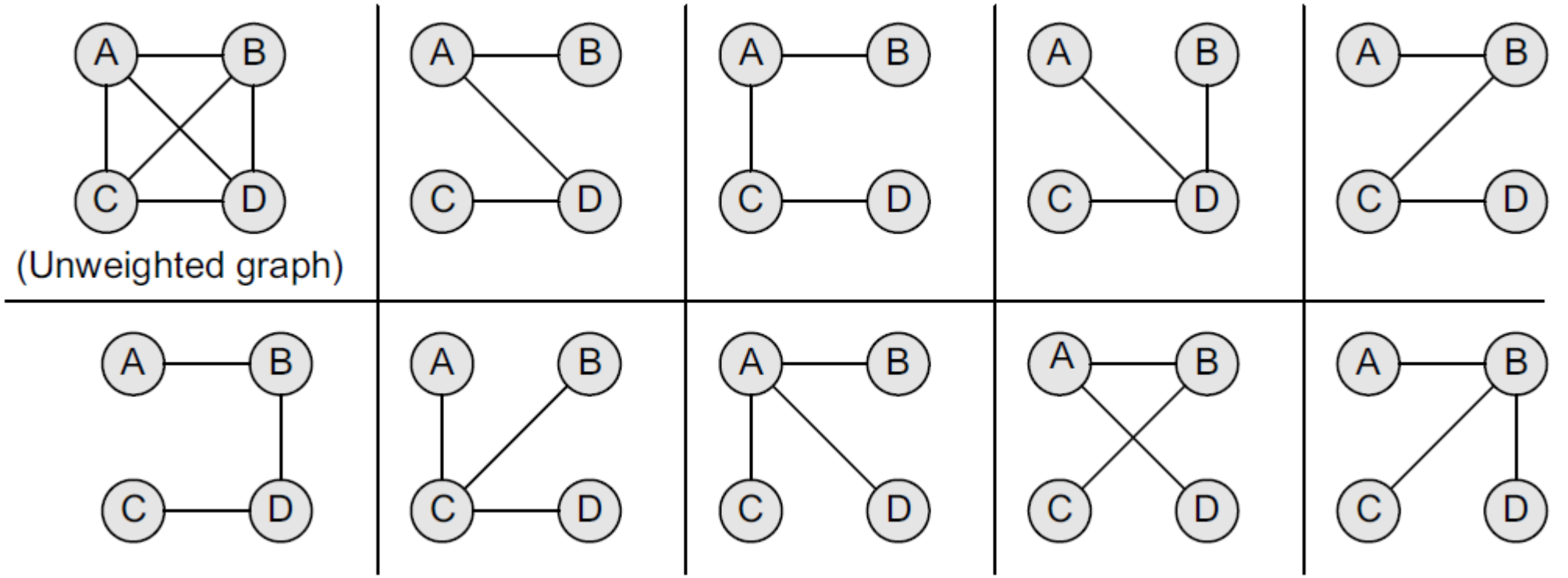
- A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together
- A graph G can have many different spanning trees
- We can assign weights to each edge, and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree
- 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

Properties of Minimum Spanning Tree

- **Possible multiplicity**
  - There can be multiple minimum spanning trees of the same weight. Particularly, if all the weights are the same, then every spanning tree will be minimum.
- **Uniqueness**
  - When each edge in the graph is assigned a different weight, then there will be only one unique minimum spanning tree
- **Minimum-cost subgraph**
  - If the edges of a graph are assigned non-negative weights, then a minimum spanning tree is in fact the minimum-cost subgraph or a tree that connects all vertices
- **Cycle property**
  - If there exists a cycle C in the graph G that has a weight larger than that of other edges of C, then this edge cannot belong to an MST
- **Usefulness**
  - Minimum spanning trees can be computed quickly and easily to provide optimal solutions. These trees create a sparse subgraph that reflects a lot about the original graph
- **Simplicity**
  - The minimum spanning tree of a weighted graph is nothing but a spanning tree of the graph which comprises of n–1 edges of minimum total weight. Note that for an unweighted graph, any spanning tree is a minimum spanning tree

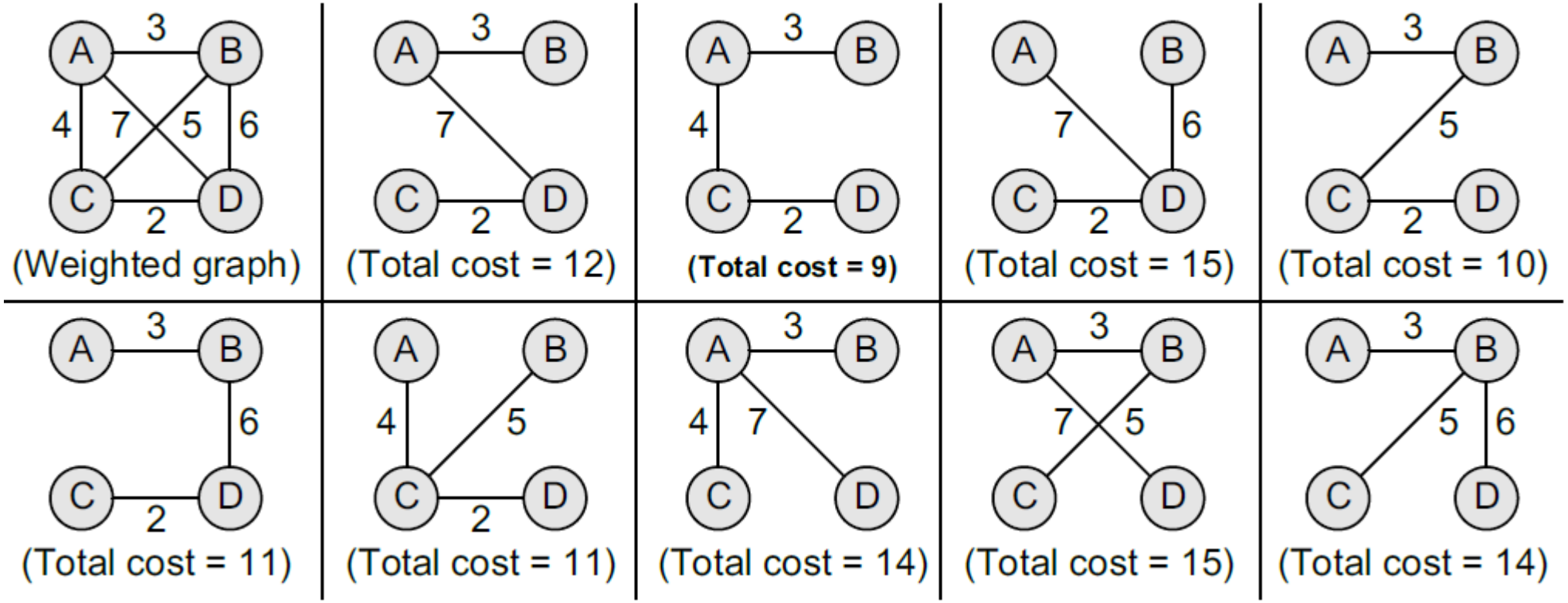
Find MST for Unweighted graph

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



Find MST for Weighted graph

- Consider a weighted graph G shown in below figure. From G, we can draw three distinct spanning trees. But only a single minimum spanning tree can be obtained, that is, the one that has the minimum weight (cost) associated with it



Applications of Minimum Spanning Trees

- MSTs are widely used for designing networks. A minimum spanning tree is used to determine the least costly paths with no cycles in this network, thereby providing a connection that has the minimum cost involved
- MSTs are used to find airline routes. While the vertices in the graph denote cities, edges represent the routes between these cities. Therefore, MSTs are used to optimize airline routes by finding the least costly path with no cycles.
- 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.

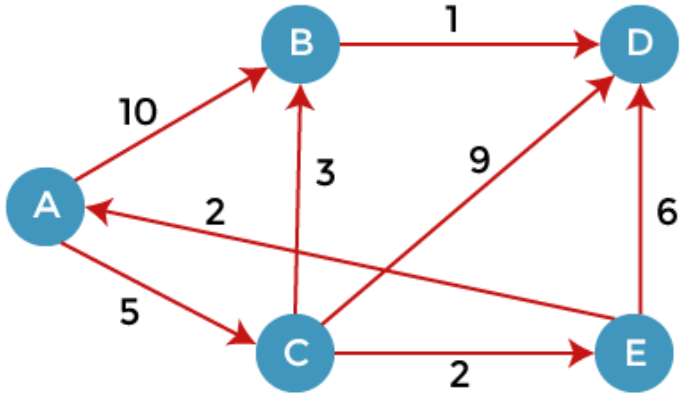
Dijkstra's Algorithm

Dijkstra’s Algorithm

- Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph
- It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph
- Dijkstra algorithm is a single-source shortest path algorithm. Here, single-source means that only one source is given, and we have to find the shortest path from the source to all the nodes
- The distance between the vertices can be calculated by using the formula:
- $d(x, y) = d(x) + c(x, y) < d(y)$
- This technique of finding shortest path is known as relaxation

Dijkstra's Algorithm Sample

- Consider the directed graph where A is the source vertex and we find the shortest path to all vertexes



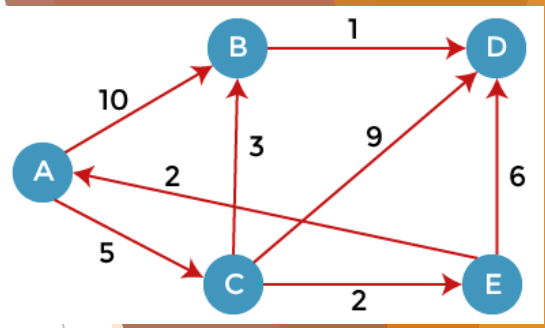
- A vertex is a source vertex so entry is filled with 0 while other vertices filled with ∞

	A	B	C	D	E
A	0	∞	∞	∞	∞

- If(d(x) + c(x, y) < d(y)) then we update d(y) = d(x) + c(x, y)

	A	B	C	D	E
A	0	∞	∞	∞	∞
		10	5	∞	∞

Dijkstra’s Algorithm Sample…

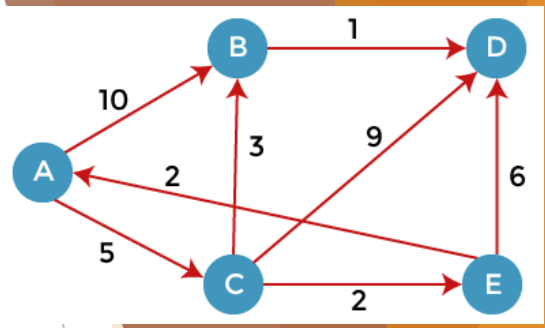


- We compare the vertices to find the vertex with the lowest value. Since the vertex C has the minimum value, i.e., 5 so vertex C will be selected
- The direct paths from the vertex C are C to B, C to D, and C to E

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
		8		14	

- We calculate the distance from C to B. Consider vertex C as 'x' and vertex B as 'y’
- d(x, y) = d(x) + c(x, y) < d(y) = (5 + 3) < 10 = 8 < 10
- We calculate the distance from C to D. Consider vertex C as 'x' and vertex D as 'y'
- d(x, y) = d(x) + c(x, y) < d(y) = (5 + 9) < ∞ = 14 < ∞

Dijkstra’s Algorithm Sample…

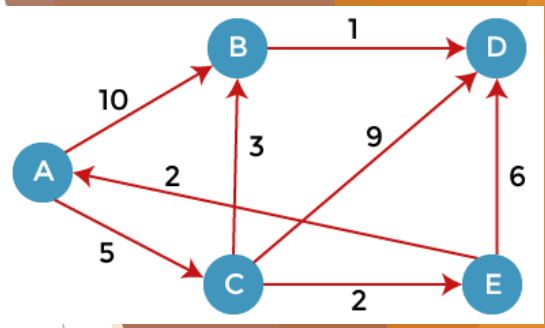


- We calculate the distance from C to E. Consider vertex C as 'x' and vertex E as 'y'
- d(x, y) = d(x) + c(x, y) < d(y) = (5 + 2) < ∞ = 7 < ∞

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7

- In the above table that 7 is the minimum value among 8, 14, and 7. Therefore, the vertex E is added on the left as shown in the above table
- The vertex E is selected so we consider all the direct paths from the vertex E
- The direct paths from the vertex E are E to A and E to D. Since the vertex A is selected, so we will not consider the path from E to A

Dijkstra's Algorithm Sample…

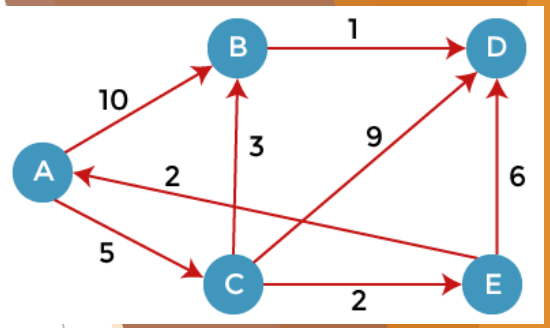


- Consider the path from E to D
- d(x, y) = d(x) + c(x, y) < d(y) = (7 + 6) < 14 = 13 < 14

A		B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	

- The value 8 is minimum among 8 and 13. Therefore, vertex B is selected. The direct path from B is B to D
- d(x, y) = d(x) + c(x, y) < d(y) = (8 + 1) < 13 = 9 < 13

Dijkstra’s Algorithm Sample…



- Since 9 is less than 13 so we update d(D) from 13 to 9. The value 9 will be added under the D column

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	
D				9	

Dijkstra's Algorithm Applications

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map

Bellman Ford Algorithm

Bellman Ford Algorithm

- Bellman ford algorithm is a single-source shortest path algorithm
- This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph
- If the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not
- In contrast to Dijkstra algorithm, bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values
- Both Dijkstra & Bellman ford algorithm fail on graphs with negative cycles

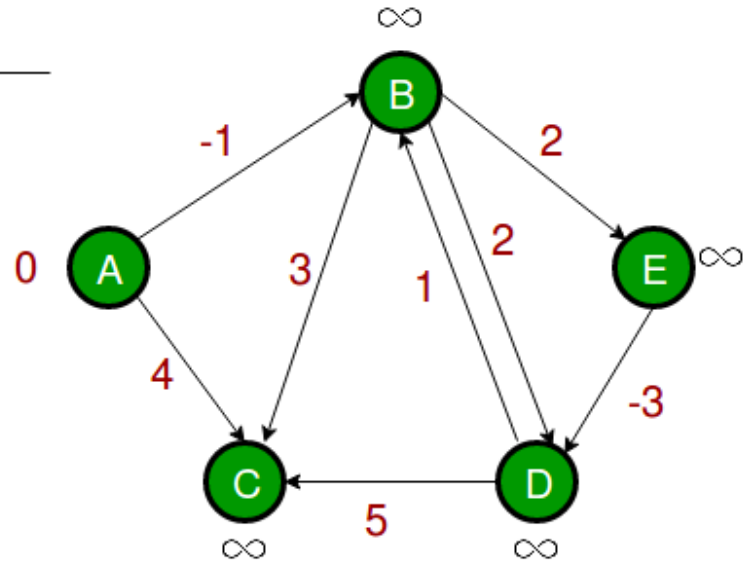
How Bellman Ford Algorithm works

- Like other Dynamic Programming Problems, the algorithm calculates the shortest paths in a bottom-up manner
- It first calculates the shortest distances which have at most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on
- After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |V| – 1 times
- The idea is, assuming that there is no negative weight cycle if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give the shortest path with at-most (i+1) edges

Bellman Ford Algorithm Sample

- Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times

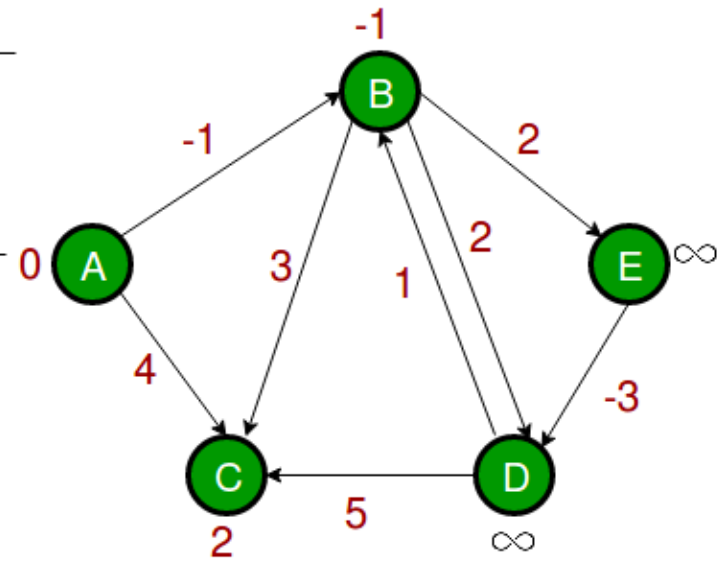
A	B	C	D	E
0	∞	∞	∞	∞



Bellman Ford Algorithm Sample…

- The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed

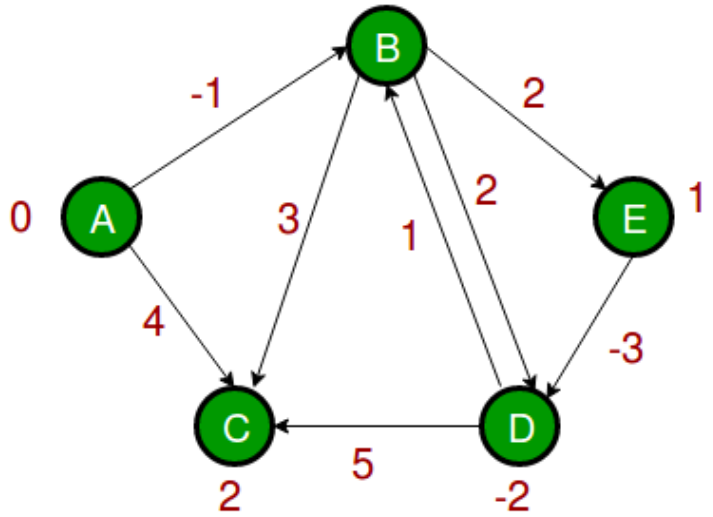
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



Bellman Ford Algorithm Sample…

- The first iteration guarantees to give all shortest paths which are at most 1 edge long. The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



Prim's Algorithm

Prim's Algorithm

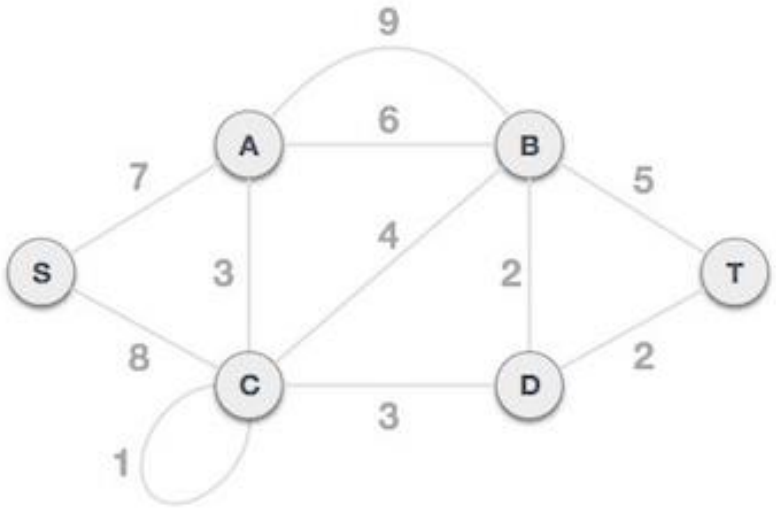
- Prim’s algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph
- In other words, the algorithm builds a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized
- We start from one vertex and keep adding edges with the lowest weight until we reach our goal

How Prim’s Algorithm works

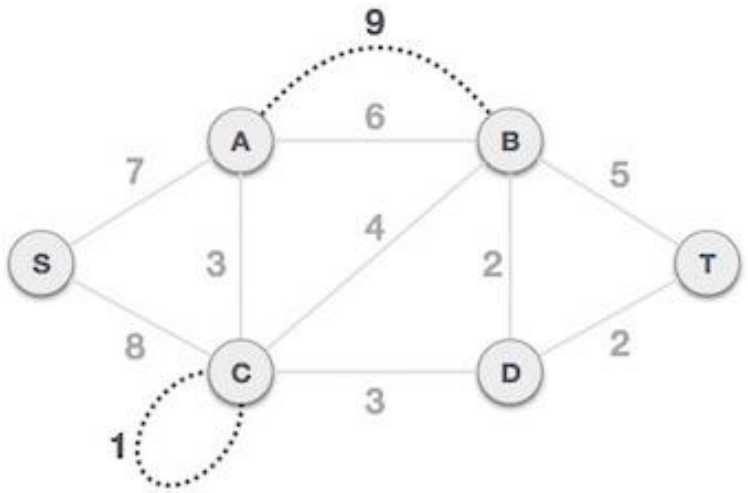
- The steps for implementing Prim's algorithm are as follows:
  1. Initialize the minimum spanning tree with a vertex chosen at random
  2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
  3. Keep repeating step 2 until we get a minimum spanning tree

Prim’s Algorithm Sample

- Construct a minimum spanning tree of the graph given below. Start the Prim’s algorithm from vertex S

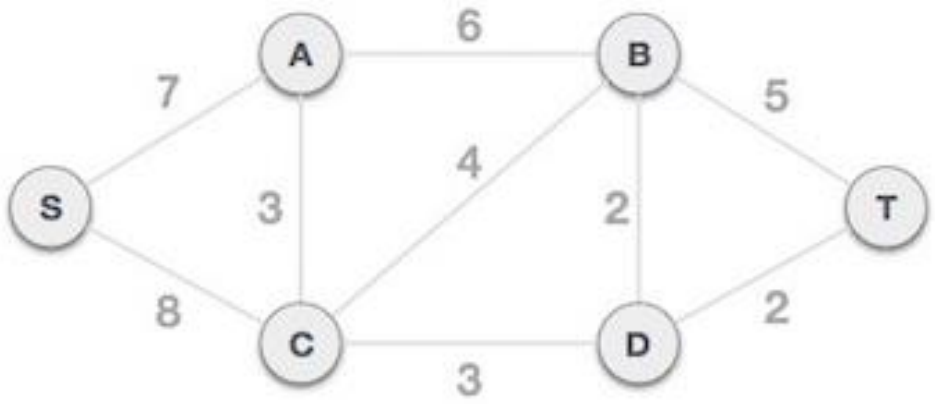


- **Remove all loops and parallel edges**
- In case of parallel edges, keep the one which has the least cost associated and remove all others

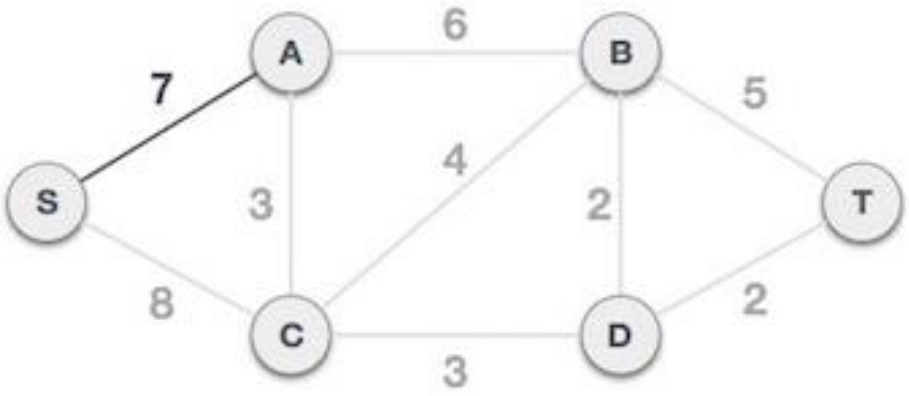


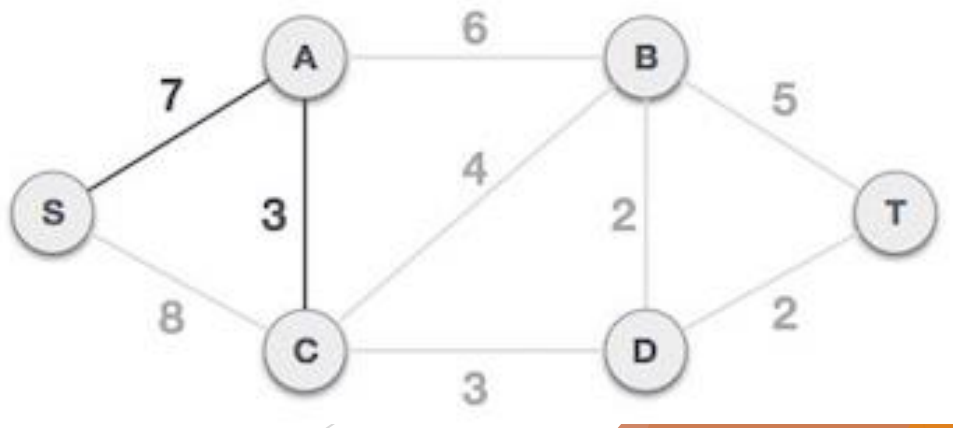
Prim’s Algorithm Sample…

- **Choose any arbitrary node as root node**
- In this case, we choose S node as the root node of Prim's spanning tree



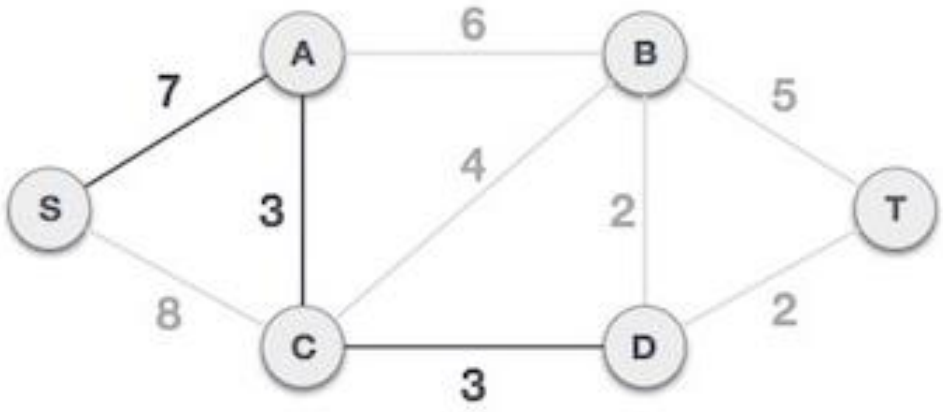
- **Check outgoing edges and select the one with less cost**

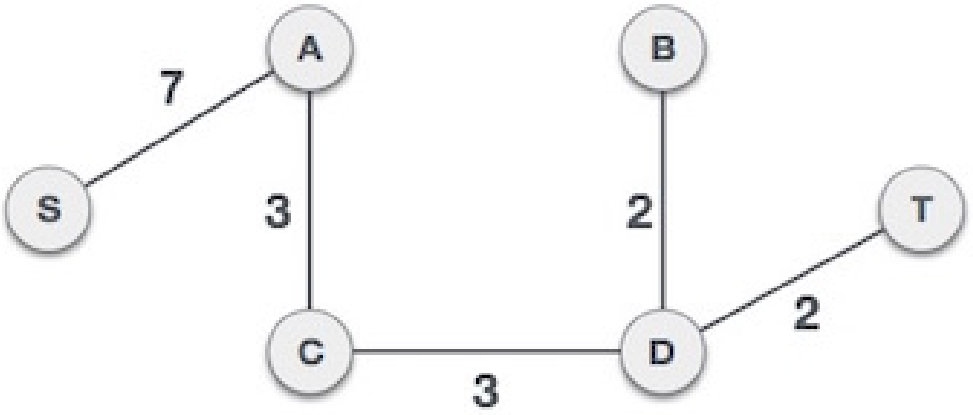




Prim's Algorithm Sample…

- We select the one which has the lowest cost and include it in the tree
- After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one
- In the next step will again yield edge 2 as the least cost again





Kruskal's Algorithm

Kruskal's Algorithm

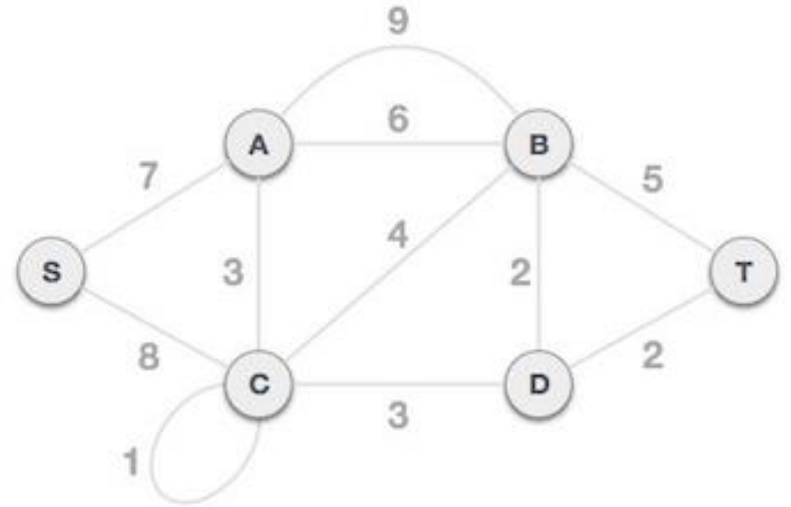
- Kruskal's Algorithm is a famous greedy algorithm
- It is used for finding the Minimum Spanning Tree (MST) of a given graph
- To apply Kruskal's algorithm, the given graph must be weighted, connected and undirected
- We start from the edges with the lowest weight and keep adding edges until we reach our goal

How Kruskal’s Algorithm works

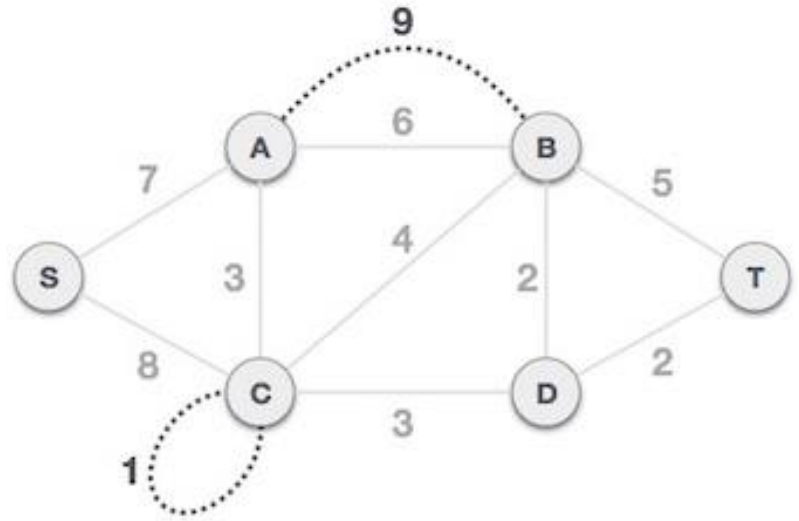
- The steps for implementing Kruskal's algorithm are as follows:
  1. Sort all the edges from low weight to high
  2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge
  3. Keep adding edges until we reach all vertices

Kruskal's Algorithm Sample

- Construct a minimum spanning tree of the graph given below using Kruskal's Algorithm

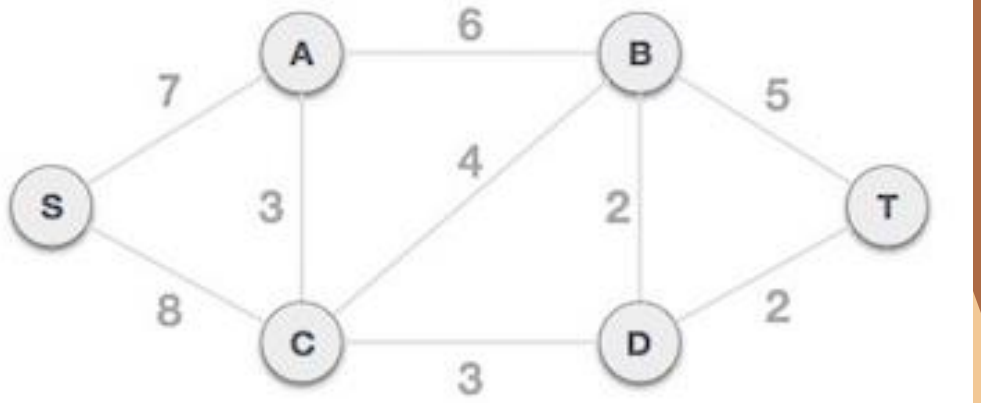


- **Remove all loops and parallel edges**
- In case of parallel edges, keep the one which has the least cost associated and remove all others



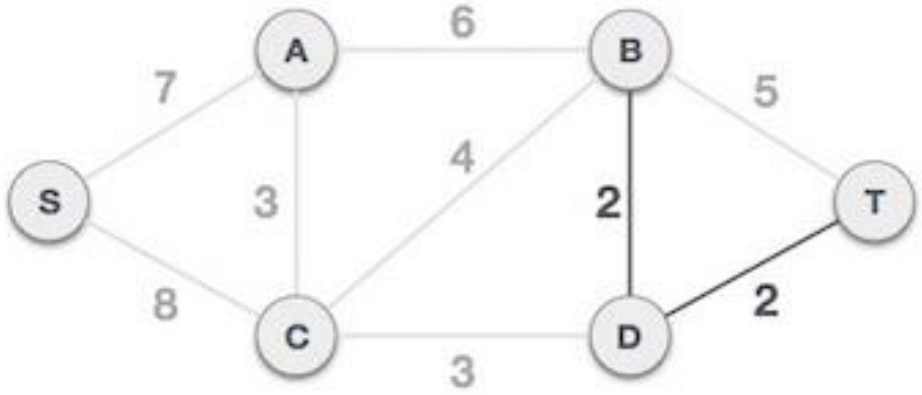
Kruskal's Algorithm Sample…

- Arrange all edges in their increasing order of weight



B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

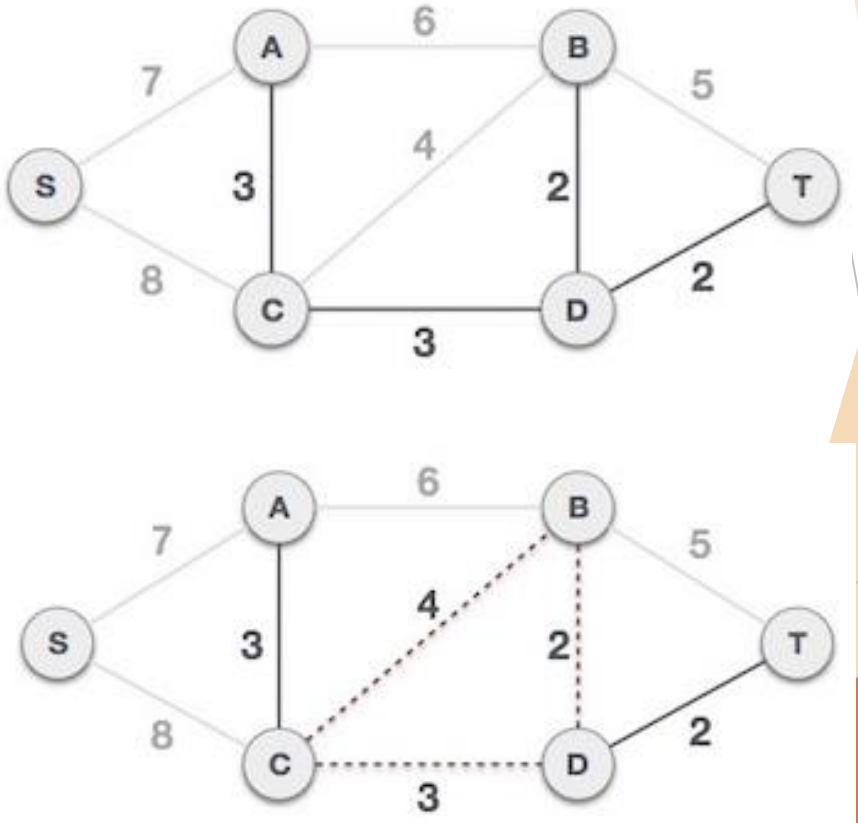
- Add the edge which has the least weightage



Kruskal's Algorithm Sample…

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

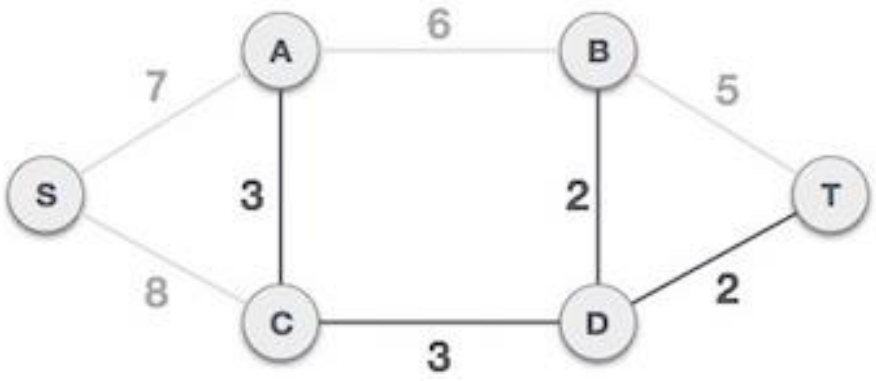
- Adding the edges does not violate spanning tree properties, so we continue to our next edge selection
- Next cost in the table is 4, and we observe that adding it will create a circuit in the graph
- Therefore, we ignore it. In the process we shall *ignore/avoid all edges that create a circuit*

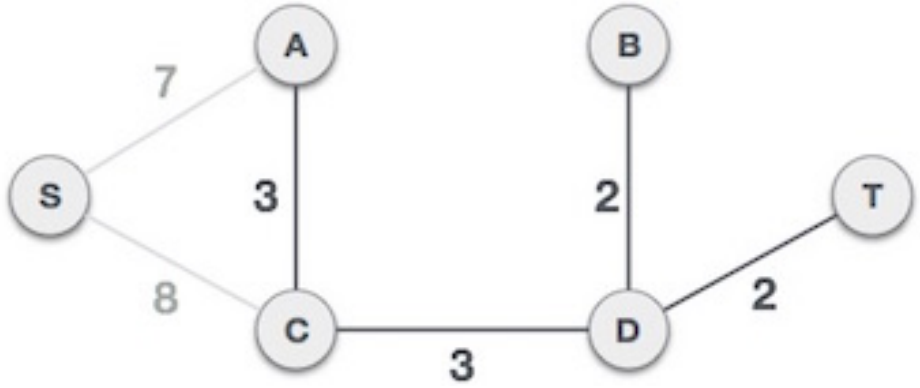


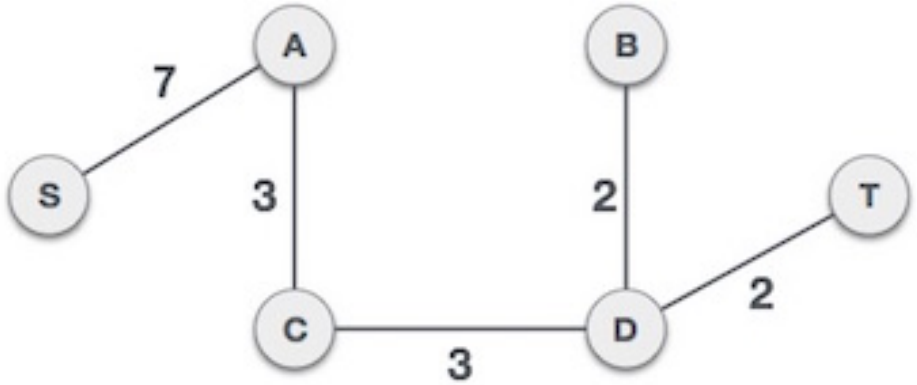
Kruskal's Algorithm Sample…

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

- We remove/ignore the edge with cost 4
- We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on







Any Questions?