Algorithms on Graph Data Structures

Hridyesh Singh Bisht

18070124030

Abstract- This article will be covering the basics of graph data structure, algorithms to represent them, traverse them, find a minimum spanning tree, finding the shortest path and sort them using topological sort.

Index Terms-
1. Graph Data Structure, Types of Graphs and Applications of Graph Data Structure
2. Ways to Represent Graphs :
    a. Edge List
    b. Adjacency Matrix
    c. Adjacency List
3. Ways to Traverse a graph:
    a. DFS Algorithm
    b. BFS Algorithm
4. Spanning Tree and Minimum Spanning Tree:
    a. Prim Algorithm
    b. Kruskal Algorithm
5. Shortest path in the graph:
    a. Dijkstra Algorithm
    b. Bellman-Ford Algorithm
6. Topological Sort:

I. INTRODUCTION

This article guides a stepwise walkthrough on what do you mean by graph data structures, types of graphs and applications of graphs. Then ways to represent graphs, ways to traverse a graph and its applications, ways to find the minimum spanning tree and it's applications, ways of finding the shortest path and its applications and sorting the graph topologically.

II. SECTION 2

This article guides a stepwise walkthrough by for writing an article. Number bulletins can be like this:
1) Read
2) For more information on the topic click on the references part at the bottom of each topic.

III. Section 3

1. Graph Data Structure:
    a. What is a graph data structure
    b. Types of graphs
    c. Applications of graph data structure
2. Ways to Represent Graphs :
    a. Edge list
    b. Adjacency matrix
    c. Adjacency list
3. Ways to Traverse a graph:
    a. DFS algorithm
    b. BFS algorithm
    c. Difference between DFS and BFS
4. Spanning Tree :
    a. Minimum spanning tree
    b. Prim algorithm
    c. Kruskal algorithm
    d. Difference between Prim and Kruskal
    e. Applications of spanning-tree
5. Shortest path in the graph:

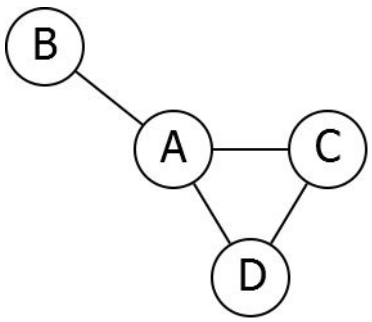
1. What is a Graph

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

A graph data structure (V, E) consists of:

1. A collection of vertices (V) or nodes.
2. A collection of edges (E) or paths.

For the following graph,



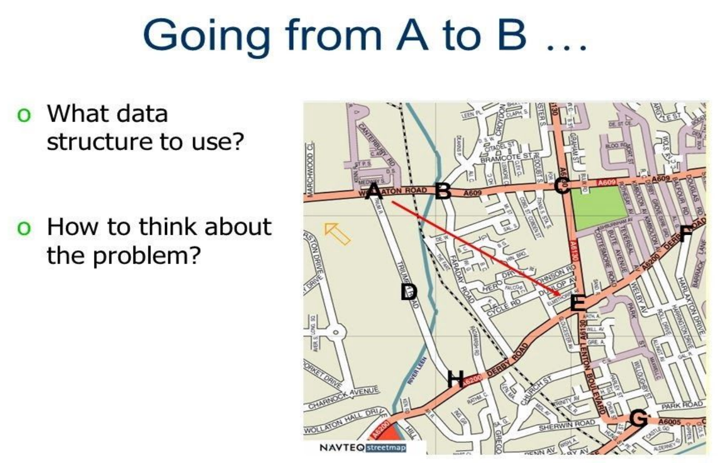
Vertices: A, B, C, D

Edges: (A, B) , (A, C) , (A, D) , (C , D)

Graph algorithm runtimes depend on |V | and |E |.

1.1. Use cases for Graph data structure:

1. Use a graph to find out whether two places on a road-map are connected and what is the shortest distance between them
2. Used to effectively store data in cloud storage solutions such as Dropbox.
3. Algorithms used in social media applications, for example, Facebook friends recommendation,



Going from A to B …

- What data structure to use?
- How to think about the problem?

1.3.Graph Terminology Used:

1. **Path:** It is the sequence of vertices in which each pair of successive vertices is connected by an edge.
2. **Cycle:** It is a path that starts and ends on the same vertex.
3. **Simple Path:** It is a path that does not cross itself that is, no vertex is repeated (except the first and the last). Also, simple paths do not contain cycles.
4. **Length of a Path:** It is the number of edges in the path. Sometimes it's the sum of weights of the edges also in case of weighted graphs.
5. **Degree of a Vertex:** It is the number of edges that are incident to the vertex.
6. **Indegree of a Vertex:** The indegree of a vertex is the number of edges for which it is head i.e. the numbers edge coming to it.
7. **Outdegree of a vertex**: The out-degree of a vertex is the number of edges for which it is the tail i.e. the number of edges going out of it. **A node whose outdegree is 0 is called a sink node.**
8. **Adjacent vertices:** If (V_i, V_j) is an edge in G, then we say that V_i and V_j are adjacent and the edge (V_i, V_j) is *incident* on V_i and V_j.

1.4.Different Types of Graphs:

1. Undirected Graph:

A graph is an undirected graph if the pairs of vertices that make up the edges are unordered pairs.

i.e. an edge(V_i, V_j) is the same as (V_j, V_i). The graph G_1 shown above is an undirected graph.

Undirected graphs might be used to represent:

1. Social networks
2. Maze exploration

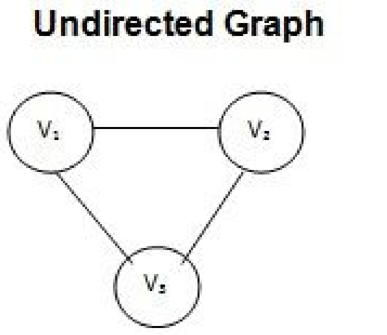
2.Directed Graph:

In a directed graph, each edge is represented by a pair of ordered vertices.

i.e. an edge has a specific direction. In such a case, edge (V_i, V_j) $\neq$ (V_j, V_i)

Directed graphs might be used to represent:

1. Streets with one-way roads.
2. Links between webpages.



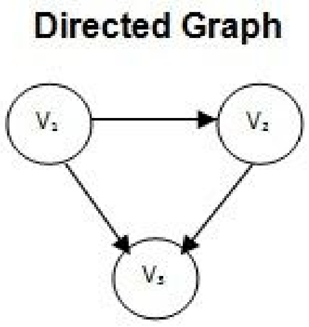
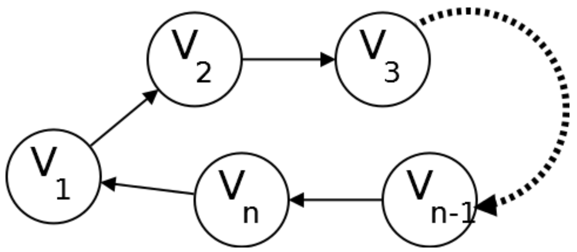


Figure 1: An Undirected Graph Figure 2: A Directed Graph

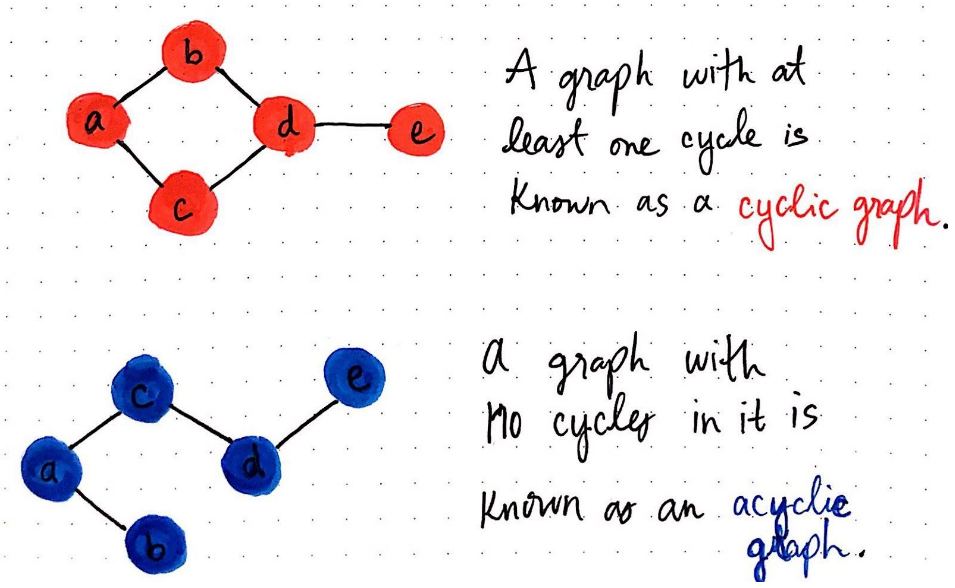
3.Cycle in a graph?

A cycle in a graph G is a sequence of vertices v 1 , v 2 , . . . , v n so that (v 1 , v 2), (v 2 , v 3), . . . , (v n− 1 , v n), (v n , v 1) are all edges.



Cycle: A cycle is a path whose first and last vertices are the same. For example, nodes abcd are cyclic.

Acyclic: A graph with no cycles is called an **acyclic** graph. A directed acyclic graph is called ***dag***.

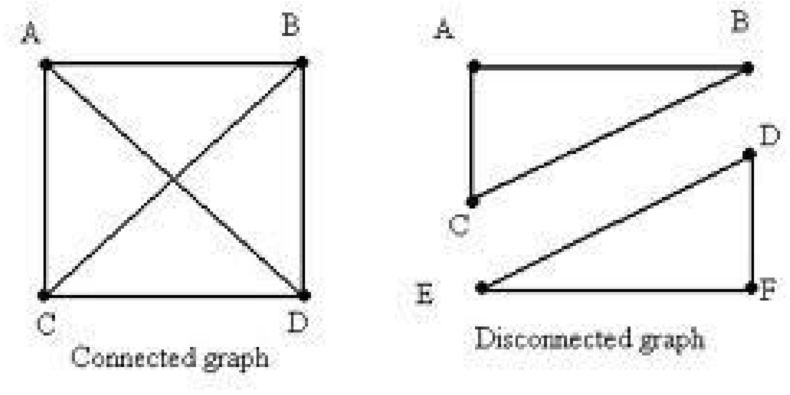

a
b
c
d
e
A graph with at least one cycle is known as a cyclic graph.
a
b
c
d
e
a graph with no cycles in it is known as an acyclic graph.

4.Connected Graph:

Two vertices V_i and V_j are said to be connected if there is a path in G from V_i to V_j.

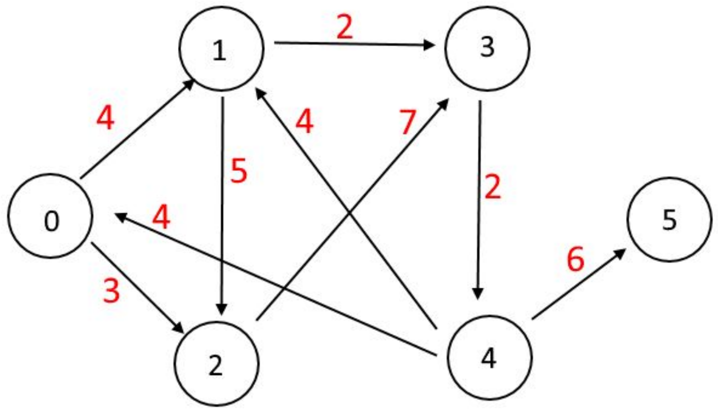
1.***Strongly Connected Graph**:* A directed graph G is said to be strongly connected if for every pair of distinct vertices V_i, V_j, there is a directed path from V_i to V_j and also from V_j to V_i.

2.***Weakly Connected Graph***: A directed graph G is said to be weakly connected there exists at least one set of distinct vertices V_i, V_j, such that there is a directed path from V_i to V_j but no path from V_j to V_i.



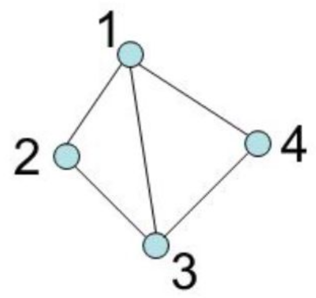
5.Weighted Graph or Network:

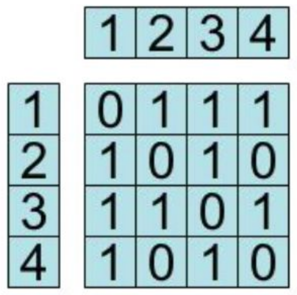
A number (weight) may be associated with each edge of a graph. Such a graph is called a weighted graph or a network.



Weighted Graph

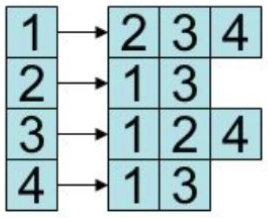
2.Ways to Represent Graphs:





$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 1 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 1 & 0 & 1 & 0 \end{array}$$

Adjacency matrix



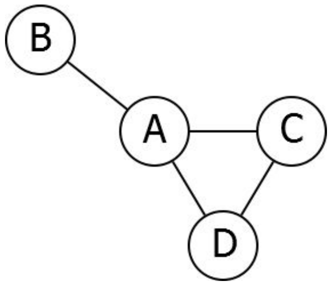
1 → 2 3 4
2 → 1 3
3 → 1 2 4
4 → 1 3

Adjacency list

1.Edge List:

To list an edge, we have an array of two vertex numbers containing the vertex numbers of the vertices that the edges are incident on.

Consider the following graph example,



Edges: (A, B) , (A, C) , (A, D) , (C , D)

Time and Space complexity:

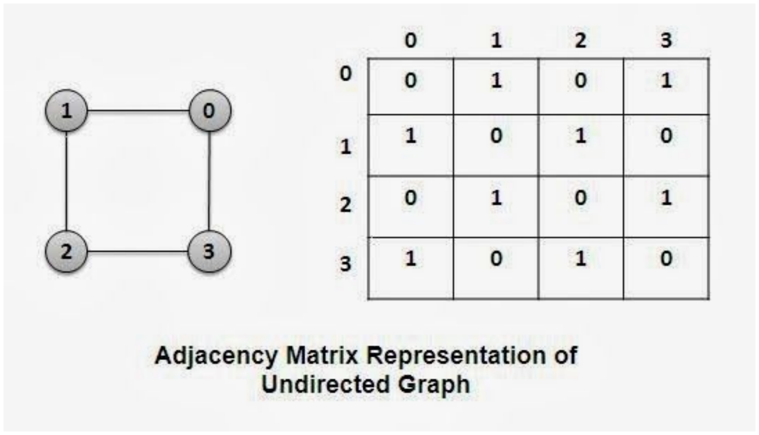
1. Total space for an edge list is Θ(E), as linear storage
2. To figure out if it is an edge or not is Θ(E), as we will have to do a linear search
3. To list out the all edges Θ(E)

2. Adjacency Matrix:

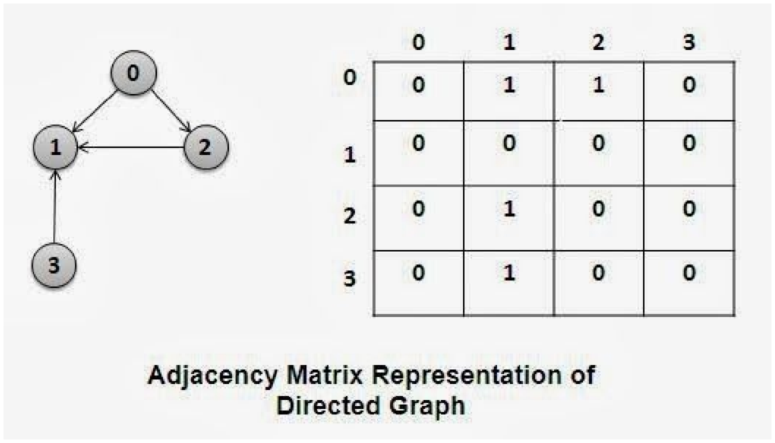
The adjacency matrix A of a graph G is a two-dimensional array of n × n elements where n is the numbers of vertices. The entries of A are defined as

1. A[i][j]= 1, if there exist an edge between vertices i and j in G.
2. A[i][j]= 0, if no edge exists between i and j.

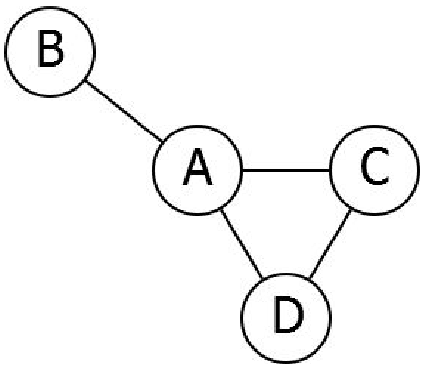
- **Undirected graph**: The degree of vertex i is the number of 1's in its row (or the sum of row i)



- **Directed graph**
    1. (Indegree) *:* A total number of 1’s in column i.
    2. (Outdegree) : Number of 1’s in row i.



Consider the following graph example,



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

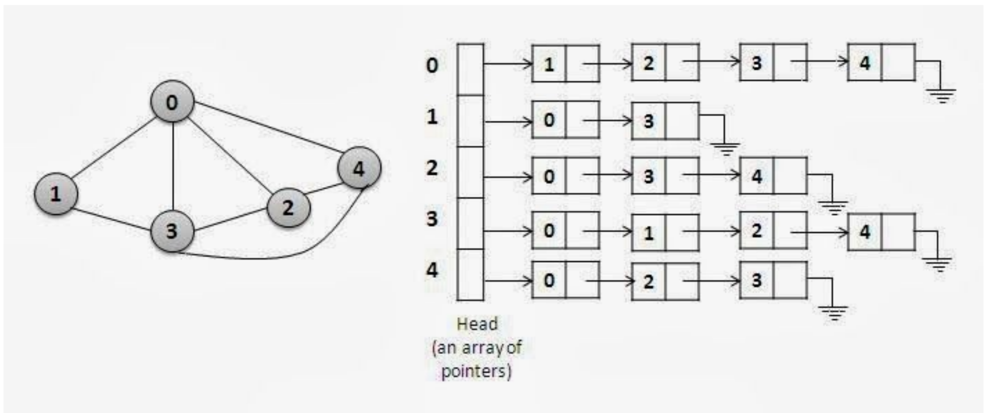
C	**1**	**0**	**0**	**1**
D	**1**	**0**	**1**	**0**

Time and Space complexity:

1. It takes Θ(V ^ 2) space, as we use a matrix of size V*V to store edges.
2. To figure out if it is an edge or not is Θ(1), as it will be done in linear time
3. To list out all the edges Θ(V ^ 2), as it as a matrix of size V*V

3.ADJACENCY LIST:

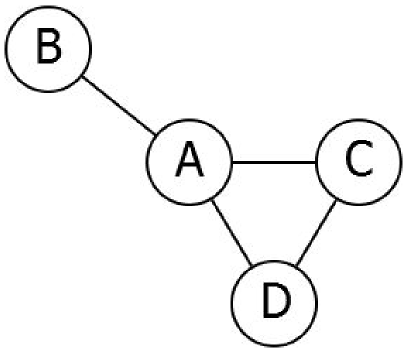
An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

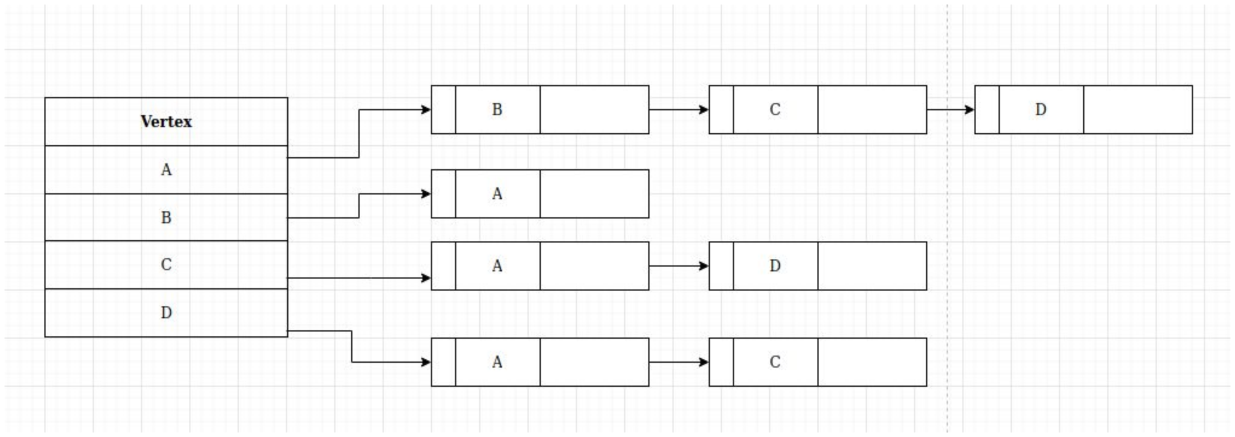


Adjacency List Representation of Graph

1. From the adjacency list, we can calculate the outdegree of any vertex i. The total number of nodes in the list 'i' is its outdegree.
2. To obtain the indegree, we can construct an inverse adjacency list. The inverse adjacency list.

Let us consider this example,





The degree of vertex i could be as high as |V|−1 vertical (if i is adjacent to all the other |V|−1 vertices) or as low as 0 (if i is isolated, with no incident edges).

Time and Space complexity:

1. To store the adjacency list, we need $\Theta(V+E)$ space as we need to store every vertex and their neighbour's space.
2. To figure out if it is an edge or not is Θ(degree)
3. To list out all the edges Θ(E)
4. To list out the neighbour edges Θ(degree)

Adjacency list majorly depends on data structure we use to store vertices, such as a linked list or hash tables.

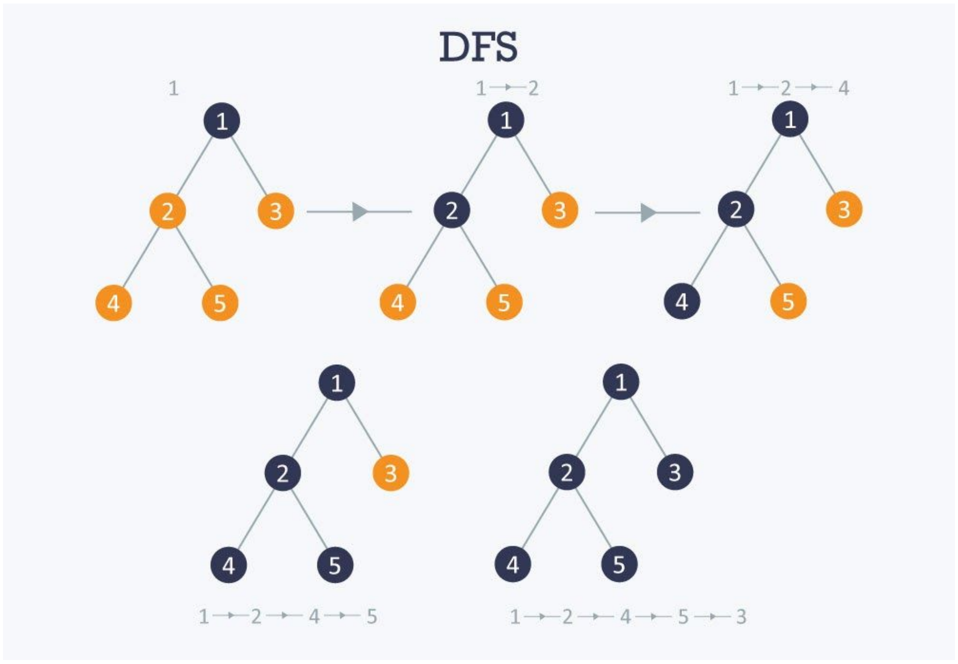
4.Reference:

1. https://runestone.academy/runestone/books/published/cppds/Graphs/AnAdjacencyMatrix.html
2. https://medium.com/the-programming-club-iit-indore/graphs-and-trees-using-c-stl-322e5779eef9

3.GRAPH TRAVERSALS:

1.DEPTH FIRST SEARCH:

We follow a path in the graph as deeply as we can go marking the vertices in the path as 'visited'. When there are no adjacent vertices that are not visited, we proceed backwards (backtrack) to a vertex in the path which has an 'unvisited' adjacent vertex and proceeds from this vertex. The process continues till all vertices have been visited.



Algorithm: Depth-first search (Graph G, Souce_Vertex S)

1. Create a stack **STK** to store the vertices.

2. Push the source vertex **S** in the stack 'STK'.

3. While the stack **STK** is not empty

4. Pop the vertex **U** from the top of the stack. i.e Vertex **U** = STK.top(), STK.pop()

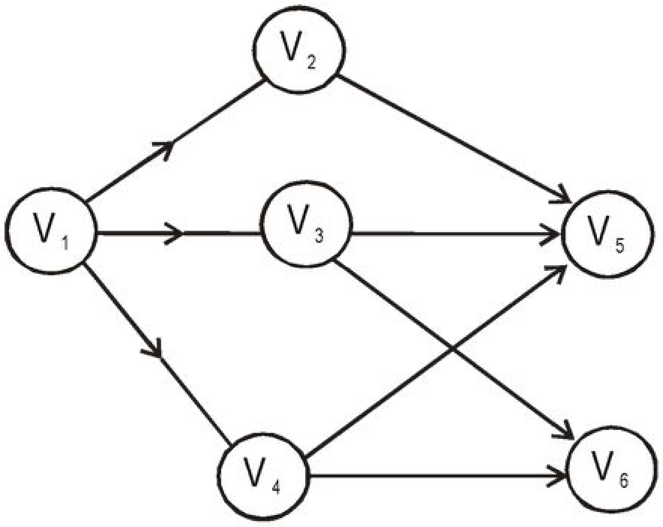
5. If the vertex **U** is not visited

6. Explore the vertex **U** and mark **U** as visited.

7. For every vertex **V** adjacent to vertex **U**

8. Push the vertex **V** in the stack STK

For example, let us consider the following graph



DFS of this graph is V_1 V_2 V_5 V_3 V_6 V_4

Time complexity:

The time complexity of depth-first search: **O(V+E) for an adjacency list implementation of a graph**. As each vertex is explored only once, all the vertices are explored in O(V) time.

We can implement DFS using various other data structures for storing graph and for searching in it.

Applications of DFS:

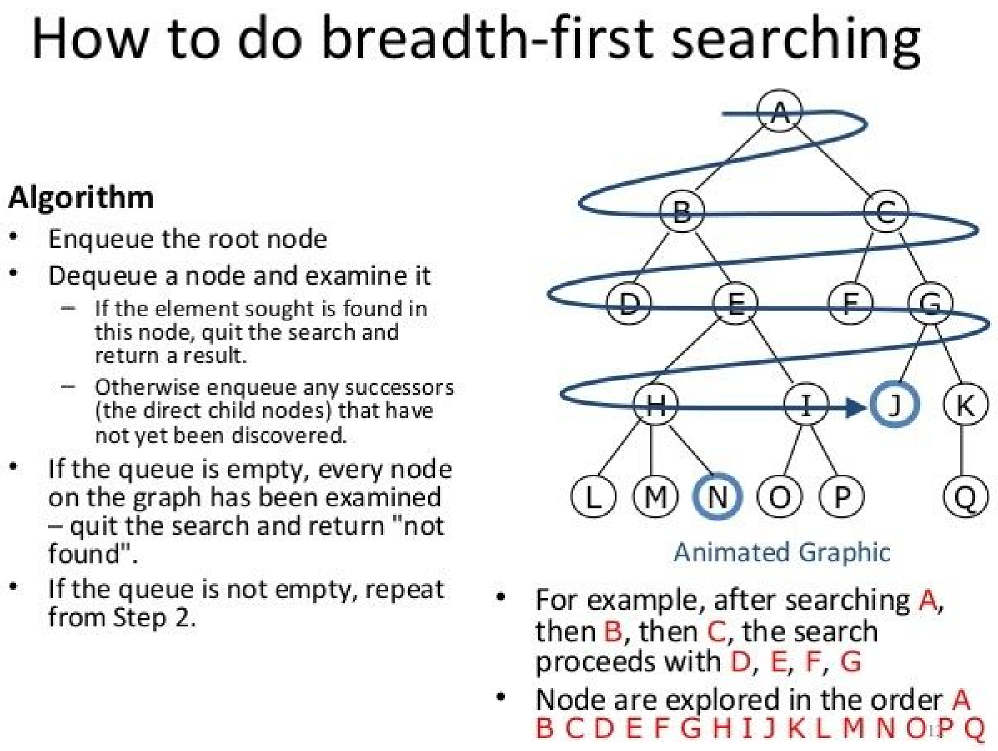
1. Minimum Spanning Tree
2. Topological sorting
3. To solve puzzle/maze which has one solution

For reference,

1. https://brilliant.org/wiki/depth-first-search-dfs/
2. https://algotree.org/algorithms/tree_graph_traversal/depth_first_search/

2.BREADTH FIRST SEARCH:

BFS is an algorithm for traversing an unweighted Graph or a Tree. BFS starts with the root node and explores each adjacent node before exploring node(s) at the next level.



How to do breadth-first searching

Algorithm

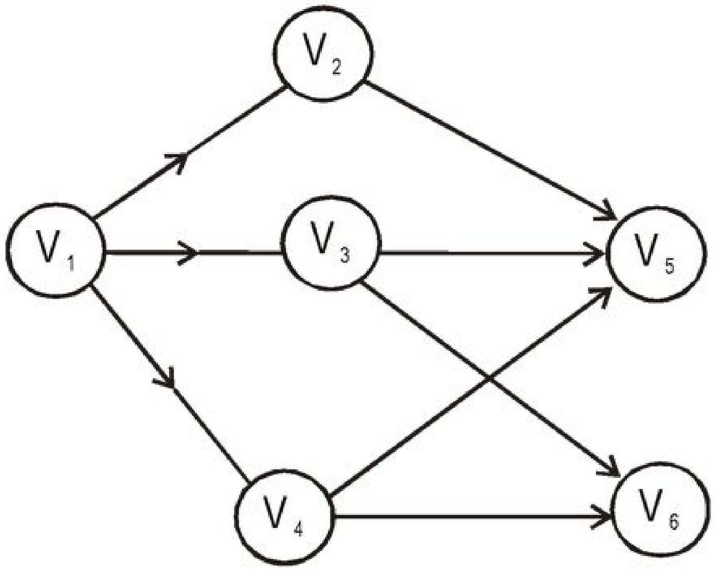
- Enqueue the root node
- Dequeue a node and examine it
  - If the element sought is found in this node, quit the search and return a result.
  - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
- If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
- If the queue is not empty, repeat from Step 2.

- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q

Algorithm: Breadth-first search (Graph G, Souce_Vertex S)

```
1. Create a queue Q to store the vertices.
2. Push the source vertex S in the queue Q.
3. Mark S as visited.
4. While the queue Q is not empty
5. Remove vertex U from the front of the queue. i.e Vertex U = Q.front(), Q.pop()
6. For every vertex V adjacent to the vertex U
7. If the vertex V is not visited
8. Explore the vertex V and mark V as visited.
9. Push the vertex V in the queue Q.
```

For example, let us consider the following graph



BFS of this graph is $V_1\ V_2\ V_3\ V_4\ V_5\ V_6$

Time complexity:

The time complexity of breadth-first search: O(V+E) for an adjacency list implementation of a graph. Performing an O(1) operation L times results to O(L) complexity. Thus, removing and adding a vertex from/to the Queue is O(1), but when you do that for V vertices, you get O(V) complexity.

We can implement BFS using various other data structures for storing graph and for searching in it.

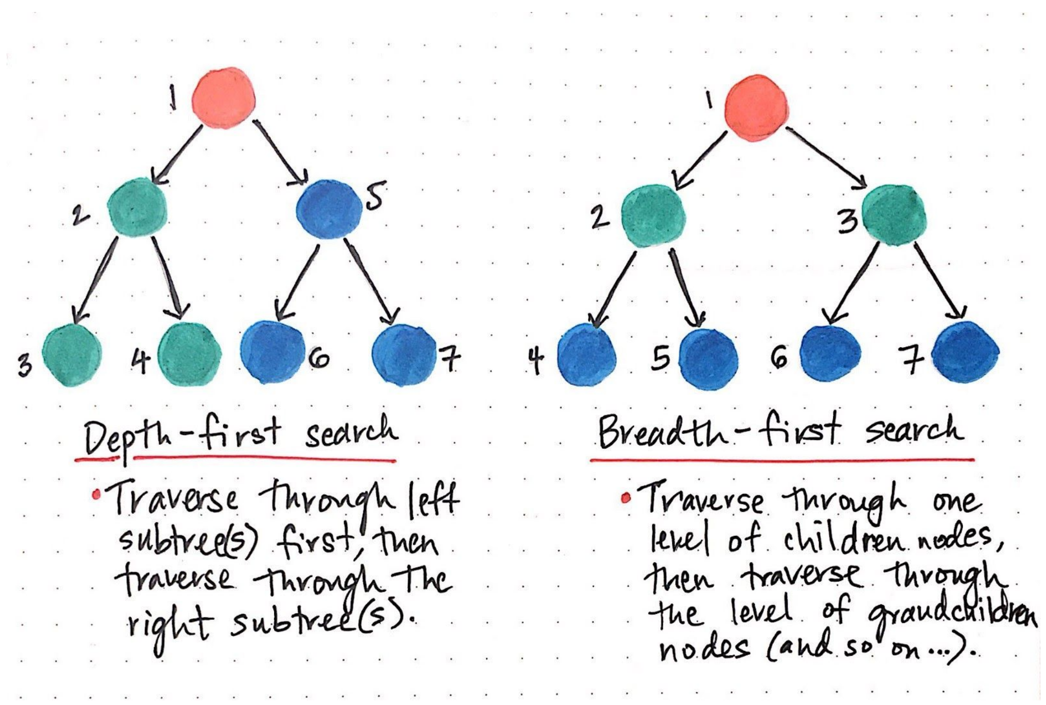
Application of BFS :

1. Minimum Spanning Tree or shortest path
2. Peer to peer networking
3. Crawler for search engine
4. Social network websites - to find people with a given distance k from a person using BFS till k levels.

For reference,

1. https://algotree.org/algorithms/tree_graph_traversal/breadth_first_search/
2. https://brilliant.org/wiki/breadth-first-search-bfs/

3.Difference between DFS and BFS

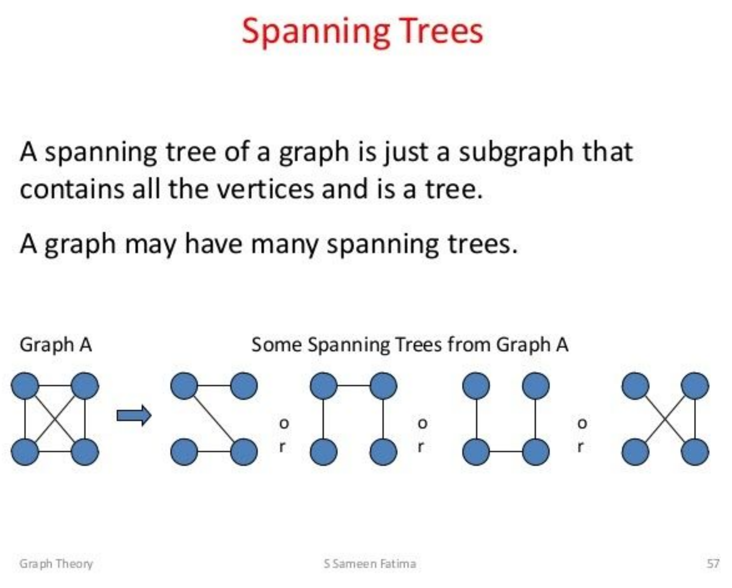

1
2
5
3
4
6
7
Depth-first search
• Traverse through left subtree(s) first, then traverse through the right subtree(s).
1
2
3
4
5
6
7
Breadth-first search
• Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

For more information,

1. https://stackoverflow.com/questions/687731/breadth-first-vs-depth-first
2. https://visualgo.net/en/mst?slide=1

4.SPANNING TREE:

Definition of Spanning Tree: Let G = (V, E) be an undirected connected graph. A subgraph t = (V, E′) of G is a spanning tree of G if and only if t is a tree.



Spanning Trees

A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree.

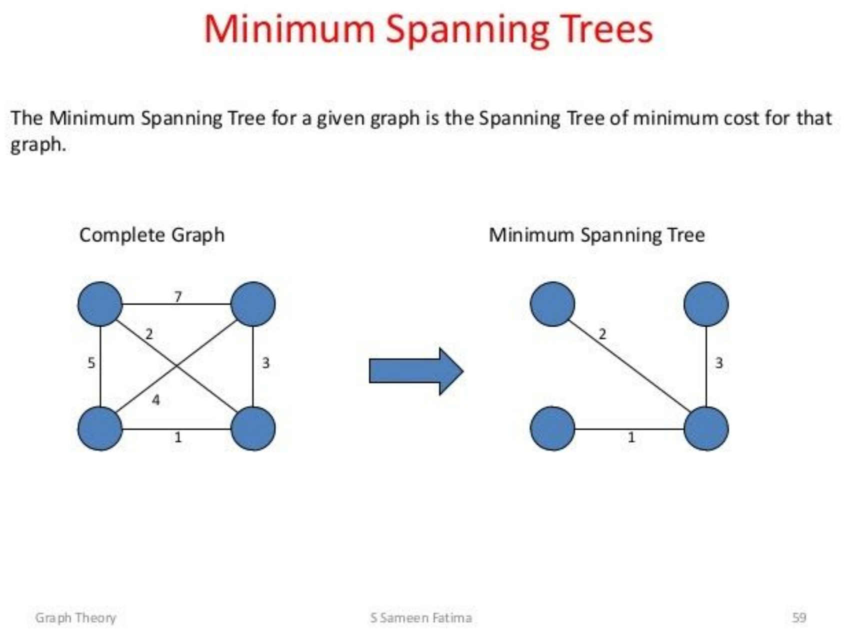
A graph may have many spanning trees.

Graph Theory S Sameen Fatima 57

4.1.MINIMUM COST SPANNING TREE:

The spanning-tree having the minimum sum of weights of edges is called minimum cost spanning tree. These weights may represent the lengths, distances, cost, etc.

Such trees are widely used in practical applications such as a network of road lines between cities, etc.

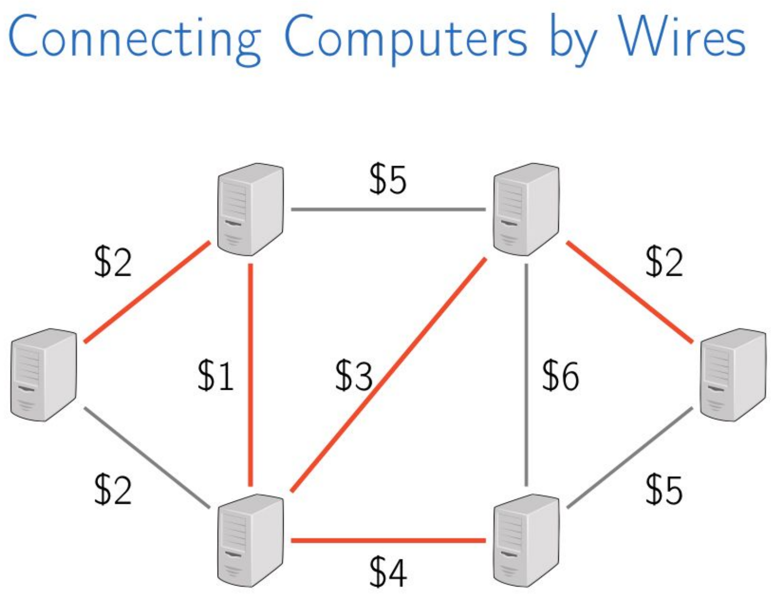


Minimum Spanning Trees

The Minimum Spanning Tree for a given graph is the Spanning Tree of minimum cost for that graph.

Complete Graph
7
2
5
3
4
1
Minimum Spanning Tree
2
3
1
Graph Theory
S Sameen Fatima
59

For example.

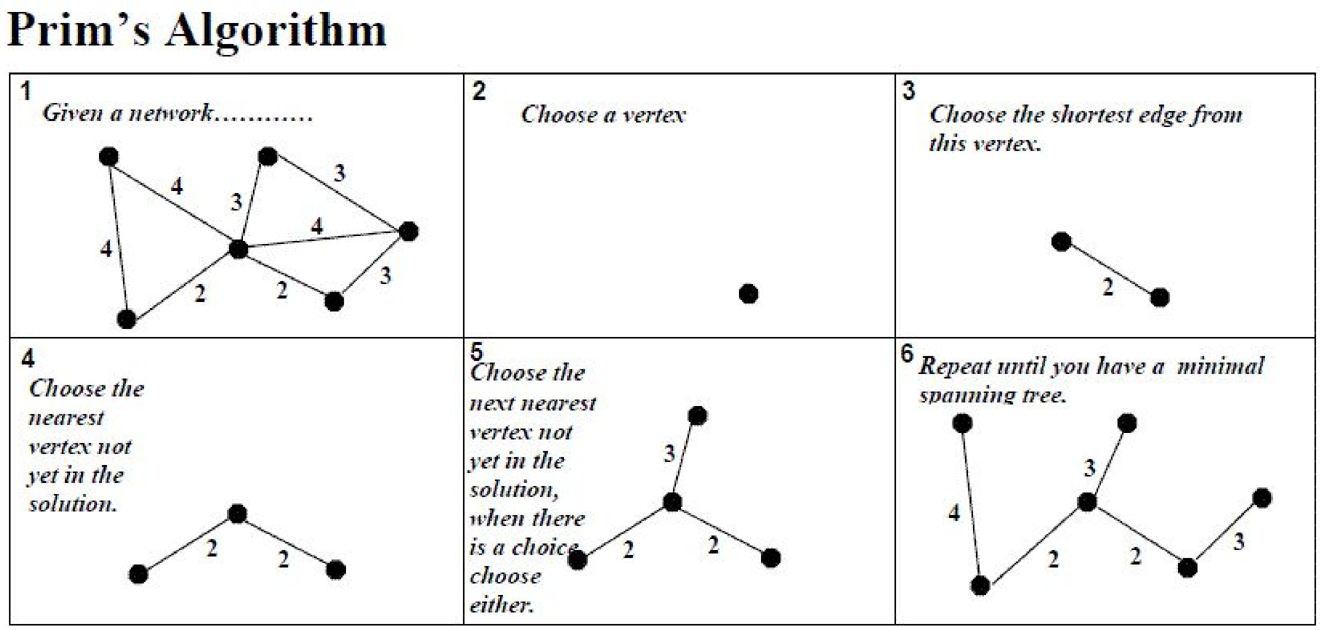


4.2.Algorithms to find the minimum spanning tree:

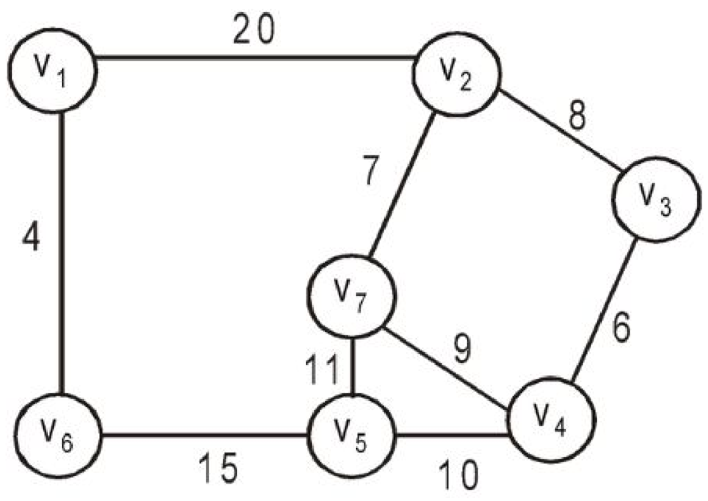
1. PRIM’S ALGORITHM:

Prim’s algorithm is a greedy algorithm. It finds a minimum spanning tree for a weighted undirected graph. This means it finds a **subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized**.

The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.


Prim’s Algorithm
1 Given a network............
2 Choose a vertex
3 Choose the shortest edge from this vertex.
4 Choose the nearest vertex not yet in the solution.
5 Choose the next nearest vertex not yet in the solution, when there is a choice choose either.
6 Repeat until you have a minimal spanning tree.

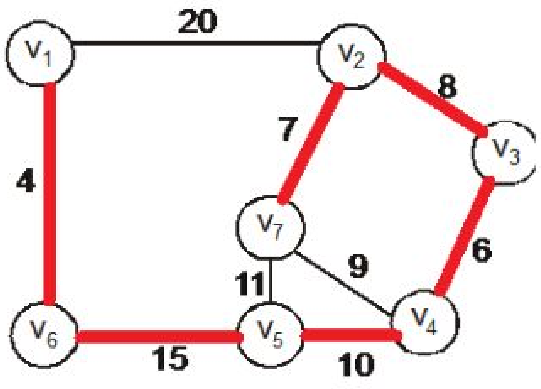
Let’s consider this graph,



All the steps to find the Minimum spanning tree using prim’s algorithm,

Step	Consider	Select	Spanning Tree
1.	(V_1, V_6)	(V_1, V_6)	
2.	(V_1, V_2), (V_5, V_6)	(V_5, V_6)	
3.	(V_1, V_2), (V_5, V_7), (V_4, V_5)	(V_4, V_5)	
4.	(V_1, V_2), (V_5, V_7), (V_3, V_4), (V_4, V_7)	(V_3, V_4)	
5.	(V_1, V_2), (V_5, V_7), (V_4, V_7), (V_2, V_3)	(V_2, V_3)	
6.	(V_1, V_2), (V_5, V_7), (V_4, V_7), (V_2, V_7)	(V_2, V_7)	
7.	(V_1, V_2), (V_5, V_7), (V_4, V_7)	(V_4, V_7)	Forms a cycle
8.	(V_5, V_7), (V_1, V_2)	(V_5, V_7)	Forms a cycle
9.	(V_1, V_2)	(V_1, V_2)	Forms a cycle

The end result of the minimum spanning tree should look like,



Time Complexity:

The Complexity depends on the data structures used for the graph and for ordering the edges by weight. for example,

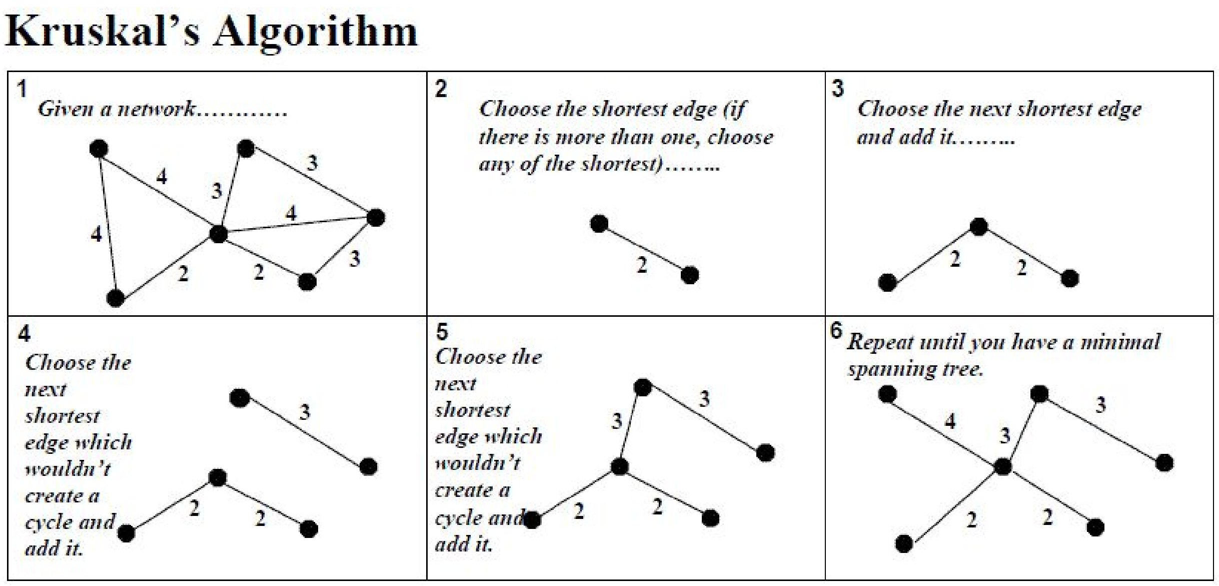
1. For an Adjacency matrix, it will be, $O(|V|^2)$
2. For an Adjacency List, it will be $O(|E|\log|V|)$

2.KRUSKAL'S ALGORITHM:

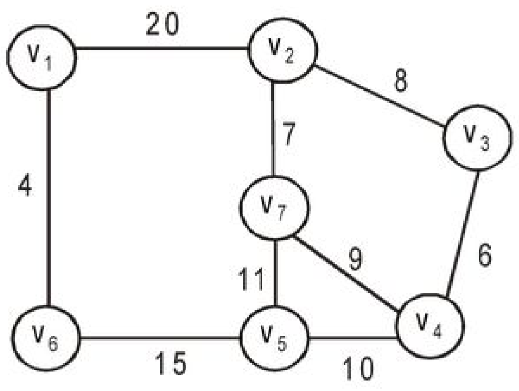
It is a greedy algorithm. It **finds a subset of the edges that forms a tree that includes every vertex**, where the total weight of all the edges in the tree is minimized.

If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).

Kruskal's Algorithm :


Kruskal's Algorithm
1 Given a network…………
2 Choose the shortest edge (if there is more than one, choose any of the shortest)……..
3 Choose the next shortest edge and add it……..
4 Choose the next shortest edge which wouldn't create a cycle and add it.
5 Choose the next shortest edge which wouldn't create a cycle and add it.
6 Repeat until you have a minimal spanning tree.

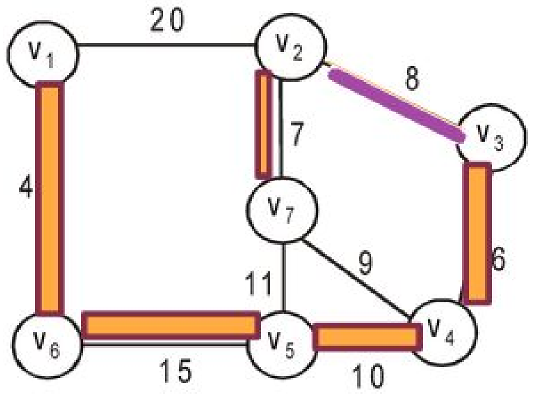
Let's consider this graph,



All the steps to find the Minimum spanning tree using Kruskal's algorithm,

Step	Consider	Spanning Tree
1	(V_1, V_6)	
2.	(V_3, V_4)	
3.	(V_2, V_7)	
4.	(V_2, V_3)	
5.	(V_4, V_7)	Forms a Cycle, Reject
6.	(V_4, V_5)	
7.	(V_5, V_7)	Forms a Cycle, Reject
8.	(V_5, V_6)	

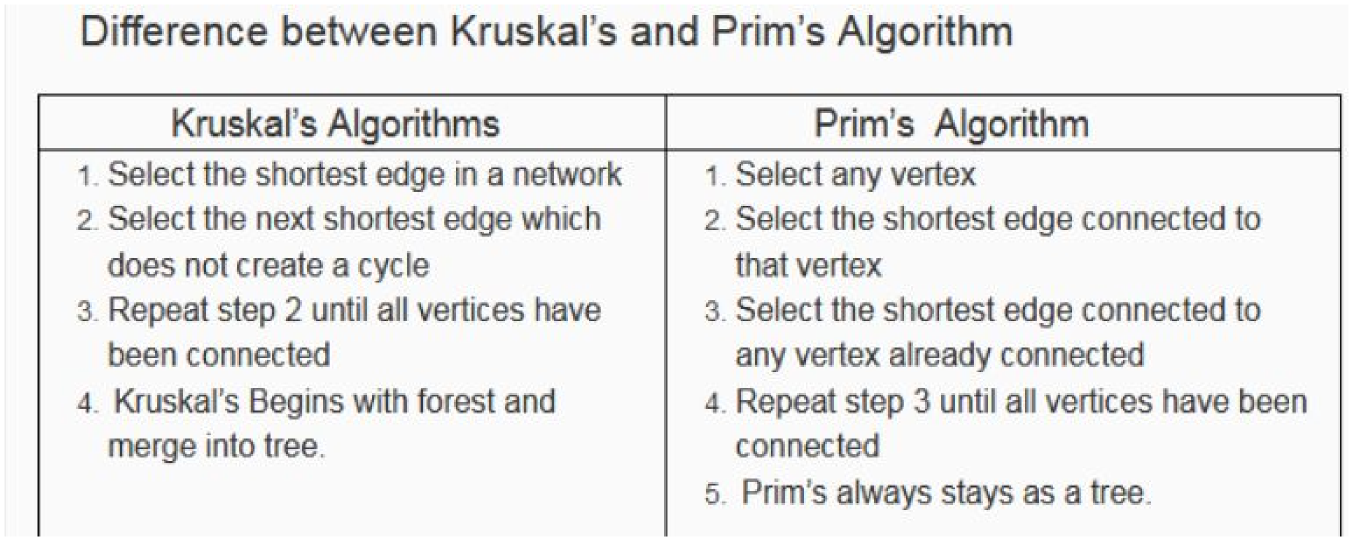
The end result of the minimum spanning tree should look like,



Time Complexity:

The most time-consuming operation is sorting because of the total complexity of the Disjoint-Set operations, which is the overall Time Complexity of the algorithm is **O(|E||LogE|)**

3.Difference between Kruskal's and Prim's Algorithm



Difference between Kruskal's and Prim's Algorithm

Kruskal's Algorithms	Prim's Algorithm
1. Select the shortest edge in a network 2. Select the next shortest edge which does not create a cycle 3. Repeat step 2 until all vertices have been connected 4. Kruskal's Begins with forest and merge into tree.	1. Select any vertex 2. Select the shortest edge connected to that vertex 3. Select the shortest edge connected to any vertex already connected 4. Repeat step 3 until all vertices have been connected 5. Prim's always stays as a tree.

4.Applications of finding minimum spanning tree:

1. Network design
2. Travelling salesman problem

5.SHORTEST PATH

A person wishing to travel from city A to city B could require the following information.

1. Is there a path from A to B?
2. If there is more than one path from A to B, which is the shortest?

To solve this problem we use the Dijkstra algorithm and Bellman-Ford Algorithm.

1.Dijktra Algorithm:

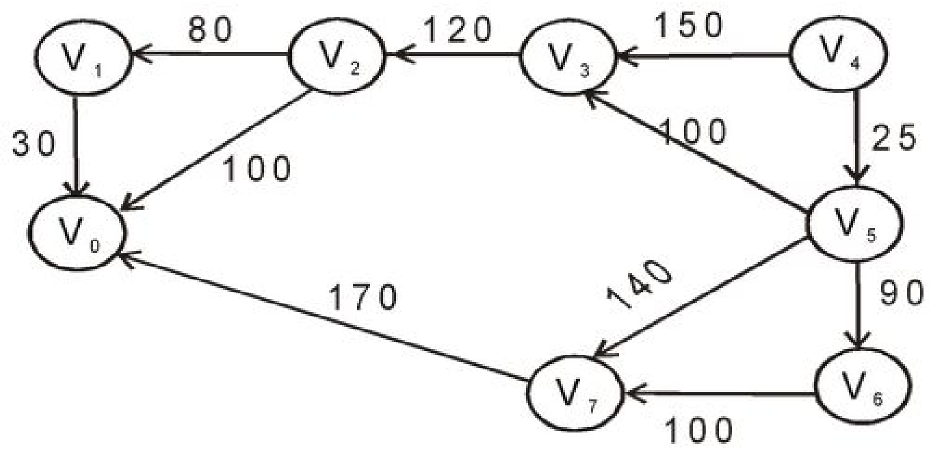
Dijkstra's algorithm finds the shortest path in a weighted graph containing only positive edge weights from a single source. It uses a priority-based dictionary or a queue to select a node/vertex nearest to the source that has not been edge relaxed.

Algorithm: Dijkstra's Shortest Path

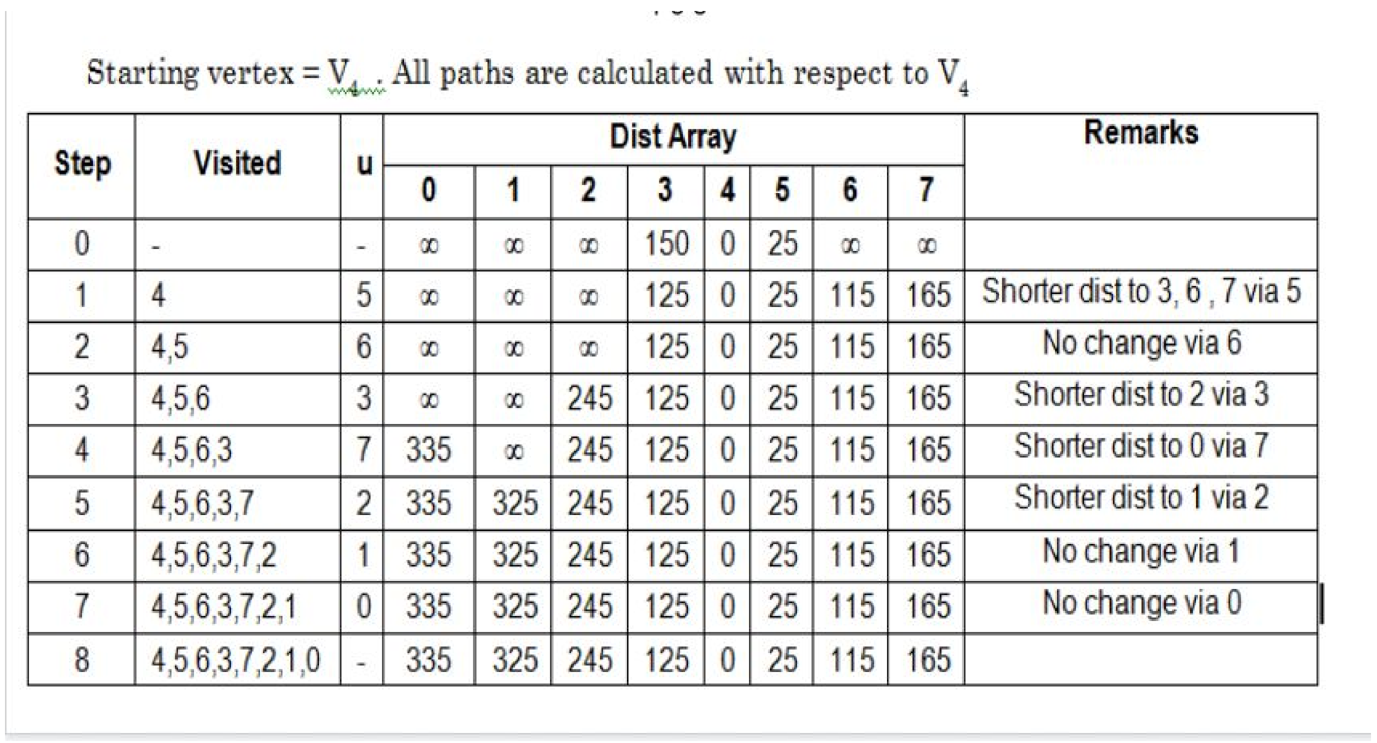
1. Start with the empty Shortest Path Tree (SPT).
2. Maintain a set SPT[] to keep track to vertices included in SPT.
3. Assign a distance value to all the vertices, (say distance []) and initialize all the distances with +∞ (Infinity) except the source vertex. This will be used to keep track of the distance of vertices from the source vertex. The distance of source vertex to source vertex will be 0.
4. Repeat the following steps until all vertices are processed.
    1. Pick the vertex ***u*** which is not in ***SPT[]*** and has minimum distance. Here we will loop through the vertices and find the vertex with minimum distance.
    2. Add vertex ***u*** to ***SPT[].***
    3. Loop over all the adjacent vertices of

4. For adjacent vertex v, if v is not in SPT[] and distance[v] > distance[u] + edge u-v ***weight*** then update ***distance[v] = distance[u] + edge u-v weight***

Let's consider the graph,



Then Dijkstra algorithm using matrix results,



Starting vertex = V_4 . All paths are calculated with respect to V_4

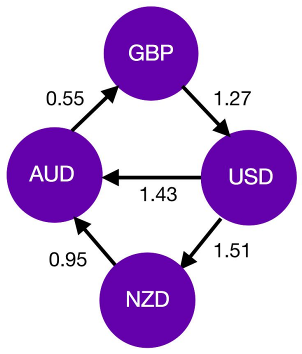
Step	Visited	u	Dist Array								Remarks
			0	1	2	3	4	5	6	7	
0	-	-	∞	∞	∞	150	0	25	∞	∞	
1	4	5	∞	∞	∞	125	0	25	115	165	Shorter dist to 3, 6 , 7 via 5
2	4,5	6	∞	∞	∞	125	0	25	115	165	No change via 6
3	4,5,6	3	∞	∞	245	125	0	25	115	165	Shorter dist to 2 via 3
4	4,5,6,3	7	335	∞	245	125	0	25	115	165	Shorter dist to 0 via 7
5	4,5,6,3,7	2	335	325	245	125	0	25	115	165	Shorter dist to 1 via 2
6	4,5,6,3,7,2	1	335	325	245	125	0	25	115	165	No change via 1
7	4,5,6,3,7,2,1	0	335	325	245	125	0	25	115	165	No change via 0
8	4,5,6,3,7,2,1,0	-	335	325	245	125	0	25	115	165	

Time Complexity:

1. O(|V|^2) for an Adjacency matrix
2. O(ElogV) for Adjacency List and Min Heap

2.Limitation of Dijkstra algorithm:

Dijkstra's algorithm relies on the fact that the shortest path from s to t goes only through vertices that are closer to s. This is no longer the case for graphs with negative edges, One of the use cases of this is currency cycle with a negative cycle.

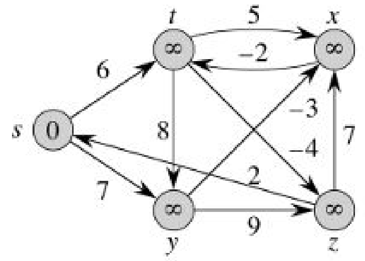


3.Bellman-Ford Algorithm:

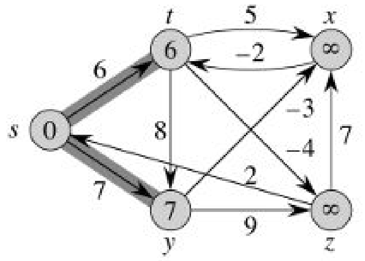
Bellman-Ford algorithm finds the shortest path (in terms of distance/cost) from a single source in a directed, weighted graph containing positive and negative edge weights.

Algorithm: Bellman-Ford:

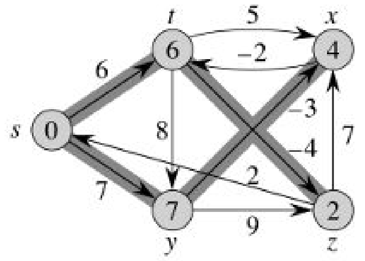
```
1.    Initialize the distance from the source node S to all other nodes as infinite (999999999) and to itself as 0.
2.    For every node in the graph do
3.    For every edge E in the EdgeList do
4.    Node_u = E.first, Node_v = E.second
5.    Weight_u_v = EdgeWeight ( Node_u, Node_v )
6.    If ( Distance [v] > Distance [u] + Weight_u_v )
7.            Distance [v] = Distance [u] + Weight_u_v
```



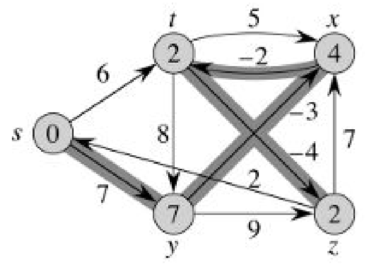
(a)



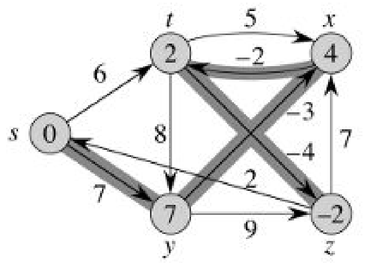
(b)



(c)



(d)



(e)

Time Complexity:

1. O(|V| * |E|) for the Adjacency list
2. O(|V| ^ 3) for the Adjacency matrix

In a graph with only positive edge weights, Dijkstra's algorithm with a priority queue / set implementation runs faster in O ((E+V) log V) than Bellman-Ford O (E.V).

6.TOPOLOGICAL SORT:

AOV network: A directed graph in which the vertices represent activities or tasks and the edges represent the precedence is called an AOV network. There should be no cycles in an AOV network i.e there should be at least one activity which does not have a predecessor(start activity).

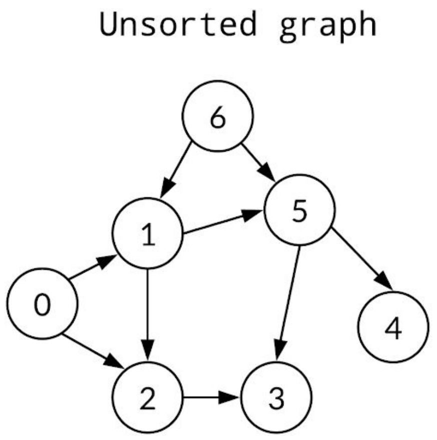
Topological sort: If we have an AOV network, then we would like to find out whether the project is feasible or not and if it is feasible, then in what order should the activities be performed so that the project can be completed. The process of converting a set of precedences represented by an AOV network into a linear list in which no later activity precedes an earlier one is called Topological Sorting. This method identifies the order of the activities.

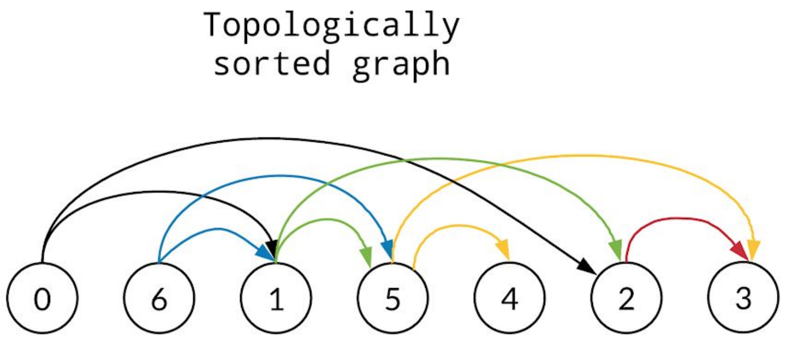
We can either use DFS or BFS for topological sort.

Algorithm: Topological Sort:

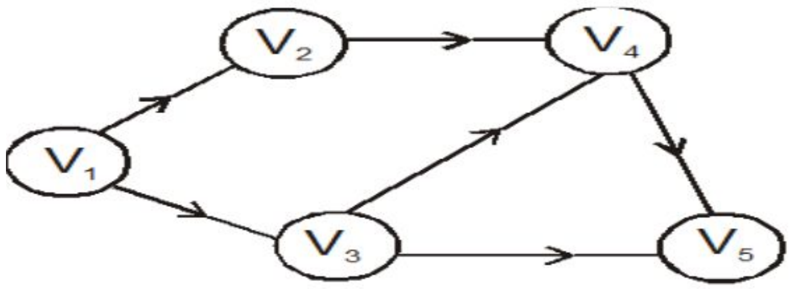
1. Use a temporary stack to store the vertex.
2. Maintain a visited [] to keep track of already visited vertices.

3. In DFS we print the vertex and make a recursive call to the adjacent vertices but here we will make the recursive call to the adjacent vertices and then push the vertex to stack.
4. Observe closely the previous step, it will ensure that vertex will be pushed to stack only when all of its adjacent vertices (descendants) are pushed into the stack.
5. Finally, print the stack.
6. For disconnected graph, Iterate through all the vertices, during iteration, at a time consider each vertex as source (if not already visited).

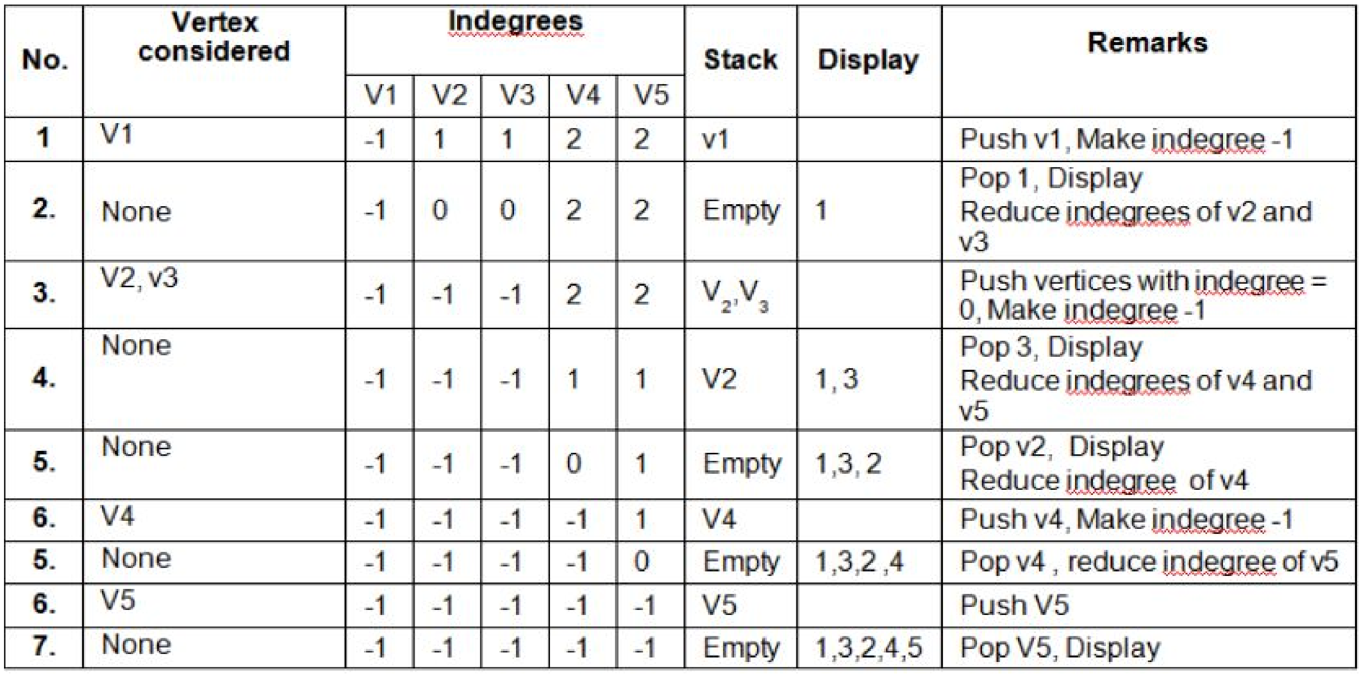




Let us consider this graph,



All the steps to perform for topological sort,



No.	Vertex considered	Indegrees					Stack	Display	Remarks
		V1	V2	V3	V4	V5			
1	V1	-1	1	1	2	2	v1		Push v1, Make indegree -1
2.	None	-1	0	0	2	2	Empty	1	Pop 1, Display Reduce indegrees of v2 and v3
3.	V2, v3	-1	-1	-1	2	2	V_2,V_3		Push vertices with indegree = 0, Make indegree -1
4.	None	-1	-1	-1	1	1	V2	1, 3	Pop 3, Display Reduce indegrees of v4 and v5
5.	None	-1	-1	-1	0	1	Empty	1,3, 2	Pop v2, Display Reduce indegree of v4
6.	V4	-1	-1	-1	-1	1	V4		Push v4, Make indegree -1
5.	None	-1	-1	-1	-1	0	Empty	1,3,2 ,4	Pop v4 , reduce indegree of v5
6.	V5	-1	-1	-1	-1	-1	V5		Push V5
7.	None	-1	-1	-1	-1	-1	Empty	1,3,2,4,5	Pop V5, Display

The topological order is: V_1, V_3, V_2, V_4, V_5

Time Complexity: O(V+E)

Applications of Topological sort:

1. Scheduling jobs from given dependencies among Jobs. F
2. Instruction Scheduling
3. Determining the order of compilation tasks to perform in makefiles, data serializations and resolving symbol dependencies in linkers.

References

[1] Research papers of Daniel Kane
[2] Research papers from Princeton
[3] Research papers from the University of Washington
[4] Reference from AlgoTree, Khan Academy and Stackoverflow
[5] GitHub repository for code

I have used a lot of images from a lot of sources, and I don't own many of the images. I just found them online and used them to explain concepts. As images are far better for understanding concepts.