Book
On
Graph Algorithms

- Algorithms Analysis & Design Project

Contents

All the above algorithms are discussed along with Example, pseudo code and code implementation and are analyzed based on time and space complexity.

Chapter 1: Introduction

Graph Data Structure:

A non linear data structure that consists of nodes (vertices) and edges. Edges are lines that connect a node to another node or to itself in a graph. Formally, Graph contains a finite set of nodes (vertices) and a set of edges which connect a pair of nodes.

Graph Terminology

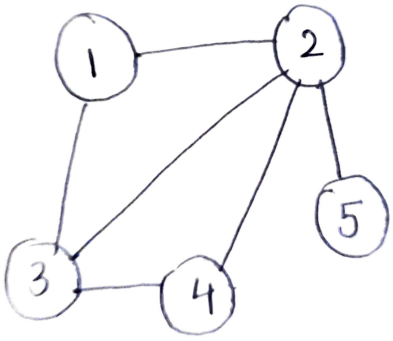
- **Neighbour Of a Vertex**
  An edge is said to join its endpoints. A vertex joined by an edge to a vertex is said to be a neighbour of v.
- **Proper Edge**
  An edge that joins two distinct vertices of a graph
- **Self Loop**
  An edge that joins a vertex to itself
- **Multi Edge**
  Collection of 2 or more edges having identical endpoints
- **Directed Edge**
  An edge whose one of the endpoints is designated as tail and other as head. And we can move from tail to head
- **Degree of a vertex**
  Degree of a vertex v is the number of proper edges incident on v plus twice the number of self loops
- **Trail**
  A trail is a walk with no repeated edges
- **Path**
  A trail with no repeated vertices(except possibly the initial and final vertices)
- **Cycle**
  A trail in which the first and last vertices are the same.

Types Of Graphs

Types graphs used in the rest of the book.

- **Simple Graph**

  A graph that does not contain either self loops or multi edges.

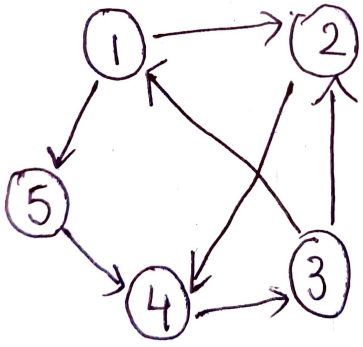


- **Multi Graph (or loopless graph)**

  A graph that contain multi edges but no self loops

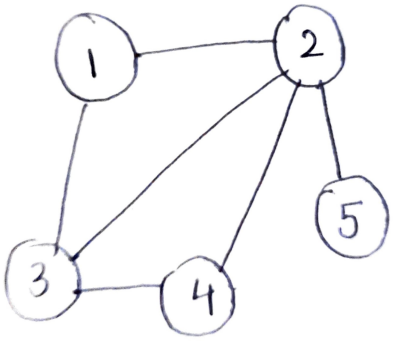
- **Directed Graph**

  Graph each of whose edges are directed



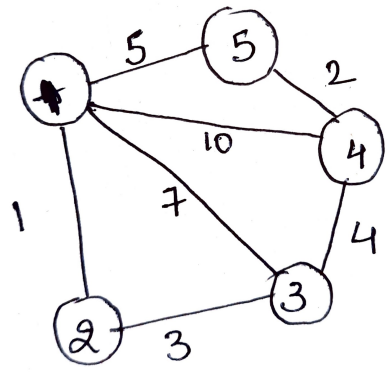
- **Undirected Graph**

  The graph in which the edges are not directed



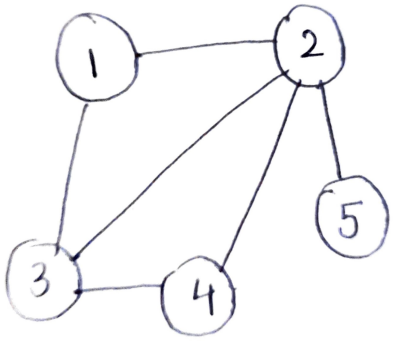
- **Weighted Graph**

  Graph in which each edge have a value



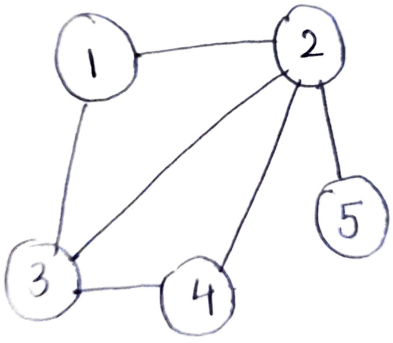
- **Un-Weighted Graph**

  Graph in which edges does not have a value



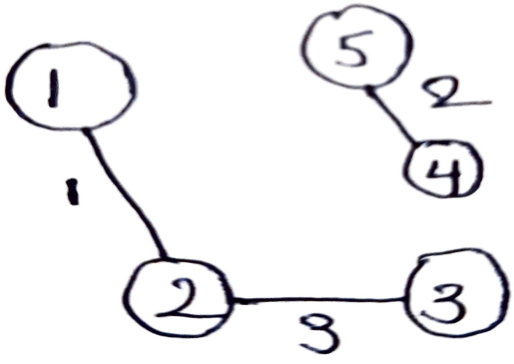
- **Connected Graph**

  A graph in which there is path between any two vertices of a graph



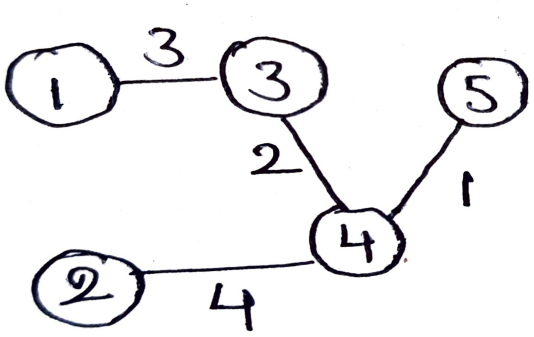
- **Disconnected Graph**

  A graph which is not connected is a disconnected graph



- **Tree**

  A connected graph with no cycles.



Graph Representation

Two most commonly used ways to represent a graph are

1. Adjacency Matrix
2. Adjacency List

- **Adjacency Matrix**

  This is a 2 dimensional array of size |V+1|*|V+1| where |V| is the number of vertices in the graph and the element Array[i][j] represents the edge between nodes i and j. When there is no edge between them this is initialized to 0.

  1. If the graph is weighted and there is an edge between nodes i and j then Array[i][j] gives the value of weight of the edge.
  2. If the un weighted and there is an edge between nodes i and j then Array[i][j] = 1

- **Adjacency List**

  An array of lists is used and the size of the array is the number of the vertices of the graph. Array[i] represents the list of vertices that are adjacent to node i.

List of edges is also used to represent the graph and this is useful mainly in Kruskal's algorithm.

In this book we assume that the vertices of the graph are indexed from 1.

Chapter 2: Search and Sort

Searching or Traversing a Graph

Two Algorithms that are widely used to search in a graph are Depth-First Search (DFS) and Breadth-First Search (BFS)

Depth-First Search:

DFS is an algorithm for searching or traversing a graph both directed or undirected.

As the name suggests it searches as deeper as possible in the graph.

- **IDEA**:

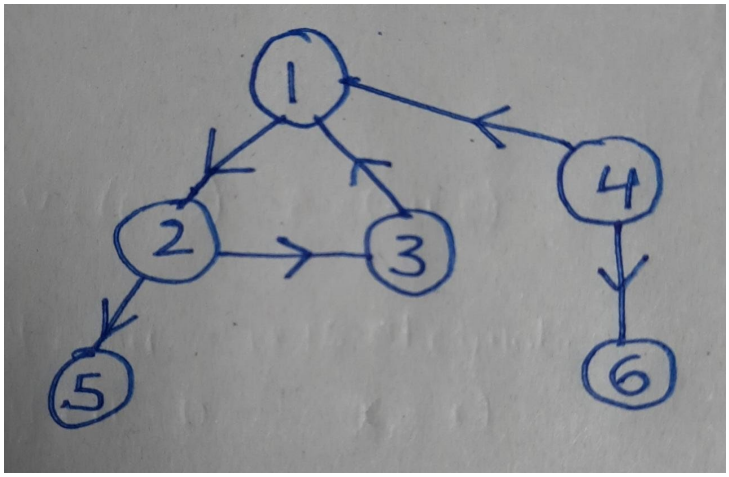
  Visits all the nodes that are not visited from recently visited node and if all the vertices that are reachable from the vertex are seen then it recurses back. Here, when we reach a node it is marked as seen, and is marked as visited when all the nodes that are reachable from the node are seen.

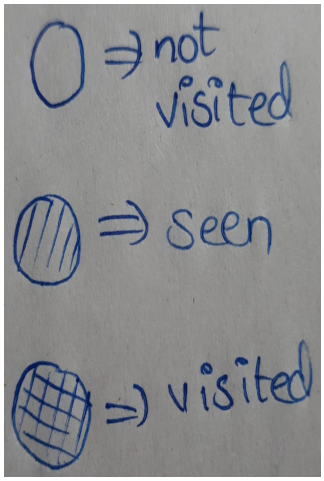
  Here we maintain the information about whether the node is seen or not because the graph may contain cycles. If the cycle is present in a graph then we may visit the node twice and keep on moving in the loop if we don't maintain the information whether the node is visited or not.

  Let us see an example to see the working of the algorithm.

- **Example**:

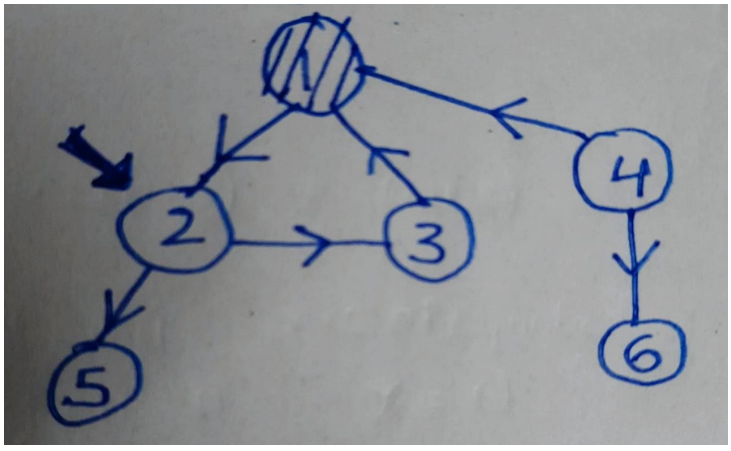
  Consider a directed graph G shown below



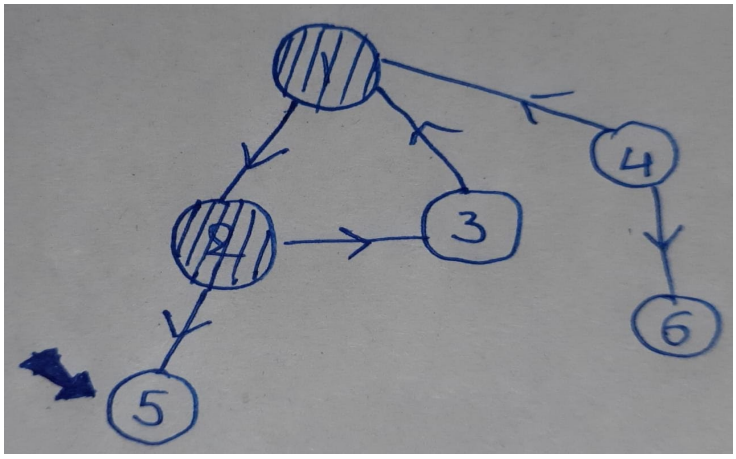


Here initially all the nodes are marked as not visited.Now lets start the algorithm from node 1.

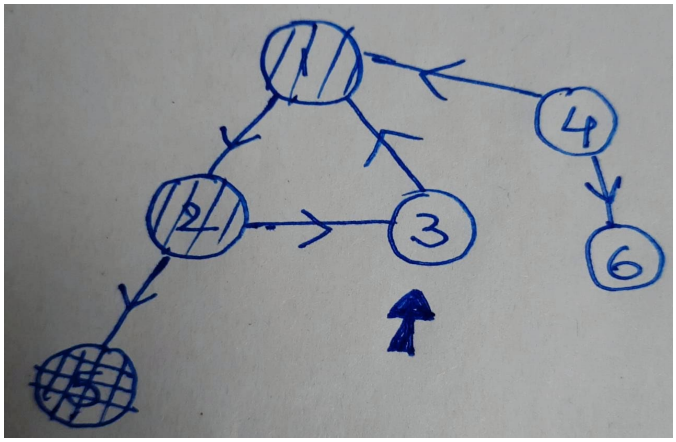
- Now, we mark the node 1 as seen and move to the next node that can be reached (node 2).



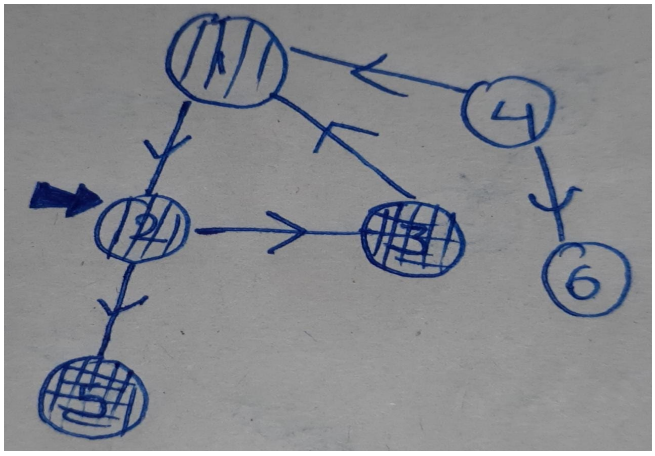
- Now, we mark the node 2 as seen and move to the next node that can be reached from 2 and as there are two nodes 3 and 5 we choose the left node.



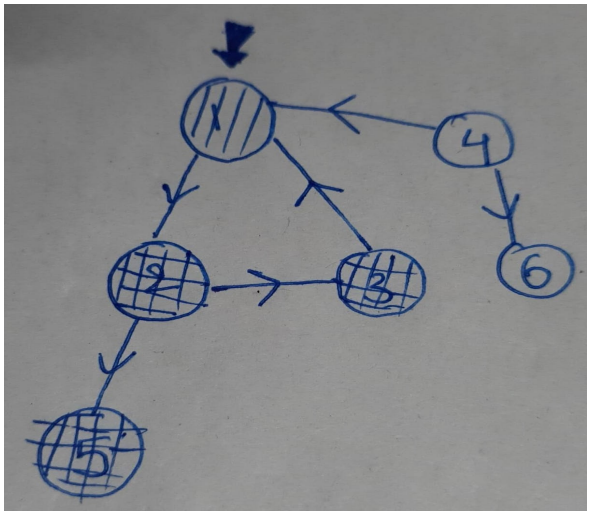
- Now as there are no nodes that can be reached from 5 we mark the node 5 as visited and recurse back to 2. And we move to node 3 that can be reached from 2.



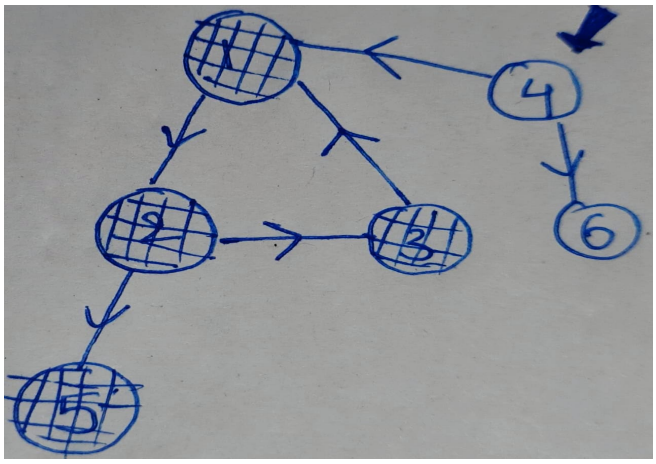
- Now as all the nodes that are reachable from 3 are seen we mark node 3 as visited and recurse back to node 2.



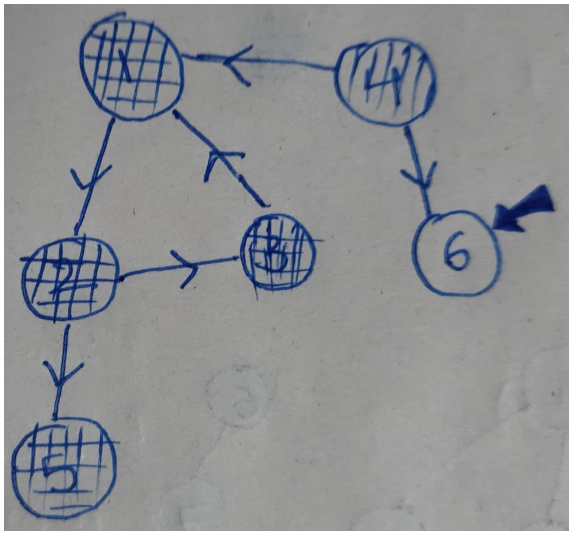
- As all the nodes that can be reached from node 2 are seen we recurse back to node 1.



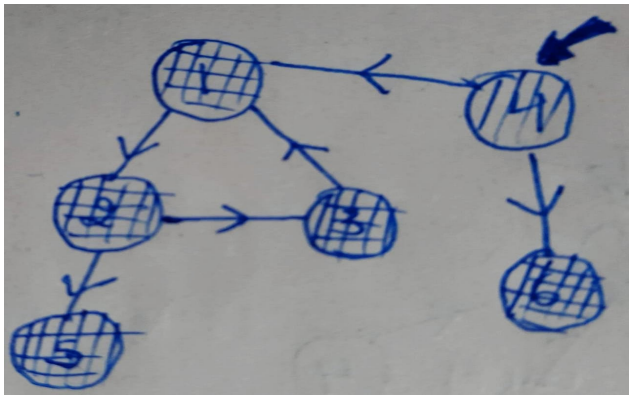
- As all the nodes that are reachable from node 1 are seen we mark 1 as visited. And as we cannot recurse back from 1 we check whether all nodes in the graph are visited and select the node that is not visited. So, now we select the node 4.



- Now, we mark 4 as seen and move to node 6 (node that is reachable from 4).



- Now as there are no nodes that can be reached from 6 we mark 6 as visited and recurse back to 4.



- As all the nodes that are reachable from 4 are seen we mark 4 as visited. As there are no nodes to recurse back and all the nodes of the graph are seen the final output of the traversal is

  1,2,5,3,4,6.
- There can be multiple traversals and the Algorithm gives one the traversal. The other traversals for the above graph can be (4,1,2,5,3,6); (6,4,1,2,5,3); ...

- **Algorithm**:
  - We maintain an array to know the status of the node (not-visited, seen or visited). Then mark every vertex of the graph as not-visited.
  - And every non visited vertex in the graph is passed as an argument to the function DFS_Traversal()
  - In the function DFS_Traversal() we mark the non-visited node as seen and then we move to the vertices that can be reached from the node. And do the same.

- After performing this on all the vertices that can be reached we mark the root vertex as visited.

- Pseudo Code:

```
DFS(Graph G,int V)
{
   // status[i] = 0 => not visited, 1 => seen, 2 => visited
   int status[V+1] = {0}; // V = no.of Vertices
   for every vertex v ∈ V  // V = set of vertices of G
       if(status[v] == 0)
           DFS_Traversal(G, v, status);
   return;
}
DFS_Traversal(Graph G, int v, int* status)
{
   status[v] = 1;
   for every u ∈ Adj[v] // => there is a vertex from v to u
       if(status[u] == 0)
           DFS_Traversal(G,u,status);
   status[v] = 2;
   return;
}
```

- Code Implementation:

  DFS_code

- Analysis
  - Time Complexity:

    According to the algorithm DFS_Traversal() is called for each vertex exactly once and in the function we check all the edges from the vertex. So, we are checking all the edges of the graph.

    Thus, Time complexity of the Algorithm is O(V+E) where V = number of vertices of the graph and E = number of the edges of the graph
  - Space Complexity:

    As we are maintaining an additional array of size V+1 to store the information about whether the node is visited or not. The Space Complexity of the algorithm is O(V).

- **Applications:**
    - Helps to find Strongly connected components in a graph.
    - Detect a cycle in a graph
    - To check if the graph is bipartite
    - Find a path between two given vertices.

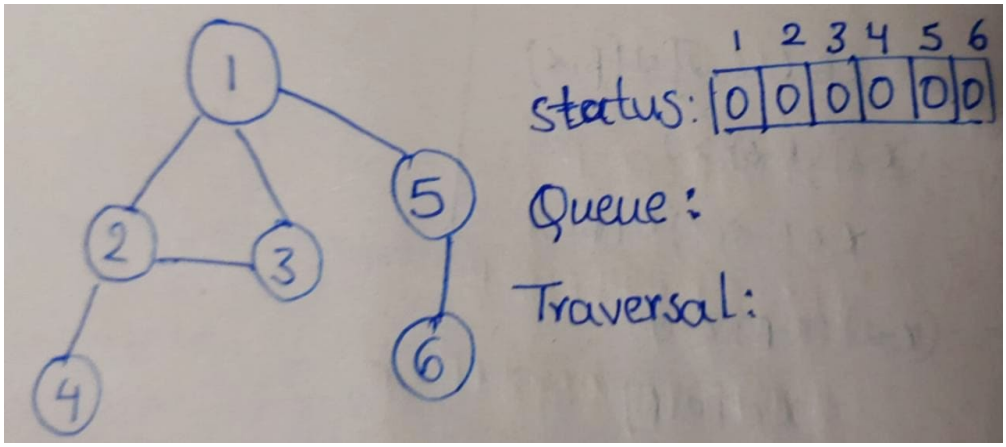
Breadth-First Search:

BFS is also an algorithm to search in or traverse a graph. As the name suggests it visits all the nodes that are present at the same level first and then advances to the next level.This continues till all the nodes of the graph or visited or stops when we find the required node.

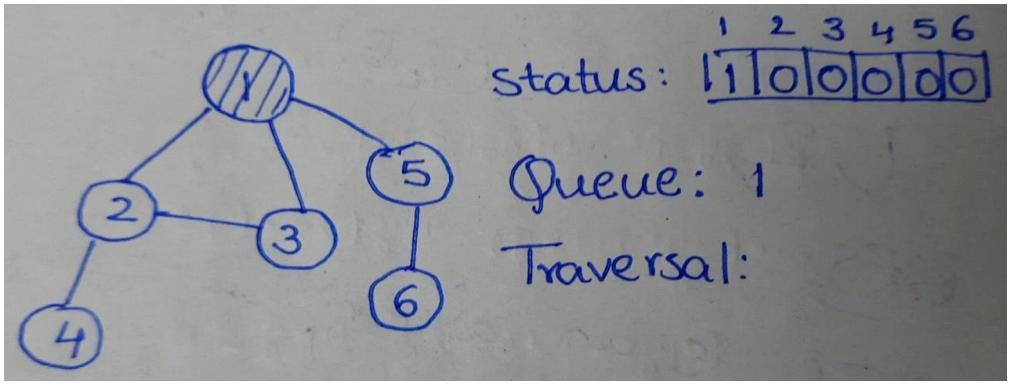
This can be used in both directed and undirected graphs.

- **Algorithm**:
    - We maintain a status array to find out whether the node is not-visited, seen or visited. And we also maintain a queue
    - Now we choose a node (start node) and then push it's neighbours on to the queue and then pop an element from the queue and push it's neighbours on to the queue and so on..
    - We do this till all the nodes are reached from the point and then we choose another node from the graph that is not visited and repeat the same process till all the nodes in the graph are visited.
    - The status of the node that is not visited is set as 0 and the status of the node that is in the queue is set as 1 and the node that is popped from the queue is set as 2.
    - The order of popping the elements is the required BFS traversal.

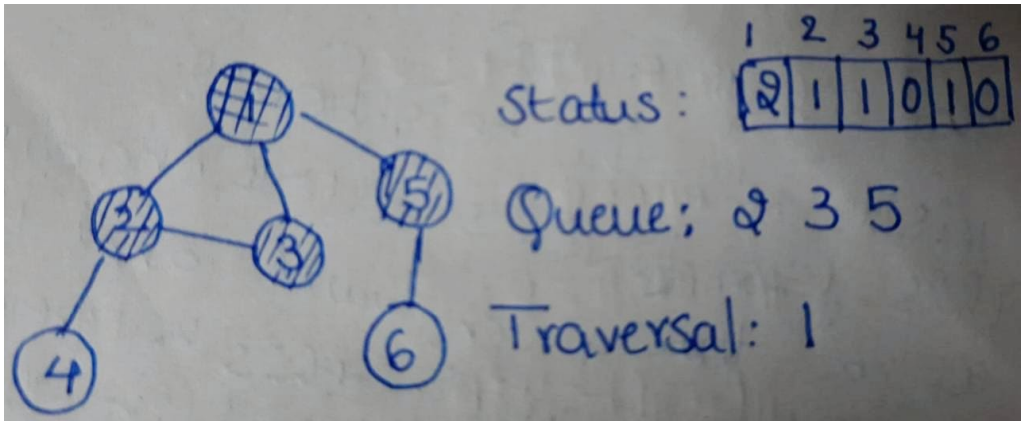
  Let us see an example to see the working of the algorithm.
- **Example**:
    - Consider an undirected graph G shown below



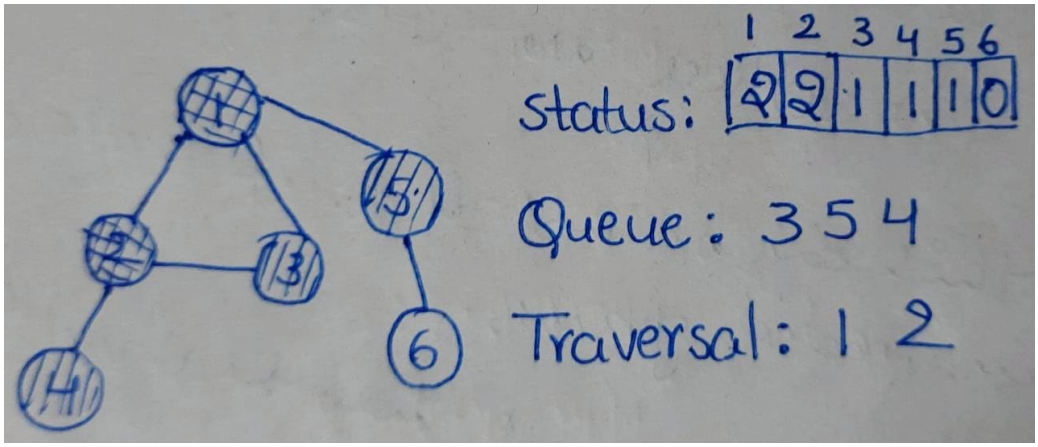
- Initially the status array is initialized to 0. Now we start with 1 and push it on to the queue and update its status.



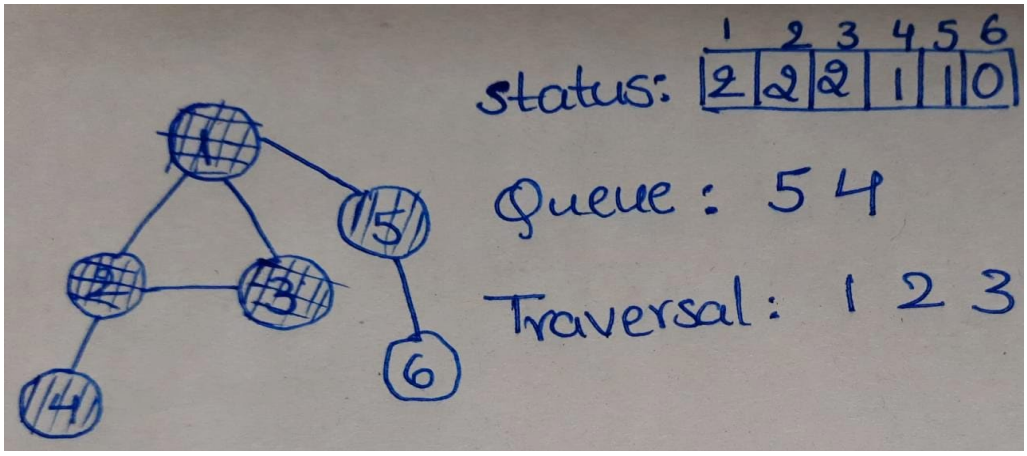
- Now we pop 1 from the queue and push its neighbours 2 3 5 onto the queue and update the status of nodes 1, 2, 3 and 5 to 2, 1, 1 and 1 respectively.



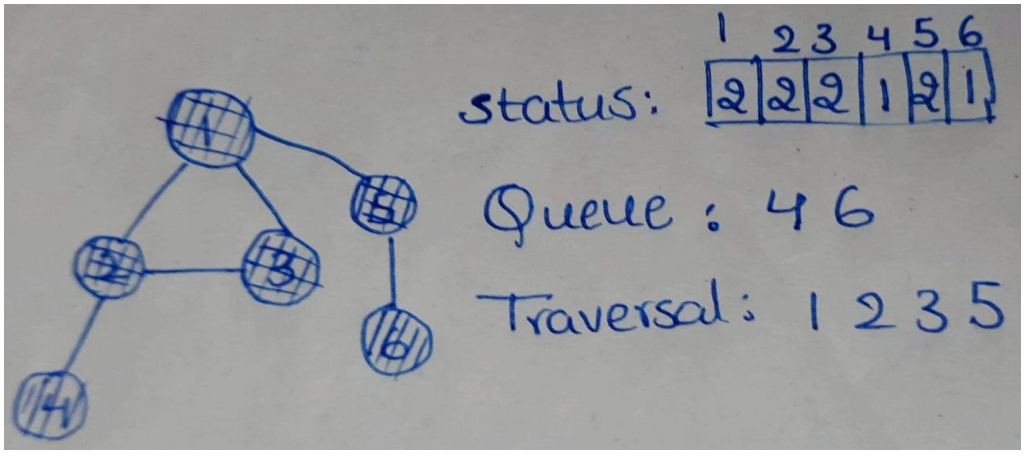
- Now we pop 2 from the queue and push its neighbours onto the queue and update their status.



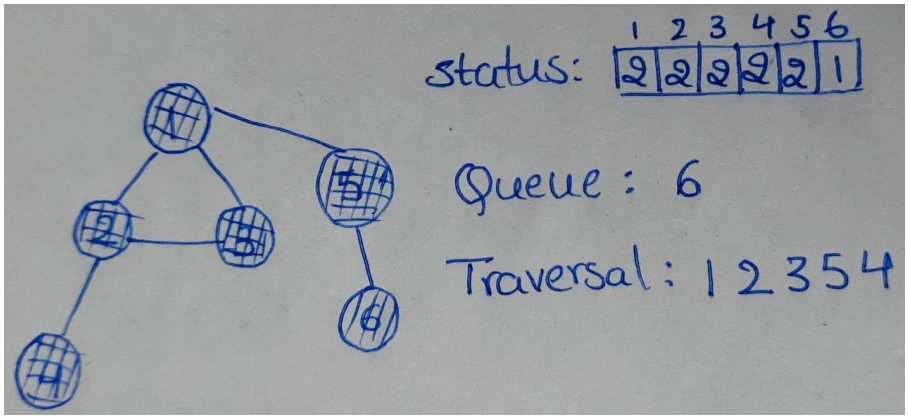
- As there are no neighbours for 3 we just pop 3 from the queue and update the status.



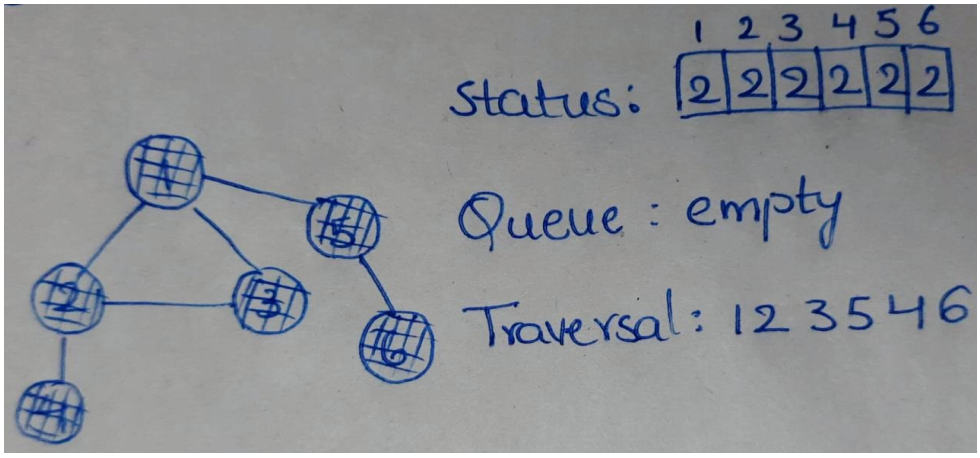
- Now we pop 5 from the queue and push 6 onto the queue and update their status.



- As there are no neighbours for 4 we just pop 4 from the queue and update the status.



- As there are no neighbours for 6 we just pop 6 from the queue and update the status.



- Now, All the nodes of the graph are visited.The BFS traversal is 1 2 3 5 4 6.

- **Pseudo Code:**

```
BFS(Graph G, int V)
{
   vector<int> status(V+1, 0);     // status array
   queue<int> que;          // queue
   for every vertex s ∈ V  // V = set of vertices of G
       if(status[s] == 0)
           que.push(s);
           status[s] = 1;
           BFS_Traversal(G, status, que);
   return;
}
```

```
BFS_Traversal(Graph G, vector<int> &status, queue<int> &que)
{
   while queue is not empty
       int v = que.front();
       que.pop();
       // pushing the neighbours of v into queue
       for every u ∈ Adj[v]      // => there is a vertex from v to u
           if(status[u] == 0)
               que.push(u);
               status[u] = 1;      // u is seen
       status[v] = 2;          // v is visited
   return;
}
```

- Code Implementation:
  [BFS_code]

- Analysis:
  - Time Complexity:
    Each vertex of the graph is pushed and popped from the queue exactly once.The operations pushing and popping from the queue are constant time operations.So, the time taken to do this is O(V).
    For each vertex we are checking all the neighbours. So, in total we are checking all the edges in the graph and the time taken to do this is O(E).
    Thus, Overall time taken by the algorithm is O(V+E) where V = no.of vertices and E = no.of edges of the graph.
  - Space Complexity:
    As we are maintaining an array and queue which in the worst case may contain V-1 elements. The space complexity of BFS is O(V).
- Applications:
  - Cycle detection in undirected graphs
  - To test if a graph is bipartite
  - Find path between two vertices
  - GPS navigation systems to find all neighbouring places.
  - Shortest path in undirected graph.

Sorting a graph:

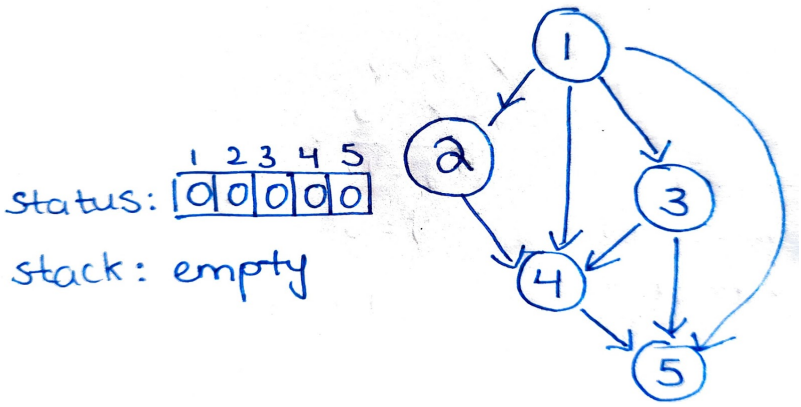
Topological Sort:

Topological sort is to linearly order vertices of a graph in such a way that if there is an edge from u to v then u comes before v in the ordering. And this sorting is only possible if a graph is directed acyclic graph (DAG).

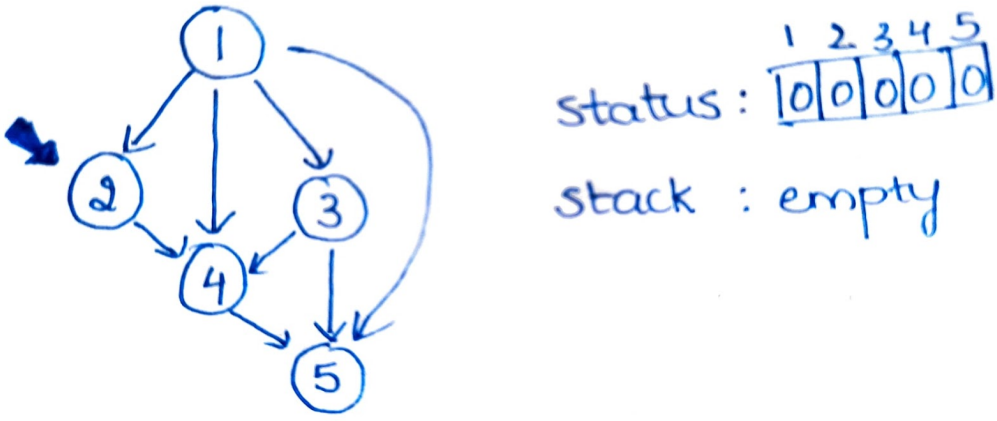
- **Algorithm:**
  - We start with a node and recurse for all its adjacent vertices and mark a node as visited if all its adjacent vertices are visited.
  - Similar to DFS and BFS we maintain an array to know whether the vertex is visited or not.
  - We also maintain a stack and push a node that is visited onto the stack.
  - We continue this till all the vertices of the graph are pushed onto the stack. Then we pop the elements of the stack and print them to get the sorted order of vertices of the graph.

  Let us see an example to see the working of the algorithm.

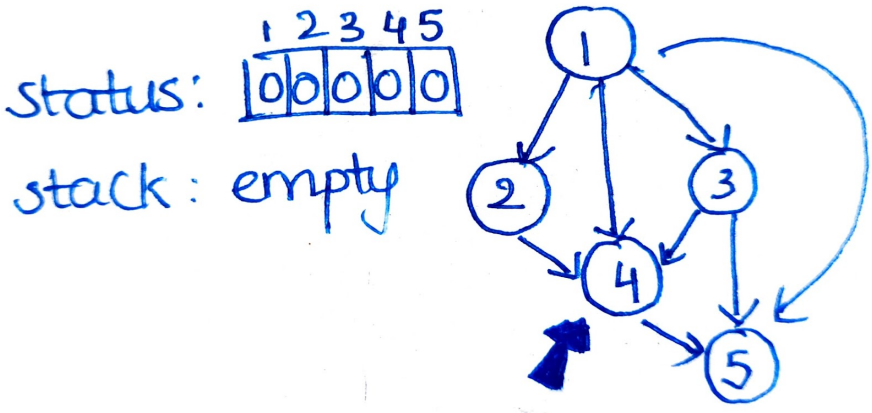
- **Example:**
  - Consider the DAG G shown below.



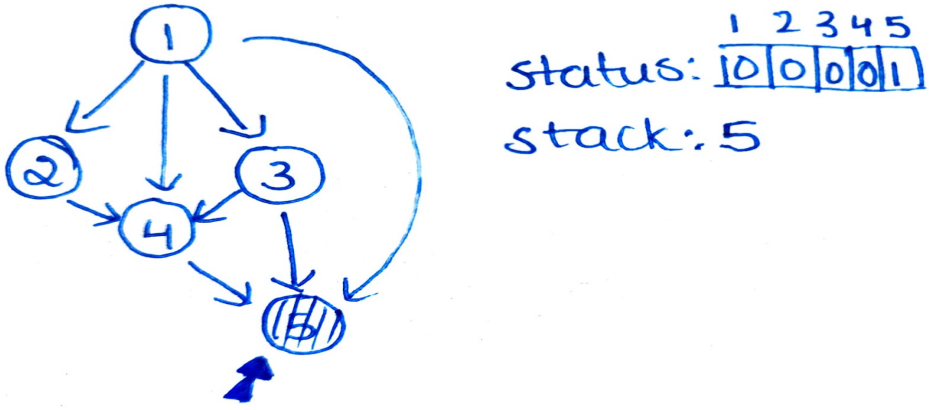
  - Now let's start from node 1 as its adjacent nodes 2, 3 and 4 are not visited. And move to one of them. Let say node 2



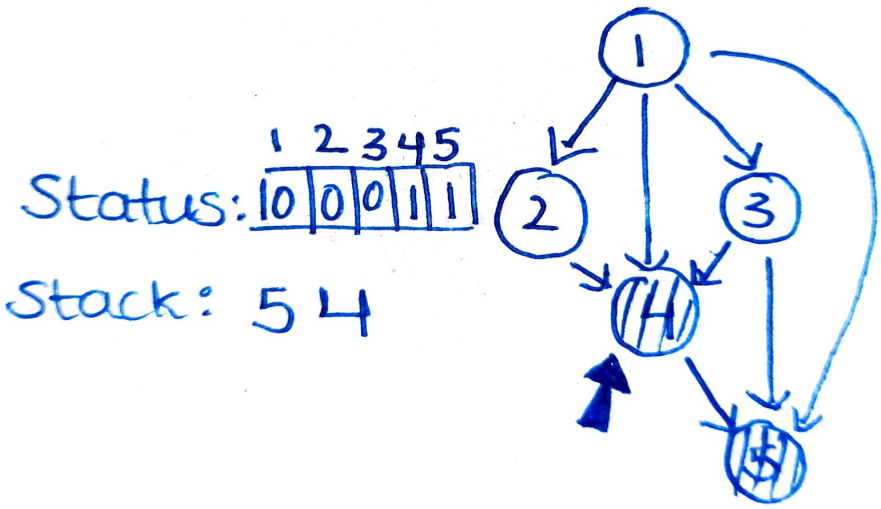
- Now we check the adjacent vertices of 2 and as they are not visited we move node 4 (adjacent vertex of 2).



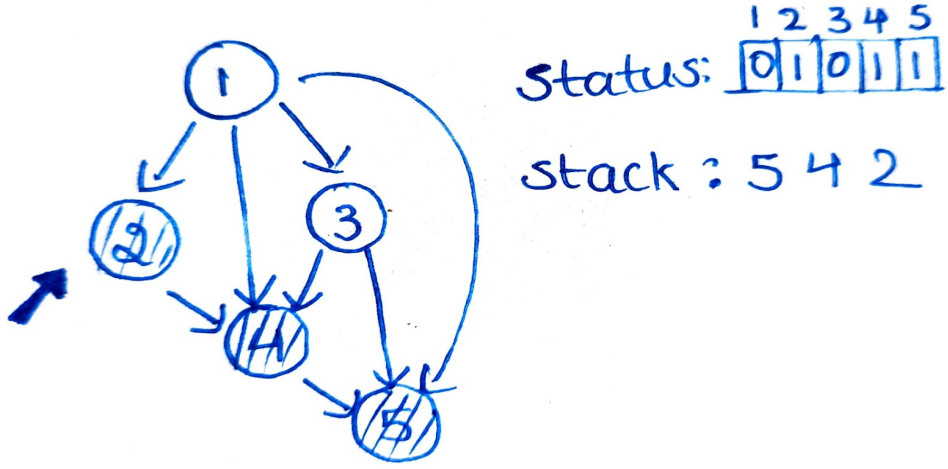
- Now we check the adjacent vertices of 4 and as they are not visited we move node 5 (adjacent vertex of 4). As there are no adjacent nodes for node 5 we mark 5 as visited and push it onto the stack.



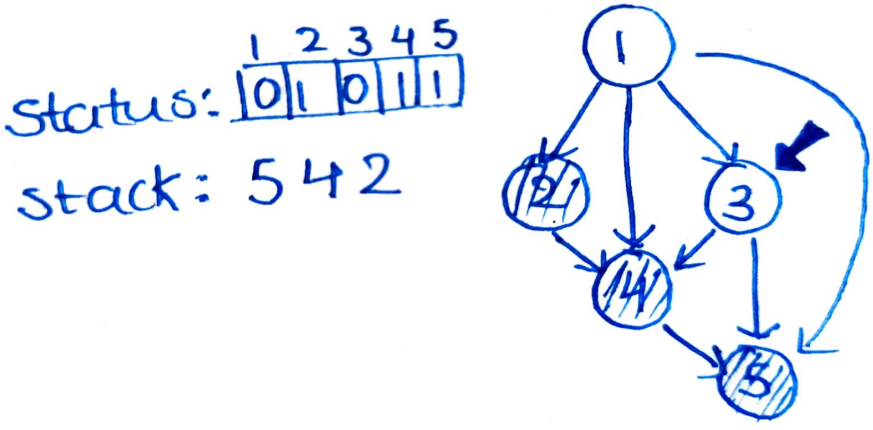
- Now we recurse back to node 4 and as all the adjacent nodes of node 4 are visited we mark 4 as visited and push it onto the stack.



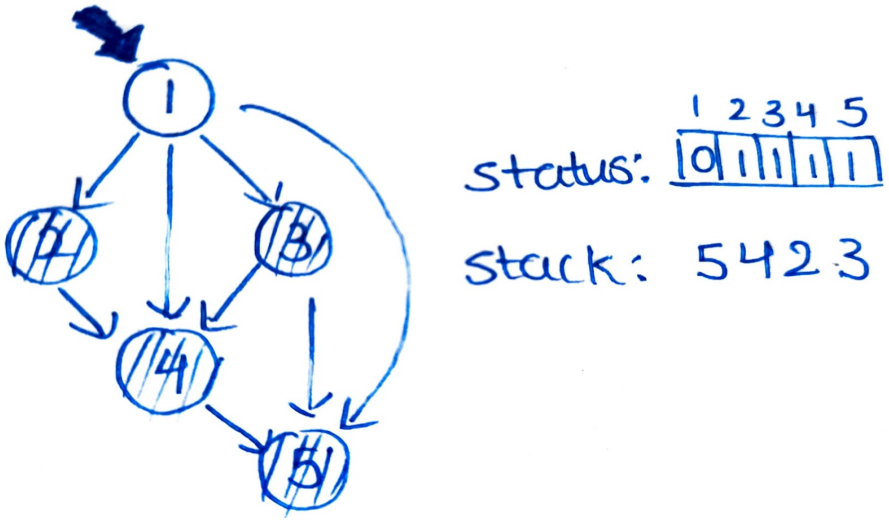
- Again we recurse back to node 2 and as all its adjacent nodes are visited we push it onto the stack and mark it as visited.



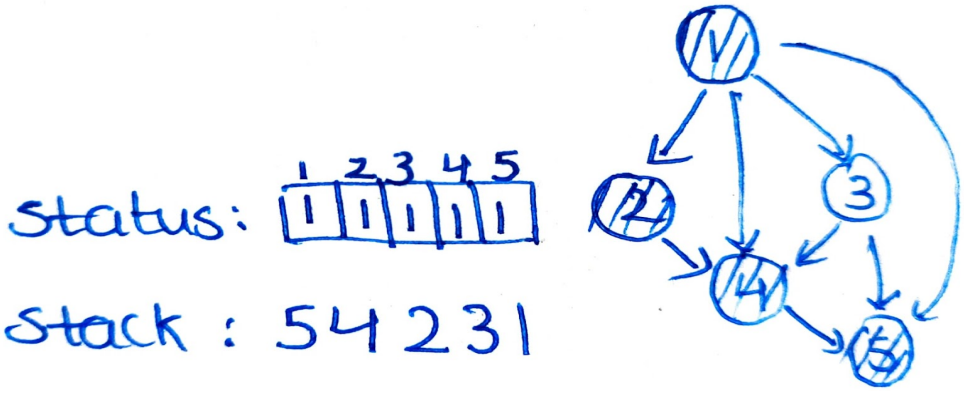
- Then we recurse back to node 1 and as node 3 that is adjacent to node 1 is not visited we move to node 3.



- As all the adjacent vertices of node 3 are visited we mark it as visited and push it onto stack and recurse back to node 1.



- Now as all its adjacent nodes are visited we push 1 onto the stack.



- There is no node to recurse back and as all the vertices are pushed onto the stack we pop out the stack to get the order. Here the order is 1 3 2 4 5

- **Pseudo Code:**

```
Topological_Sort(Graph G,int v,vector<bool>& status,stack<int>& stck)
{
   for every u ∈ Adj[v]        // for every adjacent node of v
       if(status[u] == 0)  // if u is not visited
           Topological_Sort(G,u,status,stck);

   stck.push(v);     // push onto stack
   status[v] = 1;      // mark v as visited

   return;
}
```

```
TSort(Graph G, int V)
{
      vector<bool> status(V+1, 0);
      stack <int> stck;

      for every vertex v ∈ V
            if(status[v] == 0)
                  Topological_Sort(G,v,status,stck);
      return;
}
```

- Code Implementation:
  TopologicalSort_Code

- Analysis:
  - Time Complexity:
    The Algorithm is similar to DFS as we are performing the traversal and pushing the node onto the stack without printing it. And the push and pop operations are constant time operations.So, the time complexity of the algorithm is the same as time complexity of DFS, that is O(V+E) where V = no.of vertices and E = no.of edges.
  - Space Complexity:
    The space complexity of the algorithm is O(V) as we are maintaining a stack whose maximum length is V.

- Applications:
  - Used for scheduling jobs from the given dependencies among them.
  - Used to order compilation tasks.

Chapter 3: Shortest Paths

In our daily life, finding the shortest path between two places is important inorder to reach the destination earlier. This problem can be solved using graph algorithms (Bellman-Ford Algorithm, Dijkstra and Floyd-Warshall Algorithm).

Here Bellman-Ford and Dijkstra are single source shortest path algorithms where Folyd-Warshall is All pair shortest path algorithm.

In an unweighted graph we can easily find the shortest path using BFS as the length of the path is equal to the number of edges in the path. To find the shortest path in a weighted graph we use the above mentioned algorithms.

Single Source Shortest Paths

Bellman-Ford Algorithm

Finds the shortest path from the given starting point(source) to all other points in the graph.

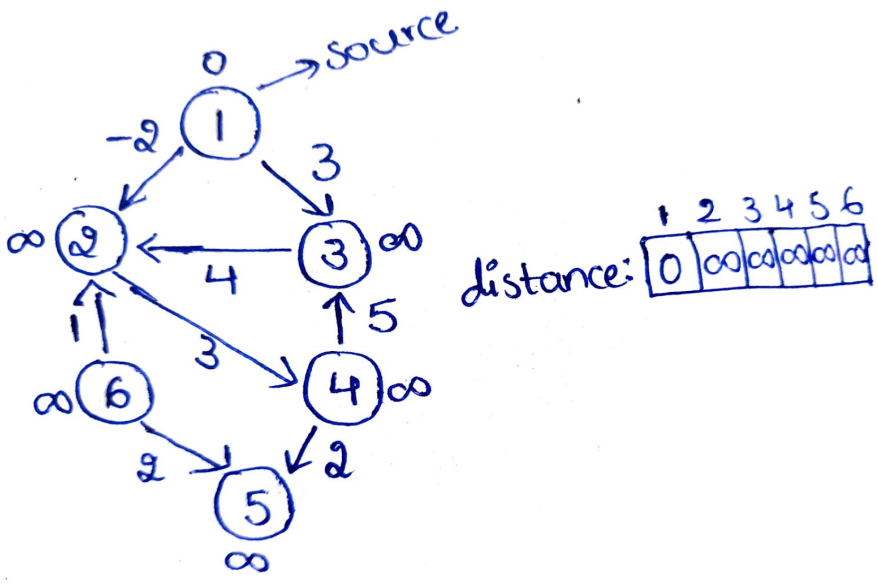
IDEA:

We initially assume that all points are at infinite distance from the source and then we reduce the distances by finding edges that decrease the length of the path until it finds the actual shortest path.

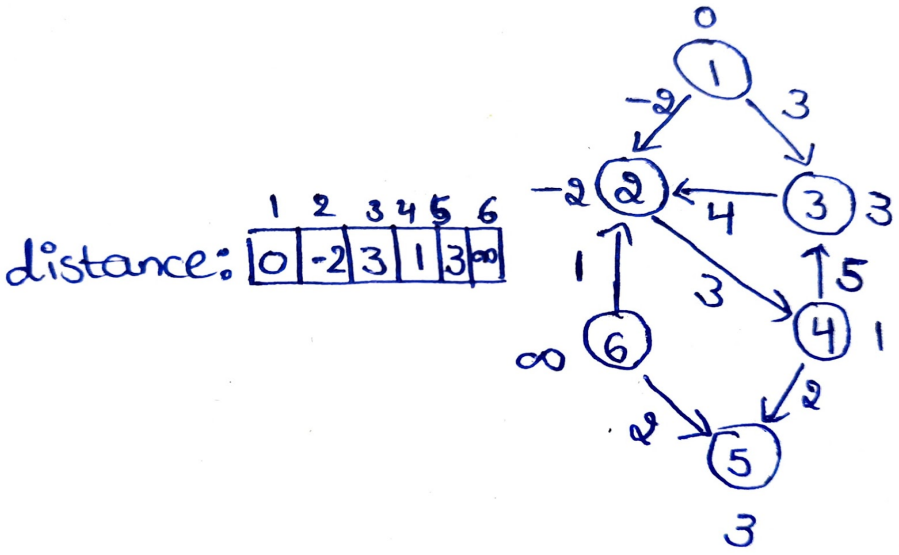
This Algorithm works for all the graphs and if there are any negative cycles in the graph it detects them. As if there is a negative cycle in the graph then the shortest path is not defined because we can loop multiple times and reduce the path further.

Let us see an example to see the working of the algorithm.

- **Example:**
    - Consider an indirect graph G = (V, E) shown below.



- Initially the distance from source to all other nodes is considered to be infinity.
- After 1st iteration according to the algorithm we get



- And the result after another iteration is also the same. So, the above distance array contains the distances of all the shortest paths from source.

- **Algorithm**:
  - Initialize elements of array(distance) of length |V| to infinity and distance[source] to zero.
  - Then for every edge (u,v) in the graph check if distance[v] is greater than distance[u] + weight of the edge between u and v. If this is true then update the value of distance[v] to

distance[u] + weight of the edge between u and v. Repeat this |V|-1 times.
    - We can break from the loop if there is no change in the distance array.
    - The values in the array are shortest distances from the source to the corresponding point if there is no negative cycle.
    - Now, repeat the loop once more, if the distances decrease then there is a negative cycle in the graph. Otherwise the values in the array are the shortest distances from the source.
    - We can also get the actual path by maintaining the vertex from which we get the shortest distance along with distance.

- Pseudo Code:

```
Bellman_Ford(Graph G, int s, int V)
{
   vector<int> distance(V+1, INFINITE);
   distance[s] = 0;

   for(int i = 1;i < V; i++)
       for every (u,v,w) ∈ E    // here w is the weight of the edge
           if(distance[v] > distance[u]+w && distance[u] != INFINITE)
               distance[v] = distance[u] + w;

   for every (u,v,w) ∈ E

       if(distance[v] > distance[u] + w && distance[u] != INFINITE)
       {
           cout << "Negative cycle detected" << endl;
           return;
       }

   for(int i = 1;i <= V+1; i++)
       cout << distance[i] << " ";
   cout << endl;
   return;
}
```

- **Code Implementation:**
  BellmanFord_Code

- **Analysis:**
  - **Time Complexity:**
    - As In the worst case we check all the edges |V| times the time complexity of the algorithm is O(V*E) where V = no.of vertices and E = no.of edges of the graph.
    - The best case time complexity of the algorithm is O(E) here we get all the shortest paths in one iteration.
  - **Space Complexity:**
    - As we need to maintain an array of length V, the space complexity of the algorithm is O(V).

Dijkstra Algorithm: (A greedy algorithm)

Is a single source shortest path algorithm. Given a vertex it finds the shortest path from the vertex to all other vertices in the graph. Works for both directed and undirected graphs that do not contain negative edges.

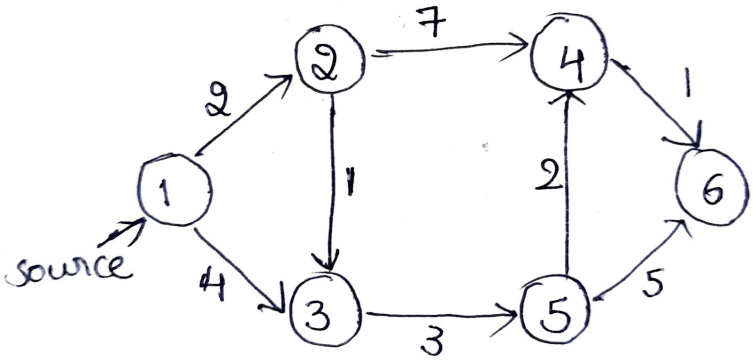
IDEA:

We initialize the distances of all the vertices from source to infinity and then update the distances of the neighbours of the source node. Next, we choose a vertex that is at the shortest distance from the source node and update the distances of the neighbours from the node. We do this till all the nodes are visited.

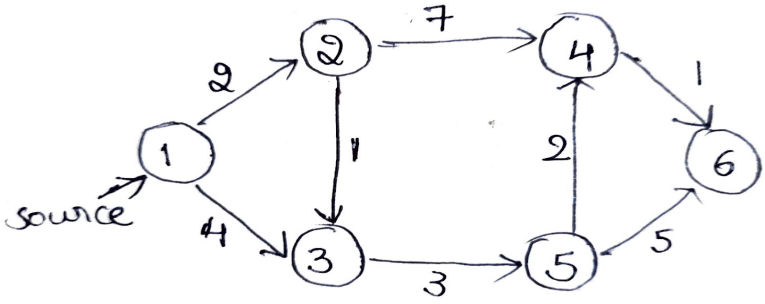
Then the resulting distances in the array give the shortest distances of vertices from the source node.

Let us see an example to see the working of the algorithm.

- **Example:**
  - Consider the weighted, directed graph given below



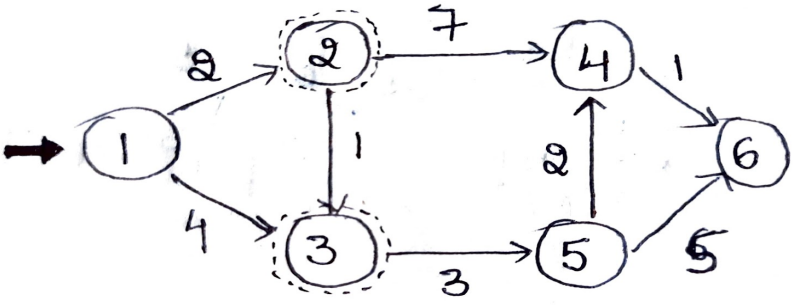
  - Initially the distance of all the vertices from the source is infinity.



distance:

1	2	3	4	5	6
0	∞	∞	∞	∞	∞

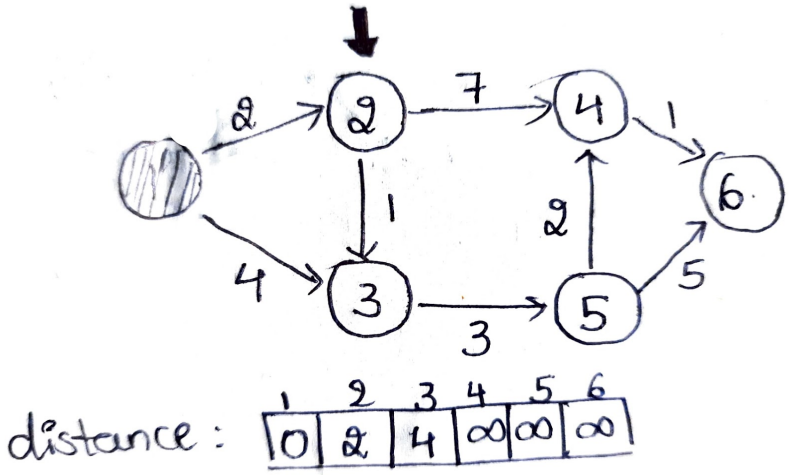
  - Now, we update the distance of the neighbours of node 1.



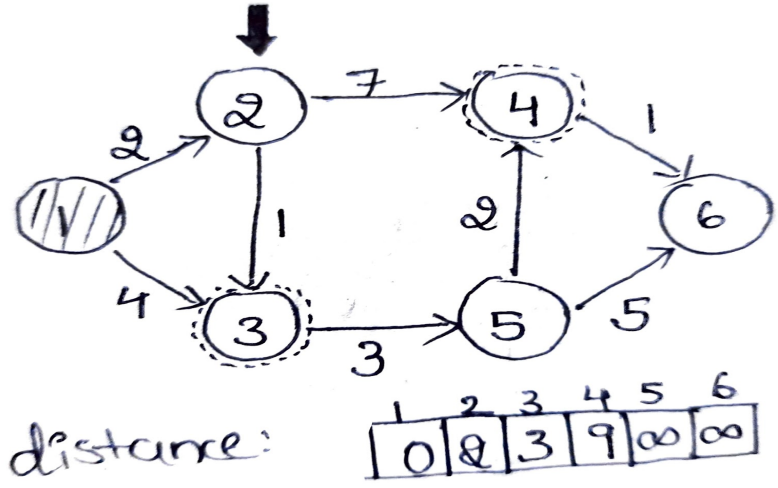
distance:

1	2	3	4	5	6
0	2	4	∞	∞	∞

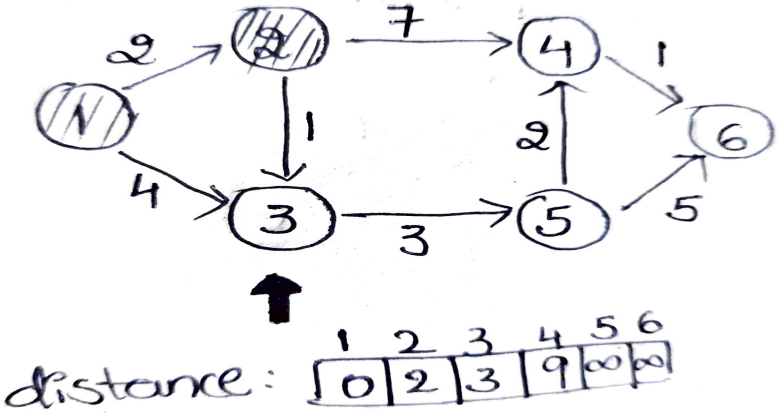
  - Then, we move to a neighbour that is at the shortest distance from node 1 that is, node 2.



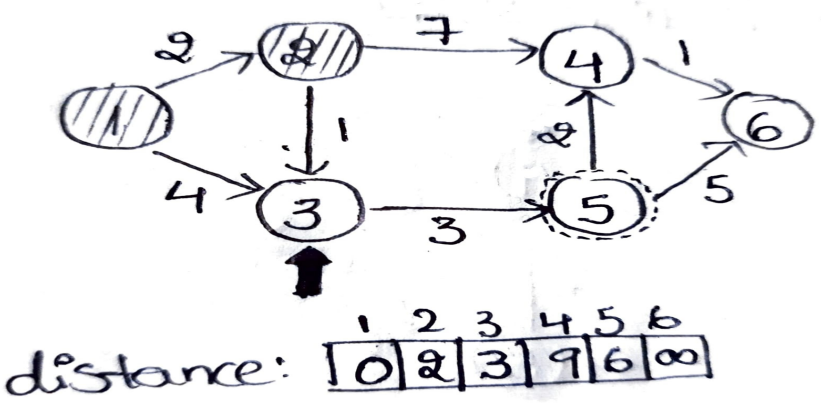
- Now, we update the distance of the neighbours of node 1.



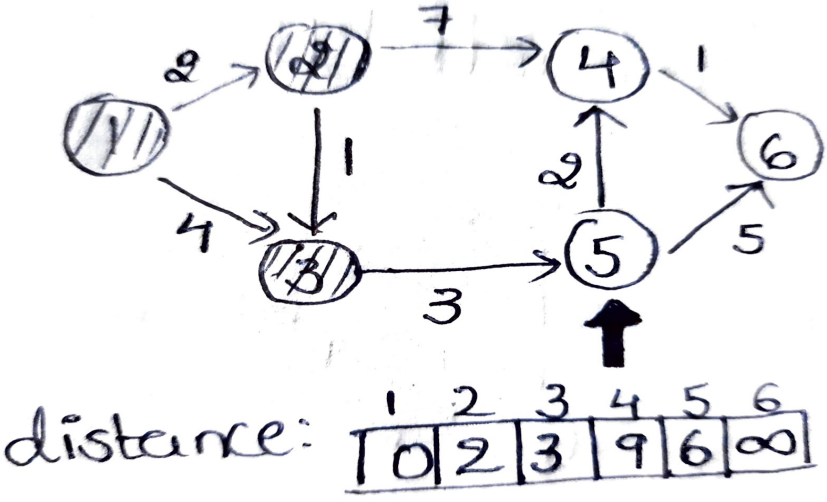
- Then, we move to a neighbour that is at the shortest distance from node 2 that is, node 3.



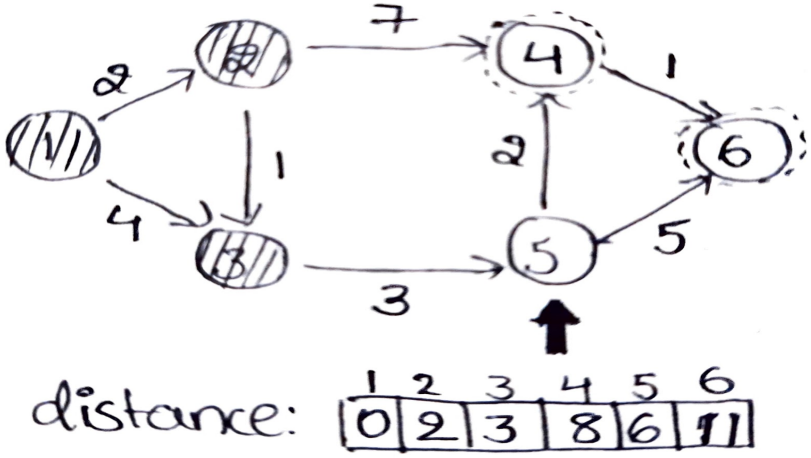
- Now, we update the distance of the neighbours of node 3.



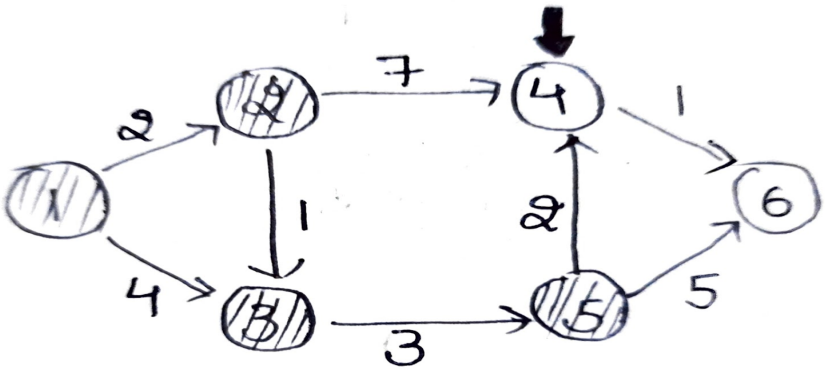
- Then, we move to a neighbour that is at the shortest distance from node 3 that is, node 5.



- Now, we update the distance of the neighbours of node 5.



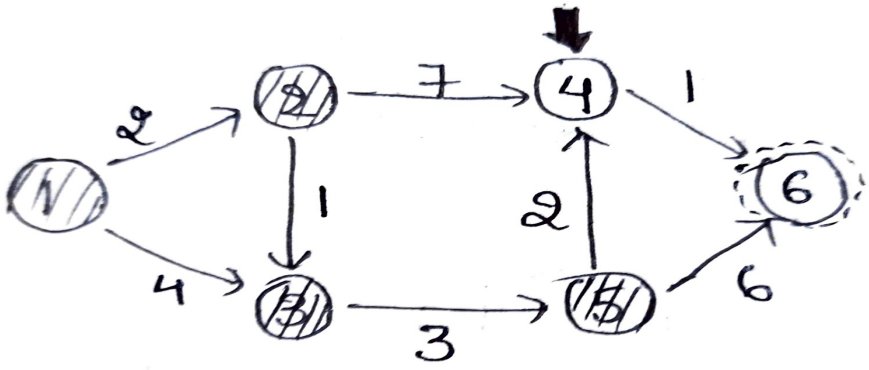
- Then, we move to a neighbour that is at the shortest distance from node 5 that is, node 4.



distance:

1	2	3	4	5	6
0	2	3	8	6	11

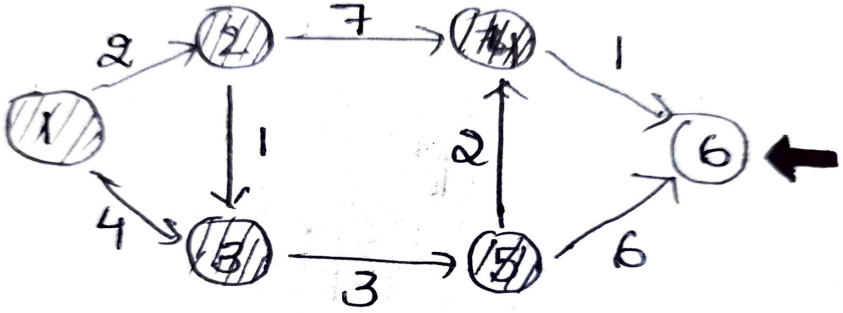
- Now, we update the distance of the neighbours of node 4.



distance:

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

- Then, we move to a neighbour that is at the shortest distance from node 4 that is, node 6.



distance:

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

- Now as there are no edges from node 6 we mark node 6 as visited. As all the nodes of the graph are visited. The distance array we get are the shortest distances of the vertices from the source node.

- Algorithm:
    - We maintain a priority queue (min heap) so that we get the vertex with minimum distance from source in O(1). And we also maintain an array that keeps track of the position of the vertex in the priority queue so that we can update the distance in O(1).
    - Now, we initialize all the distances of vertices from the source to infinity and the distance of source to 0. And use the UP_Heapify() to heapify the priority queue so that source node is at the root.
    - Then, extract the minimum value from the priority queue and swap the min node with the last element of the queue and decrease the size of the queue.Now, we restore the priority queue property by using the Down_heap() function that brings the largest value to the bottom of the priority queue.
    - Now, for the neighbours of the min node we update the distance in the priority queue and use UP_Heapify() function to move the lowest value to the top of the priority queue.
    - We repeat the above two steps till all the vertices of the priority queue are popped.
    - Now, the distances in the priority queue are the shortest distances of vertices from the source vertex.

- Pseudo Code:

```
Dijkstra(Graph G, int source, int V)
{
   edge_weight pque[V]; // creating the priority queue
   int positions[V];//Array that keep track of positions in priority
queue
   for i in V:     // initializing the distances
       pque[i].vertex = i+1;
       positions[i] = i;
       if(i+1 != s)
           pque[i].weight = INFINITY;
       else
           pque[i].weight = 0;
```

```
    UP_Heapify with respective the source vertex
    while queue is not empty
        min_node = extract min value from pque
        swap(min_node, last node)
        Update the position array
        size_of_queue--
        Down_Heapify(root)
        if(distance(u) != INFINITY) // here u is the vertex of min_node
            for every (u,v,w) ∈ E
                if(distance(v) > distance(u) + w)
                    distance(v) = distance(u)+w
                    UP_heapify(v)
    // printing the distances
    for(int i = 0; i < V; i++)
        cout << pque[positions[i]].weight << " ";
    cout << endl;
}
```

- Code Implementation:
  [Dijkshtra_Code]

- Analysis:
  - Time Complexity:
    - As the loops are similar to BFS the time complexity will be O(|V|+|E|) but the operations in the loop UP_heapify takes O(log|V|) so, the total time complexity of the Dijkstra Algorithm is O((|V|+|E|)log|V|) i.e, O(|E|log|V|). (since, |V| is less than |E|)
    - The time complexity can be further reduced using Fibonacci heaps to O(|E| + |V|log|V|).
  - Space Complexity:
    - As we additionally need to maintain a priority queue and an array the space complexity of dijkstra algorithms is O(|V|).
- Applications:
  - In G-Maps to find the shortest path between two locations.
  - In social networking websites like facebook to suggest friends.

- **Drawbacks**:
    - May or may not work on the graphs that have negative edges. To handle these graphs Bellman-Ford Algorithm is used.

All Pairs Shortest Paths

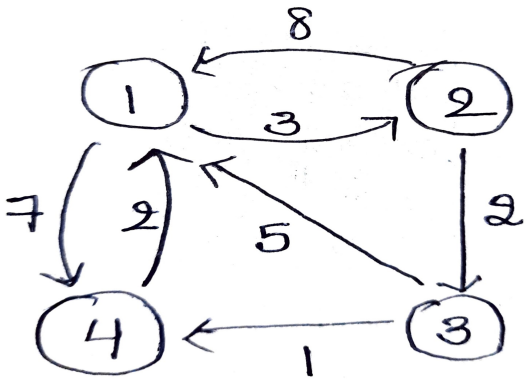
Floyd Warshall Algorithm

All Pairs shortest path algorithm. This finds the shortest path between all two vertices of the graph. Works on both directed and undirected graphs that contain negative edges but not negative cycles.

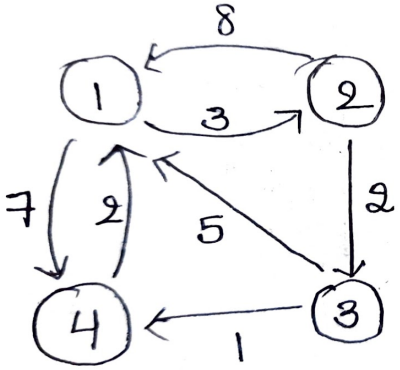
- **Algorithm:**
    - We represent the graph using an adjacency matrix. And initialize all the distances to infinity and modify the vertex to itself as 0.
    - Then modify the distances based on the given input(distance between two vertices when there are no intermediate vertices).
    - Now, we check whether choosing node 1 as an intermediate vertex reduces the distance between vertices and update the distance if it reduces the distance.
    - Now, on the updated matrix we choose node 2 as intermediate matrix and modify the distances. Similarly we perform this for all other vertices in the graph.
    - And the resulting matrix contains the distances of shortest paths from any vertex to other vertex in the graph. This works as we are considering all the intermediate vertices that reduce the path between two vertices.

Let us see an example to see the working of the algorithm.

- **Example:**
  - Consider the directed graph G given below.



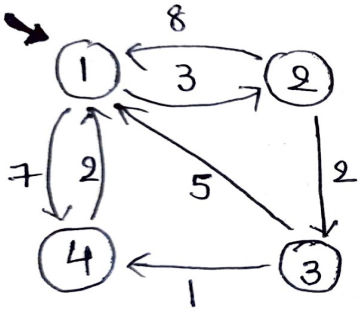
  - Initially when no intermediate vertices are considered



	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

distance matrix

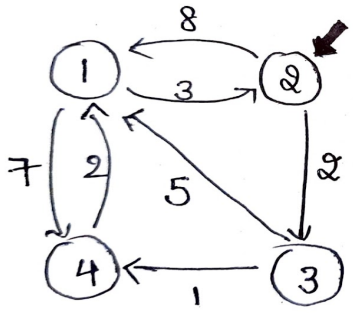
  - Now, when the node 1 is considered the above distance matrix is modified as



	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

distance matrix when 1 is intermidiate

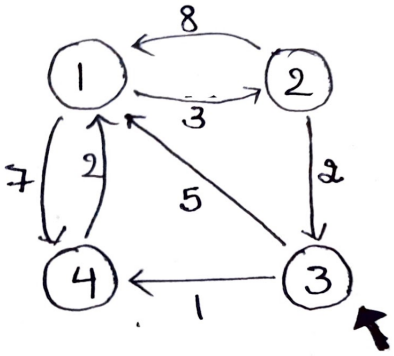
  - Then on considering node 2 as an intermediate vertex we get



	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

distance matrix when 2 is intermidate vertex

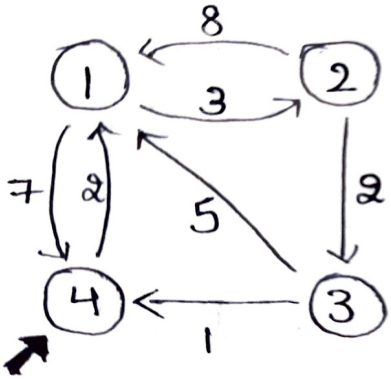
- When node 3 is considered as intermediate then the distance matrix is



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

distance matrix

- Finally on considering node 4 as intermediate vertex, the distance matrix (the final result) is



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

distance matrix

- Pseudo Code

```
void Floyd_Warshall(int** distance)
for u in V:          // initialize the distance matrix
   for v in V:
       if(j == i)
           distance[i][j] = 0;
       else
           distance[i][j] = INFINITY;
```

```
for (u,v,w) ∈ E:        // When there are no intermediate vertices
   distance[u][v] = min(w, distance[u][v]);

for n in V:     // n is the intermediate node
   for u in V:     // updating the values of the matrix
       for v in V:
           distance[u][v] = min(distance[u][v], distance[u][n] +
distance[n][v]);
```

- Code Implementation
  - FloydWarshall_Code

- Analysis
  - Time Complexity:

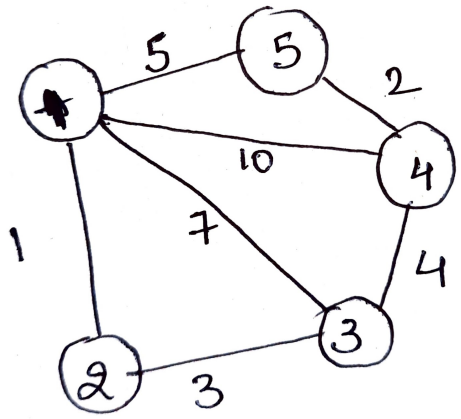
    As there is a 3 nested for loop and each runs |V| times, the complexity of the Algorithm is O(|V|^3).
  - Space Complexity
    - As we maintain a 2D matrix (distance array) of size |V+1|*|V+1|, the space complexity of the algorithm is O(|V|^2).
- Applications
  - Helps in finding shortest paths
  - Used in inversion of real matrices
  - To check whether the graph is bipartite or not.

Chapter 4: Minimum Spanning Trees

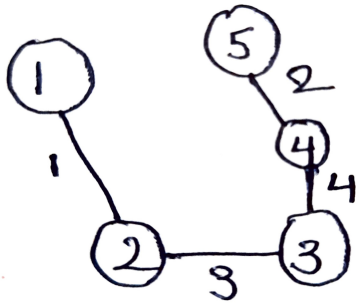
Spanning Tree:

A tree that connects all the vertices of the graph with the minimum number of edges possible. So, spanning trees never contain a cycle. Since it is a tree it is always connected. Thus, disconnected graphs do not have spanning trees. But every connected component of the graph has a spanning tree.

There can be more than one spanning tree possible for a graph.



Graph G

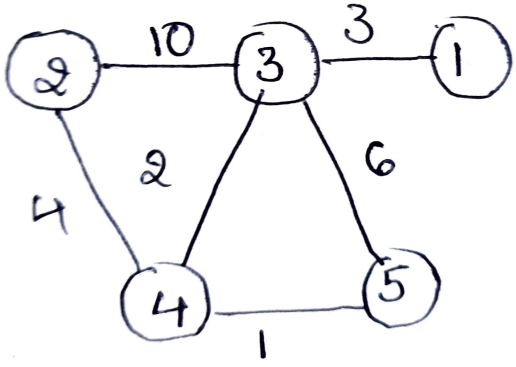


Spanning Tree

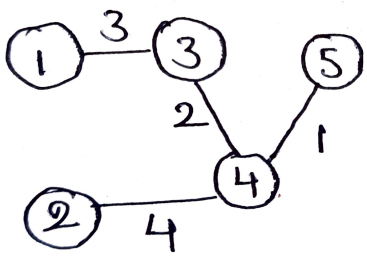
Minimum Spanning Tree:

This is defined for weighted graphs, A spanning tree having minimum weight is defined as a minimum spanning tree. There can be more than one minimum spanning trees for a graph.

In the above given example the spanning tree is a minimum spanning tree of Graph G.



Graph G

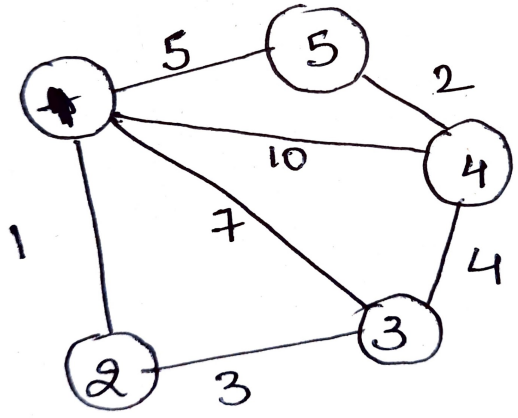


Minimum spanning tree

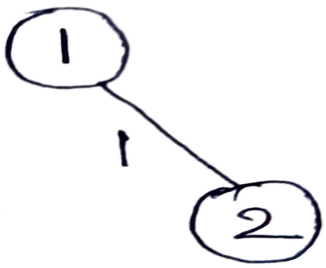
Kruskal Algorithm:

Used to find a minimum spanning tree in the given graph. Will work both on connected and disconnected graphs but not on directed graphs as it fails to detect cycles in a directed graph.

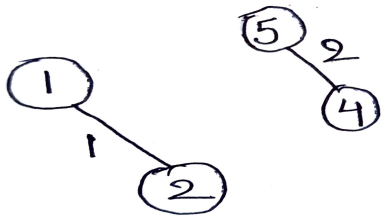
- **Algorithm:**
  - We store the input graph as a list of edges and then sort the list in increasing order based on the edge weights.
  - Now, we start from the edge with least weight and add the next edge if it does not form a cycle.
  - We repeat the step 2 until we add |V|-1 edges (as the minimum spanning tree of a graph has |V|-1 edges).
  - Here we detect whether a cycle is present or not by union-find method and to optimize it we use path compression method.
  - Let us see an example to see the working of the algorithm.
- **Example**:
  - Consider the undirected graph G given below.



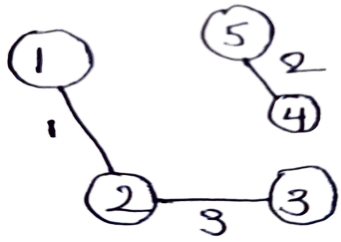
  - Now we sort the edges of the graph based on weights and select the edge with minimum weight. Here edge between nodes 1 and 2.



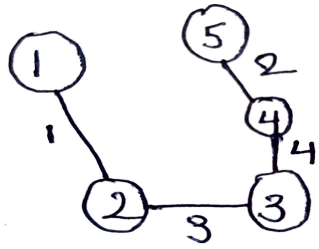
- Then we select the next smallest edge that is the edge between nodes 4 and 5 and add it as it does not form a cycle.



- Next we select the edge be nodes 2 and 3. As it does not form a cycle we add it.



- Then the next smallest edge is between nodes 3 and 4 as it does not form any cycle we add it.



- This is the resulting minimum spanning tree as the number of edges of the tree are |V|+1. And the weight of the tree is 10.

- Pseudo Code

```
Kruskal(Graph G)
{
   sort(list of edges);
   for (u,v,w) in List_of_Edges
       find the parent(King :P) of u and v
       if(parent(u) != parent(v))      // No cycle
           union(parent(u), parent(v)) based on rank//Increase kingdom
           add edge to the MST
}
```

- Code Implementation
  - Kruskal_Code

- Analysis
  - Time Complexity

    We need O(|E|log|E|) (i.e O(|E|log|V|)) time to sort the list of edges.Then for every edge we perform union based on rank and find along with path compression and this in worst case takes O(log|V|) time. As we do this for all the edges the time complexity to perform this task would be O(|V|log|V|). Thus, total time complexity of the algorithm is O((|V|+|E|)log|V|).
  - Space Complexity

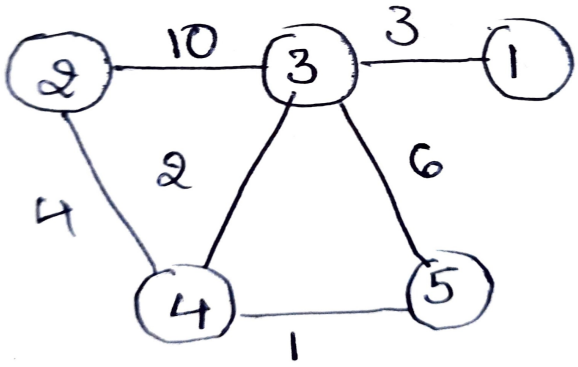
    As we are maintain an array to store rank and parent of the vertex this adds O(|V|) of space complexity and O(|E|) space complexity is added while dealing with edges.

Prim's Algorithm

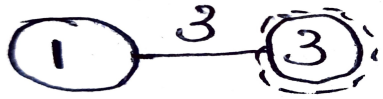
This is similar to the Dijkstra algorithm for finding the shortest path. This algorithm works on undirected graphs but does not work on directed graphs as it assumes all the vertices are connected. In this we partition vertices into two sets. One set contains the vertices that belong to MST and another contains vertices that do not belong to MST. In code we implement this using priority queue(min heap) to reduce the time complexity.

- Algorithm
  - We start with an initial vertex and initialize its value to 0 and all other values of the vertices to infinity. We maintain a priority queue that contains the vertices and their corresponding values and the priority queue is maintained based on the values of vertices.
  - Now, we pop the vertex that has minimum value from the queue and update the values of its neighbours to the weight of the edge between them if the value of the neighbour is greater than the weight of the edge between them.(Here, the vertex that is popped is part of the MST).
  - We repeat the above step till all the vertices are part of the MST(i.e, till the queue is empty).
  - Let us see an example to see the working of the algorithm.

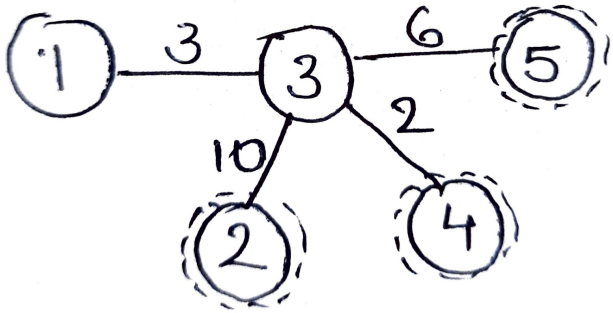
- Example
  - Consider the undirected graph given below



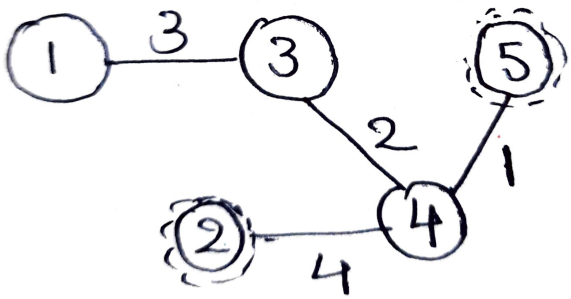
  - Consider, node 1 to be the initial vertex then adjacent vertex to node 1 is 3. So we update the distance in the priority queue. (Only vertices with finite value in the priority queue are shown in the diagram).



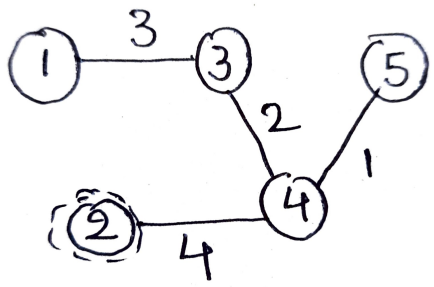
  - Now, we pop 1 from the priority queue and select the vertex with minimum value in the queue. Here node 3.



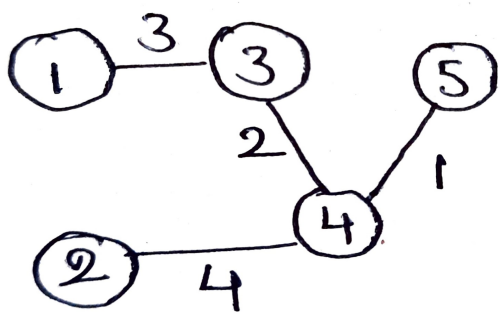
  - Now, we update the values of neighbours of node 3 that are not part of the queue. Pop node 3 and select a node that has a minimum value in the queue (i.e node 4).



- Now, we update the values of neighbours of node 4 that are not part of the queue. Pop it from the queue and select a node that has a minimum value in the queue (i.e node 5).



- As there are no neighbours for node 5 that are part of the queue we pop it from the queue and choose the vertex with minimum value from the queue(i.e, node 2).



- This is the resulting MST as all elements of the queue are popped. And the weight of the MST is 10.

- Pseudo Code

```
Prims(Graph G)
{
   vertex_value priority_q[V]; // creating the priority queue
   int positions[V];  // array that keep track of positions in
priority queue
   int weight = 0;     // weight of the MST
   for i in V:     // initializing the distances
       priority_q[i].vertex = i+1;
       positions[i] = i;
       if(i+1 != 1)
           priority_q[i].value = INFINITY;
       else
           priority_q[i] = 0;
```

```
UP_Heapify with respective the vertex 1

  while queue is not empty
      min_node = extract min value from pque
      swap(min_node, last node)
      size_of_queue--
      Down_Heapify(root)

      for v in Adj_list[u]:  // here w is distance between u and v
          if(value(v) > w)
              value(v) = w;
              UP_Heapify(v);
      weight += min_node.value;
}
```

- Code Implementation
  [Prims_Code]

- Analysis
  - Time Complexity
    - As the algorithm is same as dijkstra with a minor modification, the time complexity is same as Dijkstra Algorithm.i.e, O(|E|log|V|).
  - Space Complexity
    - As we additionally need to maintain a priority queue and an array the space complexity of Prim's algorithm is O(|V|).

Chapter 5: Cycles

Cycle:

A trail in which the first and last vertices are the same.

Directed Cycle:

A directed trail in which the first and last vertices are the same in a directed graph.

Undirected Cycle:

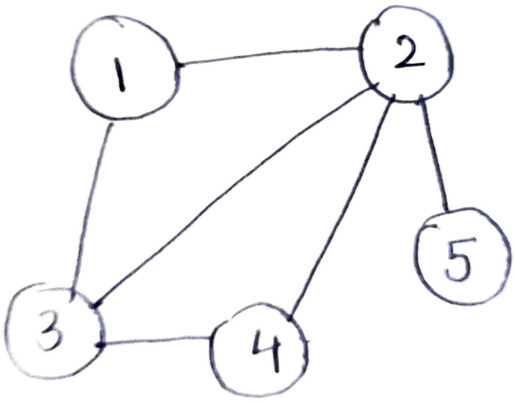
An undirected trail in which the first and last vertices are the same in an undirected graph.

Refer Depth First Search before this section as this modifies DFS.

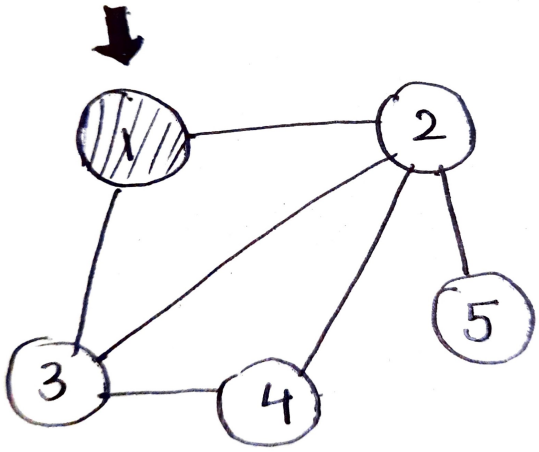
Cycle Detection in Undirected Graph:

As discussed in chapter 1 we use DFS to detect a cycle in the undirected graph.

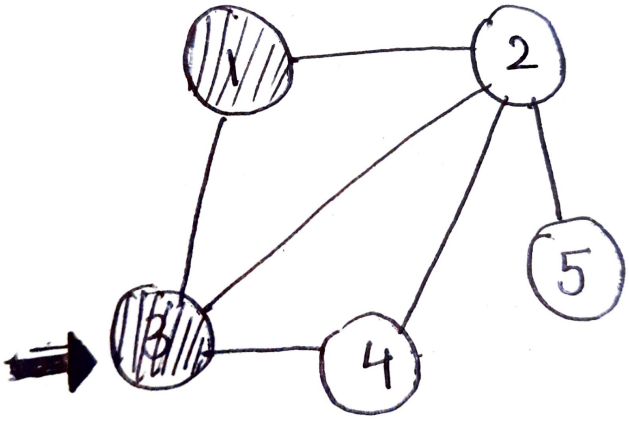
- **Algorithm:**
  - To the DFS_Traversal() we pass an extra argument parent. The parent of the node on which we apply DFS_Traversal().
  - Then in DFS_Traversal() we traverse on the adjacent vertices of the node(v). If the adjacent vertex is not visited then we apply DFS_Traveral on that node with v as it's parent. If the node is visited and it is not the parent of v then it indicates a cycle and we return. Else we move to the next adjacent vertex.
  - We repeat this till all the vertices of the graph are visited.
  - Lets see the working of the algorithm with an example.
- **Example**:
  - Consider the undirected graph given below.



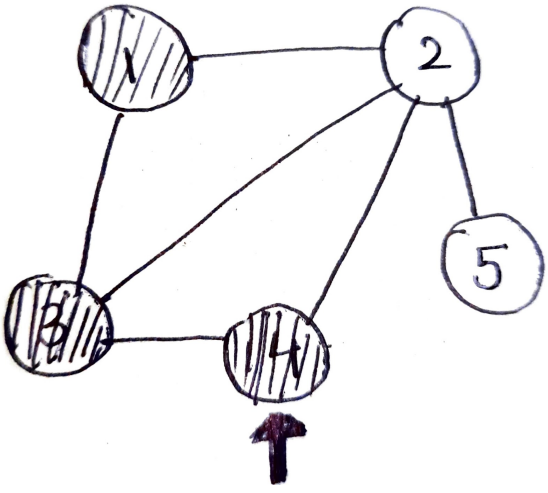
- Now, we start DFS with node 1 and the parent of node 1 is -1 as it is considered as the root vertex. We mark node 1 as seen.



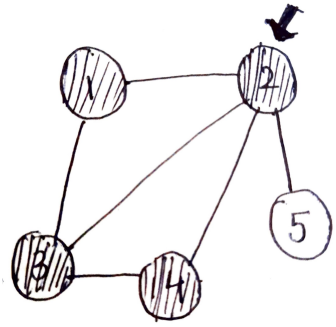
- Now, we move to node 3, an adjacent vertex of node 1. As node 3 is unseen we mark it seen and perform DFS_Traversal with node 1 as its parent.



- Then we move to node 4, an adjacent vertex of node 3. As node 4 is unseen we mark it as seen and perform DFS_Traversal() with node 3 as parent.



- Next we move to node 2, an adjacent vertex of node 4. As node 2 is unseen we mark it as seen and perform DFS_Traversal() with node 4 as parent.



- Then, we move to node 1, an adjacent vertex of node 2. As node 1 is seen and it is not the parent of 2, This forms a cycle (1->3->4->2->1). So we return 1 from the function.

- Pseudo Code:

```
DFS(Graph G, int V)
   int status[V + 1] = {0}; // V = no.of vertices
   int check = 0;
   for every vertex v ∈ V  // V = set of vertices of G
       if(status[v] == 0)
            check = DFS_Traversal(G, v, status, -1);
            if(check == true)
                return 1;
return 0;


DFS_Traversal(Graph G, int v, int* status, int parent)
   status[v] = 1;
   int check = 0;
   for every u ∈ Adj[v]      // => there is a vertex from v to u
       if(status[u] == 0)    //
           check = DFS_Traversal(G,u,status, v);
           if(check == 1);   // return if there is a cycle
               return;

       else if neighbour is visited != parent(v)
           return 1;
   status[v] = 2;

   return 0;
```

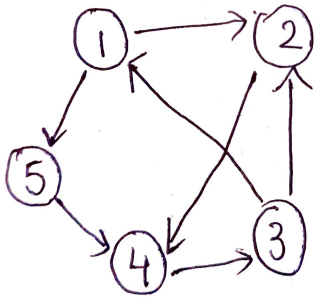
- Code Implementation
  - Undirected_Cycle_Code

- Analysis
  - Time and Space complexity are the same as DFS.

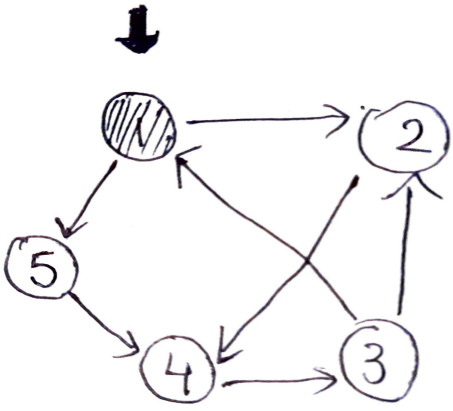
Cycle Detection in a directed graph:

To detect the cycle in a directed graph we use DFS as discussed in chapter 1. We modify the DFS_Traversal() function.

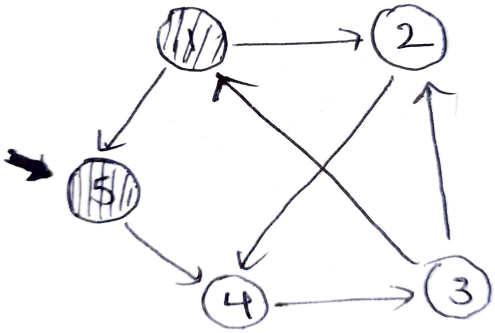
- Algorithm
  - In DFS_Traversal() we while traversing we check whether the adjacent vertex is seen or not.
  - If the vertex is seen then it indicates the presence of cycle.So, we return 1 from the function. If the vertex is not seen then we perform DFS_Traversal() on that vertex.
  - We repeat this till all the vertices are visited in the graph.
  - Let us see the working of the algorithm with an example.
- Example
  - Consider the directed graph G given below



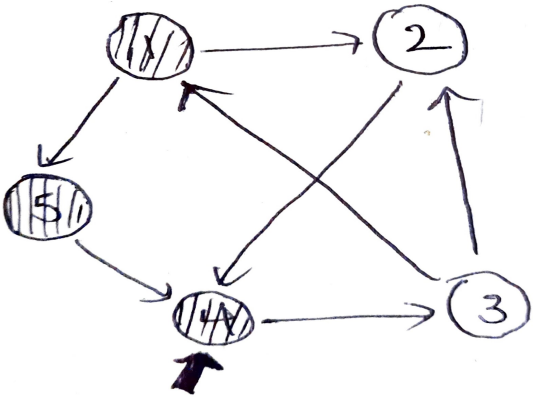
  - Now, we start DFS_Traversal from Node 1 and mark it as seen.



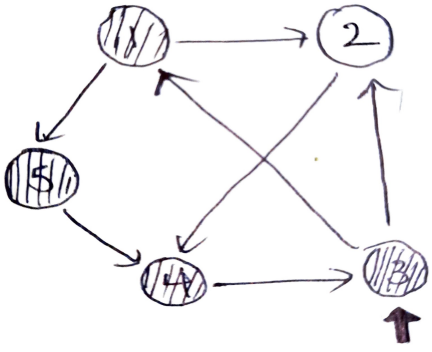
- Then we move to one of the adjacent vertices of node 1 i.e, node 5. And mark it as seen.



- As node 5 has only 1 adjacent vertex node 4, we move to node 4 and mark it as seen.



- As node 4 has only 1 adjacent vertex node 3, we move to node 3 and mark it as seen.



- Now, we move to node 1(adjacent vertex of node 3) as it is already seen. This indicates the presence of a cycle(1->5->4->3->1).

- Pseudo Code

```
DFS(Graph G, int V)
{
  // status[i] = 0 => not visited, 1 => seen, 2 => visited
   int check = 0;
   int status[V + 1] = {0}; // V = no.of vertices
   for every vertex v ∈ V  // V = set of vertices of G
       if(status[v] == 0)
            check = DFS_Traversal(G, v, status);    // initially the
vertex does not have parent
            if(check == 1)
                return 1;
   return 0;
}
DFS_Traversal(Graph G, int v, int* status)
{
   status[v] = 1;
   int check = 0;
   for every u ∈ Adj[v]      // => there is a vertex from v to u
       if(status[u] == 0)    // if the vertex is not visited we
perform DFS
       {
           check = DFS_Traversal(G,u,status);
           if(check == 1);   // return if there is a cycle
               return;
       }
       else if neighbour is seen
           return 1;            // we return
   status[v] = 2;
   return 0;
}
```

- Code Implementation
  - Directed_Cycle_Code
- Analysis
  - The Time and space complexities are the same as DFS.

And as discussed in the chapter Shortest Paths we can detect a negative cycle in a graph using Bellman-Ford Algorithm.

The End...