**Week #3**

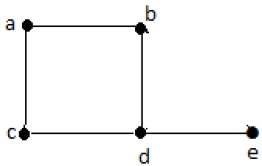
**Graph Theory**

# **| Intro to graph theory**

A graph is a representation of a set of objects connected by links. The interconnected objects are represented by points termed as **Nodes**, and the links that connect the nodes are called **Edges**.

In the graph below we have:

* 5 nodes {a, b, c, d, e}
* 5 edges {ab, ac, bd, cd, de}



# 

# 

# 

# 

# **| Edge Types**

As mentioned above an Edge is a link between two Nodes.

## **Directed Edges**

A directed edge is an edge said to have a specific direction from one node to another and cant be traversed in the opposite direction.

If we have a directed edge from node A to node B that means we can only move from node A to node B.

## **Undirected Edges**

An undirected edge from one node to another and can be traversed in any direction.

If we have an undirected edge between node A to node B that means we can move from node A to node B OR from node B to node A.

## **Weighted Edges**

In applications, the weight may be a measure of the length of the edge, the capacity of the edge, the energy required to move between the nodes connected by the edge, etc.

Weighted edges may be either directed or undirected.

If we have a directed weighted edge between node A and node B where its cost = 5, that means we can only move from A to B with cost = 5.

## **Unweighted Edges**

An unweighted edge is an edge between two nodes that doesn't have a cost or a weight.

Unweighted edges may be either directed or undirected.

# 

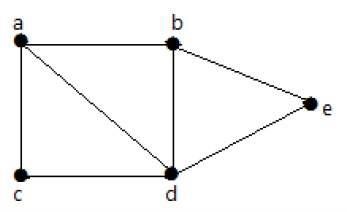
# 

# **| Graph Connectivity**

A graph is said to be connected if there is a path between every pair of nodes. That is called the connectivity of a graph. A graph with multiple disconnected nodes and edges is said to be a disconnected graph.

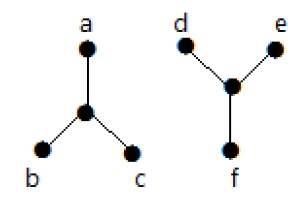
Connected or Disconnected?

**Example 1**



It is possible to travel from one node to any other node. For example, one can traverse from node ‘a’ to node ‘e’ using the path ‘a-b-e’.

**Example 2**



In the following example, traversing from node ‘a’ to node ‘f’ is not possible because there is no path between them directly or indirectly. Hence it is a disconnected graph.

# 

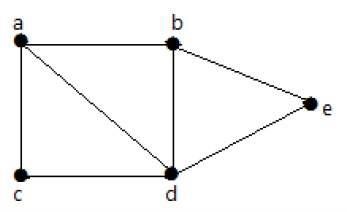
# **| Graph Components**

A component is a subgraph in which each pair of nodes is connected with each other via a path.

A connected graph is considered as a graph with only one component.

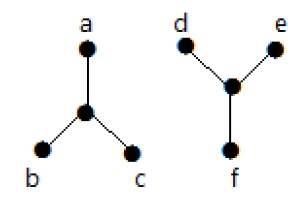
If we have a disconnected graph the number of components in that graph is the number of connected subgraphs in that graph.

**Example 1**



1 component {a, b, c, d, e}

**Example 2**



2 components {a, b, c} {d, e, f}

## | Types of graphs

### Directed Graph:

Directed means that its edges have directions (figure shown below). Directed means that its edges have directions (figure shown below), so if we picture it as roads then someone standing at node can go to node but the opposite cannot happen.

### 

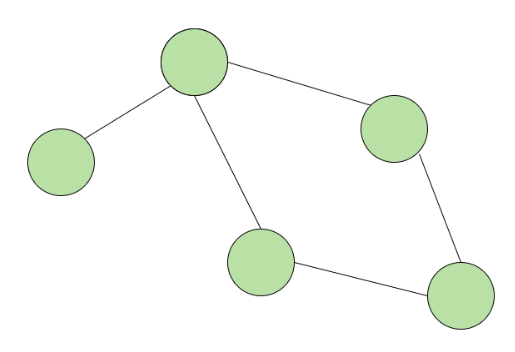
**Figure no.1**

### Undirected Graph:

Undirected graph means that the edges between nodes do not have a direction, in other words, if you use an edge to go from node to node you can use the same edge to go from node to node .

### Cyclic and Acyclic:

It is clear that cyclic is the opposite of acyclic which means that a cyclic graph is a graph that contains at least one cycle, or a graph that is a cycle, with varying definitions of cycles. A cycle means that if you start from a node you can return to the same node without using the edge more than once.



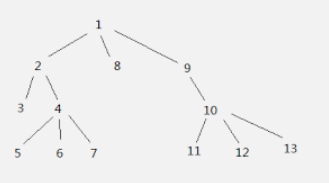
**Figure no.2**

### Directed Acyclic Graph (DAG):

The graph should be directed and acyclic. Acyclic means that there are no loops which mean in the graph (figure no.1) you cannot draw an edge between node and .

### Trees:

A tree is an undirected graph in which any two vertices are connected by **exactly one path (edge)**, in other words, it is a connected acyclic undirected graph. So it **cannot have** loops.



**Figure no.3**

### Weighted and UnWeighted Graph:

A weighted graph means that each edge has a number associated with it called **weight**, you can imagine this number as the cost to use this edge, or if picture edges as roads you can think of it as the length of the road. It is obvious that the Unweighted graph is the opposite which means edges do not have weights.

### Connected Graph:

A Connected graph means that you can start traversing this graph from any node and reach all the other nodes.

## | Graph representation

### Adjacency matrix:

An adjacency matrix is a way of representing the graph as a matrix so in a matrix for example at row and column there should be a number or a boolean representing whether there is an edge going from node to node but not the opposite, this number could also represent the weight of the edge if it is a weighted graph. The time and space complexity of building this adjacency matrix is where is the number of edges.

| int main() {  int n, m;  cin >> n >> m;  int adjacencyMatrix[n][m];  *//set the matrix to -1 so we know if there is an edge between nodes*  memset(adjacencyMatrix, -1, sizeof(adjacencyMatrix));  for (int i = 0; i < m; ++i) {  *//input the two nodes that the edge connect and the weight of the edge*  int u , v , w;  cin >> u >> v >> w;  adjacencyMatrix[u][v] = w;  adjacencyMatrix[v][u] = w;  } } |
| --- |

### 

### Adjacency list:

An adjacency list is that each node has its own list of nodes that it is connected to, so we can represent this as an array of vectors where the index of the array is the node and the vector in this index is the list of nodes connected to .

Also if the graph is weighted then you can add the nodes as pairs to the list where the pair contains the node is connected to and the weight of the edge between them. The

time and space complexity for this implementation is .

| int main() {  int n , m;  cin >> n >> m;  vector<pair<int, int>> adjacency\_list[n];  for (int i = 0; i < m; ++i) {  *//the two nodes and the weight of the edge*  int u , v , w;  cin >> u >> v >> w;  *//add u to the list of nodes connected to node v and vise versa*  adjacency\_list[u].emplace\_back(v, w);  adjacency\_list[v].emplace\_back(u , w);  } } |
| --- |

# **| DFS**

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

A standard DFS implementation puts each vertex of the graph into one of two categories:

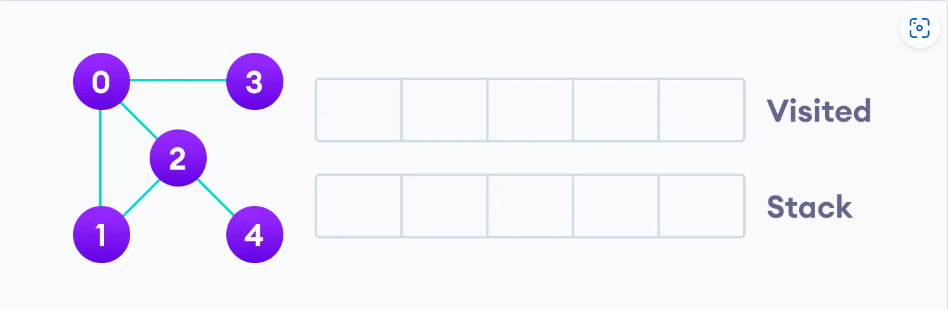
* Visited
* Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

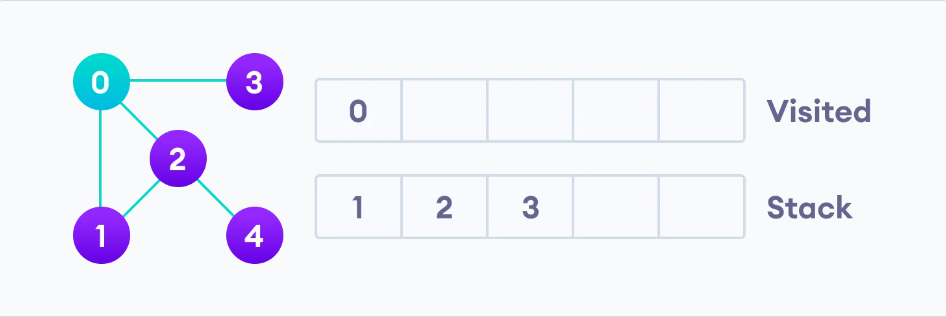
The DFS algorithm works as follows:

* Start by putting any one of the graph's vertices on top of a stack.
* Take the top item of the stack and add it to the visited list.
* Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
* Keep repeating steps 2 and 3 until the stack is empty.

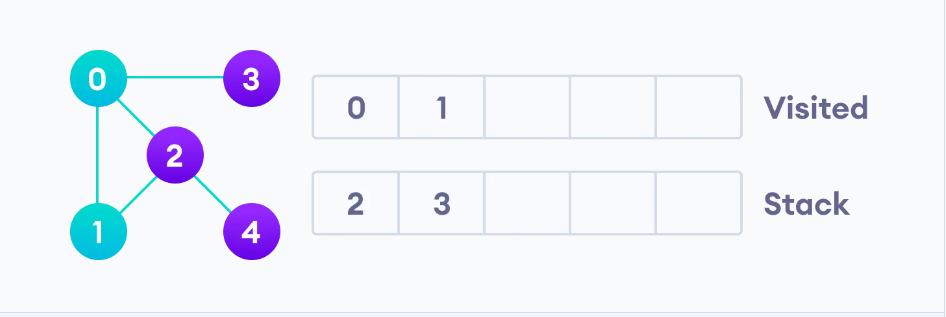
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



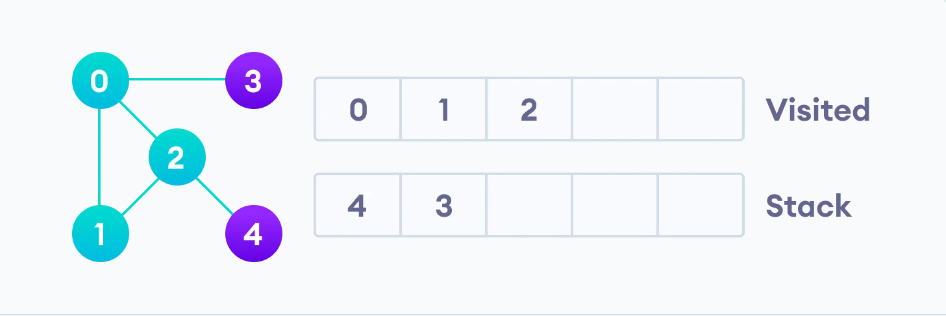
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

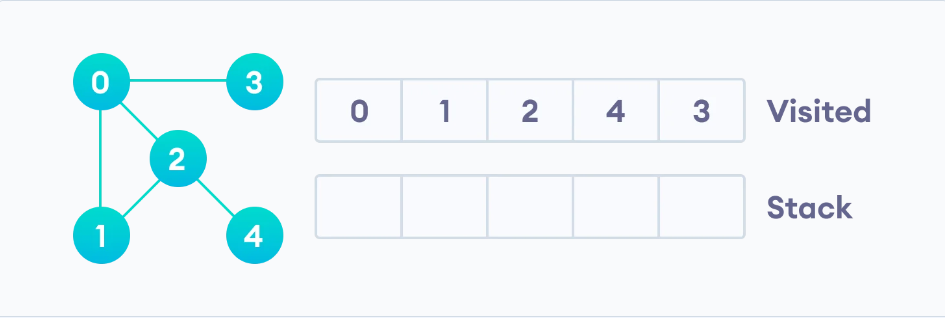


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.





After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



# 

# 

# 

# 

# **| Applications of DFS Algorithm**

* For finding the path
* To test if the graph is bipartite
* For finding the strongly connected components of a graph
* For detecting cycles in a graph

# **| Code**

| vector<int> adj[100]; bool vis[100], cycle;  void dfs(int node) {  vis[node] = true;   for (int child : adj[node]) {  if (!vis[child])  dfs(child);  } } |
| --- |

**Time Complexity : O(V + E),** Where **V** is the number of nodes and **E** is the number of edges.

# **| Cycle detection in Undirected Graph**

To detect for a cycle in an undirected graph, keep a visited array to mark all the visited nodes until now, if you have node **U** and traversed from it to node **V,** there can be a cycle if the following 2 conditions are met :

1- Node **V** is already visited.

2- Node **V** is not the parent of node **U.**

So, we can add a parameter to our **dfs** function that holds the parent for the current node.



| vector<int> adj[100]; bool vis[100], cycle;  void dfs(int node, int parent) {  vis[node] = true;    for (int child : adj[node]) {  if (!vis[child])  dfs(child, node);  else if (vis[child] && child != parent)  cycle = true;  } } |
| --- |

# **| BFS**

**BFS** algorithm is a graph traversal algorithm just like the **DFS,** but it’s not recursive.

A standard BFS implementation puts each vertex of the graph into one of two categories:

* Visited
* Not Visited

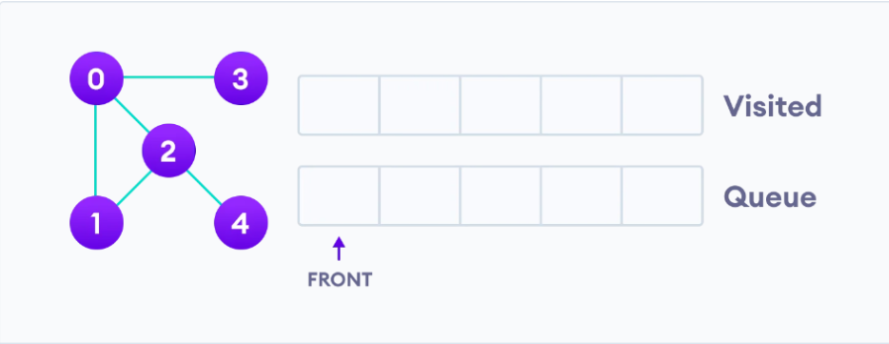
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

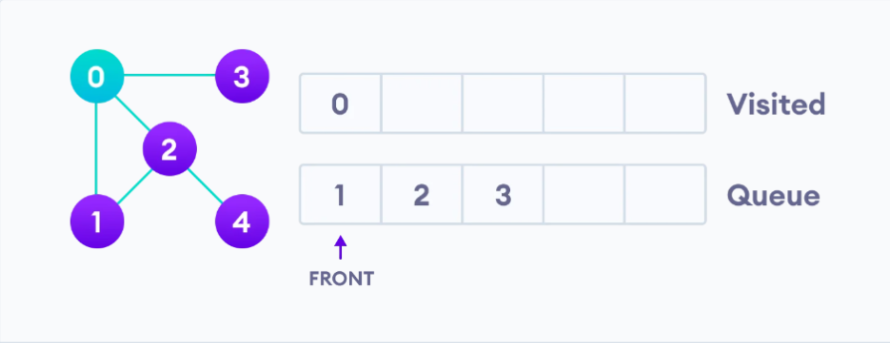
* Start by putting any one of the graph's vertices at the back of a queue.
* Take the front item of the queue and add it to the visited list.
* Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
* Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

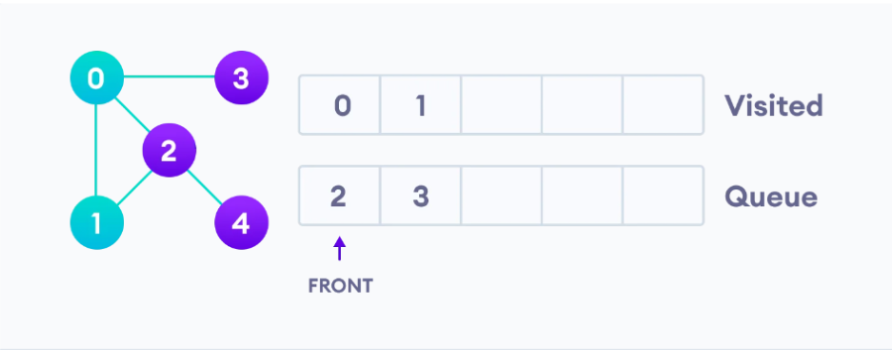
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

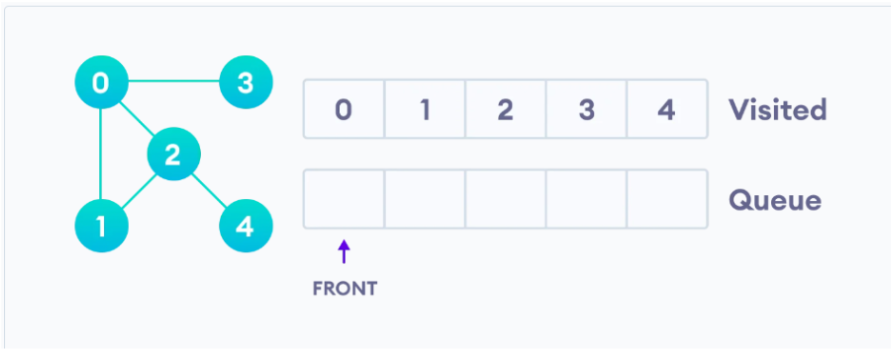


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.





Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Since the queue is empty, we have completed the Breadth First Traversal of the graph.

# **| Applications of BFS Algorithm**

* To build index by search index
* For GPS navigation
* Path finding algorithms
* In Ford-Fulkerson algorithm to find maximum flow in a network
* Cycle detection in an undirected graph
* In minimum spanning tree.

# **| Code**

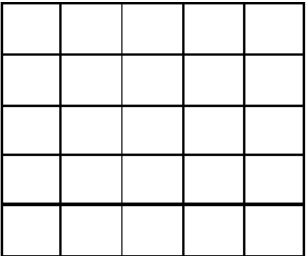
| vector<int> adj[100]; bool vis[100];  void bfs(int start) {  queue<int> q;  q.push(start);  vis[start] = true;    while (!q.empty()) {  int node = q.front();  q.pop();    for (int child : adj[node]) {  if (!vis[child]) {  q.push(child);  vis[child] = true;  }  }  } } |
| --- |

**Time Complexity : O(V + E),** Where **V** is the number of nodes and **E** is the number of edges.

# 

# **| Traversing in a 2D Grid**

The **BFS** algorithm can be used to traverse through a 2D grid, we can assume that the position we are currently standing on is a node and it’s children are the four adjacent positions.



Let’s take the above grid as an example, if you are standing on point **(3,2)** you can move to the following four points, **(3,1), (3,3), (2,2), (4,2).**

So, if you are on point **(i,j),** you can move to **(i - 1, j), (i + 1, j), (i, j + 1), (i, j - 1),** but first you need to check if the point you are traversing to is in borders.

| int dx[] = {1, -1, 0, 0, -1, -1, 1, 1}; int dy[] = {0, 0, 1, -1, -1, 1, 1, -1}; bool vis[100][100]; bool inBorders(int i, int j) {  return i >= 1 && i <= 100 && j >= 1 && j <= 100; } void bfs(int stx, int sty) {  queue<pair<int,int>> q;  q.push({stx, sty});  vis[stx][sty] = true;  while (!q.empty()) {  int x, y;  tie(x, y) = q.front();  q.pop();  for (int i = 0; i < 4; i++) {  int newx = x + dx[i];  int newy = y + dy[i];  if (inBorders(newx, newy) && !vis[newx][newy]) {  q.push({newx, newy});  vis[newx][newy] = true;  }  }  } } |
| --- |