# **Graph**

* **Use of data structure to make graph**

unordered\_map<T,list<T>>adj;

* **Bfs and dfs**

**1)BFS:-** Here queue is used . There may be different combinations and those combinations are formed based on either we select the left node or right node.

**Code/simple program:-**

for(int startNode=0;startNode<numNodes;startNode++){//this for loop is for disconnected components

        if(!visited[startNode]){

            bfsQueue.push(startNode);

            visited[startNode] = true;

            while (!bfsQueue.empty()) {

                int currentNode = bfsQueue.front();

                bfsQueue.pop();

                cout << currentNode << " ";

                for (int neighbor = 0; neighbor < numNodes; ++neighbor) {

                    if (graph[currentNode][neighbor] == 1 && !visited[neighbor]) {

                        bfsQueue.push(neighbor);

                        visited[neighbor] = true;

                    }

                }

            }

        }

    }

**2)DFS:-**Here stack is used. So to avoid confusion , keep one thing remember that the node you have selected is expanded in its depth and then the siblings are expanded.

**Code/simple program:-**

for (int startNode = 0; startNode < numNodes; ++startNode) {

        if (!visited[startNode]) {

            dfsStack.push(startNode);

            visited[startNode] = true;

            while (!dfsStack.empty()) {

                int currentNode = dfsStack.top();

                dfsStack.pop();

                cout << currentNode << " ";

                for (int neighbor = 0; neighbor < numNodes; ++neighbor) {

                    if (graph[currentNode][neighbor] == 1 && !visited[neighbor]) {

                        dfsStack.push(neighbor);

                        visited[neighbor] = true;

                    }

                }

            }

        }

    }

BFS:O(V+E) (adjacency list),O(V2) (adjacency matrix)

DFS: O(V+E) (adjacency list), O(V2) (adjacency matrix)

* **Detect Cycle using bfs and dfs in Undirected Graph:-**

**Condition for cycle detection:-**

We have to maintain one DS in order the maintain the parent of each node

So if node is visited and that node is not a parent of current node then there is cycle

Present in graph.

**Code for bfs:-**

bool BFS(int\*\* &graph, int numNodes) {

    vector<bool> visited(numNodes, false);

    queue<int> bfsQueue;

    unordered\_map<int, int> parent;

    for (int startNode = 0; startNode < numNodes; startNode++) {

        if (!visited[startNode]) {

            bfsQueue.push(startNode);

            visited[startNode] = true;

            // parent[startNode] = -1;

            while (!bfsQueue.empty()) {

                int currentNode = bfsQueue.front();

                bfsQueue.pop();

                for (int neighbor = 0; neighbor < numNodes; ++neighbor) {

                    if (graph[currentNode][neighbor] == 1) {

                        if (!visited[neighbor]) {

                            bfsQueue.push(neighbor);

                            visited[neighbor] = true;

                            parent[neighbor] = currentNode;

                        } else if (neighbor != parent[currentNode]) {

                            // A cycle is detected

                            return true;

                        }

                    }

                }

            }

        }

    }

    return false;

}

**Code for dfs:-**

bool DFS(int\*\* graph, int numNodes) {

    vector<bool> visited(numNodes, false);

    stack<int> dfsStack;

    unordered\_map<int, int> parent;

    cout << "DFS traversal for the entire graph: ";

    for (int startNode = 0; startNode < numNodes; ++startNode) {

        if (!visited[startNode]) {

            dfsStack.push(startNode);

            visited[startNode] = true;

            while (!dfsStack.empty()) {

                int currentNode = dfsStack.top();

                dfsStack.pop();

                cout << currentNode << " ";

                for (int neighbor = 0; neighbor < numNodes; ++neighbor) {

                    if(graph[currentNode][neighbor]==1){

                        if (graph[currentNode][neighbor] == 1 && !visited[neighbor]) {

                            dfsStack.push(neighbor);

                            visited[neighbor] = true;

                            parent[neighbor]=currentNode;

                        }

                        else if(neighbor!=parent[currentNode]){

                            return true;

                        }

                    }

                }

            }

        }

    }

    cout << endl;

    return false;

}

* Topological sorting

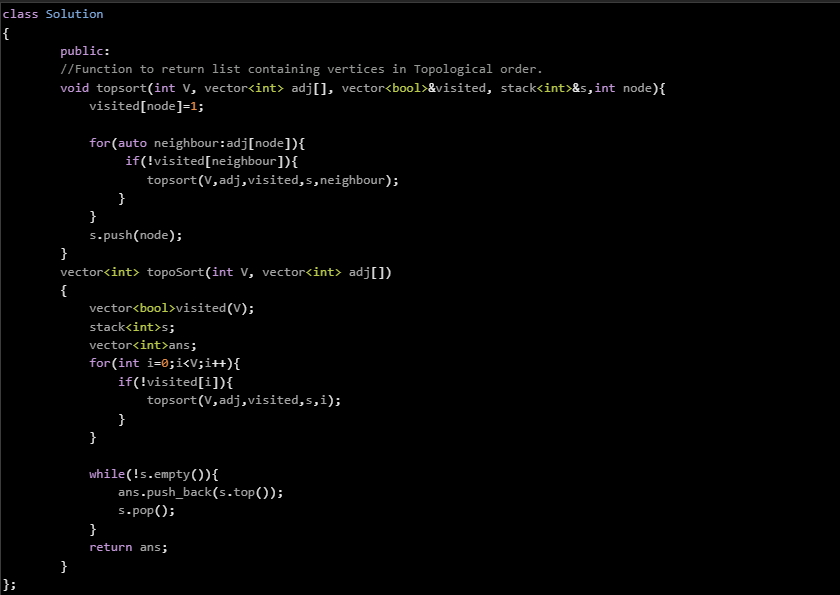
Consider very simple adj list as,u->v,z. Topological sort is like traversal to all nodes like bfs and dfs But there is a catch that in topological sort sequence the parent node will always come first than its child nodes like u,v,z is valid topological sequence

Code

Topological sort is done using dfs.

Logic for topological sort is simple. Just we have to maintain one stack which will have input when there is node child node present to expand.

Here ,recursive approach is imp.



Topological sort is done of the DAG only ie directed acyclic graph. So this says that this logic can be used to detect the cycle in the graph. Like if Topological sort is possible the cycle is not present and vice versa.