**Batch:** B1 **Roll Number:** 1914078 **Name:** Devansh Shah

**Experiment Number:** 2

**Aim of the Experiment:** To implement BFS and DFS - Uninformed search algorithm in state space

**Theory**

Intelligent agents are supposed to maximize the performance measure. The problem solving

agents start with the activity of goal formulation where, it organizes the behavior by limiting the

objectives that the agent is trying to achieve. Then comes the problem formulation which the

process of deciding what actions and state to consider, given a goal. These (legal) actions when

applied to the initial state, gives us the entire state-space.

The state-space leaves the agent with several immediate options of unknown value where the

agent can decide what to do next by first examining different possible sequences of actions that

lead to states of known value, and then choosing the best one.

The algorithms in uninformed search category:

**BFS:**

These algorithms search trees of nodes, whether that tree is explicit or implicit (generated on the

go). The basic principle is that a node is taken from a data structure, its successors examined and

added to the data structure. By manipulating the data structure, the tree is explored in level by

level order.

This method selects the deepest unexpanded node in the search tree for expansion.

**Algorithm :**

1. Enqueue the root node.

2. 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.

3. If the queue is empty, every node on the graph has been examined – quit the search and return

"not found".

4. If the queue is not empty, repeat from Step 2

**DFS:**

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves

exhaustive searches of all the nodes by going ahead, if possible, else by backtracking. Here, the

word backtrack means that when you are moving forward and there are no more nodes along the

current path, you move backwards on the same path to find nodes to traverse. All the nodes will

be visited on the current path till all the unvisited nodes have been traversed after which the next

path will be selected.

In depth-first search, the frontier acts like a last-in first-out stack. The elements are added to the

stack one at a time. The one selected and taken off the frontier at any time is the last element that

was added.

**Algorithm**

1. If the initial state is a goal state, quit and return success.

2. Otherwise, loop until success or failure is signalled.

Generate a state, say E, and let it be the successor of the initial state. If there is no

successor, signal failure.

Call Depth-First Search with E as the initial state.

If success is returned, signal success. Otherwise continue in this loop.

The advantage of depth-first Search is that memory requirement is only linear with respect to the

search graph. This is in contrast with breadth-first search which requires more space. The reason

is that the algorithm only needs to store a stack of nodes on the path from the root to the current

node.

The depth-first search is time-limited rather than space-limited.

**Program/ Steps:**

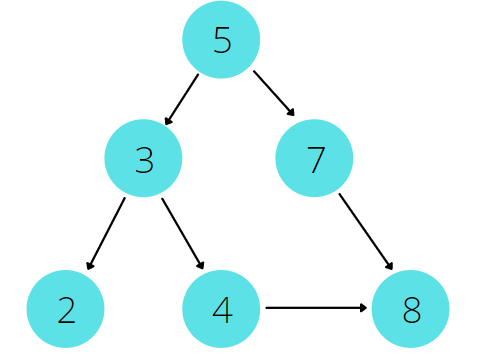
1. Implement Mentioned algorithm for BFS for graph search

2. Implement the mentioned algorithm for DFS for graph search

3. Output must show the contents of Fringe and Visited nodes for each iteration of graph

traversal. Also, finally it must print the path traversed.

Graph:



**CODE:**

#include <bits/stdc++.h>

using namespace std;

//creates a graph

void addEdge(vector<int> adj[],int u , int v){

    adj[u].push\_back(v);

    adj[v].push\_back(u);

}

//display the graph (adjacency list representation)

void displayGraph(vector<int> adj[],int V){

    cout<<endl<<"Adjacency list representation of the graph: "<<endl;

    for(int i=0;i<V;i++){

        cout<<i;

        for(auto x : adj[i]){

            cout<< "->"<<x;

        }

        cout<<endl;

    }

    cout<<endl;

}

//bfs function

void bfs(vector<int> adj[], int destination,int source, int V){

    cout<<"\*\*BFS\*\*"<<endl;

    vector<int> path;

    bool found = false;

    // queue using deque...done to iterate through queue which //is not possible using inbuilt queue in STL

    deque<int> q;

    q.push\_back(source);

    vector<int> vis(V,0); //visited vector for keeping track o//f visited nodes

    vis[source] = 1;

    cout<<"Queue: "<<q[0]<<endl;

    while(!q.empty()){

        int td = q.front();

        if(td == destination){

            found = true;

            break;

        }

        else{

            path.push\_back(td);

        }

        q.pop\_front();

        for(auto v : adj[td]){

            if(!vis[v]){

                vis[v] = 1;

                q.push\_back(v);

            }

        }

        cout<<"Visited nodes till now: ";

        for(int i=0;i<vis.size();i++){

            if(vis[i] == 1){

                cout<<i<<" ";

            }

        }

       cout<<endl<<"Queue: ";

       for(int i=0;i<q.size();i++){

           cout<<q[i]<<" ";

       }

        cout<<endl;

    }

    if(found){

        cout<<endl<<"Element found"<<endl;

    }

    else{

        cout<<endl<<"Element not found"<<endl;

    }

    cout<<"BFS Path: ";

    for(auto p : path){

        cout<< p <<" ";

    }

    if(found)

        cout<<destination<<endl<<endl;

}

//dfs fucntion

void dfs(vector<int> adj[], int destination,int source, int V){

    cout<<"\*\*DFS\*\*"<<endl;

    vector<int> path;

    bool found = false;

    //stack implemented using deque... done to iterate through //stack which is not possible using inbuilt stack in STL

    deque<int> st;

    st.push\_front(source);

    cout<<"Stack: "<<source;

    vector<int> vis(V,0); //visited vector for keeping track o//f visited nodes

    vis[source] = 1;

    while(!st.empty()){

        int td = st.front();

        if(td == destination){

            found = true;

            break;

        }

        else{

            path.push\_back(td);

        }

        st.pop\_front();

        for(auto x: adj[td]){

            if(!vis[x]){

                vis[x] = 1;

                st.push\_front(x);

            }

        }

        cout<<endl;

        cout<<"Visited nodes till now: ";

        if(found == false){

            for(int i=0;i<vis.size();i++){

                if(vis[i] == 1){

                    cout<<i<<" ";

                }

            }

        }

        cout<<endl<<"Stack: ";

        for(int i=0;i<st.size();i++){

           cout<<st[i]<<" ";

        }

    }

    cout<<endl;

    if(found){

        cout<<endl<<"Element found"<<endl;

    }

    else{

        cout<<endl<<"Element not found"<<endl;

    }

    cout<<"DFS Path: ";

    for(auto p : path){

        cout<< p <<" ";

    }

    if(found)

        cout<<destination<<endl;

}

int main(){

    int V,E;

    cout<<"Enter the number of vertices: ";

    cin>>V;

    vector<int> adj[V];

    cout<<"Enter the number of edges: ";

    cin>>E;

    for(int i=0;i<E;i++){

        int a,b;

        cout<<"Enter the vertices of edge "<<i+1<<": ";

        cin>>a>>b;

        addEdge(adj,a,b);

    }

    displayGraph(adj,V);

    int destination,source ;

    cout<<"Enter the source: ";

    cin>>source;

    cout<<"Enter the element to be searched: ";

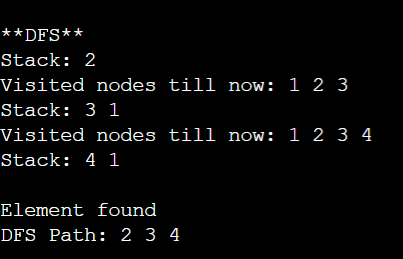
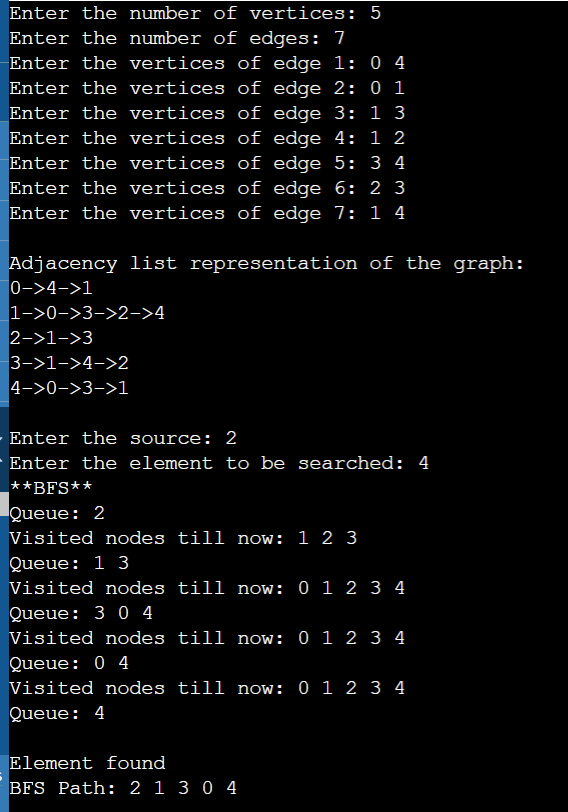
    cin>>destination;

    bfs(adj, destination, source, V);

    dfs(adj, destination, source, V);

    return 0;

}

**OUTPUT:** 

**Outcomes:**

**CO2:** Analyze and formalise the problem (as state space, graph etc) and select the appropriate search method and write the algorithm.

**Conclusion (based on the Results and outcomes achieved):**

Through this experiment we formalized the problem and wrote the programs for breadth-first search and depth-first search for graphs.

**References:**

* Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach,

Second Edition, Pearson Publication

* Elaine Rich, Kevin Knight, Artificial Intelligence, Tata McGraw Hill, 1999.