|  |
| --- |
| Experiment No.2 |
| Study and Implementation of Depth first search for problem solving. |
| Date of Performance: |
| Date of Submission: |

**Aim:** Study and Implementation of Depth first search for problem solving.

**Objective:** To study the uninformed searching techniques and its implementation for problem solving.

**Theory:**

**Artificial Intelligence** is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

* A search problem consists of:
  + **A State Space.** Set of all possible states where you can be.
  + **A Start State.** The state from where the search begins.
  + **A Goal Test.** A function that looks at the current state returns whether or not it is the goal state.
* The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
* This plan is achieved through search algorithms.

**Depth First Search:** DFS is an uninformed search method. It is also called blind search. Uninformed search strategies use only the information available in the problem definition. A search strategy is defined by picking the order of node expansion. Depth First Search (DFS) searches deeper into the problem space. It 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.

**The basic idea is as follows:**

Pick a starting node and push all its adjacent nodes into a stack.

Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty.

However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

**Algorithm:**

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

1. Visited
2. Not Visited

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

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. 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.

1. Keep repeating steps 2 and 3 until the stack is empty.

**Pseudocode:**

#include<stdio.h>

#include<stdlib.h>

#define MAX 100

#define initial 1

#define visited 2

int n; /\* Number of nodes in the graph \*/

int adj[MAX][MAX]; /\*Adjacency Matrix\*/

int state[MAX]; /\*Can be initial or visited \*/

void DF\_Traversal();

void DFS(int v);

void create\_graph();

int stack[MAX];

int top = -1;

void push(int v);

int pop();

int isEmpty\_stack();

main()

{

create\_graph();

DF\_Traversal();

}/\*End of main()\*/

void DF\_Traversal()

{

int v;

for(v=0; v<n; v++)

state[v]=initial;

printf("\nEnter starting node for Depth First Search : ");

scanf("%d",&v);

DFS(v);

printf("\n");

}/\*End of DF\_Traversal( )\*/

void DFS(int v)

{

int i;

push(v);

while(!isEmpty\_stack())

{

v = pop();

if(state[v]==initial)

{

printf("%d ",v);

state[v]=visited;

}

for(i=n-1; i>=0; i--)

{

if(adj[v][i]==1 && state[i]==initial)

push(i);

}

}

}/\*End of DFS( )\*/

void push(int v)

{

if(top == (MAX-1))

{

printf("\nStack Overflow\n");

return;

}

top=top+1;

stack[top] = v;

}/\*End of push()\*/

int pop()

{

int v;

if(top == -1)

{

printf("\nStack Underflow\n");

exit(1);

}

else

{

v = stack[top];

top=top-1;

return v;

}

}/\*End of pop()\*/

int isEmpty\_stack( )

{

if(top == -1)

return 1;

else

return 0;

}/\*End if isEmpty\_stack()\*/

void create\_graph()

{

int i,max\_edges,origin,destin;

printf("\nEnter number of nodes : ");

scanf("%d",&n);

max\_edges=n\*(n-1);

for(i=1;i<=max\_edges;i++)

{

printf("\nEnter edge %d( -1 -1 to quit ) : ",i);

scanf("%d %d",&origin,&destin);

if( (origin == -1) && (destin == -1) )

break;

if( origin >= n || destin >= n || origin<0 || destin<0)

{

printf("\nInvalid edge!\n");

i--;

}

else

{

adj[origin][destin] = 1;

}

}

}

OUTPUT : :

Enter number of nodes : 6

Enter edge 1( -1 -1 to quit ) : 0 1

Enter edge 2( -1 -1 to quit ) : 0 2

Enter edge 3( -1 -1 to quit ) : 0 3

Enter edge 4( -1 -1 to quit ) : 1 3

Enter edge 5( -1 -1 to quit ) : 3 4

Enter edge 6( -1 -1 to quit ) : 4 2

Enter edge 7( -1 -1 to quit ) : 5 5

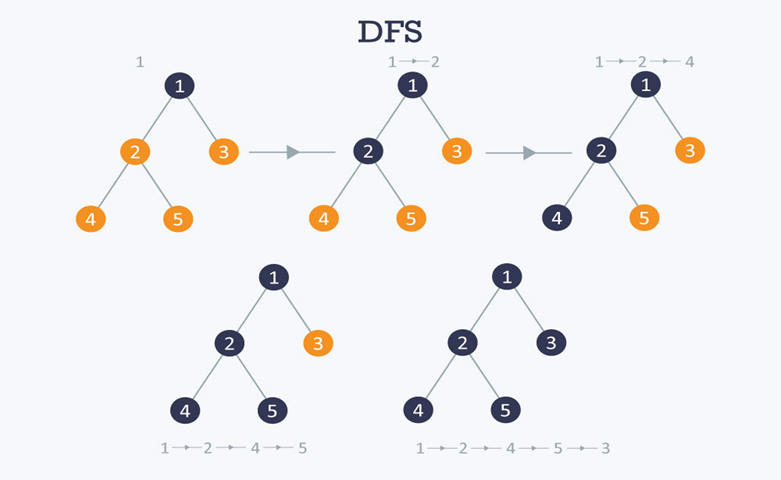
Enter edge 8( -1 -1 to quit ) : -1 -1

Enter starting node for Depth First Search : 0

0 1 3 4 2

Process returned 0

**DFS Working: Example**

****

**Path: 1 🡪 2🡪 4🡪 5🡪 3**

**Searching Strategies are evaluated along the following dimensions:**

1. **Completeness:** does it always find a solution if one exists?
2. **Time complexity:** number of nodes generated
3. **Space complexity:** maximum number of nodes in memory
4. **Optimality:** does it always find a least-cost solution?

**Properties of depth-first search:**

1. Complete:- No: fails in infinite-depth spaces, spaces with loops.
2. Time Complexity: O(bm)
3. Space Complexity: O(bm), i.e., linear space!
4. Optimal: No

**Advantages of Depth-First Search:**

1. Memory requirement is only linear with respect to the search graph.
2. The time complexity of a depth-first Search to depth d is O(b^d)
3. If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

**Disadvantages of Depth-First Search:**

1. There is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree.
2. Depth-First Search is not guaranteed to find the solution.
3. No guarantee to find a optimum solution, if more than one solution exists.

**Applications**

**How to find connected components using DFS?**

A graph is said to be disconnected if it is not connected, i.e. if two nodes exist in the graph such that there is no edge in between those nodes. In an undirected graph, a connected component is a set of vertices in a graph that are linked to each other by paths.

Consider the example given in the diagram. Graph G is a disconnected graph and has the following 3 connected components.

* First connected component is 1 🡪 2 🡪 3 as they are linked to each other
* Second connected component 4 🡪 5
* Third connected component is vertex 6

**Conclusion:** Hence we have implemented DFS and was executed successfully. The algorithm demonstrated its ability to traverse a graph or tree by exploring nodes in a depth-first manner. By visiting nodes and backtracking when necessary, DFS can efficiently search for solutions in various problem domains.