Student Name: ______________________________ **Roll No:** ____________ **Section:** ___________

Experiment No. 14

Lab 14 – Breadth First, Depth First Search Algorithm, Uninformed and Salesman Travelling Algorithm.

In this lab, we will learn about BFS, DFS, Uninformed Search and Travelling Salesman Algorithm.

Graph traversal is the process of moving from the root node to all other nodes in the graph. The order of nodes given by a traversal will vary depending upon the algorithm used for the process. As such, graph traversal is done using two main algorithms: Breadth-First Search and Depth-First Search. In this article, we are going to go over the basics of one of the traversals, breadth first search in java, understand BFS algorithm with example and BFS implementation with java code. Also, If you are stuck anywhere with your java programming, our experts will be happy to provide you with any type of Java homework help.

What is a Breadth-First Search?

Breadth-First Search (BFS) relies on the traversal of nodes by the addition of the neighbor of every node starting from the root node to the traversal queue. BFS for a graph is similar to a tree, the only difference being graphs might contain cycles. Unlike depth-first search, all of the neighbor nodes at a certain depth are explored before moving on to the next level.

BFS Algorithm

The general process of exploring a graph using breadth-first search includes the following steps:-

1. Take the input for the adjacency matrix or adjacency list for the graph.
2. Initialize a queue.
3. Enqueue the root node (in other words, put the root node into the beginning of the queue).
4. Dequeue the head (or first element) of the queue, then enqueue all of its neighboring nodes, starting from left to right. If a node has no neighboring nodes which need to be explored, simply dequeue the head and continue the process. (Note: If a neighbor which is already explored or in the queue appears, don't enqueue it – simply skip it.)
5. Keep repeating this process till the queue is empty.

(In case people are not familiar with enqueue and dequeue operations, enqueue operation adds an element to the end of the queue while dequeue operation deletes the element at the start of the queue.)

Edge Cases

Now, there's always the risk that the graph being explored has one or more cycles. This means that there's a chance of getting back to a node that we have already explored. How do we determine if a node has been explored or not? It's simple – we simply maintain an array for all the nodes. The array at the beginning of the process will have all of its elements initialized to 0 (or false). Once a node is explored once, the corresponding element in the array will be set to 1 (or true). We simply enqueue nodes if the value of their corresponding element in the array is 0 (or false).

Student Name: ______________________________ **Roll No:** ______________ **Section:** ____________

Another case to consider while you're traversing all the nodes of a graph using the BFS in java. What happens when the graph being provided as input is a disconnected graph? In other words, what happens when the graph has not one, but multiple components which are not connected? It's simple. As described above, we will maintain an array for all of the elements in the graph. The idea is to iteratively perform the BFS algorithm for every element of the array which is not set to 1 after the initial run of the BFS algorithm is completed. This is even more relevant in the case of directed graphs because of the addition of the concept of "direction" of traversal.

Applications

Breadth-First Search has a lot of utility in the real world because of the flexibility of the algorithm. These include follows:

- Discovery of peer nodes in a peer-to-peer network. Most torrent clients like BitTorrent, uTorrent, qBittorent use this mechanism for discovering something called "seeds" and "peers" in the network.
- Web crawling uses graph traversal techniques for building the index. The process considers the source page as the root node and starts traversing from there to all secondary pages linked with the source page (and this process continues). Breadth-First Search has an innate advantage here because of the reduced depth of the recursion tree.
- GPS navigation systems use Breadth-First Search for finding neighboring locations using the GPS.
- Garbage collection is done with Cheney's algorithm which utilizes the concept of breadth-first search.
- Algorithms for finding minimum spanning tree and shortest path in a graph using Breadth-first search.

Program 1: Implement Breadth First Algorithm.

```
import java.io.*;
import java.util.*;

class Graph
{
    private int V;                                      //number of nodes in the graph
    private LinkedList<Integer> adj[];                      //adjacency list
    private Queue<Integer> queue;                           //maintaining a queue

    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; i++)
        {
            adj[i] = new LinkedList<>();
        }
        queue = new LinkedList<Integer>();
    }


    void addEdge(int v,int w)
```

Student Name: ____________________________________ **Roll No:** ________________ **Section:** ______________

```
    {
        adj[v].add(w);                          //adding an edge to the adjacency
list (edges are bidirectional in this example)
    }

    void BFS(int n)
    {

        boolean nodes[] = new boolean[V];       //initialize boolean array for
holding the data
        int a = 0;

        nodes[n]=true;
        queue.add(n);                   //root node is added to the top of the queue

        while (queue.size() != 0)
        {
            n = queue.poll();               //remove the top element of the queue
            System.out.print(n+" ");            //print the top element of the queue

            for (int i = 0; i < adj[n].size(); i++)  //iterate through the linked
list and push all neighbors into queue
            {
                a = adj[n].get(i);
                if (!nodes[a])                  //only insert nodes into queue if
they have not been explored already
                {
                    nodes[a] = true;
                    queue.add(a);
                }
            }
        }
    }

    public static void main(String args[])
    {
        Graph graph = new Graph(6);

        graph.addEdge(0, 1);
        graph.addEdge(0, 3);
        graph.addEdge(0, 4);
        graph.addEdge(4, 5);
        graph.addEdge(3, 5);
        graph.addEdge(1, 2);
        graph.addEdge(1, 0);
        graph.addEdge(2, 1);
        graph.addEdge(4, 1);
        graph.addEdge(3, 1);
        graph.addEdge(5, 4);
        graph.addEdge(5, 3);

        System.out.println("The Breadth First Traversal of the graph is as follows
:");
```

Student Name: ___________________________________ Roll No: _______________ Section: _____________

```
        graph.BFS(0);
    }
}
```

Output:

Program 2: Implement BFS algorithm using JAVA alternative technique.

```
import java.io.*;
import java.util.*;
public class BFSTraversal
{
    private int node;       /* total number number of nodes in the graph */
    private LinkedList<Integer> adj[];      /* adjacency list */
    private Queue<Integer> que;           /* maintaining a queue */
    BFSTraversal(int v)
    {
        node = v;
        adj = new LinkedList[node];
        for (int i=0; i<v; i++)
        {
            adj[i] = new LinkedList<>();
        }
        que = new LinkedList<Integer>();
    }
    void insertEdge(int v,int w)
    {
        adj[v].add(w);      /* adding an edge to the adjacency list (edges are bidire
ctional in this example) */
    }
    void BFS(int n)
    {
        boolean nodes[] = new boolean[node];       /* initialize boolean array for ho
lding the data */
        int a = 0;
        nodes[n]=true;
        que.add(n);       /* root node is added to the top of the queue */
        while (que.size() != 0)
        {
            n = que.poll();        /* remove the top element of the queue */
            System.out.print(n+" ");    /* print the top element of the queue */
            for (int i = 0; i < adj[n].size(); i++)  /* iterate through the linked li
st and push all neighbors into queue */
            {
                a = adj[n].get(i);
```

Student Name: ___________________________________ Roll No: _______________ Section: _____________

```
                if (!nodes[a])      /* only insert nodes into queue if they have not
been explored already */
                {
                    nodes[a] = true;
                    que.add(a);
                }
            }
        }
    }
    public static void main(String args[])
    {
        BFSTraversal graph = new BFSTraversal(6);
        graph.insertEdge(0, 1);
        graph.insertEdge(0, 3);
        graph.insertEdge(0, 4);
        graph.insertEdge(4, 5);
        graph.insertEdge(3, 5);
        graph.insertEdge(1, 2);
        graph.insertEdge(1, 0);
        graph.insertEdge(2, 1);
        graph.insertEdge(4, 1);
        graph.insertEdge(3, 1);
        graph.insertEdge(5, 4);
        graph.insertEdge(5, 3);
        System.out.println("Breadth First Traversal for the graph is:");
        graph.BFS(0);
    }
}
```

Output:

What is Depth First Search?

Depth First Search (DFS) is an algorithm of graph traversal which starts exploring from a source node (generally the root node) and then explores as many nodes as possible before backtracking. Unlike breadth-first search, exploration of nodes is very non-uniform by nature.

DFS Algorithm

The general process of exploring a graph using depth first search includes the following steps:

1. Take the input for the adjacency matrix or adjacency list for the graph.
2. Initialize a stack.
3. Push the root node (in other words, put the root node into the beginning of the stack).
4. If the root node has no neighbors, stop here. Else push the leftmost neighboring node which hasn't already been explored into the stack. Continue this process till a node is encountered which has no

neighbors (or whose neighbors have all been added to the stack already) – stop the process then, pop the head, and then continue the process for the node which is popped.

Keep repeating this process till the stack becomes empty.

It should be noted that Depth first search in java does not work in a uniform way like Breadth first search, and tracing out a traversal might be harder.

There's still another problem to solve. What happens if the graph given is a disconnected graph (meaning that it has multiple connected components instead of a single component)? This would mean that the results obtained would be skewed because all nodes would never be explored. The solution is to iterate through the unexplored nodes and manually use the DFS algorithm to explore each component individually. Of course, this means that one would need to take the help of an array to mark the nodes which have already been explored up to a certain point.

Applications

Depth First Search has a lot of utility in the real world because of the flexibility of the algorithm. These include:

- All traversal methods can be used for the detection of cycles in graphs. Cycle detection is done using DFS by checking for back edges.
- Both DFS and BFS can be used for producing the minimum spanning tree and for finding the shortest paths between all pairs of nodes (or vertices) of the graph.
- DFS can be used for topological sorting of a graph. In topological sorting, the nodes of the graph are arranged in the order in which they appear on the edges of the graph.
- DFS can be used to check if a graph is bipartite or not. A bipartite graph is such that all nodes in the graph can be divided into two sets such that the edges of the graph connect one vertex from each set.
- DFS can be used for finding the strongly connected components of a graph. Strongly connected components are such that all of the nodes in the component are connected to one another.

Implementing Depth First Search in Java

There are multiple ways to implement DFS in Java. We will be using an adjacency list for the representation of the graph and will be covering both recursive as well as an iterative approach for implementation of the algorithm. The graph used for the demonstration of the code will be the same as the one used for the above example.

Recursive implementation

The recursive implementation for the DFS algorithm in java is as follows:

Student Name: ________________________________ Roll No: ______________ Section: ____________

Program 3: Implement DFS algorithm using JAVA recursive technique

```
import java.io.*;
import java.util.*;


class Graph {
    private int V;                              //number of nodes

    private LinkedList<Integer> adj[];              //adjacency list

    public Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
{
          adj[i] = new LinkedList();
        }

    void addEdge(int v, int w)
    {
        adj[v].add(w);                                  //adding an edge to the
adjacency list (edges are bidirectional in this example)
    }


    void DFSUtil(int vertex, boolean nodes[])
    {

        nodes[vertex] = true;                          //mark the node as explored
        System.out.print(vertex + " ");
        int a = 0;

        for (int i = 0; i < adj[vertex].size(); i++)  //iterate through the
linked list and then propagate to the next few nodes
            {
                a = adj[vertex].get(i);
                if (!nodes[a])                      //only propagate to next nodes
which haven't been explored
                {
                    DFSUtil(a, nodes);
                }
            }
    }

    void DFS(int v)
```

Student Name: ______________________ Roll No: ______________ Section: ____________

```
    {
        boolean already[] = new boolean[V];          //initialize a new
boolean array to store the details of explored nodes
        DFSUtil(v, already);
    }

    public static void main(String args[])
    {
        Graph g = new Graph(6);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 0);
        g.addEdge(1, 3);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 4);
        g.addEdge(3, 5);
        g.addEdge(4, 3);
        g.addEdge(5, 3);

        System.out.println(
            "Following is Depth First Traversal: ");

        g.DFS(0);
    }
}
```

Output:

Program 4: Implement DFS algorithm using JAVA iterative technique

```
import java.util.*;


 class Graph
    {
        int V;                                       //number of nodes

        LinkedList<Integer>[] adj;                   //adjacency list

        Graph(int V)
        {
            this.V = V;
            adj = new LinkedList[V];

            for (int i = 0; i < adj.length; i++)
                adj[i] = new LinkedList<Integer>();

        }

        void addEdge(int v, int w)
        {
            adj[v].add(w);                                  //adding an edge to the 
adjacency list (edges are bidirectional in this example)
        }

        void DFS(int n)
        {
            boolean nodes[] = new boolean[V];

            Stack<Integer> stack = new Stack<>();

            stack.push(n);                                        //push root node to 
the stack
            int a = 0;

            while(!stack.empty())
            {
                n = stack.peek();                           //extract the top element 
of the stack
                stack.pop();                                //remove the top element 
from the stack

                if(nodes[n] == false)
                {
                    System.out.print(n + " ");
                    nodes[n] = true;
                }

                for (int i = 0; i < adj[n].size(); i++)  //iterate through the 
linked list and then propagate to the next few nodes
            {
                a = adj[n].get(i);
                if (!nodes[a])                        //only push those nodes to the 
stack which aren't in it already
                {
                    stack.push(a);                          //push the top 
element to the stack
```

```
                }
            }

            }
        }

    public static void main(String[] args)
    {
        Graph g = new Graph(6);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 0);
        g.addEdge(1, 3);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 4);
        g.addEdge(3, 5);
        g.addEdge(4, 3);
        g.addEdge(5, 3);

        System.out.println("Following is the Depth First Traversal");
        g.DFS(0);
    }
}
```

Output:

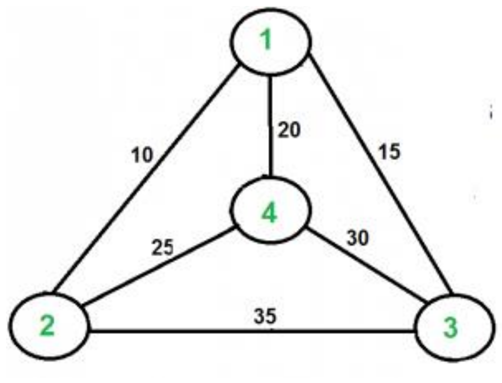
Travelling Salesman Problem

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns back to the starting point.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in the figure. A TSP tour in the graph is 1 -> 2 -> 4 -> 3 -> 1. The cost of the tour is 10 + 25 + 30 + 15 which is 80.

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.



Output of Given Graph:

Minimum weight Hamiltonian Cycle : 10 + 25 + 30 + 15 = 80

Algorithm of TSP

1. Consider city 1 (let say 0th node) as the starting and ending point. Since route is cyclic, we can consider any point as starting point.
2. Start traversing from the source to its adjacent nodes in DFS manner.
3. Calculate cost of every traversal and keep track of minimum cost and keep on updating the value of minimum cost stored value.
4. Return the permutation with minimum cost.

Program 5: Implement the TSP based on above approach.

```
// Java implementation of the approach
class GFG
{

        // Function to find the minimum weight
        // Hamiltonian Cycle
        static int tsp(int[][] graph, boolean[] v,
                                int currPos, int n,
                                int count, int cost, int ans)
        {

                // If last node is reached and it has a link
                // to the starting node i.e the source then
                // keep the minimum value out of the total cost
                // of traversal and "ans"
                // Finally return to check for more possible values
```

Student Name: ________________________________ Roll No: _______________ Section: ____________

```
            if (count == n && graph[currPos][0] > 0)
            {
                    ans = Math.min(ans, cost + graph[currPos][0]);
                    return ans;
            }

            // BACKTRACKING STEP
            // Loop to traverse the adjacency list
            // of currPos node and increasing the count
            // by 1 and cost by graph[currPos,i] value
            for (int i = 0; i < n; i++)
            {
                    if (v[i] == false && graph[currPos][i] > 0)
                    {

                            // Mark as visited
                            v[i] = true;
                            ans = tsp(graph, v, i, n, count + 1,
                                            cost + graph[currPos][i], ans);

                            // Mark ith node as unvisited
                            v[i] = false;
                    }
            }
            return ans;
    }

    // Driver code
    public static void main(String[] args)
    {

            // n is the number of nodes i.e. V
            int n = 4;

            int[][] graph = {{0, 10, 15, 20},
                                            {10, 0, 35, 25},
                                            {15, 35, 0, 30},
                                            {20, 25, 30, 0}};

            // Boolean array to check if a node
            // has been visited or not
            boolean[] v = new boolean[n];

            // Mark 0th node as visited
            v[0] = true;
            int ans = Integer.MAX_VALUE;
```

Student Name: ______________________________ Roll No: ______________ Section: ____________

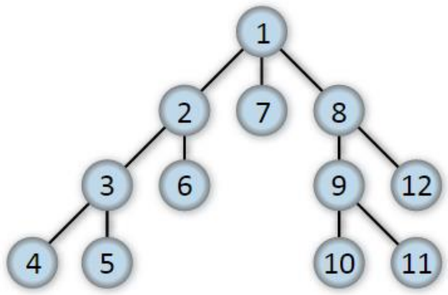
```
                // Find the minimum weight Hamiltonian Cycle
                ans = tsp(graph, v, 0, n, 1, 0, ans);

                // ans is the minimum weight Hamiltonian Cycle
                System.out.println(ans);
        }
}
```

Output:

Student Name: ______________________________ Roll No: _____________ Section: ____________

Programming Exercise

1. Perform DFS and BFS algorithms on the following graphs and compare the results on the basis of time of execution.

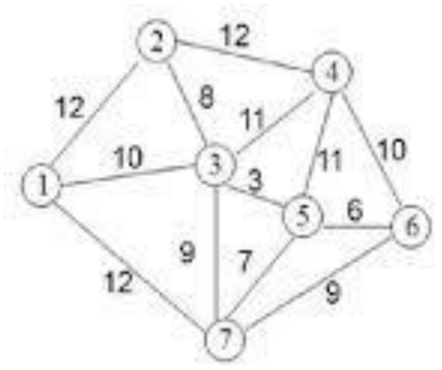


2. Use the following matrix to solve the travelling salesman problem.

To city

$$\begin{array}{lc|ccccc|}
 & & \textbf{A} & \textbf{B} & \textbf{C} & \textbf{D} & \textbf{E} \\
 & \textbf{A} & \alpha & 2 & 5 & 7 & 1 \\
 & \textbf{B} & 6 & \alpha & 3 & 8 & 2 \\
\textbf{From city} & \textbf{C} & 8 & 7 & \alpha & 4 & 7 \\
 & \textbf{D} & 12 & 4 & 6 & \alpha & 5 \\
 & \textbf{E} & 1 & 3 & 2 & 8 & \alpha
\end{array}$$

3. Use the graph to show your solution if you are using travelling salesman algorithm.


12
12
8
11
10
11
10
3
6
9
7
12
9