Program 1: BFS

Breadth-First Search (BFS) is based on traversing nodes by adding the neighbors of each node to the traversal queue starting from the root node. The BFS for a graph is similar to that of a tree, with the exception that graphs may have cycles. In contrast to depth-first search, all neighbor nodes at a given depth are investigated before proceeding to the next level.

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 bidirectional in this example) \*/

    }

    void BFS(int n)

    {

        boolean nodes[] = new boolean[node];       /\* initialize boolean array for holding 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 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;

                    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:

Breadth First Traversal for the graph is:

0 1 3 4 2 5

Practical 2: DFS

The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.The space complexity of the DFS algorithm is O(V).

import java.io.\*;

import java.util.\*;

// This class represents a

// directed graph using adjacency

// list representation

class Graph {

    private int V; // No. of vertices

    // Array of lists for

    // Adjacency List Representation

    private LinkedList<Integer> adj[];

    // Constructor

    @SuppressWarnings("unchecked") Graph(int v)

    {

        V = v;

        adj = new LinkedList[v];

        for (int i = 0; i < v; ++i)

            adj[i] = new LinkedList();

    }

    // Function to add an edge into the graph

    void addEdge(int v, int w)

    {

        adj[v].add(w); // Add w to v's list.

    }

    // A function used by DFS

    void DFSUtil(int v, boolean visited[])

    {

        // Mark the current node as visited and print it

        visited[v] = true;

        System.out.print(v + " ");

        // Recur for all the vertices adjacent to this

        // vertex

        Iterator<Integer> i = adj[v].listIterator();

        while (i.hasNext()) {

            int n = i.next();

            if (!visited[n])

                DFSUtil(n, visited);

        }

    }

    // The function to do DFS traversal.

    // It uses recursive

    // DFSUtil()

    void DFS(int v)

    {

        // Mark all the vertices as

        // not visited(set as

        // false by default in java)

        boolean visited[] = new boolean[V];

        // Call the recursive helper

        // function to print DFS

        // traversal

        DFSUtil(v, visited);

    }

    // Driver's Code

    public static void main(String args[])

    {

        Graph g = new Graph(4);

        g.addEdge(0, 1);

        g.addEdge(0, 2);

        g.addEdge(1, 2);

        g.addEdge(2, 0);

        g.addEdge(2, 3);

        g.addEdge(3, 3);

        System.out.println(

            "Following is Depth First Traversal "

            + "(starting from vertex 2)");

        // Function call

        g.DFS(2);

    }

}

Following is Depth First Traversal (starting from vertex 2)

2 0 1 3

Practical 3:

import java.io.\*;

import java.lang.\*;

import java.util.\*;

class MST {

    // Number of vertices in the graph

    private static final int V = 5;

    // A utility function to find the vertex with minimum

    // key value, from the set of vertices not yet included

    // in MST

    int minKey(int key[], Boolean mstSet[])

    {

        // Initialize min value

        int min = Integer.MAX\_VALUE, min\_index = -1;

        for (int v = 0; v < V; v++)

            if (mstSet[v] == false && key[v] < min) {

                min = key[v];

                min\_index = v;

            }

        return min\_index;

    }

    // A utility function to print the constructed MST

    // stored in parent[]

    void printMST(int parent[], int graph[][])

    {

        System.out.println("Edge \tWeight");

        for (int i = 1; i < V; i++)

            System.out.println(parent[i] + " - " + i + "\t"

                            + graph[i][parent[i]]);

    }

    // Function to construct and print MST for a graph

    // represented using adjacency matrix representation

    void primMST(int graph[][])

    {

        // Array to store constructed MST

        int parent[] = new int[V];

        // Key values used to pick minimum weight edge in

        // cut

        int key[] = new int[V];

        // To represent set of vertices included in MST

        Boolean mstSet[] = new Boolean[V];

        // Initialize all keys as INFINITE

        for (int i = 0; i < V; i++) {

            key[i] = Integer.MAX\_VALUE;

            mstSet[i] = false;

        }

        // Always include first 1st vertex in MST.

        key[0] = 0; // Make key 0 so that this vertex is

        // picked as first vertex

        parent[0] = -1; // First node is always root of MST

        // The MST will have V vertices

        for (int count = 0; count < V - 1; count++) {

            // Pick thd minimum key vertex from the set of

            // vertices not yet included in MST

            int u = minKey(key, mstSet);

            // Add the picked vertex to the MST Set

            mstSet[u] = true;

            // Update key value and parent index of the

            // adjacent vertices of the picked vertex.

            // Consider only those vertices which are not

            // yet included in MST

            for (int v = 0; v < V; v++)

                // graph[u][v] is non zero only for adjacent

                // vertices of m mstSet[v] is false for

                // vertices not yet included in MST Update

                // the key only if graph[u][v] is smaller

                // than key[v]

                if (graph[u][v] != 0 && mstSet[v] == false

                    && graph[u][v] < key[v]) {

                    parent[v] = u;

                    key[v] = graph[u][v];

                }

        }

        // print the constructed MST

        printMST(parent, graph);

    }

    public static void main(String[] args)

    {

        /\* Let us create the following graph

        2 3

        (0)--(1)--(2)

        | / \ |

        6| 8/ \5 |7

        | /  \ |

        (3)-------(4)

            9        \*/

        MST t = new MST();

        int graph[][] = new int[][] { { 0, 2, 0, 6, 0 },

                                    { 2, 0, 3, 8, 5 },

                                    { 0, 3, 0, 0, 7 },

                                    { 6, 8, 0, 0, 9 },

                                    { 0, 5, 7, 9, 0 } };

        // Print the solution

        t.primMST(graph);

    }

}

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

Practical 4: A\* Algorithm

import java.util.PriorityQueue;

import java.util.HashSet;

import java.util.Set;

import java.util.List;

import java.util.Comparator;

import java.util.ArrayList;

import java.util.Collections;

public class AstarSearchAlgo{

        //h scores is the stright-line distance from the current city to Bucharest

        public static void main(String[] args){

                //initialize the graph base on the Romania map

                Node n1 = new Node("Arad",366);

                Node n2 = new Node("Zerind",374);

                Node n3 = new Node("Oradea",380);

                Node n4 = new Node("Sibiu",253);

                Node n5 = new Node("Fagaras",178);

                Node n6 = new Node("Rimnicu Vilcea",193);

                Node n7 = new Node("Pitesti",98);

                Node n8 = new Node("Timisoara",329);

                Node n9 = new Node("Lugoj",244);

                Node n10 = new Node("Mehadia",241);

                Node n11 = new Node("Drobeta",242);

                Node n12 = new Node("Craiova",160);

                Node n13 = new Node("Bucharest",0);

                        Node n14 = new Node("Giurgiu",77);

                //initialize the edges

                //Arad

                n1.adjacencies = new Edge[]{

                        new Edge(n2,75),

                        new Edge(n4,140),

                        new Edge(n8,118)

                };

                 //Zerind

                n2.adjacencies = new Edge[]{

                        new Edge(n1,75),

                        new Edge(n3,71)

                };

                 //Oradea

                n3.adjacencies = new Edge[]{

                        new Edge(n2,71),

                        new Edge(n4,151)

                };

                 //Sibiu

                n4.adjacencies = new Edge[]{

                        new Edge(n1,140),

                        new Edge(n5,99),

                        new Edge(n3,151),

                        new Edge(n6,80),

                };

                 //Fagaras

                n5.adjacencies = new Edge[]{

                        new Edge(n4,99),

                        //178

                        new Edge(n13,211)

                };

                 //Rimnicu Vilcea

                n6.adjacencies = new Edge[]{

                        new Edge(n4,80),

                        new Edge(n7,97),

                        new Edge(n12,146)

                };

                 //Pitesti

                n7.adjacencies = new Edge[]{

                        new Edge(n6,97),

                        new Edge(n13,101),

                        new Edge(n12,138)

                };

                 //Timisoara

                n8.adjacencies = new Edge[]{

                        new Edge(n1,118),

                        new Edge(n9,111)

                };

                 //Lugoj

                n9.adjacencies = new Edge[]{

                        new Edge(n8,111),

                        new Edge(n10,70)

                };

                 //Mehadia

                n10.adjacencies = new Edge[]{

                        new Edge(n9,70),

                        new Edge(n11,75)

                };

                 //Drobeta

                n11.adjacencies = new Edge[]{

                        new Edge(n10,75),

                        new Edge(n12,120)

                };

                 //Craiova

                n12.adjacencies = new Edge[]{

                        new Edge(n11,120),

                        new Edge(n6,146),

                        new Edge(n7,138)

                };

                //Bucharest

                n13.adjacencies = new Edge[]{

                        new Edge(n7,101),

                        new Edge(n14,90),

                        new Edge(n5,211)

                };

                 //Giurgiu

                n14.adjacencies = new Edge[]{

                        new Edge(n13,90)

                };

                AstarSearch(n1,n13);

                List<Node> path = printPath(n13);

                        System.out.println("Path: " + path);

        }

        public static List<Node> printPath(Node target){

                List<Node> path = new ArrayList<Node>();

        for(Node node = target; node!=null; node = node.parent){

            path.add(node);

        }

        Collections.reverse(path);

        return path;

        }

        public static void AstarSearch(Node source, Node goal){

                Set<Node> explored = new HashSet<Node>();

                PriorityQueue<Node> queue = new PriorityQueue<Node>(20,

                        new Comparator<Node>(){

                                 //override compare method

                 public int compare(Node i, Node j){

                    if(i.f\_scores > j.f\_scores){

                        return 1;

                    }

                    else if (i.f\_scores < j.f\_scores){

                        return -1;

                    }

                    else{

                        return 0;

                    }

                 }

                        }

                        );

                //cost from start

                source.g\_scores = 0;

                queue.add(source);

                boolean found = false;

                while((!queue.isEmpty())&&(!found)){

                        //the node in having the lowest f\_score value

                        Node current = queue.poll();

                        explored.add(current);

                        //goal found

                        if(current.value.equals(goal.value)){

                                found = true;

                        }

                        //check every child of current node

                        for(Edge e : current.adjacencies){

                                Node child = e.target;

                                double cost = e.cost;

                                double temp\_g\_scores = current.g\_scores + cost;

                                double temp\_f\_scores = temp\_g\_scores + child.h\_scores;

                                /\*if child node has been evaluated and

                                the newer f\_score is higher, skip\*/

                                if((explored.contains(child)) &&

                                        (temp\_f\_scores >= child.f\_scores)){

                                        continue;

                                }

                                /\*else if child node is not in queue or

                                newer f\_score is lower\*/

                                else if((!queue.contains(child)) ||

                                        (temp\_f\_scores < child.f\_scores)){

                                        child.parent = current;

                                        child.g\_scores = temp\_g\_scores;

                                        child.f\_scores = temp\_f\_scores;

                                        if(queue.contains(child)){

                                                queue.remove(child);

                                        }

                                        queue.add(child);

                                }

                        }

                }

        }

}

class Node{

        public final String value;

        public double g\_scores;

        public final double h\_scores;

        public double f\_scores = 0;

        public Edge[] adjacencies;

        public Node parent;

        public Node(String val, double hVal){

                value = val;

                h\_scores = hVal;

        }

        public String toString(){

                return value;

        }

}

class Edge{

        public final double cost;

        public final Node target;

        public Edge(Node targetNode, double costVal){

                target = targetNode;

                cost = costVal;

        }

}

Output: Path: [Arad, Sibiu, Rimnicu Vilcea, Pitesti, Bucharest]