<u>Experiment: 4</u>

Aim : Program on uninformed search methods

Objective : To make students understand uninformed search methods(Uniform Cost Search)

Theory :

Uniform cost search

Breadth-first search finds the shallowest goal state, but this may not always be the least-cost solution for a general path cost function. Uniform cost search modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe (as measured by the path cost g(n)), rather than the lowest-depth node.
It is easy to see that breadth-first search is just uniform cost search with g(n) = DEPTH(«).
When certain conditions are met, the first solution that is found is guaranteed to be the cheapest solution, because if there were a cheaper path that was a solution, it would have been ,expanded earlier, and thus would have been found first.

<u>Procedure</u>

UniformCostSearch(Graph, start, goal)
node ← start
cost ← 0
frontier ← priority queue containing node only
explored ← empty set
do
if frontier is empty
return failure
node ← frontier.pop()
if node is goal
return solution
explored.add(node)
for each of node's neighbors n
if n is not in explored
if n is not in frontier
frontier.add(n)

CODE:

```
import java.util.PriorityQueue;
import java.util.HashSet;
import java.util.Set;
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
import java.util.Comparator;

//diff between uniform cost search and dijkstra algo is that UCS has a goal
```

```
public class UniformCostSearchAlgo{
  public static void main(String[] args){
    //initialize the graph base on the Romania map
    Node n1 = new Node("Arad");
    Node n2 = new Node("Zerind");
    Node n3 = new Node("Oradea");
    Node n4 = new Node("Sibiu");
    Node n5 = new Node("Fagaras");
    Node n6 = new Node("Rimnicu Vilcea");
    Node n7 = new Node("Pitesti");
    Node n8 = new Node("Timisoara");
    Node n9 = new Node("Lugoj");
    Node n10 = new Node("Mehadia");
    Node n11 = new Node("Drobeta");
    Node n12 = new Node("Craiova");
    Node n13 = new Node("Bucharest");
    Node n14 = new Node("Giurgiu");

    //initialize the edges
    n1.adjacencies = new Edge[]{
      new Edge(n2,75),
      new Edge(n4,140),
      new Edge(n8,118)
    };

    n2.adjacencies = new Edge[]{
      new Edge(n1,75),
      new Edge(n3,71)
    };

    n3.adjacencies = new Edge[]{
      new Edge(n2,71),
      new Edge(n4,151)
    };

    n4.adjacencies = new Edge[]{
      new Edge(n1,140),
      new Edge(n5,99),
      new Edge(n3,151),
      new Edge(n6,80),
    };

    n5.adjacencies = new Edge[]{
      new Edge(n4,99),
      new Edge(n13,211)
    };

    n6.adjacencies = new Edge[]{
      new Edge(n4,80),
      new Edge(n7,97),
      new Edge(n12,146)
```

```
    };

    n7.adjacencies = new Edge[]{
        new Edge(n6,97),
        new Edge(n13,101),
        new Edge(n12,138)
    };

    n8.adjacencies = new Edge[]{
        new Edge(n1,118),
        new Edge(n9,111)
    };

    n9.adjacencies = new Edge[]{
        new Edge(n8,111),
        new Edge(n10,70)
    };

    n10.adjacencies = new Edge[]{
        new Edge(n9,70),
        new Edge(n11,75)
    };

    n11.adjacencies = new Edge[]{
        new Edge(n10,75),
        new Edge(n12,120)
    };

    n12.adjacencies = new Edge[]{
        new Edge(n11,120),
        new Edge(n6,146),
        new Edge(n7,138)
    };

    n13.adjacencies = new Edge[]{
        new Edge(n7,101),
        new Edge(n14,90),
        new Edge(n5,211)
    };

    n14.adjacencies = new Edge[]{
        new Edge(n13,90)
    };
    UniformCostSearch(n1,n13);

    List<Node> path = printPath(n13);

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

}
```

```
public static void UniformCostSearch(Node source, Node goal){
   source.pathCost = 0;
   PriorityQueue<Node> queue = new PriorityQueue<Node>(20,
      new Comparator<Node>(){

         //override compare method
         public int compare(Node i, Node j){
            if(i.pathCost > j.pathCost){
               return 1;
            }
            else if (i.pathCost < j.pathCost){
               return -1;
            }

            else{
               return 0;
            }
         }
      }
   );
   queue.add(source);
   Set<Node> explored = new HashSet<Node>();
   boolean found = false;

   //while frontier is not empty
   do{
      Node current = queue.poll();
      explored.add(current);
      if(current.value.equals(goal.value)){
         found = true;
      }
      for(Edge e: current.adjacencies){
         Node child = e.target;
         double cost = e.cost;
         child.pathCost = current.pathCost + cost;
         if(!explored.contains(child) && !queue.contains(child)){
            child.parent = current;
            queue.add(child);
            System.out.println(child);
            System.out.println(queue);
            System.out.println();
         }
         else if((queue.contains(child))&&(child.pathCost>current.pathCost)){
            child.parent=current;
            current = child;
         }
      }
   }while(!queue.isEmpty());
}
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;
    }
}

class Node{
    public final String value;
    public double pathCost;
    public Edge[] adjacencies;
    public Node parent;
    public Node(String val){
        value = val;
    }
    public String toString(){
        return value;
    }
}
class Edge{
    public final double cost;
    public final Node target;
    public Edge(Node targetNode, double costVal){
        cost = costVal;
        target = targetNode;
    }
}
```

```
C:\pooja>javac UniformCostSearchAlgo.java

C:\pooja>java UniformCostSearchAlgo
Zerind
[Zerind]

Sibiu
[Zerind, Sibiu]

Timisoara
[Zerind, Sibiu, Timisoara]

Oradea
[Timisoara, Sibiu, Oradea]

Lugoj
[Sibiu, Oradea, Lugoj]

Fagaras
[Oradea, Lugoj, Fagaras]

Rimnicu Vilcea
[Oradea, Lugoj, Fagaras, Rimnicu Vilcea]

Mehadia
[Fagaras, Rimnicu Vilcea, Mehadia]

Bucharest
[Mehadia, Rimnicu Vilcea, Bucharest]

Drobeta
[Rimnicu Vilcea, Bucharest, Drobeta]

Pitesti
[Drobeta, Bucharest, Pitesti]

Craiova
[Drobeta, Bucharest, Pitesti, Craiova]

Giurgiu
[Pitesti, Craiova, Giurgiu]

Path: [Arad, Sibiu, Fagaras, Bucharest]

C:\pooja>
```