CYPRUS INTERNATIONAL UNIVERSITY

FACULTY OF ENGINEERING

CMPE415
PROJECT REPORT
ARTIFICAL INTELLIGENCE

STUDENT 1

NAME : YASSER

SURNAME: ELKABBOUT

NUMBER : 20131196

STUDENT 2

NAME : ABDALLAH FAHMI NEMER

SURNAME: NASSER

NUMBER : 20121604

1. The aim of the project

The problem:

The aim of this project was to write a program that will implement the A* Search Algorithm to find the solution path from a starting node to a goal node.

The input:

Two text files were given, “tree.txt” and “heuristic.txt”. “tree.txt” defines the search tree where each line will contain a parent-child relation and a path cost between them. Each data is separated with a space.

e.g.
```
A B 5
A C 3
B D 6
```

The first character in the first line will be the start node (A in here) and the goal node will be “G”.

“heuristic.txt” will define the heuristic, h(n), values. Each line contains the heuristic value of each node. Each data is separated with a space.

e.g.
```
A 20
B 15
C 18
```

The output:

The program should give the solution path and the path cost from start node to goal.

Allowed programming languages:

C++, C# and Java programming languages are allowed to be used to implement the search algorithm.

2. The programming language used

We decided to use Java over C# and C++ as Java is having a well defined and structured library when it comes to trees and graphs. It will be more efficient to implement such an algorithm use a high level language like Java.

Furthermore, this program can run on any operating system as it can be launched by using a JVM. In contrast, using C# will limit us to Microsoft’s operating system, windows.

However, C++ would be also a good choice. But, the implementation of classes and the way of abstraction is much more higher in Java. That’s why, we think that Java is the best choice.

3. Definition: A* Search Algorithm

The most widely known form of best-first search is called A* Search. It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

F(n) = g(n) + h(n)

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have:

F(n)= estimated cost of the cheapest solution through n.

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest of f(n).

It turns out that this strategy is more reasonable: provided that the heuristic function h(n) satisfies certain conditions, A* search is both complete and optimal. The algorithm is identical to Uniform-cost-search except that A* uses g(h) and h(n) instead of g(h). [1]

The way of how the A* Search works is illustrated in figure 1. Each time, the node having the lowest f(n) value is the node expanded.

The targets nodes will be expanded until reaching the goal destination, which is Bucharest.

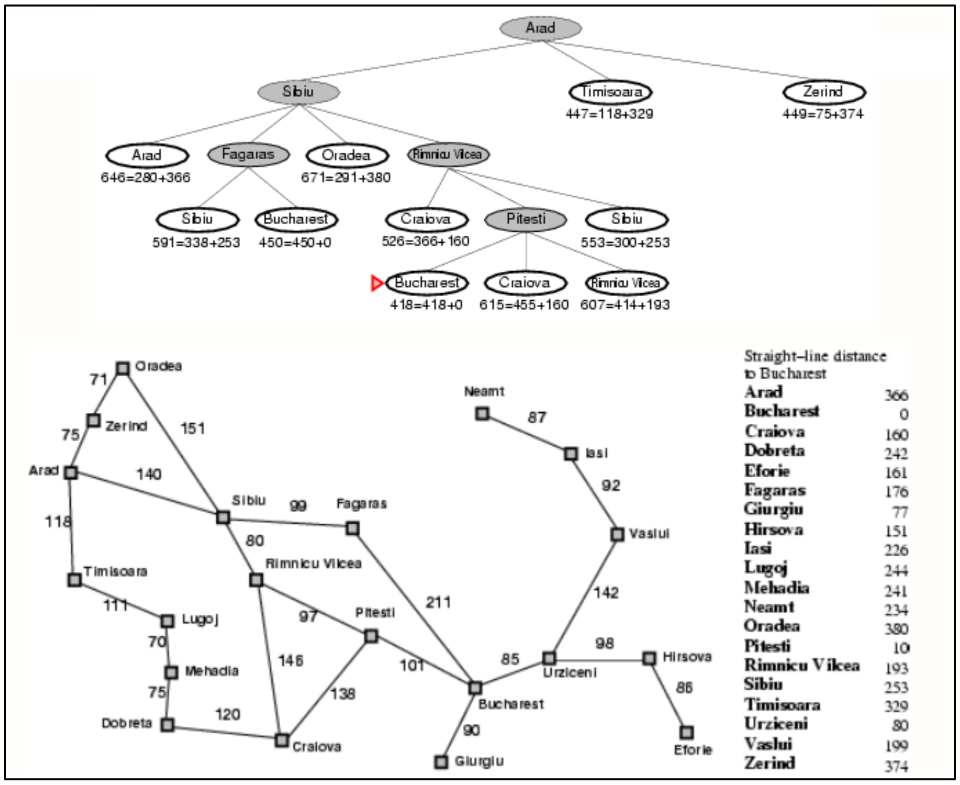

Figure1. A* Search Algorithm and Romania’s map

4. The pseudocode of this algorithm

The pseudocode of this algorithm is shown in figure 2. This pseudocode was a big help while implementing the algorithm in Java.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
   return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), $\infty$)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new $f$-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors $\leftarrow$ [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, $\infty$
  for each s in successors do  /* update $f$ with value from previous search, if any */
      $s.f \leftarrow \max(s.g + s.h,\ node.f)$
  loop do
      best $\leftarrow$ the lowest $f$-value node in successors
      if $best.f > f\_limit$ then return failure, $best.f$
      alternative $\leftarrow$ the second-lowest $f$-value among successors
      result, $best.f \leftarrow$ RBFS(problem, best, min(f_limit, alternative))
      if result $\neq$ failure then return result
```

Figure2. A* Search pseudocode

5. Advantages of A* Search

Noted for its performance and accuracy, it enjoys widespread use. Peter Hart, Nils Nilsson and Bertram Raphael first described the algorithm in 1968. It is an extension of Edsger Dijkstra's 1959 algorithm. A* achieves better performance (with respect to time) by using heuristics. In 1964 Nils Nilsson invented a heuristic based approach to increase the speed of Dijkstra's algorithm. This algorithm was called A1. In 1967 Bertram Raphael made dramatic improvements upon this algorithm, but failed to show optimality. He called this algorithm A2. Then in 1968 Peter E. Hart introduced an argument that proved A2 was optimal when using a consistent heuristic with only minor changes. His proof of the algorithm also included a section that showed that the new A2 algorithm was the best algorithm possible given the conditions. He thus named the new algorithm in Kleene star syntax to be the algorithm that starts with A and includes all possible version numbers or A* [2].

As A* traverses the graph, it follows a path of the lowest known cost, keeping a sorted priority queue of alternate path segments along the way. If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. This process continues until the goal is reached.[2]
A* uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals).[2]

A* best-first search is known as a complete and optimal algorithm:

- Complete: Yes, unless there are infinitely many nodes with f <= f(g).
- Time/Space: Exponential complexity.
- Optimal: Yes.
- Optimal Efficient: Yes (no algorithm with the same heuristic is guaranteed to expand fewer nodes).

6. Disadvantages of A* Search

The performance of heuristic search algorithms depends on the quality of the heuristic function. If the heuristic values provided are not admissible, the function will return some unexpected solutions. It doesn’t guarantee to give the optimal and the complete solution and may enter a loop, as a dead end.

By Choosing the heuristic values, one should be very careful and precise.

Further more, as this algorithm uses a lot of memory while operating (exponential complexity), some other algorithms might be more memory efficient like simple-MBA*. The latter finds the optimal reachable solution given the memory constraint.

7. The source code

As requested, the source code of our project is attached. It is within the appendix.

The main Search algorithm is called:

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

- We used a comparator to sort the queue according to the f(n) values.
- After this, the node with the least f(n) was added to the queue.
- If it is the goal node, the program will stop and add it to the solution path.
- Otherwise, the program will check the edges of the popped node.
- The edge with the least f(n) function is chosen.
- The total path cost value is incremented.
- The target node is expanded and added to the queue.
- And it iterated in this way, until finding the goal node.

References

[1] Nosrati, Karimi. “Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches”. World Applied Programming, Vol (2), No (4), April 2012. 251-256. ISSN: 2222-2510.
[2] Stuart Russel, Peter Norvig. “Artificial Intelligence: A Moder Approach, third edition”. Prentice Hall.

Appendix

```
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.InputStreamReader;
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{

        public static List <TreeNode> TreeAStar = new ArrayList <TreeNode>();
        public static ArrayList<String> Values1= new ArrayList <String>();
        public static ArrayList<String> Values2=new ArrayList <String>();
        public static ArrayList<String> Values3=new ArrayList<String>();

        public static void main(String[] args){

           try{


          FileInputStream fstream = new FileInputStream("heuristic1.txt");
          DataInputStream in = new DataInputStream(fstream);
          BufferedReader br = new BufferedReader(new InputStreamReader(in));
          String strLine;
          TreeNode node = null;

          while ((strLine = br.readLine()) != null)   {
          String[] tokens = strLine.split(" ");
          node = new TreeNode(tokens[0]);
          node.heuristic = Integer.parseInt(tokens[1]);
          TreeAStar.add(node);

          }
         in.close();
       }

       catch (Exception e){
           System.err.println("Error: "+ e.getMessage());
       }

       try{

          FileInputStream fstream = new FileInputStream("tree1.txt");
          DataInputStream in = new DataInputStream(fstream);
          BufferedReader br = new BufferedReader(new InputStreamReader(in));
          String strLine;
          TreeNode node = null;

          while ((strLine = br.readLine()) != null)   {
          String[] tokens = strLine.split(" ");

          for(TreeNode currentNode : TreeAStar){

          if(currentNode.letter.equals(tokens[0]))
          currentNode.addChildren((tokens[1]));

        if(currentNode.letter.equals(tokens[1])){
           currentNode.pathCostFromParent=Integer.parseInt(tokens[2]);
           currentNode.parent=tokens[0];
        }
        }

}

in.close();

}catch (Exception e){
  System.err.println("Error: " + e.getMessage());
}



      for (TreeNode currentNode: TreeAStar){

      System.out.println("Node: "+ currentNode.letter +"\n"
                    +"Heuristic: " +currentNode.heuristic +"\n" +
                    "Parent: " +currentNode.parent +"\n"+
                    "path Cost from Parent: " +currentNode.pathCostFromParent+"\n"+
                    "Childrens: "+currentNode.childrens +"\n-----------");

  }

      ////////////////////////////////////////////////////////////////////////////////////////////////////////


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

      for(TreeNode currentNode: TreeAStar){

          nodesMap.add(new Node(currentNode.letter, currentNode.heuristic));

      }

       try{

       FileInputStream fstream = new FileInputStream("tree1.txt");
       DataInputStream in = new DataInputStream(fstream);
       BufferedReader br = new BufferedReader(new InputStreamReader(in));
       String strLine;


       while ((strLine = br.readLine()) != null)   {
       String[] tokens = strLine.split(" ");

       for(Node currentNode:nodesMap){

          if(currentNode.value.equals(tokens[0])){

               Values1.add(tokens[0]);
               Values2.add(tokens[1]);
               Values3.add(tokens[2]);


          }
       }
```

```
        }

    in.close();

    }catch (Exception e){
      System.err.println("Error: " + e.getMessage());
    }

            List <struct> structs = new ArrayList <struct>();
             for(int i=0;i<Values1.size();i++){

            for(int x=0; x<nodesMap.size();x++){


                if(nodesMap.get(x).value.equals(Values1.get(i)))
                    structs.add(new struct(nodesMap.get(x)));



            }
            }

             for(int i=0;i<Values2.size();i++){

            for(int x=0; x<nodesMap.size();x++){


                if(nodesMap.get(x).value.equals(Values2.get(i)))
                    structs.get(i).node2=nodesMap.get(x);

            }
            }

            for(int i=0;i<Values2.size();i++){

            for(int x=0; x<nodesMap.size();x++){


                if(nodesMap.get(x).value.equals(Values2.get(i)))
                    structs.get(i).cost= Double.parseDouble(Values3.get(i));

            }
            }


          for(int i=0;i<structs.size();i++){
               for (int x=0; x<nodesMap.size();x++){

                   if(structs.get(i).node1.value.equals(nodesMap.get(x).value)){
                       nodesMap.get(x).edgs.add(
                               new Edge (
                                       (structs.get(i).node2),
                                       (structs.get(i).cost))
                       );


                   }


               }
```

```
  }


  int goalIndex = 0;
  for(Node currentNode:nodesMap){

     int size=currentNode.edgs.size();;


     if(size ==3){


      currentNode.adjacencies = new Edge[]{
      currentNode.edgs.get(0), currentNode.edgs.get(1), currentNode.edgs.get(2)
  };


  }

      if(size ==2){


      currentNode.adjacencies = new Edge[]{
      currentNode.edgs.get(0), currentNode.edgs.get(1)
  };


  }

         if(size ==1){


      currentNode.adjacencies = new Edge[]{
      currentNode.edgs.get(0)
  };


  }

            if(size ==0){


      currentNode.adjacencies = new Edge[]{

  };


  }




  }

  //Index of target Node
 for(int i=0;i<nodesMap.size();i++){

    if(nodesMap.get(i).value.equals("G"))
        goalIndex=i;
}
```

```
        AstarSearch(nodesMap.get(0),nodesMap.get(goalIndex));

        List<Node> path = printPath(nodesMap.get(goalIndex));

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

                System.out.println(nodesMap.get(1).edgs.size());


}

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);

                        }

                }

        }

}
public static 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 List <Edge> edgs = new ArrayList <Edge>();
public  String firstVariable;
public Integer secondVariable;

public Node(String val, double hVal){
        value = val;
        h_scores = hVal;
}

public String toString(){
        return value;
        }

}

        public static class struct{
            Node node1;
            Node node2;
            double cost;

            public struct( Node node1){

                this.node1=node1;


            }
            public struct( Node node2, int cost){

                this.node2=node2;


            }

            public struct(){}


        }

        public static class Edge{
        public final double cost;
        public final Node target;

        public Edge(Node targetNode, double costVal){
                target = targetNode;
                cost = costVal;


        }
}


        // TreeNode Class
    public static class TreeNode {

      //Node's variables
      public List<String> childrens = new ArrayList<String>();
      public String parent = null;
      public String letter;
      public int fFunction;
      public int heuristic;
      public int pathCostFromParent;
      public int depthLevel;
      public boolean flag = false;


      // Node constructors
       public TreeNode(){

      }

        public TreeNode(String letter, int heuristic){

          this.letter=letter;
```

```
            this.heuristic=heuristic;
        }

        public TreeNode(String letter, int heuristic, String parent){

            this.letter=letter;
            this.heuristic=heuristic;
            this.parent = parent;
        }

        public TreeNode(String parent, String letter, int pathCost){
            this.parent=parent;
            this.letter=letter;
            this.pathCostFromParent=pathCost;

        }

        public TreeNode(String letter){
            this.letter=letter;
        }

         public void addChildren(String child){
            this.childrens.add(child);

        }

      }

      public static class populateTree{


      }




  }
```