**Assignment 1**

**Problem Statement:**

Use Heuristic Search Techniques to Implement Best first

search (Best-Solution but not always optimal) and A\* algorithm (Always gives

optimal solution)

**Theory**

Heuristic Search

A heuristic is a method that

• might not always find the best solution

• but is guaranteed to find a good solution in reasonable time.

• By sacrificing completeness it increases efficiency.

• Useful in solving tough problems which could not be solved any other way.

• solutions take an infinite time or very long time to compute.

The classic example of heuristic search methods is the travelling salesman

problem.

The heuristic function is a way to inform the search about the direction to

a goal. It provides an informed way to guess which neighbor of a node will lead

to a goal. There is nothing magical about a heuristic function. It must use only

information that can be readily obtained about a node. Typically a trade-off

exists between the amount of work it takes to derive a heuristic value for a node

and how accurately the heuristic value of a node measures the actual path cost

from the node to a goal.

**Hill Climbing**

Here the generate and test method is augmented by an heuristic function which

measures the closeness of the current state to the goal state.

1. Evaluate the initial state if it is goal state quit otherwise current state is

initial state.

2. Select a new operator for this state and generate a new state.

3. Evaluate the new state

(a) if it is closer to goal state than current state make it current state

(b) if it is no better ignore

4. If the current state is goal state or no new operators available, quit. Oth-

erwise repeat from 2.1.3.1

**A\* Algorithm**

The A\* algorithm combines features of uniform-cost search and pure heuristic

search to efficiently compute optimal solutions. A\* algorithm is a best-first

search algorithm in which the cost associated with a node is

f (n) = g(n) + h(n)

where g(n) is the cost of the path from the initial state to node n and h(n) is

the heuristic estimate or the cost or a path from node n to a goal. Thus, f(n)

estimates the lowest total cost of any solution path going through node n. At

each point a node with lowest f value is chosen for expansion. Ties among nodes

of equal f value should be broken in favor of nodes with lower h values. The

algorithm terminates when a goal is chosen for expansion.

The main drawback of A\* algorithm and indeed of any best-first search is

its memory requirement. Since at least the entire open list must be saved, A\*

algorithm is severely space-limited in practice, and is no more practical than

best-first search algorithm on current machines. For example, while it can be

run successfully on the eight puzzle, it exhausts available memory in a matter

of minutes on the fifteen puzzle.

**Best First Search**

Best-first search is a search algorithm, which explores a graph by expanding

the most promising node chosen according to a specified rule. Judea Pearl

described best-first search as estimating the promise of node n by a “heuristic

evaluation function f(n) which, in general, may depend on the description of

n, the description of the goal, the information gathered by the search up to

that point, and most important, on any extra knowledge about the problem

domain.”

Some authors have used “best-first search” to refer specifically to a search

with a heuristic that attempts to predict how close the end of a path is to a

solution, so that paths, which are judged closer to a solution, are extended first.

This specific type of search is called greedy best-first search. Efficient selection

of the current best candidate for extension is typically implemented using a

priority queue.

The end result is that best-first search will visit what it thinks are the most

promising cells first, which gives best-first some of the nice properties of both

BFS and DFS. However, this leaves best-first search vulnerable to bad heuristics,

or certain types of mazes which exploit weaknesses of certain heuristics.1.4

**Source Code A\***

import java.util.Map;

java.util.Map.Entry;

java.util.Scanner;

java.util.TreeMap;

public class BFSfinal {

//graph, heuristic, parent arrays store values starting at index 1

java.util.Map;

java.util.Scanner;

java.util.TreeMap;

public class Astar {

Scanner sc = new Scanner(System.in);

int nodes,start,goal,closedCount=0;

int graph[][] = new int[100][100];

int heuristic[] = new int[100];

int closed[] = new int[10];

int parent[] = new int[10];

int distancearray[] = new int[10];

TreeMap<Integer, Integer> openList= new TreeMap<>();

void input(){

System.out.println("Enter the number of elements of the graph");

nodes = sc.nextInt();

System.out.println("Enter the graph adjacency values");

for(int i=1;i<=nodes;i++){

for(int j=1;j<=nodes;j++){

graph[i][j] = sc.nextInt();

}

}

System.out.println("Enter the input graph elements heuristic

values");

for(int i=1;i<=nodes;i++){

heuristic[i] = sc.nextInt();

}

System.out.println("Enter the start state and the goal state");

start = sc.nextInt();

goal = sc.nextInt();

parent[start] = -999;

}

boolean closedListCheck(int key){

for(int i=0;i<closedCount;i++){if(closed[i]==key){

return true;

}

}

return false;

}

boolean openListCheck(int key){

for(Map.Entry<Integer,Integer> entry:openList.entrySet()){

if(entry.getKey()==key){

return true;

}

}

return false;

}

int returnTheSmallestValueKey(){

Map.Entry<Integer,Integer> minValue = openList.firstEntry();

for(Map.Entry<Integer,Integer> entry: openList.entrySet()){

if(minValue.getValue() > entry.getValue()){

minValue=entry ;

}

}

return minValue.getKey();

}

void processAstar(){

int distance=0;

distancearray[start] =distance;

openList.put(start, heuristic[start]);

int currentNodeKey = start;

while(!openList.isEmpty()){

if(currentNodeKey == goal){

closed[closedCount] = currentNodeKey;

distancearray[currentNodeKey] =

distancearray[currentNodeKey];

closedCount++;

break;

}else{

for(int j=1;j<=nodes;j++){

if(graph[currentNodeKey][j]!=-1){

if(!closedListCheck(j)){

if(!openListCheck(j)){

openList.put(j,heuristic[j]+graph[currentNodeKey][j]+distancearray[currenthe current

cost of reaching the node is less than what

is previously present \*/

// Gives the value of the node with key = j

int value = openList.get(j);

if(value >

heuristic[j]+graph[currentNodeKey][j]+distancearray[currentNodeKey]){

parent[j] = currentNodeKey;

openList.replace(j, value,

heuristic[j]+graph[currentNodeKey][j]+distancearray[currentNodeKe

distancearray[j] =

graph[currentNodeKey][j]+distancearray[currentNodeKey];

}

}

}

}

}

}

closed[closedCount] = currentNodeKey;

closedCount++;

openList.remove(currentNodeKey);

currentNodeKey = returnTheSmallestValueKey();

}

}

void printClosedList(){

System.out.println("The nodes selected from BFS are as follows:

");

for(int i=0;i<closedCount;i++){

System.out.print(closed[i]+" ");

}

System.out.println("\nThe actual path from the start to the goal

state is as follows:");

int currentId = closed[closedCount-1];

while(true){

System.out.print(currentId+"<= ");

currentId = parent[currentId];

if(currentId == -999){break;}

}

System.out.println("The distances are as follows: ");

for(int i=1;i<8;i++){

System.out.print(distancearray[i]+" ");

}

}

public static void main(String args[]){

Astar astar = new Astar();

astar.input();

astar.processAstar();

astar.printClosedList();}

}

**BFS**

import java.io.\*;

import java.util.\*;

class bfs

{

int nodes,source,dest,hue[],mat[][];

ArrayList<Integer> open=new ArrayList<Integer>();

ArrayList<Integer> close=new ArrayList<Integer>();

LinkedList<Integer> std =new LinkedList<Integer>();

HashMap<Integer,Integer> familylist=new HashMap<Integer,Integer>();

public void accept()

{

Scanner sc=new Scanner(System.in);

System.out.println("Enter the number of nodes\n==>>");

nodes=sc.nextInt();

System.out.println("Enter the source\n==>>");

source=sc.nextInt();

System.out.println("Enter the destination==>>");

dest=sc.nextInt();

System.out.println("Enter the huerisitic values of each

node==>>");

hue= new int[nodes];

for(int i=0;i<nodes;i++)

{

hue[i]=sc.nextInt();

}

mat=new int[nodes][nodes];

System.out.println("Enter the adjacency matrix of the

graph/tree");

for(int i=0;i<nodes;i++)

{

for(int j=0;j<nodes;j++)

{

mat[i][j]=0;

}

System.out.println("Enter the number of adjacent values to

Row "+(i+1)+"\n==>>");

int val=sc.nextInt();

System.out.println("Enter the adjacent values to Row

"+(i+1)+"\n==>>");

for(int r=0;r<val;r++)

{

int k=sc.nextInt();

mat[i][k-1]=1;}

}

}

public void put(int node)

{

for(int i=0;i<nodes;i++)

{

if(mat[node-1][i]==1&&!(open.contains(i+1))&&!(close.contains(i+1)))

{

open.add(i+1);

familylist.put(i+1,node);

}

}

}

public int gets(int s)

{

return hue[s-1];

}

public void sort()

{

for(int i=0;i<open.size()-1;i++)

{

for(int j=0;j<open.size()-1;j++)

{

if(gets(open.get(j))>gets(open.get(j+1)))

{

int c=open.get(j);

open.remove(j);

open.add(j+1,c);

}

}

}

}

public void path()

{

open.add(source);

while(!(open.get(0)==dest))

{

int node=open.get(0);

put(node);

open.remove(0);

close.add(node);

sort();

}

close.add(open.get(0));

open.remove(0);

System.out.println("\n\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ANSWER\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n\n\n");

System.out.println("The elements of the closed path are\n==>>");

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

System.out.print(close.get(i)+" ");

}

System.out.println("\n\nThe elements of the open path

are\n==>>");

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

{

System.out.print(open.get(i)+" ");

}

System.out.println("\n\nThe path from source to destination is

==>\n");

int key =dest;

std.addFirst(dest);

while(!(key==source))

{

std.addFirst(familylist.get(key));

key=familylist.get(key);

}

for(int i=0;i<std.size()-1;i++)

{

System.out.print(std.get(i)+"-->");

}

System.out.print(std.get(std.size()-1));

}

public static void main(String args[])

{

bfs obj =new bfs();

obj.accept();

obj.path();

}

}

**Output**

Enter

7

Enter

-1 13

13 -1

17 -1

-1 11

-1 27

-1 -1

0 0 0

Enter

50 40

the number of elements of the graph

the graph adjacency values

17 -1 -1 -1 -1

-1 11 27 -1 -1

-1 -1 -1 19 -1

-1 -1 13 18 -1

-1 13 -1 -1 26

19 18 -1 -1 25

0 0 0 0

the input graph elements heuristic values

50 30 30 30 0Enter the start state and the goal state

1 7

The nodes selected from A\* are as follows:

1 2 4 3 6 7

The actual path from the start to the goal state is as follows:

7<= 6<= 3<= 1<=

The distances are as follows:

0 13 17 24 37 36 61

Output

Best first Search

Enter the number of vertex

4

Enter Adjacency matrix

0

1

0

1

1

0

1

0

0

1

0

1

1

0

1

0

Adjacency Matrix :

0101

1010

0101

1010

Enter Heuristic Value for each vertex :

3

1)3

4

2)4

5

3)52

4)2

1)3

2)4

3)5

4)2

Enter the Source vertex

2

Graph traversal:

2

1

4

3

**Conclusion**

In this assignment we observed the facets of AI search techniques, particularly

the A\* technique and BFS.That is the heuristic search techniques are researched

on.