Artificial Intelligence Assignment

Ogesoka Tasie
oot4@aber.ac.uk
Department of Computer Science
Aberystwyth University - CS26520
(made in LaTeX)

March 9, 2017

Abstract

A report on problem definitions with searches and the application of heuristics.

1 Part 1: Problem Definition and Search

Introduction There are three angels, three devils and a river. The angels and devils are together on one side of the river and are trying to cross to the other side. There is a boat present which can be used to go back and forth across the river but, can only hold up to two persons at any time and must have at least one person to operate the boat. No group of angels can be left outnumbered by a group of devils however, a group of angels and no devils or a group of devils and no angels is permitted.

1.1

Start State In this problem, there can only be one possible staring position. Considering that the side of the river the angels and devils need to get to is the **goal**, we can say that all devils and angels, and the boat can have their initial positions at the opposite side to the goal. This would be the root node. Below is a graphical representation.

Figure 1: *Initial State*

Goal State The goal state would only be satisfied if, and only if, all devils and angels are together on the **goal** side of the river. In this problem we know what the goal state we are trying to achieve is. Below is a graphical representation.

Figure 2: *Goal State*

State Space The state space can be simply defined as all the possible and legal parts of the search. In this problem, all the legal combinations of angels and devils on either side of the river, produced by operations, make up the state space.

Operators Operators allow us to get from one part of the search to another. For this problem an operator would be a combination of angels and devils on the boat, or a single angel or devil, example $Move\{Angel - Devil\}$.

Path Cost The path cost is a measure of what it takes to get from one part of the search space to another. In heuristics, we also use the distance or weighted value between the current state to the goal state as a path cost. We can evaluate the path cost as being the number of moves required to reach the next state. The path cost in this case would then be **1** for each move made across the river, meaning $Move\{Angel - Devil\} = 1$.

Goal Function The goal function informs the algorithm that the goal state has been reached. The goal function would simple be whether the goal state is present in the state array or a boolean that turns true when the goal state is equivalent to the current state in the algorithm.

1.2

Set of operators If we had a set of move operations **M** with combinations of angels **A** and devils **D** then a complete set of operators would be: $\mathbf{M}\{A, D, AD, AA, DD\}$. Below is the first two levels of the search tree for this problem.

Used Operators The operators used between the root node to level one in our search tree are: DD, AD, D. The operator used between level 1 and level 2 is simply A or D.

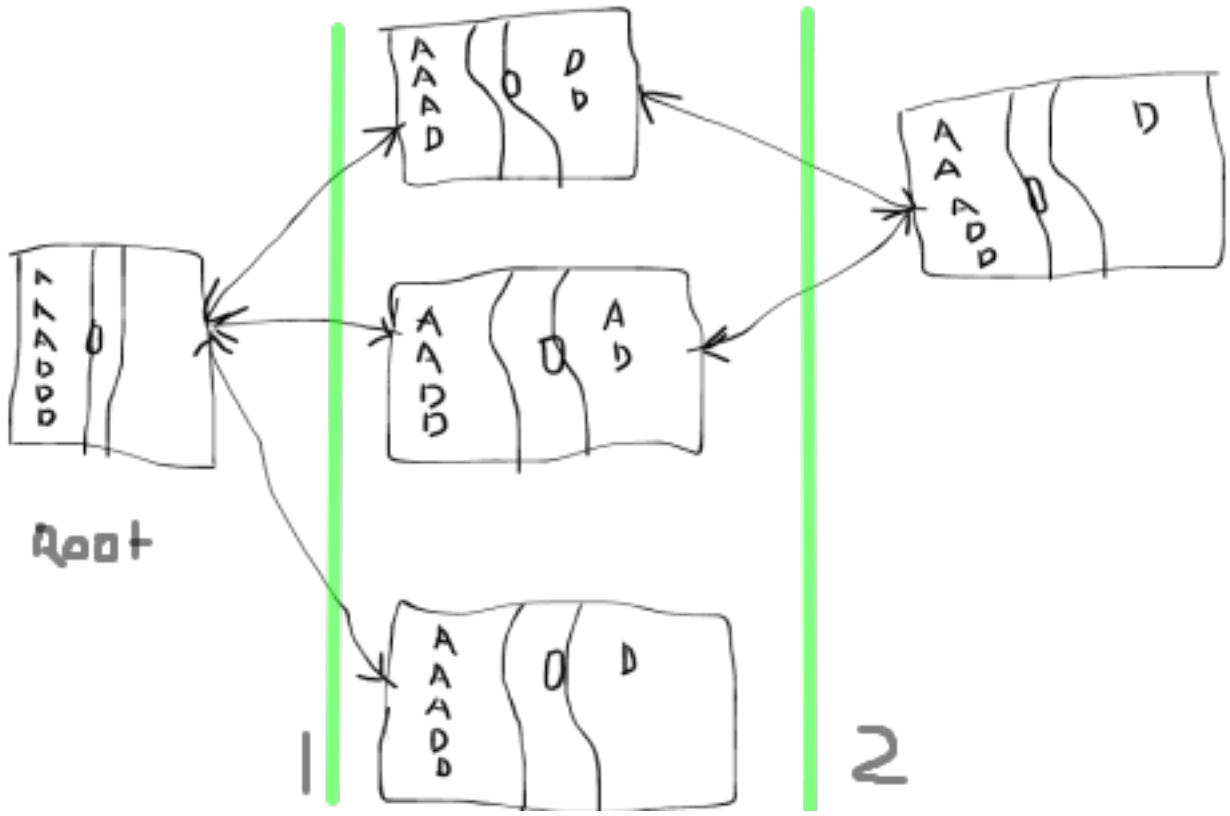

Figure 3: *Search Tree*

1.3

Bidirectional Searches From the analysis of the problem above I believe that a Bidirectional search would be best suited. Bidirectional searches start from the initial and goal state and work towards each other until they meet. One of their requirements is that the goal state is known, and seeing as we know what our goal state should look like, this criteria is fulfilled. Heuristics work better in larger complex problems and I have overlooked them simply because the problem at hand does not have varying path costs or a large state space to work with. Using a heuristic would be over doing it and unnecessary. We simply require an algorithm to produce our known goal state and produce a solution. There are not many solutions in this problem as discussed above which reduces the neccessity for a heuristic.

Properties of the State Tree By examination, the goal state is in actual fact identical to the initial state which tells us that within the state space, certain states will mirror each other, if not all. This guarantees that our two searches will meet halfway within the search tree.

Time Complexity Bidirectional searches are fast and have a time complexity of $O(b^{d/2})$. This reduction is due to the fact that the searches only need to go halfway through the search tree. They also take up less memory.

Space Complexity One draw back with Bidirectional searches are that they have increasing space complexity of $O(b^{d/2})$. This is because at least one of the searches must be breath-first to identify common states and must store the whole tree for one of the search directions. A hash table is used to compare new nodes with generated nodes from the other tree. Because our problem is not overly complex however, this would be tolerable.

Optimality Bidirectional searches are only optimal and complete if it is able to search backwards. Our problem is identical on either the initial side of the search tree and the goal side of the search tree meaning we are highly likely to find an optimal solution because we can search backwards. One issue that could be encountered is that the implementation of the algorithm must have additional logic to decide which of the search trees to expand during processing this however is not an issue because the two halves in this problems state tree are identical in depth and branching factor.

Comparisons Bidirectional searches in comparison to other uninformed algorithms work best when you have small symmetrical state trees and, identical branching factors and depths. These are some of the main points to using a bidirectional search for this problem as I believe.

2 Part 2: Solving Mazes with A* and Heuristics

2.1

Manhattan Distance The Manhattan Distance heuristic works by finding the difference in the distance between the goal and current node, along the x-axis and y-axis. The sum of the value is the total distance between the two nodes. Using coordinate data, if (x_g, y_g) are the goal nodes coordinates and (x_c, y_c) are the current nodes coordinates then, the manhattan distance $dist$ can be represented: $x = x_c - x_g, y = y_c - y_g, dist = x + y$

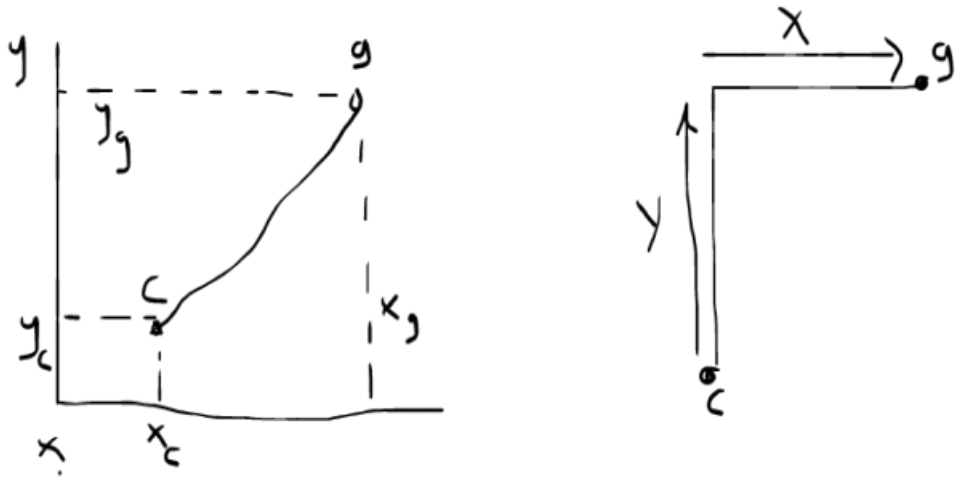

Figure 4: *Manhattan Distance*

Arc Distance The Arc Distance heuristic works by finding the curved distance between the current node and the goal node with use of the Euclidean Distance. The Euclidean Distance finds the distance between the goal node and the current node by using pythagoras. the difference in the distance between the goal and current node along the axes is found but, rather than finding their sum, we square the x difference and y distance, add these two numbers together and find the square root of this value which becomes the hypotenuse.

The Arc Distance uses the hypotensuse as the distance between two points on a perfect circle, the point where the goal is and the point where the current node is. The length of an arc is $\theta/360 \cdot 2\pi r$, seeing as the hypotenuse is twice the radius and θ is 180° along the hypotenuse, we can simplify this equation to: $(\pi h)/2)$ or $(\pi h) \cdot 0.5rad$

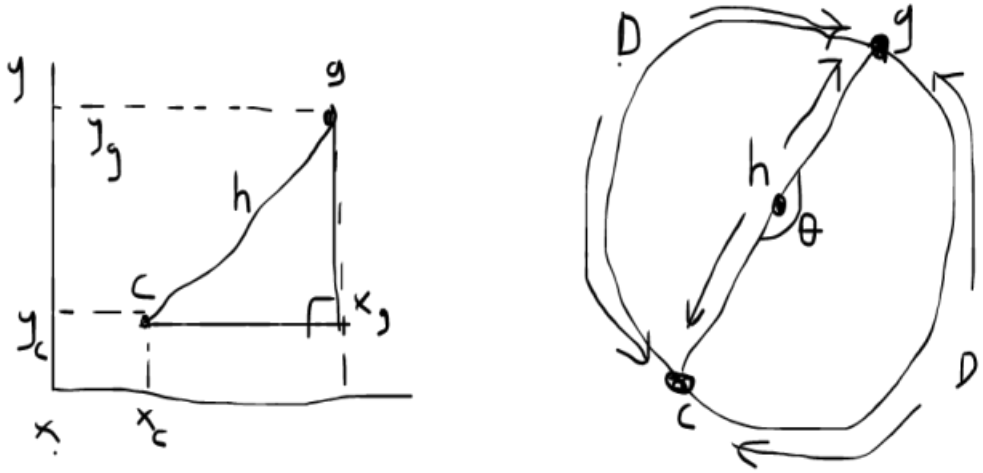

Figure 5: *Arc Distance*

2.2

Informdness and Admissibility The Manhattan Distance is an admissible heuristic which means it never over estimates how far the goal is from the current node, in comparison to the Arc Distance which is non-admissible. This is due to their different algorithimic designs, because the Arc Distance finds a curve, it assumes that it needs to move through a longer route to get to the goal unlike the Manhattan Distance.

To really compare these two heuristics based on how they reach the goal, we need to consider their informdness. Given two heuristic functions $h_1(n)$ and $h_2(n)$, $h_2(n)$ is more informed if $h_1(n) \leq h_2(n)$ and $h_2(n)$ is better for search. However, because the Arc Distance is an overestimating algorithm, it is already considered better than the Manhattan Distance heuristic because it always overestimates the path cost.

2.3

Testing Included in the appendix are a few of the test results for the algorithms Manhattan Distance, Arc Distance and Breadth-First.

Manhattan The Manhattan Distance struggles to produce good solutions especially when confronted with larger more complex mazes though can come across an optimal solution. It actively seeks good solutions that solve the problem even if they are not optimal. Though it does not overestimate the path cost it is more informed than BFS and produces better solutions. It appears to make decisions on whether to continue searching the curent depth or whether to move on to the next depth rather than prioritizing the breadth of the tree. This also could explain why it may not find the best soution because it is not as well informed as the Arc Distance heuristic.

Arc Distance The Arc Distance is very good at finding solutions and even optimal solutions. We can see from the results that the branching factor is low so, it does not search the majority of nodes at a depth level. This is probably due to its non-admissible nature. It does not traverse too far down the depth of the tree and as a result is quick, uses less memory and can handle larger more complex problems in comparison to the Manhattan Distance beause it does not store as many nodes as the other two searches. The solution lengths are relatively short and according to the results it can consistently find good solutions even when presented with a demanding maze.

Breadth-First Breadth-First has an average ability in finding good solutions. The smaller the problem the faster it works, the larger the problem the slower it becomes. It does not try to find an optimal solution, rather focuses on solving the problem as it has a relatively high depth and node count. It tends to search the majority part of the tree which results in a high task load for the system. It is very poor at solving complex problems because it expands an immense number of nodes, seeing as it is not trying to predict where the solution lies like Arc distance and Manhattan Distance. It is the slowest and the most spacious amongst the three algorithms.

3 Appendix

```java
package maze.ai.heuristics;

import maze.ai.core.BestFirstHeuristic;
import maze.core.MazeExplorer;

//Using the current node and the goal node, the manhattan distance finds the sum
//of the distance between the two points along the axes X and Y.
//
//ie the sum of, the difference between the x coordinate of the current and goal node, and, the y coordinate
//of the current and goal node.
public class ManhattanDistance implements BestFirstHeuristic<MazeExplorer> {

    @Override
    public int getHeuristic(MazeExplorer node, MazeExplorer goal) {

        double x = node.getLocation().X() - goal.getLocation().X();
        double y = node.getLocation().Y() - goal.getLocation().Y();

        return (int) (x + y);
    }
}
```

Figure 6: *Manhattan Distance Code*

```java
package maze.ai.heuristics;

import maze.ai.core.BestFirstHeuristic;
import maze.core.MazeExplorer;

public class ArcDistance implements BestFirstHeuristic<MazeExplorer> {

    // Uses the arc length between the goal node and the current node.
    // The Euclidean Distance is first calculated, which tells us the
    //non-obstructed distance between the current node and goal node.
    //We then imagine a circle running through each point and find the length
    //of the arc between the two points.
    //The Arc Length uses the angle between the two points divided by 360.
    //Because we always find the distance between x and y between the current and goal nodes,
    //and then apply the pythagorem theorem, we always end up with a hypoteneuse.
    //If we draw a circle between the ends of the hypotenuse we get double the radius,
    //running in a straight line meaning the angle between the two of them is 180.
    //the formula for finding the arc length is always  theta/360 multiplied by 2*pi*radius.
    //2*radius is the same as hypoteneuse h and 180/360 is 0.5 therefore
    // we can use the formula 0.5*3.14*sqrt(x^2+ y^2). This will find the arc distance between
    //the current node and the goal node.
    @Override
    public int getHeuristic(MazeExplorer node, MazeExplorer goal) {

        double x = node.getLocation().X() - goal.getLocation().X();
        double y = node.getLocation().Y() - goal.getLocation().Y();

        double z = Math.sqrt(x * x + y * y);
        return (int) (0.5 * Math.PI * z);
    }
}
```

Figure 7: *Arc Distance Code*

Nodes	Depth	Branching Factor(b*)	Solution Length	Test Description
197	64	1.03	65	10*10, 100% Perfection, 0 Treasures
287	18	1.25	19	10*10, 50% Perfection, 0 Treasures
324	18	1.26	19	10*10, 0% Perfection, 0 Treasures
797	128	1.02	129	20*20, 100% Perfection, 0 Treasures
1228	38	1.14	39	20*20, 50% Perfection, 0 Treasures
1408	38	1.14	39	20*20, 0% Perfection, 0 Treasures
1797	484	1	485	30*30, 100% Perfection, 0 Treasures
2705	58	1.09	59	30*30, 50% Perfection, 0 Treasures
3175	58	1.1	59	30*30, 0% Perfection, 0 Treasures
797	82	1.05	77	10*10, 100% Perfection, 5 Treasures
279615	46	1.26	37	10*10, 50% Perfection, 10 Treasures
-	-	-	-	10*10, 0% Perfection, 15 Treasures
16891	390	1.01	391	20*20, 100% Perfection, 5 Treasures
917350	83	1.15	69	20*20, 50% Perfection, 10 Treasures
18173	525	1	499	30*30, 100% Perfection, 5 Treasures
2816289	130	1.1	127	30*30, 50% Perfecttion, 10 Treasures

ManhattanDistance

Figure 8: *Manhattan Distance Tests*

Nodes	Depth	Branching Factor(b*)	Solution Length	Test Description
197	60	1.03	61	10*10, 100% Perfection, 0 Treasures
138	18	1.19	19	10*10, 50% Perfection, 0 Treasures
125	18	1.18	19	10*10, 0% Perfection, 0 Treasures
748	134	1.02	135	20*20, 100% Perfection, 0 Treasures
242	38	1.08	39	20*20, 50% Perfection, 0 Treasures
364	38	1.09	39	20*20, 0% Perfection, 0 Treasures
1793	476	1	477	30*30, 100% Perfection, 0 Treasures
406	58	1.05	59	30*30, 50% Perfection, 0 Treasures
578	58	1.06	59	30*30, 0% Perfection, 0 Treasures
2070	84	1.05	85	10*10, 100% Perfection, 5 Treasures
165696	49	1.23	49	10*10, 50% Perfection, 10 Treasures
3647794	39	1.42	39	10*10, 0% Perfection, 15 Treasures
6823	214	1.02	215	20*20, 100% Perfection, 5 Treasures
556448	70	1.17	71	20*20, 50% Perfection, 10 Treasures
7730	414	1.01	415	30*30, 100% Perfection, 5 Treasures
1744612	118	1.1	119	30*30, 50% Perfecttion, 10 Treasures

Arc Distance

Figure 9: *Arc Distance Tests*

Nodes	Depth	Branching Factor(b*)	Solution Length	Test Description
197	163	1.09	29	10*10, 100% Perfection, 0 Treasures
284	18	1.25	19	10*10, 50% Perfection, 0 Treasures
336	18	1.26	19	10*10, 0% Perfection, 0 Treasures
797	222	1	223	20*20, 100% Perfection, 0 Treasures
1238	38	1.14	39	20*20, 50% Perfection, 0 Treasures
1454	38	1.14	39	20*20, 0% Perfection, 0 Treasures
1507	213	1.01	213	30*30, 100% Perfection, 0 Treasures
2885	58	1.1	59	30*30, 50% Perfection, 0 Treasures
3278	58	1.1	59	30*30, 0% Perfection, 0 Treasures
1598	79	1.05	79	10*10, 100% Perfection, 5 Treasures
333998	37	1.36	37	10*10, 50% Perfection, 10 Treasures
-	-	-	-	10*10, 0% Perfection, 15 Treasures
9942	269	1.01	269	20*20, 100% Perfection, 5 Treasures
1284030	87	1.14	87	20*20, 50% Perfection, 10 Treasures
26079	612	1	613	30*30, 100% Perfection, 5 Treasures
91768	95	1.09	95	30*30, 50% Perfecttion, 10 Treasures

Breadth-First

Figure 10: *Breadth-First Tests*