COMP 472 Project 1 – Report

Reynald Servera

40043437 – reynald.servera@hotmail.com

1. Introduction

1.1. Context

For my first project in COMP-472 (Introduction to AI), I am given the task to write a program that will solve the Indonesian Dot Puzzle with the help of State Space Representation. Three algorithms in particular must be implemented as the program's search functions: Depth-First Search (DFS), Best-First Search (BFS) and A*. As DFS is an uninformed search, a maximum depth (max_d) must be set in order to limit the search space to a reasonable set. Similarly, for the informed searches, a maximum search path length (max_l) must be placed for these algorithms to product results in a reasonable amount of time. A text file will serve as an input file that will provide all the necessary information for the puzzles. Output files will be produced by the program that will contain the solutions (if any) and search paths for each algorithm. This project is to be coded in Python 3.7 and must be compatible with Concordia's laboratory computers. Therefore, the program must be able to run through the Anaconda command prompt. To develop my script, I decided to use Sublime Text 3 as my text editor. Test input files are to be created with Notepad in the appropriate format provided in the project handout. Input validation in the script is not necessary as all input files are assumed to be correctly formatted. I have done the whole project alone and do not belong to a team.

1.2. Technical Details

My program consists of two classes. First off, I created the Vertex class that will serve as nodes in the State Space. Each vertex will have its own index, list of neighbors, state (open or closed), puzzle (an n x n list, where n is the dimension of the puzzle), depth level, list of heuristic information (BFS_info), parent node and last move. Next, I also created the Graph class that will serve as a network of nodes (a tree in this case). A graph will have a dictionary of vertices (identified by their index), a current depth level, a boolean variable that will indicate if a solution has been found so far, a list that will contain the sequence of moves towards the solution, a list of the currently open nodes, a list of the currently closed nodes and the current search length. [1]

2. Heuristics & their Difficulties

2.1. Most Amount of White Tokens

The main heuristic function I thought of was to prioritize boards with the most amount of white tokens. The idea behind this heuristic was simple. The more white tokens were present on the board, the less black tokens I would have to deal with. In my opinion, this happens to be the strong point of this heuristic. As the search goes on, this heuristic will allow the search algorithm to naturally select nodes with smaller amounts of black tokens which should theoretically bring you closer to the goal state of all white tokens. The main weakness behind this heuristic is that it fails to take into consideration the fact that every move does not solely affect the token it pivots, but also the tokens around it. Hence, the nature of the game usually requires the player to value token positions rather than the sheer number of each token.

To implement this idea, I had to find a way to compare the value of each vertex with each other. Since a vertex already held a puzzle state as one of its class variables, I knew that important information like the number of white tokens and the first occurrence of a white token could be derived from it. I created another class variable for the vertex class, BFS_info, which was basically a list of two integers: one to store the number of white tokens and another to store the index of the first occurrence of a white token in the puzzle. Now that every vertex had the ability to carry these vital pieces of information, quantifying the value of a vertex was easy. To do so, I had to read the puzzle state contained in the vertex and count number of white tokens as well as keep track of the first occurrence of a white token. Once this was done, the next thing to do was to design a mechanism that would prioritize vertices that were highly classed according to this heuristic. To do so, I added the class variable, openList, to the Graph class. This served as a container for all the nodes that were not visited in the graph yet. It is important to note that the most important thing about this list is not necessarily the list itself, but the way it is filled. Whenever a new node was introduced into a graph (created after a move is made on a previous board), a vertex with a new board state is created and the heuristic function is immediately ran in order to calculate its heuristic value (BFS_info). Once this value is calculated, the vertex is compared with all the other vertices in the open list and is placed accordingly depending on its heuristic value. The bigger the amount of white tokens in the puzzle, the closer to the start of the list that vertex becomes. In case of a tie, its position is further determined by the earliest occurrence of a white token (the earlier it is, the closer it is to the start of the list). Finally, to know which vertex to search next, the script I designed simply always picks the first vertex in the open list. Once it is done searching that node, it is popped from the open list and placed into a closed list. If a solution is not found before the maximum search path length, the search stops and the puzzle is declared to have no solution.

2.2. Most Amount of Black Tokens

Initially, the first heuristic I thought about after reading the rules of the Indonesian Dot Puzzle was to target boards with the biggest number of black tokens. The main idea behind this was to eliminate the most black tokens I could with every single move. After running a couple of test cases, I quickly noticed that the results produced for every puzzle were nothing short from disappointing. I took a step back to re-evaluate the logic behind my algorithm and realized that the heuristic was counterintuitive. Despite having the strength of clearing large amounts of black tokens per move, the algorithm also inherited a huge weakness. The heuristic of prioritizing boards with the biggest amount of black tokens never made any progress towards the goal state. While the goal state was to fill your board with white tokens, this heuristic constantly prioritized moves that would fill the board up with black tokens. It was at this moment that I realized that the heuristic that I was looking for was the opposite of the one I originally thought of.

The implementation of this heuristic was basically the same as the first one I mentioned. Instead of counting the number of black tokens on a given board state, I would simply have to count the number of white tokens instead.

3. Results

3.1. Test 1 (Main Heuristic – Most White Tokens)

Input: 5 4 200 0000000000000000111010101

Results:

DFS: With max_d set to 4, my DFS algorithm was able to find a 3-move solution to this puzzle after searching 14270 nodes.

BFS (most white tokens): With max_l set to 200, my BFS algorithm was not able to find a solution to this puzzle.

A* (most white tokens): With max_l set to 200, my A* algorithm was able to find a 3-move solution to this puzzle after searching 9 nodes.

3.2. Test 2 (Main Heuristic – Most White Tokens)

Input: 3 5 200 110101011

Results:

DFS: With max_d set to 5, my DFS algorithm was able to find a 5-move solution to this puzzle after searching 13 nodes.

BFS (most white tokens): With max_l set to 200, my BFS algorithm was able to find a 2-move solution to this puzzle after searching 3 nodes.

A* (most white tokens): With max_l set to 200, my A* algorithm was able to find a 2-move solution to this puzzle after searching 3 nodes.

3.3. Test 3 (Main Heuristic – Most White Tokens)

Input: 3 5 2000 111101011

Results:

DFS: With max_d set to 5, my DFS algorithm was able to find a 3-move solution to this puzzle after searching 1996 nodes.

BFS (most white tokens): With max_l set to 200, my BFS algorithm was not able to find a solution to this puzzle.

A* (most white tokens): With max_l set to 200, my A* algorithm was able to find a 3-move solution to this puzzle after searching 61 nodes.

3.4. Test 4 (Different Heuristic – Most Black Tokens)

Input: 3 5 2000 111101011

Results:

<u>DFS</u>: With max_d set to 5, my DFS algorithm was not able to find a solution to this puzzle after searching 1996 nodes.

<u>BFS (most white tokens)</u>: With max_l set to 2000, my BFS algorithm was not able to find a solution to this puzzle.

<u>A* (most white tokens)</u>: With max_l set to 2000, my A* algorithm was not able to find a solution to this puzzle.

3.5. Test 5 (Main Heuristic – Most White Tokens)

Input: 2 5 2000 1111

Results:

<u>DFS</u>: With max_d set to 5, my DFS algorithm was able to find a 4-move solution to this puzzle after searching 39 nodes.

<u>BFS (most white tokens)</u>: With max_l set to 2000, my BFS algorithm was not able to find a solution to this puzzle.

<u>A* (most white tokens)</u>: With max_l set to 2000, my A* algorithm was able to find a 4-move solution to this puzzle after searching 63 nodes.

3.6. Test 6 (Different Heuristic – Most Black Tokens)

Input: 2 5 2000 1111

Results:

<u>DFS</u>: With max_d set to 5, my DFS algorithm was able to find a 4-move solution to this puzzle after searching 39 nodes.

<u>BFS (most black tokens)</u>: With max_l set to 2000, my BFS algorithm was able to find a 4-move solution to this puzzle after searching 1671 nodes.

<u>A* (most black tokens)</u>: With max_l set to 2000, my A* algorithm was able to find a 4-move solution to this puzzle after searching 702 nodes.

4. Analysis

4.1. Comparing Heuristics

H0 = Heuristic Zero (prioritizes the board with the most white tokens)

H1 = Heuristic Zero (prioritizes the board with the most black tokens)

As mentioned before, H1 was the heuristic I first thought of when tackling this project. Although it may have initially sounded intuitive, its results were disappointing enough to make me realize that what I was looking for was the complete opposite heuristic.

If we look at test 3 and test 4, the same exact puzzle was tested with the two different heuristics, H0 and H1. Looking at the results from the BFS test, we can see that both algorithms were not able to find a solution within the given search limit of 2000. On the other hand, when we look at the results from the A* test, we can see that H0 was able to find a solution after only search 61 nodes while H1 was not able to find a solution at all after searching 2000 nodes.

If we look at test 5 and test 6, the same puzzle is once again tested with the two different heuristics. Weirdly enough, H0 was not able to find a solution while using the BFS algorithm yet H1 was able to find one after searching 1671 nodes. However, if we look at the results from the A* algorithm, we notice how H0 finds a solution after searching 63 nodes while H1 finds a solution after searching 702 nodes.

To sum it up, it seems like H0 seems to generally be the better heuristic (as expected), but we should not discount the fact that H1 may sometimes be capable of finding solutions to puzzles that even H0 cannot solve at specific search lengths. This can be seen in test 5 and test 6.

4.2. Comparing Search Algorithms

DFS is naturally a good algorithm when trying to find out if a puzzle can be solved through a maximum of (max_d – 1) moves. Obviously, its biggest downside is that it is an uninformed search which means that it usually finds a solution only after exhausting the whole state space. As seen in most test cases (namely test 1, test 2 and test 3), DFS tends to consistently fail in terms of speed when compared to informed searches such as BFS and A*. Although, if we look at the results generated from test 5, we notice that there may be some situations where the DFS algorithm may find a solution before an informed search algorithm. This seems to be the case if the solution is made solely from a sequence of low index moves (such as indexes 1, 2 and 3 in a 3x3 puzzle) as they are tested earlier in the DFS algorithm.

BFS is an informed search, meaning that it usually makes much smarter decisions than an uninformed search algorithm such as DFS. It is important to note that this is only consistent if the algorithm is paired with a good heuristic function that will quantify the efficiency of every move. If we look at the results from test 2, we can see how the BFS algorithm coupled with H0 was able to find a better (shorter) solution to the one found by DFS. In addition, the solution found by BFS was calculated after searching less nodes than its competitor (3 nodes vs 13 nodes).

Finally, we have A*. A* is very similar to BFS except for one thing. It takes into consideration the depth of the current node. This characteristic is extremely important as allows each node to be properly quantified regarding where they reside in the state space. In BFS, it did not matter if a specific node took 1000 moves to get to. If the heuristic function scored it highly, they it would be prioritized in the search. The A* algorithm fixes that issue by (in my case) subtracting the depth level of the node from its heuristic value. As a result, even if a node scored 100 heuristic units higher than the others, it would not matter if it took that node 100 moves to get to. In most of the test cases, A* seemed to have found the best solution in the quickest amount of time compared to DFS and BFS. Although, like all other algorithms, there may be some exceptions (such as test 5) where even DFS can prove to be more efficient than A*.

5. References

1. Joe, J., *Python: DFS Depth First Search*, https://www.youtube.com/watch?v=QVcsSaGeSH0, last accessed 2020/02/29
2. Russell & Norvig. (2020) *Artificial Intelligence: State Space Search for Game Playing* [PowerPoint slides]. Moodle. https://moodle.concordia.ca/moodle/pluginfile.php/3828311/mod_label/intro/472-3-adversarial-Winter2020-annotated.pdf, last accessed 2020/02/29