Solving the 8-Puzzle

Alex Hazan and **Brandon Ra**
{alhazan,brra}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

The advantages of the A* algorithm can be seen in its application to a conventional 8-Puzzle. To reproduce Russell and Norvig's research, A* algorithm was implemented with two heuristics previously developed by Russell and Norvig in 2003 as its basis. Heuristic 1 counts the misplaced tiles, and heuristic 2 returns the sum of the Manhattan Distances of the puzzle tiles relative to their goal position. Randomized sample problems of varying solutions depths were solved with A*. For each problem sample, the number of nodes generated and the effective branching factor was recorded. The results reveal that heuristic 2 offers a faster and more efficient solution than heuristic 1 at every solution depth.

1 Introduction

Search algorithms are inherent to everyone's daily lives. Through recent advancements in computer science and technology, people have begun to optimize search problems and research its application to real life. For instance in geographic information systems, search algorithms are utilized for travel-routing and navigating (Zeng and Church 2009).

This paper similarly tackled the conventional 8-Puzzle problem in order to reproduce Russell and Norvig's research in 2003. We implemented an A* algorithm to solve the 8-Puzzle and investigated the effects of the heuristics underlying the program. The 8-Puzzle game is defined as a sliding puzzle that consist of a 3 x 3 board with randomly ordered numbers 1 through 8, with one empty tile. A*s objective is to determine the shortest solution path to reach a goal state, which would be given to the algorithm as a parameter along with its starting state. The A* was designed based on the two heuristic functions previously provided by Russell and Norvig in 2003 on solving the 8-Puzzle (Russell and Norvig 2003).

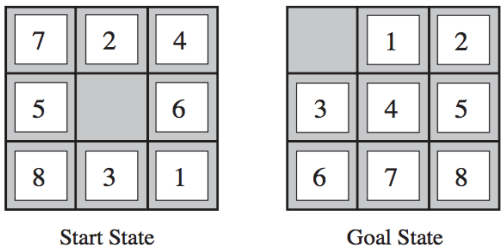

Figure 1: An example start state and goal state of a 8-Puzzle problem (Russell and Norvig 2003)

To examine the effectiveness of the heuristics, we analyzed h_1 and h_2's impacts on A*'s efficiency in deciphering the 8-Puzzle.

2 Background

Board Class

To organize our python implementation for a solution to the 8-Puzzle, we initiated a Board class to represent the state of the puzzle within a specific game. The state of the Board is stored in a *values* attribute as a list of integers, with 0 representing the empty space. To access a tile at a certain position in this list, simply multiply the row by the dimension of the board and then add the column value to give the position of the tile in the one dimensional array. The class has a *makeMove*() method for moving a tile on the board. To more easily make moves on a node, each board object keeps track of the row and the column position of the empty tile, in the attributes *eRow* and *eCol* respectively. The attribute *ePos* is calculated by utilizing the arithmetic method described above using *eRow* and *eCol*. The Board object also keeps track of the cost it took to get to that state in the attribute *pCost*, and the heuristic cost of a potential next move in attribute *hCost*. The total cost of the board that is used to determine how "fit" a Board is relative to its goal state, is held in the *tCost* attribute, which is computed by adding *pCost* and *hCost* together. The Board class also has an *isGoal*() function, which compares the values of the current board with the values of a given goal Board, returning True if the values match.

We overrode the hashing mechanism of our Board class, causing each Board object to be hashed by its values attribute. This way, we could easily look into a set or list and compare Board states with each other. We also reimplemented the equals operator for the Board class, again comparing the values attribute for each Board to see if they are equal. Our *makeMove*() function returns None when the program tries to make an illegal move, so our equals comparator for Board takes that into consideration, allowing for the comparison of a Board with a NoneType (which will result in False) and the comparison of two NoneTypes (which

will result in True). The class also has its own unique print function, which allows the user to print a Board in manner that is pleasing to the eye.

Code Setup

Since this project was implemented in Python, we utilized the language's various tools to help aid our computation. In the case that we needed to initialize a list of a certain size with uniform contents, such as in the beginning of our *makeMove*() function, we use Python's list comprehension. The completion of A* relies on the presence of a priority queue, which we obtained by using the PriorityQueue object from the Queue package. Whenever we needed to create random states or moves, we utilized the random package.

To aid with our experiments, we developed a *randomPuzzle*() function, which will generate a goal state from a given start Board that is reached by performing a specified number of moves. We also implemented a *getH*() function, which will return the heuristic value of a given board in one of two ways, depending on the user's request.

Heuristics

Our *getH*() function implements one of two heuristics, h_1 and h_2. h_1 is computed by counting how many tiles are out of position. h_2 is computed by computing the sum of Manhattan distances for each tile in the initial board relative to the goal state. Manhattan distance is defined as the sum of the vertical and horizontal distances between the two tiles.

3 Experiments

We started our experiments by implementing the A* using a priority queue. The A* function is supplied with the starting state, goal state, and option parameters. Including the states in the parameter, all nodes that A* traverse over are initialized as a Board class described previously. The option parameter specifies which heuristic will be applied to the A* function.

Each nodes that are to be investigated were stored in a priority queue. The value of the nodes $f(n)$ were calculated as the sum of the number of steps taken from origin to reach current node $g(n)$, and the heuristic value of current node $h(n)$.

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

The algorithm's main component is a loop that pops a board to be explored from the priority queue, and then checks if the node is the goal state. If yes, the solution cost is returned, whereas if false, all legal children of the node is generated. Child is defined as a state that can be reached after playing one legal move from its parent state. A separate set is maintained to record the nodes that were previously explored to prevent re-visiting nodes already analyzed.

Upon completion of A*, we delved into our primary goal of determining the impacts of the heuristics on A*'s efficiency. As a standard for comparison, the total number of nodes generated by A* to reach a solution and the resultant effective branching factor was used to determine the efficiency of the two heuristics. This is because adept algorithms would return lower values for both measurements.

We investigated 8-Puzzle problems with solution depth of even numbers between 2 to 24. 100 problems' data were collected for each solution depth, arriving at a grand total of 1200 sample data collected. To generate randomized sample problems, we began by creating random starting states. After, we executed a number of moves and the resulting board was used as the goal state to ensure that every problem produced will have a valid solution path. The number of moves made from starting state was $i + \frac{i}{2}$ where i varied over a range of even numbers from 24 to 2. $i + \frac{i}{2}$ is used because the goal board generated after 24 moves from an initial state does not guarantee a solution depth of 24, since the moves may have led to nodes that were previously visited, causing a small loop. Thus, this makes it very difficult for a board problem to have a solution depth equal to the number of moves it made. With the $\frac{i}{2}$ addition to the number of moves, it allows a higher chance for the targeted solution-depth-8-Puzzle to be generated. To further raise the chance of properly generating desired problems, we also had a separate set that recorded the nodes ventured through different moves when a random puzzle is being generated. Utilizing this set, we prevented the function from visiting a node twice, eliminating the chance of creating unnecessary loops. However, even with this addition, there is no guarantee that the generated problem's solution cost will equal the number of moves made. Thus, to even further ameliorate the runtime, we indexed the number of moves from largest to smallest. This way, while iterating over the largest move numbers, lower solution depth problem data will be filled along the way. This process may a to more than 100 problem samples to be generated for certain solution depths but we only consider the first 100 sample data for analysis.

Once all the data is collected, for each solution depth, we calculated the average number of nodes generated. With the averages, we determined the effective branching factor ($b*$) with the formula:

$$N + 1 = 1 + b + (b*)^2 + \ldots + (b*)^d$$

where N is number of nodes and d is solution depth.

4 Results

Depth	Nodes Generated		Effective Branching Factor	
	A*(h_1)	A*(h_2)	A*(h_1)	A*(h_2)
2	4	4	1.5616	1.5616
4	8	7	1.2975	1.2369
6	13	12	1.2260	1.2021
8	25	17	1.2517	1.1663
10	57	27	1.3049	1.1752
12	143	49	1.3556	1.2044
14	360	96	1.3918	1.2354
16	877	170	1.4150	1.2479
18	2131	333	1.4324	1.2675
20	5224	554	1.4513	1.2698
22	11988	1044	1.4535	1.2803
24	25987	1773	1.4552	1.2824

Table 1: The results for our Test of A* with two heuristics

Our results mirrored the trend shown in Russell and Norvig's analysis of A*, with the final data being shown in Table 1 (Russell and Norvig 2003). h_2 outperformed h_1 at every depth with regards to the number of nodes generated. The difference grew exponentially as the depth neared 24. h_2 also produced smaller effective branching factors at every level. Moreover, our implementation of Russell and Norvig's experiment produced better results than the original as both the number of nodes generated and effective branching factor undershot the original data at every level (Russell and Norvig 2003).

The search cost of h_2 being less than that of h_1 is reasonable as h_1 only counts the tiles misplaced. A tile that is one move away from its position in the goal state is treated as the same as a tile that is five moves away. However for h_2, it calculates the Manhattan distances of the tiles misplaced, resolving the issue. h_2 offers an extra dimension of accuracy as it reveals all the information h_1 gives while also giving insight to the each tile's distance to its final position. As $h_1(n) \leq h_2(n)$ in all instances, h_2 strictly dominates h_1. Both heuristics will also never overestimate the true solution cost of a puzzle as both heuristics are based on a simpler version of 8-Puzzle with less restraints (Russell and Norvig 2003).

The underlying reason for the efficiency our implementation of A* likely lies within our implementation within our code. We included constraints to prevent the puzzle from visiting states that had already been tested, allowing the function to not waste time looping through the same few states over and over again. This feature may not have been presented in Russell and Norvig's design, possibly explaining the discrepancy in our results.

5 Conclusions

In this paper, we have attempted to replicate the results of Russell and Norvig's research on solving the 8-Puzzle. We successfully implemented an A* algorithm based on h_1 and h_2 heuristics originally developed and tested by Russell and Norvig. Our results were consistent with the original paper's trend that h_2 outperforms h_1 at every solution depth. Moreover, due to certain implementation specifications, our A* algorithm proved to be more efficient than that of Russell and Norvig. Possible future directions in research would be devising a more efficient heuristic for the problem or implementing the heuristics in a larger-scaled puzzle problem.

6 Contributions

The work on the experiments and ensuing paper was divided evenly amongst the two authors. Any work that was done separate was uploaded to a shared Git repository and there was frequent communication between them that maintained understanding regarding the progress that was being made, whether it was separate or joint.

References

Russell, S. J., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Pearson Education.

Zeng, W., and Church, R. L. 2009. Finding shortest paths on real road networks: the case for a*. *International Journal of Geographical Information Science* 23(4):531–543.