Comparison of Four Different Heuristics Used in the A* Algorithm

Ryan Florida

For this project, we were to implement the A* algorithm in order to optimally solve the 8-puzzle problem. The 8-puzzle problem is a problem defined by having a 3×3-grid, where there are 8 ordered tiles and one blank space. An interpretation of this can be viewed, in Figure 1, below.

0	1	2
3	4	5
6	7	8

Figure 1: Slider Puzzle with 8 tiles in its goal state.

If we let 0 represent the blank space, then what Figure 1 shows is what we shall forevermore refer to as the "goal" state. Now, suppose we are given a puzzle like that in Figure 2.

8	6	7
3	5	1
2	4	0

Figure 2: Randomized slider puzzle

The statement of the 8-puzzle problem can then be rephrased as the question: how would one go about getting the above configuration to the goal state in the least amount of moves? Fortunately, this question can be easily, and optimally, answered by using what is known as the A* algorithm. The A* algorithm is an algorithm which finds a "least-cost-path" solution to a given problem. This is to say that, given some path-cost function $f(n) = g(n) + h^*(n)$, A* will always act in a way so as to minimize $f(n)$. Now let us inspect $f(n)$ a bit more closely. The input, n, to the cost function is just a node in a "tree of possibilities"—this is to say that n is a particular configuration of the slider puzzle out of all the possible configuration—$f(n)$ then tells us how expensive n is to add to our path. So, what can be said of $g(n)$ and $h^*(n)$ then? $g(n)$ is the cumulative cost of n, which begins at the starting configuration; to put this simply: if n' is a configuration that took us 3 moves to get to, then—assuming the cost per node is 1—$g(n') = 3$. $h^*(n)$ is the cost associated with the remaining number of moves from n to the goal state. Something should be unsettling about the definition of $h^*(n)$ though: how could we possibly know how far any given configuration is from the goal state? The answer is that we cannot know $h^*(n)$ for every problem. Thus we replace $h^*(n)$ with $h(n)$ so that $f(n) = g(n) + h(n)$. $h(n)$ is what is known as a heuristic, and it is something

that is decided by whatever entity is trying to solve the problem; the heuristic normally requires a bit of background information about the problem. Only two constraints shall be placed on $h(n)$:

1. $h(n+1) > h(n)$
2. $h(n) \leq h^*(n) \quad \forall \quad n$

So, what makes for a sufficient choice of $h(n)$? For this project, we consider four different heuristics and then compare them in order to determine which, if any, is best. The four heuristics we will be using are:

1. $h(n) = 0 \quad \forall \quad n$, i.e. we are only considering path cost; this is equivalent to what is known as a uniform cost search.
2. $h(n) = N$, where N represents the number of tiles displaced from the goal state.
3. $h(n) = M$, where M is the sum of Manhattan distances of all tiles from the goal state.
4. $h(n) = M'$, where M' is the sum of Manhattan distances of all tiles from the goal state added to the number of displaced neighbors of each tile.

The code for this problem was developed with quite a few considerations being made. The first step of development was deciding how to represent the configurations of the puzzle board. To this end, it was decided that a struct was to be used which contained the following properties: $g(n)$, the path cost; $h(n)$, the heuristic cost; the total cost; where the blank space should be moved in order to get closer to the goal configuration; the number of possible configurations that can be achieved based on the current configuration; the depth of the current node; the actual configuration of the puzzle board; and the previous configuration and possible future configurations, all based on the current configuration.

The next step was deciding how to go from one configuration to the next. Two implementations were tested, one was to store each node in a closed list and the alternative was to build a tree of valid nodes. The closed list implementation worked well, but it was found to be too slow to perform in a reasonable amount of time for the displaced tiles and uniform cost search heuristics. The tree implementation, however, proved to be superior in every case. Both implementations actually have shockingly high times for the uniform cost search, but the tree implementation was still found to be significantly faster. Hence, the tree implementation was chosen because of its reduction of the large time overhead associated with the list implementation.

Next there was the question of which data structures to use for the storage of the puzzle board, the frontier, the children, the potential moves, the goal state, and the distances for the heuristics. For the puzzle board, an array of unsigned integers was used for its random-access as well as its ability of being quickly traversed. An unordered map was considered because of its constant lookup time, but there were several traversals needed to be performed in order to locate the blank tile; therefore the array was chosen. A vector was used to maintain the frontier because it too seemed to be faster to traverse than an unordered map, which was also considered for this task. The advantage of the vector over an array is that the vector has a dynamic size, whereas an array has a static size. For the children, a struct pointer array was chosen because it was known that there could be at most four different moves for a given configuration. For the list of potential moves, again an array was used for ease of traversal. Since we needed to sum over the distances corresponding to a each space on the board for the Manhattan and modified Manhattan heuristics, an array was also chosen for the distances. Finally, and unsurprisingly, an array was used to store the goal state.

Once there was an established way to store and filter through the nodes, then the

A* algorithm was be implemented. The way in which the algorithm was carried out, in pseudo-code is as follows:

1. *Create root node*
2. *Initialize the root node according to the input state*
3. *Goal-check the root node to see if the puzzle is already in the goal form*
4. *If root is goal, then display the root and quit ; else continue*
5. *Set root's cost to ∞ and add to frontier*
6. *while the goal has not been reached and the frontier is not empty, expand the current node*
7. *Add all children of the current node to the frontier, if they are neither in the closed list nor already in the frontier*
8. *Select the least-cost child as the new current node and goal-check it*
9. *Go to next iteration of while-loop*
10. *If the frontier is not empty, print the solution; else there is no solution*
11. *Quit*

Following the above algorithm, we find for each of the heuristic function's the statistics displayed in Table 1. We then see from the Table 1 that there is a clear worst heuristic to pick, h_0, which corresponds to the uniform cost search. It should be noted that the uniform cost search was so inefficient, in fact, that it would not complete most of the runs that the other heuristics handled instantly; this lead to a rather random sampling of random seeds as compared to the others, which all ran on the same random seed values. Even with these randomly chosen seed values, the uniform cost search performed horribly. This was more than likely due to the implementation even though the UCS was predicted to run much slower since it is an uninformed method, whereas the others are informed methods.

The next least-preferable method is the displaced tiles heuristic, h_N. There is nothing wrong with choosing this heuristic, it just is not the most optimal choice available to us because it grossly underestimates $h^*(n)$. $h(n)$ is supposed to underestimate $h^*(n)$, which is a constraint we impose on $h(n)$, but h_N is just very optimistic.

Looking at the final two heuristics, h_M and $h_{M'}$, we see that $h_{M'}$ actually outperforms h_M in most aspects. Yet h_M seems to outperform $h_{M'}$ in terms of node depth. Upon first blush, this might seem like a trivial thing to point out, but if we look at d_{max} we actually see that $h_{M'}$ has a larger value than h_M and h_N — here we do not consider h_0 because its stats are not comparable to the others'. This is actually quite a problem because it implies that $h_{M'}$ does not perform optimally in every case. Unfortunately, this means that $h_{M'}$ was a bit too pessimistic and actually over-estimated $h^*(n)$. Hence $h_{M'}$ is not a valid heuristic to use.

Thus we conclude that h_M is the most optimal heuristic to use and h_0 is the worst—

$h_{M'}$ is not considered here since it was found to be sub-optimal.

Table 1: Statistics for Each Heuristic

	h_0	h_N	h_M	$h_{M'}$
V_{min}	11	10	10	10
V_{median}	14575.5	5309.5	1005.5	909.5
V_{μ}	45311.26	29557.81	3370.27	1920.8
V_{max}	1703301	682415	68643	20094
V_{σ}	184304.525	79807.168	8179.046	2848.202
N_{min}	6	5	5	5
N_{median}	4151.5	1357.5	255	236
N_{μ}	11822.1	7510.25	854.23	493.9
N_{max}	427144	171426	17286	5029
N_{σ}	46709.917	20178.506	2057.677	721.328
d_{min}	2	4	4	4
d_{median}	14	18	18	19
d_{μ}	13.62	18.04	18.04	18.36
d_{max}	24	28	28	30
d_{σ}	3.728	4.613	4.613	4.836
b_{min}	1.703	1.342	1.236	1.237
b_{median}	1.759	1.484	1.358	1.338
b_{μ}	1.772	1.478	1.358	1.339
b_{max}	2.449	1.545	1.495	1.495
b_{σ}	0.102	0.044	0.048	0.05