Josh Morris
A* Search Report
CSCI 4350
Dr. Phillips

The 8 Puzzle problem is one that many are very familiar with. Many people received them as a stocking stuffer or from a prize box in elementary school. The game consists of 8 tiles numbered one through eight in a three-by-three grid that can be slid around using the empty tile space. The goal of the game, given a scrambled board, is to arrange the tiles in a descending fashion starting with the empty space in the upper left hand corner, like so:

	1	2
3	4	5
6	7	8

Solving such a puzzle can be tricky and quite time consuming, thus making it a good candidate for exploring various search algorithms to find the solution.

To solve this problem, an A* Star search algorithm was used. A* search is an informed search meaning it tries to estimate the cost of the current state to the goal in addition to the cost from the start state to the current state. Mathematically, this looks like the following equation:

$$\mathbf{f}(\text{curNode}) = \mathbf{c}(\text{startNode}, \text{curNode}) + \mathbf{h}(\text{curNode})$$

Where **c** represents the cost to the current node from the start node; **h** ,known as the heuristic, represents the cost from current node to the goal; and **f** is the total cost of the path from the start node to the goal passing through the current node. It is nearly always impossible to create a completely accurate heuristic function for a problem and the 8 puzzle is no exception. For this reason, the heuristic function is typically a simplification of the problem being solved. In addition to being a simplification, the heuristic must also be admissible and consistent. A heuristic function is admissible if it never overestimates the cost to reach the goal and is always optimistic. A heuristic function is considered consistent if **f**(n) is nondecreasing along any path.

The implementation of A* for this project follows the typical structure of a search algorithm. It utilizes a frontier of unexpanded node configurations implemented as an ordered set of node pointers and a closed list of expanded configurations implemented as a set of board configurations. A while loop continuously process the first element in the frontier until the goal is reached or the frontier is empty, meaning there is no solution given the starting configuration. Nodes were implemented as a struct for ease of keeping track of the blank coordinate, previous node, board configuration, **f**(n), and **c**(n). Since the object of this lab assignment is to contrast various heuristic functions, heuristic functions were passed as function pointers to the A* function for easy code reuse. An emphasis was placed on using pointers as much as possible to avoid copying of data, a very costly operation when passing STL data structures to functions.

For the assignment, students were asked to implement no function, the number of tiles displaced, the sum of manhattan distances of displaced tiles, and one function of our own choosing for selectable heuristic by a program user. For my heuristic, I chose to use the sum of the number of displaced tiles and manhattan distance. My choice was motivated by the idea, that if this heuristic was admissible, it was consistently dominate the other two heuristic functions and give even better metrics. Dominance is defined as if there are two heuristics, **h1** and **h2**, **h1** dominates **h2** if **h1** is greater than **h2**.

In this experiment we use four metrics to evaluate the effectiveness of a heuristic. First, is the total number of nodes expanded, **V**. This metric shows how many nodes an algorithm must expand in order to reach the solution. The smaller this number is, the more memory efficient the algorithm is at estimating the total cost. Second, is the total number of nodes in memory, **N**. This is a measure of memory efficiency of the heuristic and also cost estimation effectiveness. Third, is depth, **d**, of the solution. If a heuristic is admissible and consistent, then it will give the same depth as other admissible and consistent heuristics. The final metric is the branching factor, **b**. The branching factor is an estimate around the average number of children a node has. It calculated using the following equation:

$$N = b^d$$

The closer the branching factor is to one, the more efficient the heuristic is in finding the solution.

For this experiment, each heuristic was tested using 100 randomly generated boards that were shuffled using 100 random moves from the goal state. The program output the total number of nodes expanded, the total number of nodes in memory, the depth of the solution, the branching factor, and the board configuration of each move to solve the puzzle.

No Heuristic

	V	**N**	**d**	**b**
Minimum	29.0	48.0	4.0	1.596
Median	64967.5	95716.5	20.0	1.792
Mean	93847.99	124312.3	18.32	1.828
Maximum	455288.0	487046.0	28.0	2.632
Standard Deviation	101100.669	123278.494	5.151	0.155

Number of Tiles Displaced

	V	N	d	b
Minimum	5.0	10.0	4.0	1.414
Median	2477.0	4024.5	20.0	1.520
Mean	5704.18	8856.16	18.32	1.521
Maximum	75754.0	107564.0	28.0	1.778
Standard Deviation	9678.748	14224.502	5.151	0.034

Manhattan Distance

	V	N	d	b
Minimum	5.0	10.0	4.0	1.275
Median	258.0	421.0	20.0	1.374
Mean	498.46	796.66	18.32	1.381
Maximum	3193.0	4955.0	28.0	1.778
Standard Deviation	597.511	939.429	5.151	0.0624

Sum of Number of Displaced Tiles and Manhattan Distance

	V	N	d	b
Minimum	5.0	10.0	4.0	1.221
Median	327.0	540.0	20.0	1.376
Mean	533.14	866.14	18.66	1.384
Maximum	3739.0	5874.0	30.0	1.778
Standard Deviation	667.138	1062.195	5.426	0.071

It can be seen that overall the fastest and most memory efficient heuristic is the sum of manhattan distances. It outperforms no heuristics by a factor of approximately 227 and

outperforms number of displaced tiles by a factor of approximately 10 in terms of total number of nodes in memory. Furthermore, it expands far fewer nodes than any of the other methods meaning it is objectively much faster at finding the shortest path to the solution. As a heuristic, manhattan distance consistently dominates no heuristic and the number of displaced tiles.

The heuristic of my own design performs close to that of the sum of manhattan distances; however, it is found to be suboptimal. This can be seen in the average depth of the solution. No heuristic, manhattan distance and number of tiles displaced are all consistent and admissible meaning they are optimal. This is further proven by the fact they all have the same mean depth of solution for the hundred test cases. The sum of manhattan distance with the number of tiles displaced has a slightly higher average than the other three. This seems to indicate that on average it found the optimal solution, but in some cases it did not.

The limitation of this problem is that it cannot quickly determine if a board configuration is unsolvable. It would have to explore every reachable board configuration from the start configuration before determining so. Additionally, it does use a lot of memory because it must generate many states in trying to reach the solution. Larger boards would pose a significant problem. In one of my early implementations of the algorithm, I ran out of memory on some of the no heuristic cases.

The most serious improvement to my code was moving to allocating nodes once and then using pointers to reference throughout the rest of the program. I realized that when I was passing sets to functions a copy constructor was being called causing a lot of time being lost. This had the effect of significantly speeding up finding the solution.

A* search is a viable method to find a solution when a better method does not present itself. Careful selection of a heuristic goes a long way to making this an appealing approach. Due to the nature of the algorithm, careful attention must be paid to node construction, handling of memory and data structures. If implemented well, this algorithm can be quite effective indeed.