Name: Cody Falzone
SID: 860929046
CS170 Winter 17
Project 1 Report

Searches Solving 8-Piece Sliding Puzzle

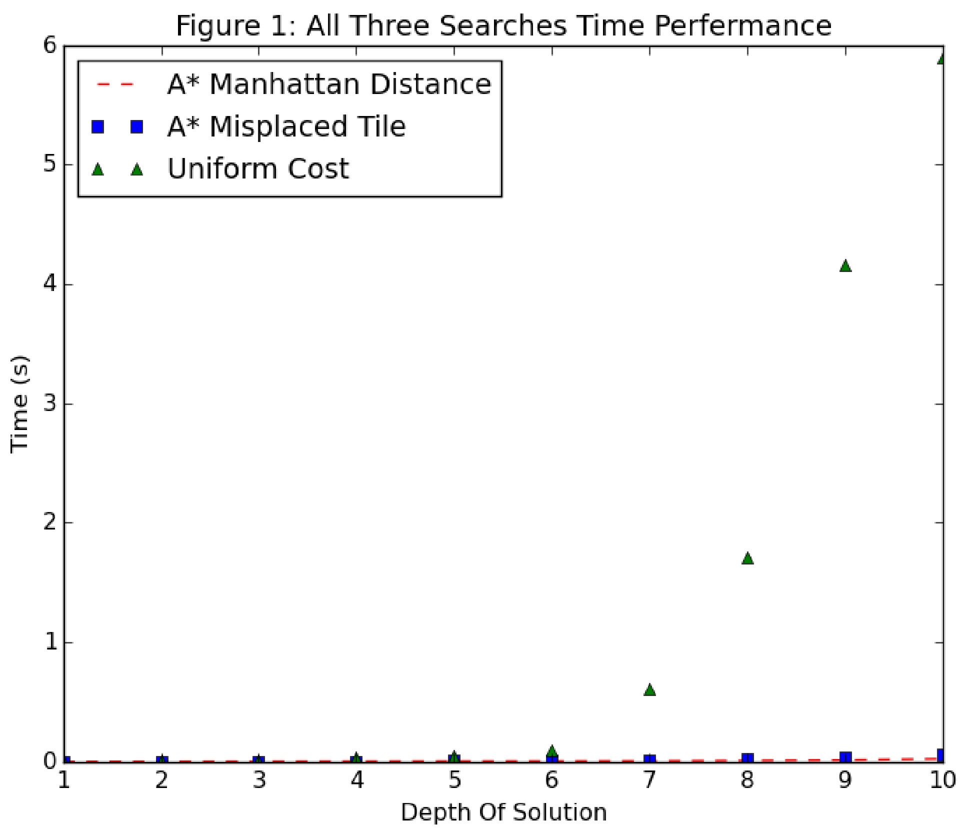
Using artificial intelligence in the computer science field is a powerful tool. An example of this is solving problems by searching. Let's consider a 8-piece sliding puzzle as the environment and three different search algorithms as the goal-based agents. The three different search algorithms that will be analyzed are Uniform Cost Search, A* with Misplaced Tile heuristic, and A* with Manhattan Distance heuristic.

The 8-piece sliding puzzle has eight pieces and nine total slots. There is one blank spot that define our actions or operators in reference to “moving” this blank spot. The positions of the 8 pieces on the puzzle is referred to as the current state of the puzzle. The search algorithms looks at available actions/operators that we may take given the current state of the puzzle and applies them iteratively to form a search tree of possibilities until they reach the goal state of the puzzle. In this search tree, nodes represent states of the puzzle. The process of evaluating the next possible states that result from the list of possible operators from the current state of the puzzle is known as expanding the node. The minimum number of moves required to get to the goal state from an initial state of the puzzle is referred to as the depth of the solution. The diameter of a search problem is given the worst initial case, what is the minimum number of moves required to reach the solution. This diameter of this 8-piece sliding puzzle problem is 31.

Uniform cost search expands the node with the next lowest path cost g(n), where g(n) is the path lowest path cost from the initial/root state to the node. A* not only takes g(n) into consideration, but also h(n), which is the estimated cost to the goal. This h(n) function is known as a heuristic function. A heuristic function for A* must underestimate the cost to the goal to be admissible. The two heuristic functions that will be discussed for this 8-piece sliding problem are the misplaced tile heuristic and manhattan distance heuristic. The misplaced tile heuristic counts how many tiles are in the incorrect spot in reference to the goal state of the puzzle, this is not including the blank tile. The manhattan distance checks the distance that an incorrect puzzle piece is from its correct spot, using the manhattan distance. Therefore, the A* cost function is $f(n) = g(n) + h(n)$. Additionally, Uniform Cost Search is just A* with the cost to the goal function h(n) equal to zero ($h(n) = 0$) .

The code is very versatile, object oriented, decoupled, and not problem specific. The code used to run these algorithms consists of multiples classes and inheritance in an object oriented coding structure. There are base classes, namely Search and Problem, that allow for a standard, general, decoupled interface of which the three different search algorithms inherit from Search and the puzzle problem inherits from Problem. There is a Node class that holds the state information of the puzzle. All three search algorithms start with the initial state of puzzle and add it to their frontier. This frontier is ordered based on the cost of the node. As described earlier, the costs of the nodes of determined by the type of search, each of them having their own cost functions f(n) using g(n) and h(n). Then, while the frontier is not empty, they pop a node off their frontier and evaluate/expand the node by applying valid operators to that nodes puzzle state. The children node's states are then checked to see if they are the goal, if they have already been expanded before, or if they are currently in the frontier waiting to be expanded. This is done successively until a goal node is reached.

The code also tests for invalid puzzles as well as returns the correct moves needed from the initial state to get to the goal. This code allows for any size puzzle to be entered.



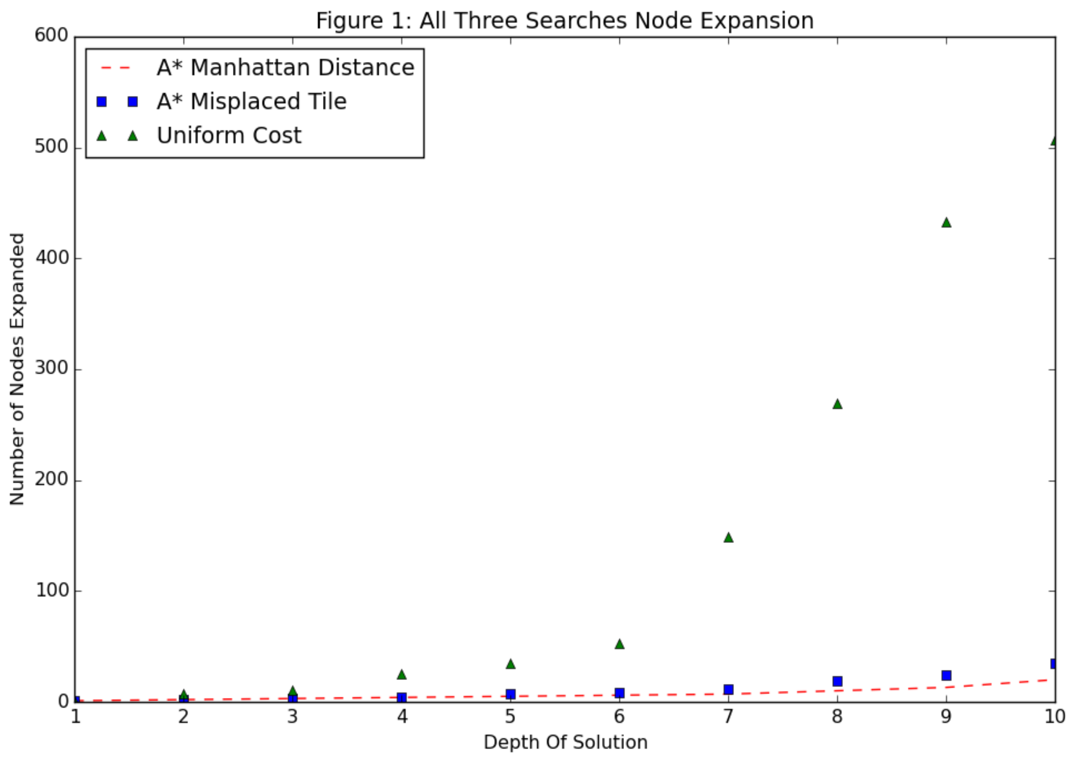


Figure 1: All Three Searches Node Expansion
A* Manhattan Distance
A* Misplaced Tile
Uniform Cost
Number of Nodes Expanded
Depth Of Solution
0
100
200
300
400
500
600
1
2
3
4
5
6
7
8
9
10

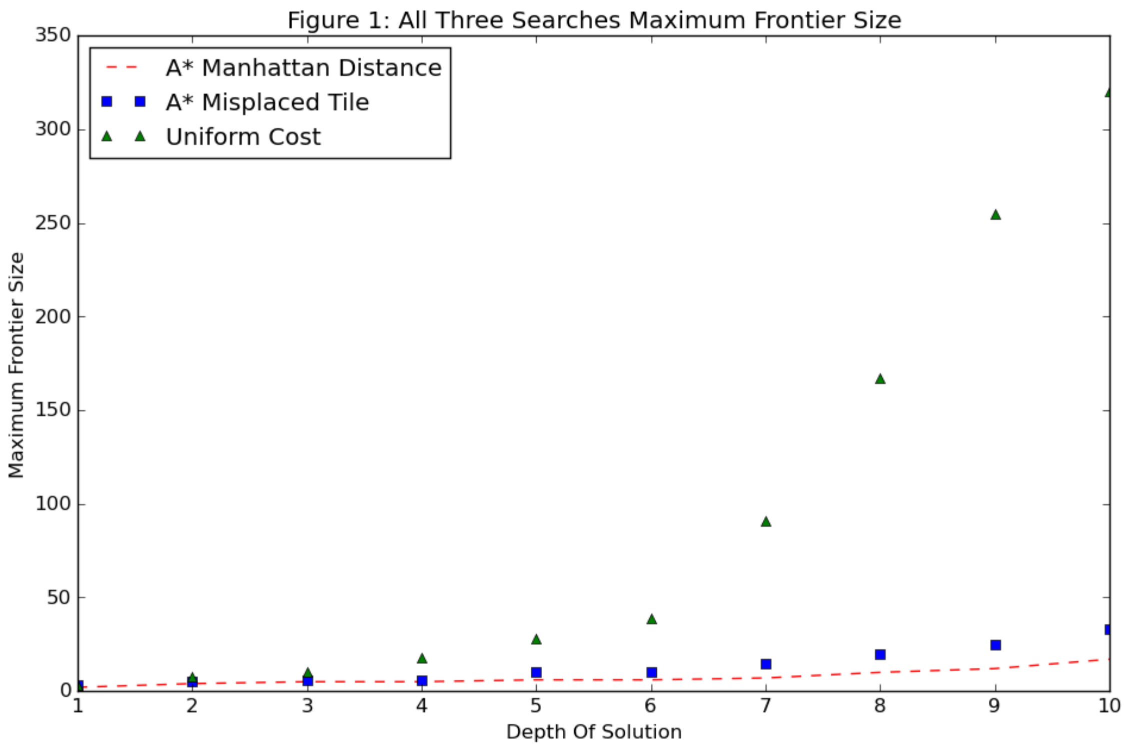


Figure 1: All Three Searches Maximum Frontier Size
A* Manhattan Distance
A* Misplaced Tile
Uniform Cost
Maximum Frontier Size
Depth Of Solution
0
50
100
150
200
250
300
350
1
2
3
4
5
6
7
8
9
10

Figure 4: A* Searches Time Perfermance

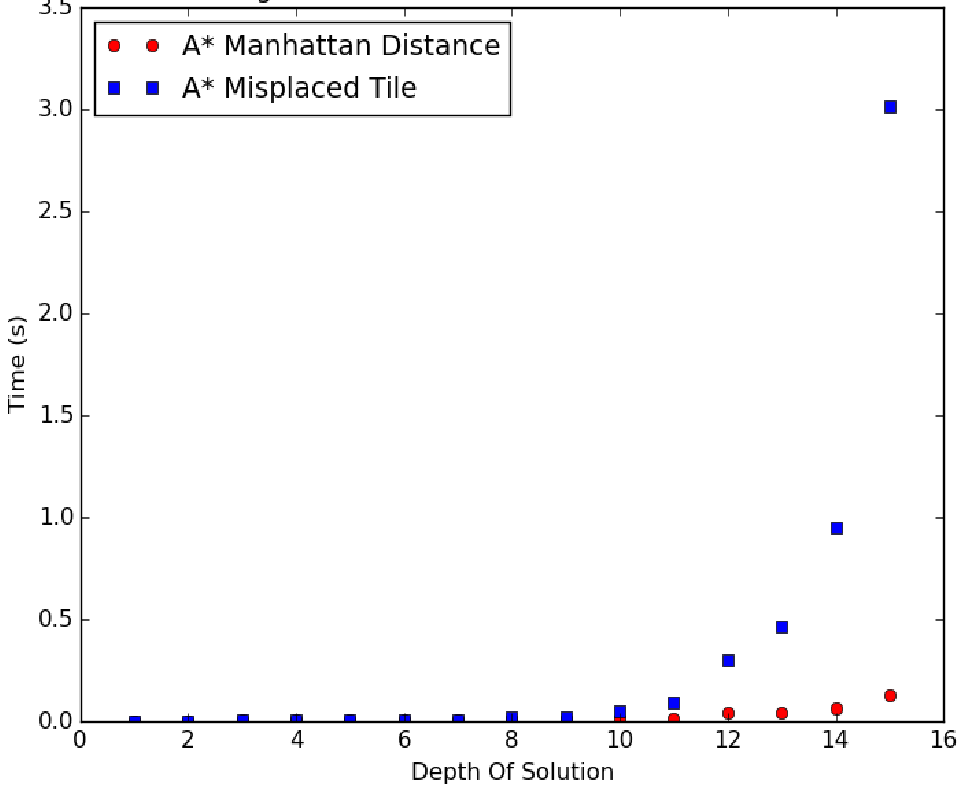
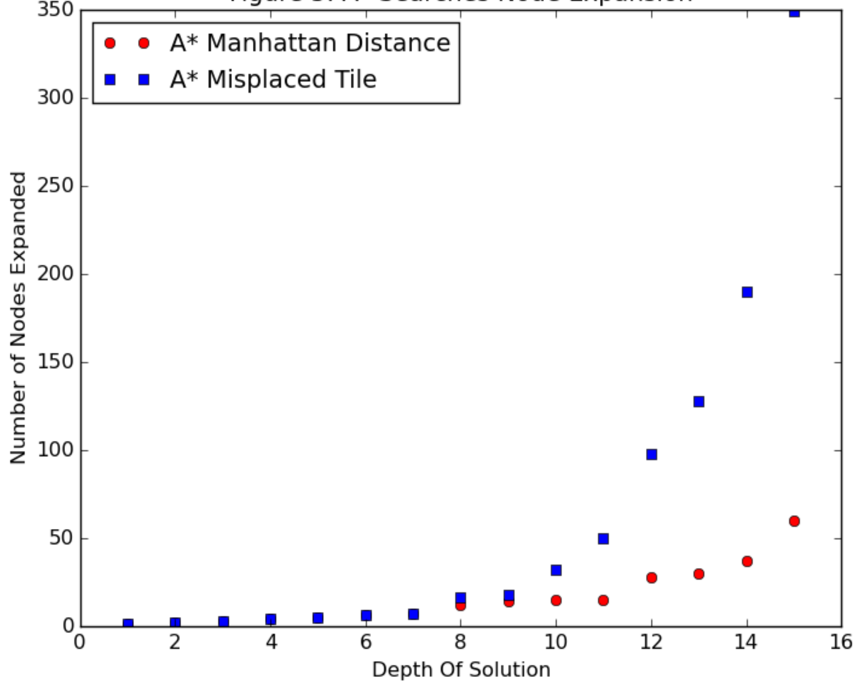
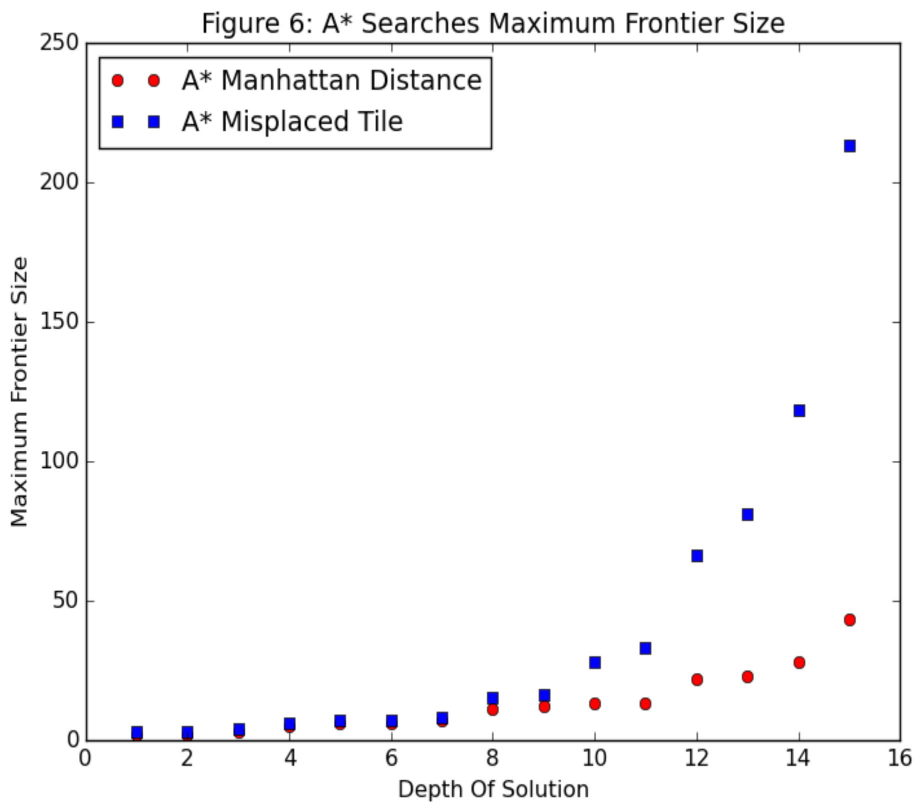


Figure 5: A* Searches Node Expansion





The criteria used to compare and contrast the searches will be the amount of time to find the goal state, the amount of nodes expanded, and the maximum size of the frontier. As shown in figure 1 above, the uniform cost is the worst algorithm by a huge factor in all three aspects of time, nodes expanded and max frontier size. It grows exponentially above the both A* algorithms. This is because Uniform Cost search does not use the h(n) function, which is the estimated cost to the goal as explained earlier.

Moving on to comparing and contrasting the two A* algorithms which is really only comparing their h(n) heuristic functions. The Manhattan Distance heuristic clearly performs better on all three aspects including time, nodes expanded, as well as max frontier size. This becomes more evident the deeper the depth of the solution is. As shown in the figures 4, 5, & 6 above, the grow rate is growing faster for the Misplaced Tile heuristic than that of the Manhattan Distance heuristic.

REFERENCES:

Fellow Class Student/Colleague : Robert Martinez

Book: Artificial Intelligence A Modern Approach 3rd Ed. Stuart J. Russel & Peter Norvig

This website was used as a reference for determining if a puzzle is solvable. I then wrote a function that calculated if a puzzle was solvable given an initial state based on the number of inversions.

Website: https://www.cs.princeton.edu/courses/archive/fall16/cos226/assignments/8puzzle.html

Various references to the python docs for use of raising errors, inheritance, methods, matplotlib, numpy, data structures, built-in functions, syntax for list comprehension, ect.

Website: https://docs.python.org/3/