Trent Callan

SID: 861117907

CS170: Intro to AI

Professor Moses Tataw

8 Puzzle Solver with A\* Search using different Heuristic Functions

Project Implementation:

To implement this project I chose to use objective-c and the cocoa touch platform, as it is where I plan to specialize after I graduate. Attached is my implementation of the three required algorithms represented as an iOS application. Because of the requirements of cocoa touch and the iOS platform this code must be compile in Xcode on mac computer. In hindsight I realize that when testing memory and computational efficiency of algorithms a mobile platform is the worst place you can choose to do so. I ran into many errors while coding this because of the memory and computation power limitations of mobile devices. In the future I would choose to do this project on a device with far more powerful hardware. As a result of my choice to use iOS as my platform my code is still optimal and complete but runs a bit slower than it would on a desktop device, but it still completes within a reasonable amount of time. To compile and run this program setup Xcode and the iPhone 6s simulator on you mac computer. Open the .xcodeproject file to open the project in Xcode and click the run button on the toolbar. This will run on your attached iOS device.

Note: you must have an itunes account to create a development provisioning profile for your iOS device that you wish to run this project on.

The 8 puzzle is a longstanding problem in AI that is can be solved using many algorithms. In this project we were asked to explore how to solve it using search techniques. The 8 puzzle is a sliding tile puzzle in which there are 8 tiles placed on a sliding board and one tile is left blank as place to slide the tiles around. The 8 puzzle problem is when you are given this board with tiles randomized and are asked to move the tiles to a certain goal configuration. To solve the 8 puzzle problem we use the A\* path finding algorithm using different heuristics to estimate the distance to the goal state. We show that these heuristics change the A\* algorithm to show the difference between blind and non-blind searches and how this affects the memory overhead and time it takes for these algorithms to find a solution to the 8 puzzle.

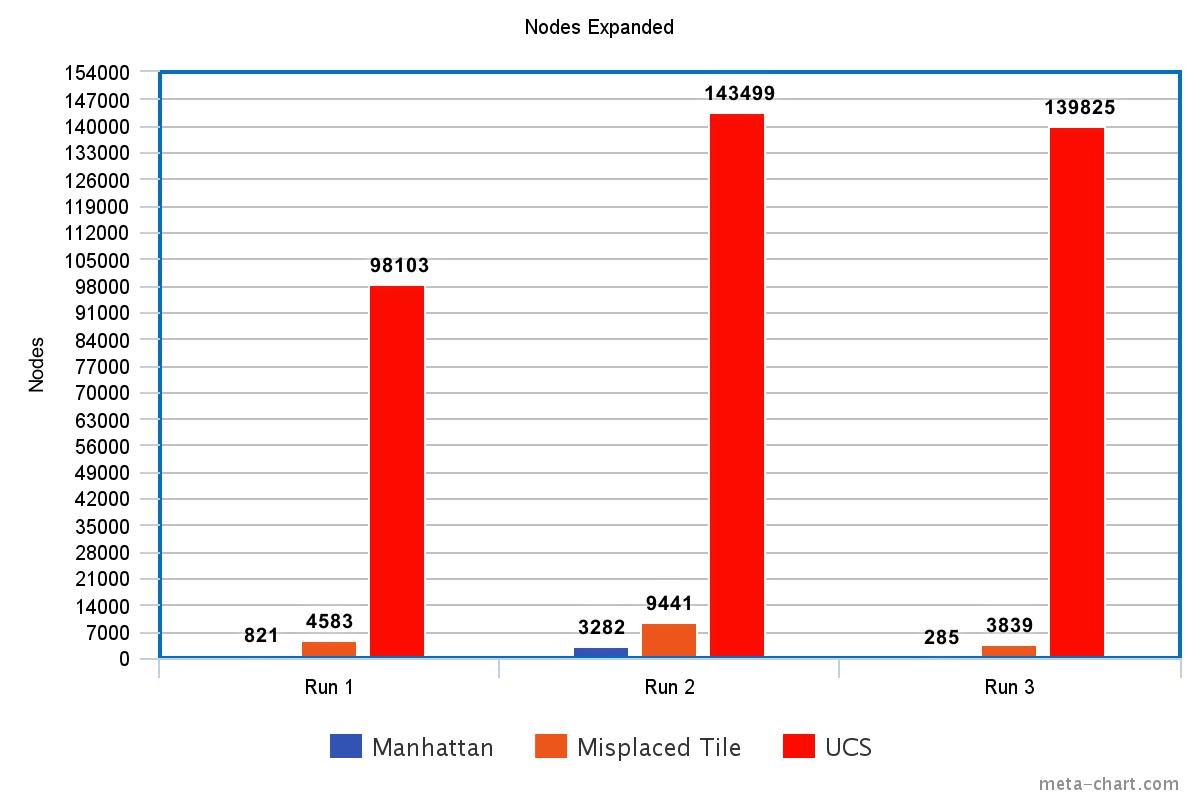
The three algorithms used in this project are the A\* algorithm with manhattan distance heuristic, the A\* algorithm with misplaced tile heuristic, and the Uniform Cost Search algorithm. We use these three different search methods to try to solve this problem. The general goal of theses algorithms is to look at an initial board state and expand that state using the list of viable moves and keep repeating this until the goal state is found. These algorithms use three different ways of choosing which nodes to expand. This choice has an enormous effect of how efficient in terms of both time and memory allocation that these algorithms have. The first and worst way to solve this problem is using a blind search. A blind search just expands nodes in order not ranking the nodes in any order or choosing the best node to expand. The blind search we use in this project is the uniform cost search, which just expands the nodes based on their depth level. The A\* algorithms work in similar ways. They choose the next node to expand by using an estimate of the cost to solve the puzzle from the current state to the goal state. This estimate is called a heuristic. The heuristic is just a best guess at how long to solve the puzzle from the current node. By choosing these nodes in the order of cost we can dramatically cut down the number of nodes searched when compared with the blind search tactics. The difference between our two implementations of the A\* algorithm is their heuristic. The first heuristic used in this project is the Misplaced tile heuristic. This heuristic estimates the cost to goal by counting the number of misplaced tiles between the current node and the goal node. The other heuristic used is the Manhattan distance heuristic. This heuristic calculates the distance to goal using the difference in the number of columns and rows between a tiles position in the current node and the goal position of that tile. It then sums up this total for all the nodes in the puzzle.

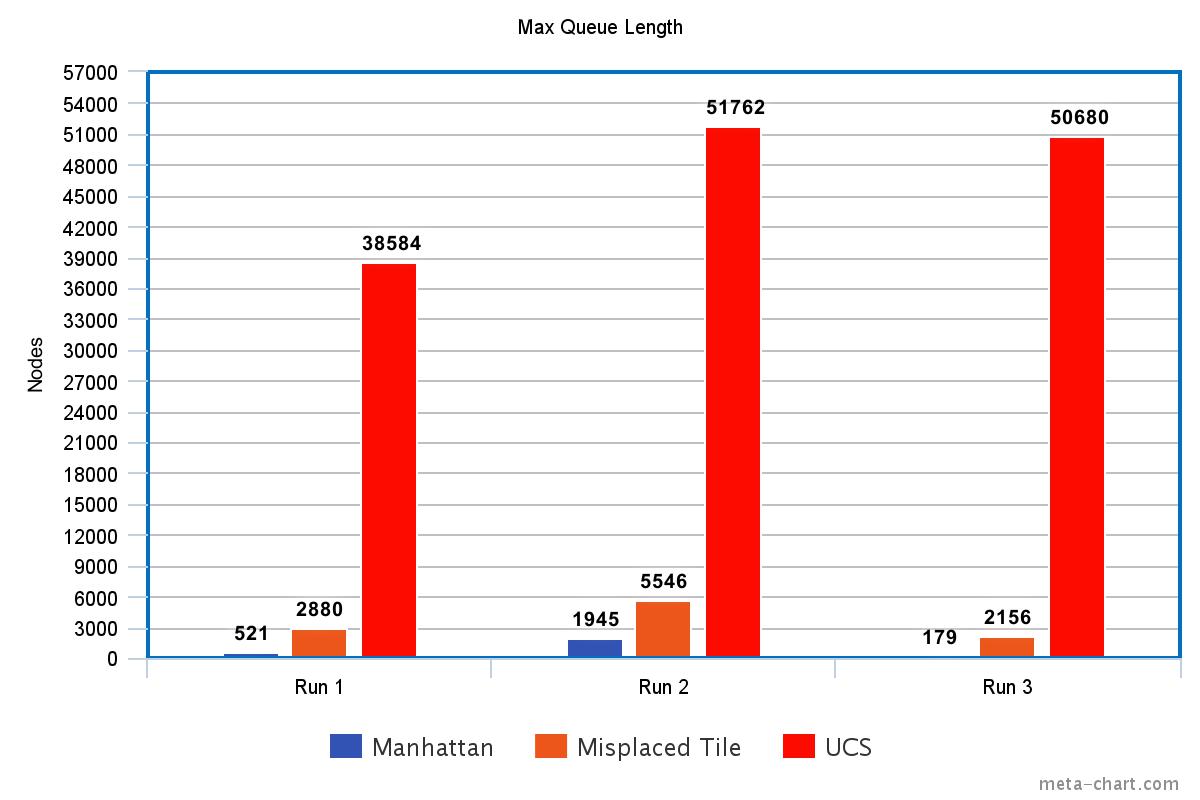
Results

Run 1 = 428603571

Run 2 = 476231580

Run 3 = 153406728





Conclusion

It’s easy to see which algorithms were better in solving this common 8 puzzle problem. The order in which the algorithms ranked in order of both speed and memory efficiency from least efficient to most efficient is, Uniform Cost Search < A\* with Misplaced Tile heuristic < A\* with Manhattan Distance heuristic. Its easy to see that the blind Uniform Cost Search performed far worse than the more intelligent A\* searches by seeing how many more nodes had to be expanded and how long the queue was. It is also generally true that blind searches are far slower than non-blind searches. Especially as the solution depth gets larger scanning all these expanded nodes takes far too long to be viable. A\* is optimal and complete and it will always give the best path to the solution. With a perfect heuristic A\* would always walk directly to the solution. The Manhattan Distance is also a better heuristic than the misplaced tile heuristic because it more accurately predicts the cost to the goal state without overestimating it. This means that it is admissible and is a better choice to use in solving the 8 puzzle problem. In conclusion, the A\* with Manhattan Distance heuristic is the best choice of these three algorithms to use to solve the 8 puzzle problem because it uses a more accurate admissible heuristic and it is an informed, non-blind search. These advantages over both A\* with misplaced tile heuristic and Uniform Cost Search make A\* with Manhattan Distance run much faster and more memory efficient.

Citations

Introduction to A\* Path finding. (n.d.). Retrieved November 01, 2016, from https://www.raywenderlich.com/4946/introduction-to-a-pathfinding

Used to understand how A\* works

IOS Foundation Documentation (n.d.). Foundation. Retrieved November 05, 2016, from http://developer.apple.com/reference/foundation

IOS Documentation