Milan Zanussi

Introduction to Artificial Intelligence

September 30, 2018

Project 1: The A* Algorithm and Its Application to the 8-Tile Sliding Puzzle

Puzzles and games comprise a large domain of intense computational problems due to their nature of leaving wide margins of possibility for players to explore, which creates a combinatorial explosion of all of the different ways players can approach the tasks involved in such puzzles and games. This unwieldy range of possibilities is a rather novel area of application for artificial intelligence and algorithmic techniques used in the field. One such application of artificial intelligence to puzzles will be the use of the A* search algorithm to find the shortest, or at least approximately the shortest, solutions to 8-tile sliding puzzles.

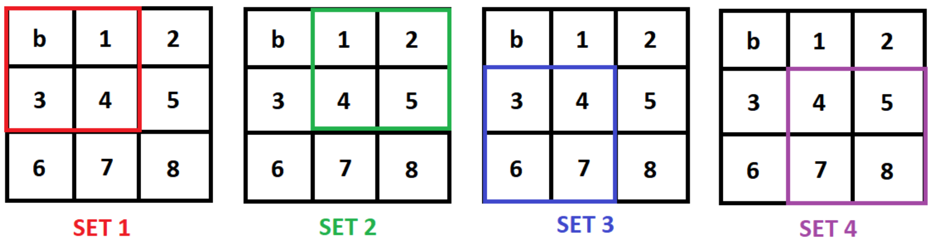
The 8-tile puzzle is a famous toy puzzle consisting of a three by three square grid with eight tiles, labelled with numbers one through eight, and one empty space. Any tile on the board can be slid into the empty space if the empty space is horizontally or vertically adjacent to the tile. The goal of the puzzle is to return its configuration into the standard initial configuration for such puzzles – that is, the empty space is in the top-left corner and the numbered tiles are ordered by their numbers from least to greatest starting with the tile labelled '1' at the top row and middle column and continuing in order from left to right on each row and continuing onto the successive rows when the end of a row is reached.

Many different pathways exist to reorienting the numbered tiles on the 8-tile slider puzzle into their appropriate positions. We can consider all of the different possible states of the puzzle – which will be all of the valid configurations that may be obtained by some series of movements away from the standard initial configuration – and we can consider all of the actions a player could take when in any of these states – that is, the rule which says that any tile may slide into the blank position so long as it is horizontally or vertically adjacent to the blank space. From this, we may construct a decision tree out of all of the possible paths which may be taken from some starting state, which may or may not be the standard initial configuration, and we can search through this decision tree to find the shortest series of moves to the standard initial state from the specified starting state given to the player. This idea of building a tree of all of the different possible ways to obtain the different states of a system is fundamental for the application of artificial intelligence to the 8-slider puzzle, but now we need to consider how we use this tree to find the shortest series of moves to solve the puzzle.

With our tree model in mind, we may consider all of the different means of searching the tree. The goal for computer scientists is to optimize the means with which the tree is searched by minimizing either the time it takes for the computer to finish the task, the space it takes to memorize the entire task, and to provide the closest path there is to being optimal. As explained previously, the algorithm which will be used is the A*-search algorithm. This algorithm is a means of searching a tree (or more generally a graph) by considering, for any node on the tree, the cost of the path used to arrive at the node – corresponding to how many slides it took to arrive at a state given a number of previous states [1, pages 92-93]. We add this path cost to some estimate of the length of the remaining path cost to attain the goal state. The function which is used to estimate the remaining path length is called the heuristic function, and depending on the heuristic used, the A*-search algorithm may obtain drastically different results.

In this project, we will use four different heuristics. These four heuristics will be:

1. $h_0(n) = 0$ for any node n : This heuristic is called the Uniform Cost Search heuristic, because it essentially makes no approximation of the remaining path length. The use of this heuristic is equivalent to a famous path finding algorithm called "Dijkstra's Algorithm" with the added rule that a goal state exists and that the algorithm should terminate when the goal is found.
2. $h_1(n)$ counts the number of tiles displaced from their positions in the standard initial state for the state corresponding to the node n. This heuristic uses the knowledge that for every displaced tile, the number of moves needed to move that tile to where it needs to be is at least one move.
3. $h_2(n)$ takes the sum of the Manhattan distances of each tile from its appropriate position in the goal state for the state corresponding to the node n. The Manhattan distance is one of a family of functions which generalizes the notion of Euclidean distance, however it is simple to think of the function as the sum of the horizontal moves which must be made to put a tile into a particular position added to the number of vertical moves which must be made to move the tile to that particular position.
4. $h_3(n)$ will rely on a diagram, but this heuristic will divide the 8-tile puzzle into four overlapping 4-tile squares and consider, if the board was in standard configuration, which squares would contain which tiles and the empty space. The function will then consider where each tile is located and will count how many squares in which the tile should be, but is not; and also count the squares in which the tiles lies, but should not lie. The function performs this operation on each tile, as well as the blank space, and then subtracts one for each tile which is adjacent to the blank tile and also displaced. This heuristic was one focal point of the project, since the other heuristics are already well-studied. This heuristic speculates that a good measure of distance from the optimal goal is a generalization of displacement where tiles are no longer categorized as singletons, but instead in overlapping groups which may characterize their positions on the 8-tile slider puzzle. A diagram of the groupings used is given by the diagram directly below.



With these heuristics, we tested the efficacy of the A* search algorithm on randomized 8-tile slider puzzles. The tests were performed on exactly one hundred randomized puzzles, and the following data was collected:

1. V : The number of nodes which were checked (expanded) in the search.
2. N : The number of nodes that were generated in the search.
3. d : The tree depth of the solution found
4. b : An approximate number of how many branching paths were generated for each node

The number of nodes expanded gives us a measure of how much time each heuristic took to search for their solutions. The number of nodes generated gives us a metric for determining how much memory was used by each of the heuristics to find a solution. We also found the solution depth to find the optimality of solutions found by each algorithm as well as to compute the branching factor in combination with the number of nodes generated.

The results gathered are given by the following table:

	Uniform Cost Heuristic				**Displaced Tile Heuristic**				**Manhattan Distance Heuristic**				**Square Inclusion Heuristic**			
	V	*N*	*d*	*b*	*V*	*N*	*d*	*b*	*V*	*N*	*d*	*b*	*V*	*N*	*d*	*b*
Min.	69	119	6	1.599	8	14	6	1.497	7	13	6	1.303	9	16	6	1.327
Max.	492103	512649	28	2.218	134170	182673	28	1.627	10664	16346	28	1.533	27564	42888	28	1.665
Median	41737	61949	18	1.803	2142.5	3405	18	1.548	365.5	596	18	1.398	1326	2172	20	1.495
Mean	85690	112798	18.08	1.823	8521.3	12756	18.08	1.548	725.8	1149	18.08	1.397	3143	5045	18.24	1.504
Std. Dev.	106407	124854	4.739	0.111	17774	25076	4.739	0.018	1276	1971	4.739	0.04	4652	7312	4.868	0.057

With the results of this table, we can see from the median and mean values of V and N values for each heuristic that the Uniform Cost Search is abysmally slower than all other algorithms. Uniform Cost checks of ten times as many nodes as the next slowest algorithm, which is the Displaced Tile Heuristic. The Displaced Tile Heuristic, while an improvement, runs substantially slower than both the Manhattan Distance heuristic and the Square Inclusion heuristic. We also note that the memory consumption seems to follow a similar trend between heuristics, but we do note that the faster heuristics expand a smaller portion of the nodes that they generate than the slower heuristics, which may reasonably seem like a good indicator for the speed of particular heuristics, since it shows that the heuristics are good at predicting which branches contain an optimal solution.

We note that the Square Inclusion Heuristic has a large average solution depth than all other solutions. We also note that all of the other heuristics are admissible[1, page 103] – that is a technical term which means that these heuristics do not overestimate the length of the path to the goal from any node state [1, page 94]. This means that our Square Inclusion Heuristic is inadmissible. Given the fact that the Square Inclusion Heuristic generates and checks more nodes than the Manhattan Distance Heuristic, this means that there should be no reason to use the Square Inclusion Heuristic other than playing an inane practical joke on an unsuspecting undergraduate student or for demonstrating that utilizing some of the topological aspects of particular problems, such as the existence of notions of distance, to create more efficient search strategies than naïve approaches which may have interesting motivations but inherently utilize less information about the environment of the problem.

Overall, the results of this project demonstrate the complexity of such literal toy problems like puzzles and games, and how algorithms and heuristics must be carefully chosen and evaluated before they are applied to such expansive problems. This has also demonstrated the importance of introducing relevant assumptions into the design of algorithms to solve such problems.

Works Cited:

[1] Russell, S. J., & Norvig, P. (2014). *Artificial Intelligence: A modern approach*. Harlow, Essex: Pearson.