A and the 8 Puzzle Problem*

Rachel Conforti
Computer Science Department
University of North Carolina at Charlotte
Charlotte, United States of America
rconfort@uncc.edu

***Abstract*— A* is an informed search algorithm that is used in many industries such as video games, GPS mapping, and natural language processing. It is one of many algorithms in artificial intelligence however it is unique due to its completeness, optimality, and efficiency. The assignment is to solve the eight puzzle problem using the A* algorithm. The A* algorithm is given the start node and the end goal and expected to solve the board. In this paper, I talk about the eight puzzle problems in more depth, an overview of A*, and an explanation of my code and results**

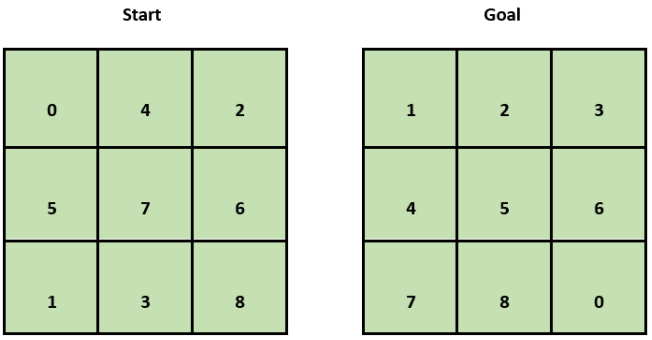
Keywords— A*, manhattan distance, path cost, 8 puzzle problem, informed search algorithm

I. Introduction

The eight puzzle problem and A* (pronounced a-star) is an interesting combination. The eight puzzle problem is the problem we want to solve using A*. A common implementation of the 8 puzzle problem in the real world is with a jumbled-up photo and it is the players' job to unscramble the photo by moving a tile into the empty slot until the picture is revealed. This is a very simple idea but trying to get a computer to solve it and solve it optimally can be a bit tricky. A* is an informed searching algorithm whose goal is to solve the board in the most efficient and optimal way possible. A* is commonly used in map transversals such as Google Maps. The reason companies use A* is because this algorithm can easily backtrack and reroute itself around obstacles or deadends. Within this paper, I will describe the 8 puzzle problem, an in-depth look into the A* algorithm, and an overview of my code. And lastly, the results of my code and how I would improve my code.

II. 8 Puzzle Problem

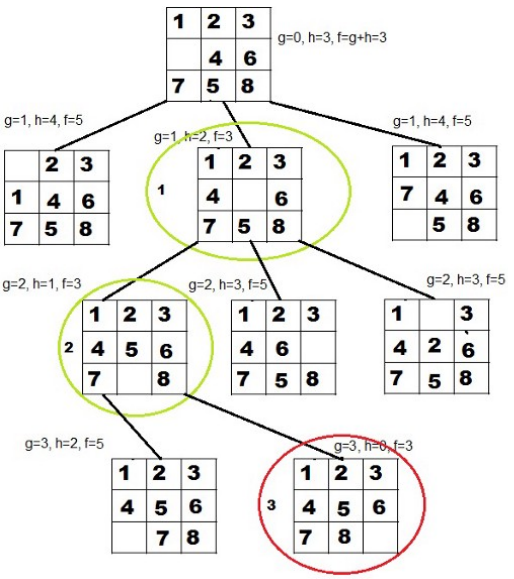
The eight-puzzle problem is a puzzle invented by Noyes Palmer Chapman in the 1870s. It consists of a three-by-three grid with eight playable tiles and one blank tile. The goal is to arrange the tiles in the same formation as the goal. For this assignment the goal is to be in the formation [1,2,3,4,5,6,7,8,0]. The only legal moves are up, right, down, and left. Below is a photo of the basics of the 8 puzzle problem.



There are many ways to solve this puzzle and all of them are valid. However, not all paths are the most optimal path to the goal. That is where A* comes in to assist since the purpose of A* is to solve the board in the least amount of moves possible. But first, we need to figure out if it is even possible to solve by taking the inversion of the board matrix. Since the board will always be odd, if the inversion count returns as odd it is not a solvable board but if the inversion is even the A* algorithms can be applied to solve the board.

III. A* Overview

In this assignment, the algorithm used was the A* algorithm. A* is an informed searching algorithm that uses heuristics to solve the problem. Meaning the algorithm will follow the option with the lowest cost till it reaches the goal node. An example of how the algorithm solves a board can be seen below:



A. G, H, and F values of A*

As shown in the photo, the board is using three values g(n), h(n), and f(n). The g(n) value represents the path cost from the start node to the next node. In this case, the g(n) value will represent the depth in the tree. So, for every step, we just add one to the previous path cost. The h(n) value represents the heuristic that estimates the cheapest path from the node to the goal node. You can calculate the heuristic in a few ways. For example, using Euclidean distance, manhattan distance, or simply the number of misplaced tiles. When you add the g(n) and h(n) values together they create the f-value that is typically shown as: f(n) = g(n) + h(n). The f-value is the key to making A* so unique and a highly efficient algorithm. This is because the heuristic would never overestimate the cost causing the algorithm to always be optimal.

B. Open list and Closed List

Within A* there are two lists utilized. One list is called the open list that allows the algorithm to remember where it is and its possible moves. The second list is called the closed list that holds all the places the algorithm has been. The open list within my assignment is a priority queue. The open list is critical to optimality and admissibility because it is constantly sorting by the lowest f-value to ensure that it is on the optimal path.

C. Why A* is a complete, optimal, and efficient algorithm

A* is a popular option because of its completeness, optimality, and efficiency. Being a complete algorithm means that A* is addressing all the possible inputs. Meaning that no matter what is thrown at the algorithm it will solve the problem. By the algorithm constantly searching through all the possible neighbors of the current board we are looking at all possible inputs for the algorithm. Being admissible means that the algorithm will never overestimate the problem because the cost to reach the goal is not higher than the lowest possible cost from the current node to the goal node. The way A* ensures its admissibility is through the heuristic. Since we are using the manhattan distance to solve the cost from the node to the goal it ensures that our algorithm is admissible. And lastly, to be consistent it means that A* is locally admissible meaning it can not overestimate the distance between any pair of nodes in the graph being searched. A* completeness, optimality, and efficiency are why it's loved and used in many different situations because no other algorithm is guaranteed to expand fewer nodes than A*.

D. The Logic of A*

The steps of the algorithm are as follows: A* takes in the starting node, appends it to the open list and closed list. Then it pops off the open list (priority queue) and gathers all the legal potential moves (up, right, left, down). From these populated moves (M), it checks to see if they are new moves by checking if M is in the open list. If M is on the open list, it means that we are already aware of this move and we do not want to add it again.

If the new move is in the open list, it will check to see if the new move has a lower path cost than the start node. If it does have a lower cost that means that is a better move so we add the new move to the open list and remove the worse node.

However, if the new move isn't in the open list we check to see if the node isn't in the closed list. If it passes this check that means it is not a node we have encountered before and we add it to the open list. After it does this for every new move it will restart by taking the new lowest f-value board by popping off the open list and expanding all the potential moves from there. This will continue until the goal node is reached. There are several ways to stop this search, but you are looking for when the heuristic equals zero, which signals the algorithm it found the goal node.

Due to its ability to solve the problem most optimally, it is a popular choice in many industries. The A* algorithm can solve the problem quickly, and even reroute itself around obstacles allowing the program to not get stuck in a loop. It is a very powerful algorithm that is used in many GPS applications.

IV. Code Overview

In this section I will break down how my code is working. I broke my code into sections that I thought would make it easier to understand my logic.

A. Node Class

The first magic method I used is the init method that creates the node class. This is built with 4 input values: value, parent, h, g, and f. Value will hold the whole board at that current state. This is to allow us to save the new move and append it to the open list. The parent value represents the board that came before the current board saved in value. This allows us to track the path from start to goal node. The h value holds the heuristic cost for the current board to the goal node. The g value is the depth of the tree, I will be adding the current path cost plus one every time it's calculated. The f value is automatically calculated within the node class which is just a combination of the g value and h value to get the full cost for each board.

After this, I used the magic method __lt__ which allows me to use less than sorting for f values. This is how the priority queue knows which order to sort the list. I built it by having it compare itself to another node object and returning a boolean value depending on the outcome. I thought this would be the best way because I originally was sorting the list every iteration with a sort and key call which is not as efficient as a priority queue.

B. Board Methods

For my board methods, they pertain to manipulating the boards read in from the text file. The main purpose of these methods is to get the boards into working form. These methods will read in the file from the text file, transform them into a three-by-three array and calculate if they are solvable.

I use two different methods to format the boards from the text file. First, using getBoards it reads in the boards from the text file one by one turning each board into it a list element. Next I call the method boardsToMatrix that takes these list elements and turns them into three-by-three numpy arrays and stores them back into the list. With these two

methods the boards are now in working order to run through the rest of my methods.

How my program figures out if the boards are solvable is by using the method getInvCount. This method intakes the board and figures out if the value i in the board is greater than the value j. If it is, it increases the count, inv_count by one. If the result of inv_count module 2 is zero then it is solvable. If it is not then is it unsolvable. By my method's calculation, all boards but the last one are solvable. The method that outputs to the console is called printSolveableBoards, and it iterates through each board in the list and uses the getInvCount method to output the corresponding results.

C. Heuristic Method

The heuristic method (named heuristic) takes in two inputs, startNode, and goalNode. I originally had my program using the misplaced tile heuristic, however, I saw that a large number of a few nodes were ending up with the same heuristic and lowering the speed. So, I decided to re-code my heuristic to use the Manhattan distance.

The way I decided to do it with a board input of a 3 by 3 array, was to figure out the location of this index in the startNode. I did so by using the np.where function call. This function call calculates the index of the value passed in and saves these indexes to 2 separate variables for x and y. I did the same thing with the goal node, figuring out where the value in the current board is and where it should be in the goal board. For example, i and j equal [2][1] which is holding the value 3. I used np.where to get the index of this value and figure out where in the index of the value 3 is in the goal node, which is [0,2]. I subtract two minus zero plus 1 minus 2 and take the absolute value of each giving me 3 meaning that the 3 is three steps out of place. And continue to add up for all nine values on the board.

D. A Star Method

The aStar method and expandNode method is where all the heavy lifting is done for this algorithm. Within the aStar, we are taking in the starting board and the goal board. We turn the start board into a Node object and append it to the open list which is a priority queue and my open list copy. The reason I have implemented two different types of open lists is that it allows me to search the openList easier. The openList is the priority queue that the algorithm is constantly popping off the lowest f-value score from. The openListCopy is just a normal list that is an unsorted copy of openList.

The next part of the aStar method is a while loop that only stops when the heuristic equals zero. I decided to use this as the termination condition because we already checked if the board is solvable, so it should only stop when it is fully solved. Within the while loop, we pop off the open list and append it to the closed list and expand that node using the expand Node method. Once the while loop is broken it will print out the path, path cost, and the number of states expanded.

E. Expand Node Method

The expandNode method takes five inputs: node, openList, openListCopy, closedList, and the goal. The expandNode method is essentially taking in the current lowest f valued board and gathering all its neighbors but invoking the getNeighbors method. The getNeighbors method is built by finding the index of the zero currently on the board and testing each direction. If the result is in bounds of the board, it is considered a valid board and returns a list of each new valid move to expandNode. The next phase of expandNode is to start a for loop that will loop through all the moves that were returned from getNeighbors. For each new move, it calculates the new g-value cost by adding one to the parent g value cost and creating a new node object. Next, using an if statement, it checks if that new node object is in the closedList meaning if we haven't already explored it, append the new node to openList. The last step is an else if statement saying if the new node is currently in the openList check if the new g value is better than the parent board and if so append the new node and remove the old node from both the openList and openListCopy. The only termination statement I have within expandNode is if the new node's heuristic value equals zero, break the loop. This is so we do not waste time expanding or looking at other possible moves when we are already at the goal node.

F. Why I think its not working and what I would do differently

There are some issues in the code and things I would do differently if I were to re-start the project knowing what I know now. The main issue within the code is, it is not solving the boards fast enough. It solves all the boards below a minute beside the last board [8, 3, 0, 5, 6, 1, 7, 4, 2]. This board solves in close to 16 minutes. This is far too long for an A* search.

The reason I believe my code is taking an extraordinary amount of time to solve the last board is due to the type of list I decided to implement. The openLists and closedLists are lists containing 3 by 3 NumPy arrays. I decided to use NumPy arrays at the beginning of the project because I was not thinking of the efficiency. This was not a good idea because when I am trying to compare my NumPy arrays to see if the new node is in either my open or closed list it slows down exponentially. I have tested by removing as many calls to check my lists as possible and every time I was successful in removing it from my code it increased in speed by minutes. I believe if I used lists instead of 2-d lists containing NumPy arrays, my code would be significantly faster. The efficiency of comparing a list to a list compared to a list containing 3 by 3 matrix NumPy arrays would be much faster resulting in much faster output.

V. Results

The results of my program is as follows:

Program Results Per Board			
Board #	Board	Path Cost	Nodes Expanded
0	0,1,3,4,2,5,7,8,6	4	5
1	1,0,3,4,2,5,7,8,6	3	4

2	1,2,3,4,0,5,7,8,6	2	3
3	1,2,3,4,5,0,7,8,6	1	2
4	1,2,3,4,5,6,7,8,0	0	0
5	3,4,0,5,6,2,7,1,8	20	946
6	1,5,6,3,7,4,0,2,8	20	613
7	8,3,0,5,6,1,7,4,2	28	10413

As you can see the first five boards are very simplistic boards that require very little effort to find the optimal pathing. However, the last three boards are requiring an exponentially larger amount of work from the algorithm. This is because the boards are relying on shuffling the board, the open list, and the closed list to ensure it's on the correct path. I believe my A* is following the optimal pathing even if it is not very quickly once it is used against more complex boards.

VI. Conclusion

This assignment was a very interesting and challenging project for me. It has been three years since I coded A* in undergrad so re-learning A* was interesting. The code I produced has flaws but it does solve the boards. I learned to keep in mind efficiency, the in and outs of A*, and how to manipulate text file read-ins to solve the eight puzzle problems. My code is also able to handle the extra credit portion of the assignment, again it might not be very fast depending on the complexity of the board, but it will solve the board.