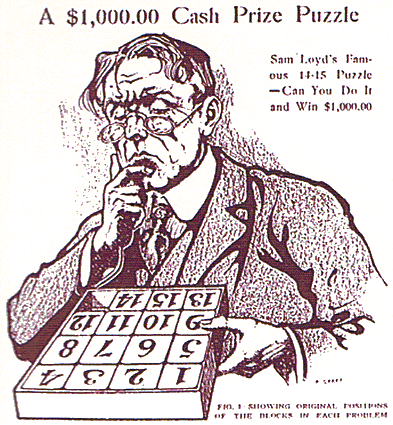
|  |
| --- |
| ITRW317 |
| N-Puzzle Solver |
| An application of A\* on the classic slide puzzle. |
| 21128707 |
| **V.W.F.Mattana** |
| **5/20/2013** |

|  |
| --- |
|  |



Contents

[Introduction 3](#_Toc356818554)

[Method 3](#_Toc356818555)

[A\* Algorithm 4](#_Toc356818556)

[Results 5](#_Toc356818557)

[Conclusion 6](#_Toc356818558)

# Introduction



The aim of this application is to simulate, and solve the 15 puzzle. The 15 puzzle is a puzzle in which 15 tiles are arranged numerically in a square frame, with one tile missing. This allows the tiles to be moved around by sliding them into the open space. This puzzle was first patented by Noyes Palmer Chapman in 1874, and since then has become quite popular, and is often known by other names, such as the slide puzzle, the Gem Puzzle, the Mystic Square, and in other versions, such as the 8 puzzle, where the square is an arrangement of 8 tiles in a square 3 tiles wide.

Interestingly, the tiles are arranged numerically when they are assembled, as this ensures the solve-ability of the puzzle. An amusing turn of events occurred in the 1980’s when Sam Loyd, a popular creator of puzzles and mathematical amusements offered $1000.00 reward to anyone who could solve his version of the 15 puzzle, in which the 14 and 15 tiles were reversed. This was a safe bet for Mr. Loyd, as this arrangement has no solution, along with half of all the possible states of the puzzle.

# Method

The first step toward a simulation of the N-Puzzle is to represent the grid in a data structure. Fortunately this is easily accomplished using a two dimensional array of length and width N. Each element is numbered, with the empty tile being numbered zero. Next the rules of the puzzle (principally mechanical) have to be modelled, and this is again fairly straight forward, in that tiles can only be exchanged with an adjacent empty (zero) tile.

At this stage it is worth noting that two methods of moves exist, the only difference being how moves are counted. In the first method, which I have implemented, one can only move a tile adjacent to the empty tile. The second method allows a slightly more realistic depiction of the puzzle, in that a whole row may be moved as one. This will yield a different count for the number of moves performed, and I personally think that a system where each move is the same is a better gauge of efficiency of an algorithm. As such the move(int z, int[][] array) method receives an integer z, which specifies the tile to attempt to move, and the array in which it should be moved.   
The method then tests if there is an adjacent element with a value of zero, and if so, exchanges the position of those two tiles, and returns the resulting board state as an array.

Then a method was written to find possible moves. This was as simple as finding the empty tile in the array, and returning an array of its adjacent tiles, as all of these tiles will be able to perform a move.

To shuffle the board, a method shuffle(int z, int[][] array) was written which receives the number of random moves that are to be made and the starting position of the grid, as an array. This method calls the method findMoveable(int[][] array), which returns an array of tiles which can be moved, generates a number between 0 and findMoveable(int[][] array).length and selects that element from findMoveable(int[][] array). This tile is them moves, and the process is repeated x times. This ensures that the problem will be solve-able, and not in any of the 15!/2 states that are unsolve-able.

This results in our starting position from which we will attempt to solve the puzzle ( restore the grid to the numerically arranges solution state). A method called solver() is responsible for this task, and will be discussed in more detail in the next section, titles A\* Algorithm, as this is the algorithm it used to resolve our dilemma. There are however a number of functions, and data structures that are used to assist solver() which are going to be discussed now. The first of these is the Heap class which is a Minimum Sorted Heap data structure, which functions as follows.

The Minimum sorted heap is a structure that can be likened to either a priority queue, or a sorted tree. I prefer the tree interpretation, as it makes more sense to me. There are only two conditions for a tree to be a sorted minimum heap. The first is that each node is smaller than the sum of its children, and that the tree is perfectly balanced, and the leaves in the last level are all in the left most position. These rules result in a structure that moves a newly added node to its correct position as soon as it is added, and rebalances itself when the top (smallest) node is removed. This structure is a prefect candidate for the open list, as will be explained shortly.

Other than the heap, there is an array List which is used for the closed list, and as a dynamic list whenever it is needed.

# A\* Algorithm

Solver() uses the A\* algorithm to find the shortest path through the state space from the initial state to the goal state. This is accomplished using the algorithm which will be explained: Firstly, there are two lists required for this algorithm, the open list, which contains states under consideration, and the closed lists which contains only states which have already been investigated. Additionnally there are three functions which will be applied with great frequency in this algorithm. The firs function is H() which determines the sum of each tile’s distance from its goal position. It can be expressed as follows:

The second function is G(), and it simply represents the number of moves since the initial state. The final function is F(), and this function is the sum of H() and G(). This is the basis on which our heuristic makes its choices.

At this stage I feel it is important to note that the states will be stored as arrays with in Node objects that also contain variables for storing H(), G(), and F(). This will save the application many cycles of cpu time when checking The F() values against previously explored nodes.

We begin the algorithm by adding the initial state (our shuffled grid) to the open list after we have determined the function F(startNode). This will be the first element on the open list. We then set The smallest F() scoring Node in the open list as the current node. As mentioned earlier, this is made trivial by using a minimum sorted heap as the data structure governing the open list, as all we have to do is pop() the first element of the open list to obtain the smallest F() scoring Node. After we have this node, we remove it from the open list.

After we have a currentNode set to the currently smallest node in the open list, we obtain the possible moves from this state, evaluate F() on all of them, and perform two tests on them. Currently these tests are trivial since the open list is empty when they are tested, but there is no better time than the present to describe the intricacies of the method. The first test is to ensure that the child node is not already on the closed list. If it is, we discard it, and continue testing at the other children. The second test tests if the child node is not already on the open list WITH a lower G() score. If this is the case, we discard the child node, and set the already existing node’s parent as the current node, and re-evaluate the node’s F(), as well as rebalance the heap. Thus only a node that is not on the closed list and has no superior copies already on the open list will reach this point, the F() value of the node are then evaluated, and it is added to the open list.

We then begin again, and select a new current node from the top of the open list (the smallest F() value).

We only stop this process when we have added our goal state to the closed list, as we will then have the shortest path from our starting state to the goal state.

This algorithm is A\* rather than just A because the heuristic that is used is always optimistic in its estimation of the distance to the goal state.

In summary the algorithm that is used is as follows:

Add start Node to the open list.

* Select the smallest F() valued Node from the open list as the current node.
* Remove the current node from the open list, and add it to the closed list.
* Find all children of the current node, and do the following to all of them:
  + If the child is on closed list, ignore it and continue.
  + If the child isn't on the open list, Make current Node the parent of this child, and record F, G, & H values for this child, & add it to the open list.
  + If the child is on the open list already, compare the G of the node on the open list to the G of this child. If the node on the open list wins (is smaller), move current Node to the parent of the winning Node, and recalculate F of the winning Node. Re-sort open list.
* Finish only if the open list is empty, or the goal state is added to the closed list.

# Results

The algorithm works well, and will find the shortest path to the goal state, but it is far from an efficient algorithm, and it can be optimised in a number of ways. However, it works admirably for its simplicity, and cannot fail if there is a possible solution. It was discovered that for N = 4, the puzzle can take a considerable number of iterations (100000+) to find the optimal solution, and as such it was decided that the 8 puzzle, or the N=3 case will be far more optimal for testing and demonstration purposes. The algorithm finds solutions averaging about 25 moves for the n=3 case, and about 50 moves for the n=4 case. This is consistent with reports of optimal solutions in the literature.

# Conclusion

I believe that the A\* algorithm is a very effective path finding algorithm, and that the heuristic that is selected has a large impact on the efficiency and speed of the algorithm completing. I feel that an iterative deepening version of the algorithm I used will almost certainly work faster than the version I used. Also a better heuristic function, will also improve runtimes. More efficient storage and access of the states can also improve performance. However all of these options will complicate the code significantly, and the essence of the searching of states might very well be lost when explaining the algorithm with this application. A direct application of this algorithm has given me a far deeper insight into the functioning of a state based search than literary descriptions alone.

