Evaluation of A* Algorithm for N x N Sliding Block Puzzles

Christopher Moore

Department of Compter Science and Engineering
University of Louisville
Louisville, USA
{ckmoor04@louisville.edu}

Introduction

In the realm of artificial intelligence and problem-solving, the A* algorithm stands as a cornerstone methodology renowned for its efficiency in finding optimal paths through complex search spaces. This paper delves into the domain of n x n sliding block puzzles, a classic problem that requires strategic maneuvering of blocks within a confined space to achieve a desired configuration. The A* algorithm, a heuristic search algorithm, has proven instrumental in addressing such intricate spatial challenges by intelligently navigating through the solution space, combining the benefits of both completeness and optimality. By scrutinizing the performance of the A* algorithm within the context of n x n sliding block puzzles, this study aims to shed light on the algorithm's effectiveness and provide valuable insights into its application in real-world problem-solving scenarios.

I. The A* Alogrithm

The A* algorithm is a widely used heuristic search algorithm that efficiently finds the shortest path between two points on a graph. It employs a combination of cost evaluation and heuristic estimation as well as to guide the search towards the most promising paths.

A* is an informed, memory-based search algorithm. The informed part refers to evaluation metrics, of which it technically has 3. $g(s)$ is the *genetic* score or traversal score of a state *s*. It is associated with the cost of traversing from the current node to that state. $h(s)$ is the *heuristic* score and will be defined later in section II. The final score is the combination of the previous two: $f(s) = g(s) + h(s)$. This is the metric used by A* to find the shortest path to the goal solution.

The memory-based aspect comes in the form of the *open* and *closed* states lists. The *open* list contains all states reachable from the current state or previous states. That way if during its exploration it reaches a ‘dead-end’ it can refer to other states previously reachable, retaining the information on the path to reach it. The *closed* list contains all the previously visited states. This prevents the algorithm from backtracking or getting caught in loops.

The following pseudo code gives a complete outline of A*

```
Function A*():
  // create initial state
  Init_state = Create_State(g_score=0)
  Init_state.setfscore() //calculate h and f

  //create open and closed lists
  Open = [Init_state]
  Closed = []

  //create list to store solution
  Solution = []

  //Start searching
  Searching = true
  While(Searching and Open is !empty):

       //get the state with lowest f score
       Open.sort(key=f_score)
       Current = open.pop(0)
       Closed.append(Current)

       // Reached Goal
       If(Current == goal):
         Searching = false
         While(Current != null)
              Solution.insert(Current)
              Current = Current.parent

       // Create Children and continue Search
       Else:
         Children = CreateChildren(Current)
         For Child in Children:
           If Child not in Closed:
              Child.setFscore()
              Open.append(Child)
```

That covers the A* algorithm at a top level. The Next section covers how we can implement the algorithm for the NxN sliding block puzzle problem.

II. Sliding Block Puzzles

The traditional sliding block puzzle is the 3 x 3 variation, where there are 8 tiles, numbered 1-8, and one missing tile. The goal is to shift the tiles around the board until the tiles are in order going right to left, top to bottom. Below is a sample of a solved and unsolved puzzle of the 3 x 3 variety:

Solved:

1	2	3
4	5	6
7	8	

Unsolved:

7	1	
6	3	2
5	4	8

[1] discussed how to go about implementing A* for the puzzle and serves as a launching point for our implementation. For A* to be implemented to solve the puzzle, we must define a few things. What is a move from a current state to a new state? What are the evaluation metrics; g-score, h-score, and f-score?

A foolish assumption to make is that to create new scores, simply find which tiles can be moved into the blank space and create new states with that swap. Searching the boards for a tile takes $O(N^2)$ time. To avoid doing this, we treat the blank tile as its own tile and swap it with the neighboring tiles to make new states. This way, the position of the blank can be stored in the state information and avoid the long search for the tile.

For A* to optimize the solution to the puzzle, it needs to know what optimal is. Since the *f* score is a combination of both *h* and *g* score, we will only need to define those. The *g* or 'generation' score is the cost to move from the previous state to this state. For our purposes, we can simply set the *g* score to be 1 plus the *g* score of the parent state. The *h* score is harder to define as it should provide information on how far away or un-optimal the current state is. The Manhattan distance heuristic for the n x n sliding block puzzle represents the sum of the horizontal and vertical distances each tile is away from its goal position. Let's denote the current state of the puzzle as *S* and S_{ij} represents the tile at row *i* and column *j* in the current state. The function $G_i(S_{ij})$ and $G_j(S_{ij})$ return the row and column indices in the goal configuration for S_{ij}. The Manhattan distance heuristic $h(S)$ can be expressed using the following equation:

$$h(S) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \left|i - G_i(S_{ij})\right| + \left|j - G_j(S_{ij})\right| \qquad (1)$$

A* has been implemented to solve the 3 x 3 sliding puzzles before like in [1], but its application is limited to just the 3x3 puzzle. How would the problem be generalized for N x N puzzle variations. Johnson, & Story [2] discovered that for all (N*N)! configurations of the puzzle only half are solvable. For an N x N puzzle, there are $\frac{(N*N)!}{2}$ possible configurations. The possible states are divided by two because not every configuration of the tiles is possible as the blocks can only be slid and not swapped.

Another way to look at the complexity of the problem is to imagine it as a tree, where the children of a node are the possible moves from that state in the tree. For a given configuration we can show the how many children states can be generated from it based on the position of the empty tile. For a 3x3 it looks like this:

2	3	2
3	4	3
2	3	2

For a 4x4:

2	3	3	2
3	4	4	3
3	4	4	3
2	3	3	2

You can see that for every N+1 increase, 4 new cells get added that create 3 children and the number of cells that create 4 children get doubled. The possible new children get doubled from 24 to 48. Jumping from a 3x3 to 4x4 creates 1.046 x 10^{13} more possible configurations of the board, which only gets worse as you continue increasing N. This increase in possible states inflates the time and memory complexity as there are more states to search and to store in the open and closed lists. Will A* still be an effective method for solving these higher order problems?

III. Methodology

An A* implementation is constructed in python to solve the N x N sliding block puzzle problem. The solution can accommodate and N x N solution. A custom class *State* is created that represents a single state in the search. The *State* class provides attributes for *g*, *h*, and *f* score as well as containing a 2-dimensional array of values for the puzzle configuration.

Before a puzzle can be solved, we first must have a solvable puzzle configuration. To do this, a function is created that manipulates the goal configuration into an unsolved one. It does this by shuffling the blank tile in a random direction up to 10,000 times. In the case of the tile being shuffled off the board, the tile remains where it is; that why its *up to* 10,000 times.

Children states are created in the algorithm by checking each of the 4 directions the blank tile could be moved in. If the tile can be moved in that direction without going out of bounds and the board doesn't appear in the *closed* list, a new *State* object is created, and all the attributes are set accordingly before being added to the *open* list.

To test the scalability of the A* algorithm, the run times and solution lengths are collected for puzzles of size N=3 and N=4. 30 problems of size N=3 are tested and the average run time and solution length is collected from those runs. The problem of size N=4 is run once with a slightly modified version that checks to see if the child states board is not already in the *open* list. The reason for this is explained in the next section.

IV. Results

To no surprise, the A* algorithm does exceptionally well with the 3x3 configuration of the puzzle. From the 30 test runs, the average time was barely less than a fourth of a second (0.2240 s) and the average solution length was 23 states (23.33)

When testing the 4x4 configuration, issues arose. The algorithm ran for over an hour before initially crashing the python kernel. This was suspected to be the fault of the *open* list expanding to large for the interpreter to handle as the algorithm was creating to many new states on its way to a solution. A change was made to check the *open* list to make sure the new state was not redundant. This would affect the run time of the algorithm but is hardly relevant when compared to the already long run time as is.

The 4x4 tests takes an obscenely long time to run. What was almost trivial to solve at one order below become impossible. The longest run I had was near 9 hours before crashing and causing a new implementation to be created. The test is left running an a follow up post on the class discussion board will be made on the thread for this project.

V. Conclusion

While A* is a superstar, pardon the pun, on problems of 3x3 complexity, it isn't a feasible solution for problems larger than that without some serious optimizations. Looking online it seems that advancements in providing the shortest solution faster haven't made much progress. The problem is NP-hard for a reason. Areas for improvement / continuation would include:

- Test different Heuristics
    - [1] lists some alternatives.
- Using better alternatives to python lists.
    - Like heaps or ques for *open* and *close*
- Implementation is another language.
    - Being able to use pointers in C could provide huge benefits when searching for states.
- Parallelizing the algorithm.

VI. References

[1] Dhondiyal, D. (2016, May 3). Implementing A-star(a*) to solve N-Puzzle. Insight into programming algorithms. https://algorithmsinsight.wordpress.com/graph-theory-2/a-star-in-general/implementing-a-star-to-solve-n-puzzle/

[2] Wm. Woolsey Johnson, & Story, W. E. (1879). Notes on the "15" Puzzle. *American Journal of Mathematics*, *2*(4), 397–404. https://doi.org/10.2307/2369492