The Eight-Puzzle Solver with A* Algorithm

Jiaqi Xiong jiaqixiong01137@gmail.com
04-November-2018

Table of Contents

Instructed by: Dr Eamonn Keogh eamonn@cs.ucr.edu

Introduction

Some people (including me) in their childhood are fond of games, among which the 8-Puzzle prevails. The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration as shown in the Figure 1 below.

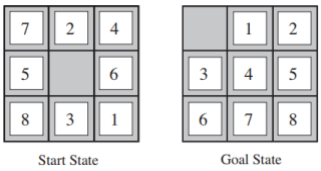


Figure 1: A typical instance of the 8-puzzle.

When it comes to create an intelligent machine to solve this kind of problem automatedly, however, the average solution cost for a randomly generated 8-puzzle instance is about 22 steps and the branching factor is about 3. (When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three.) This means if we use an exhaustive tree search, as what we did with 8-queen, to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states! Thus, we should rethink better algorithms.

The following report presents my findings about A* algorithm through the process of project completion. It explores Uniform Cost Search, and the Misplaced Tile and Manhattan Distance heuristics applied to A*. To compare these 3 algorithms, they were tested using language " c++" in visual studio community 2017, and the full code for the project can be found on my GitHub.

Algorithm

Since combining Uniform Cost Search, where enqueue nodes in order of cost g(n), and Hill Climbing Search, where enqueue nodes in order of estimated distance h(n) to goal, creates optimal, complete and very fast A* algorithm, intuitively, the main idea of A* algorithm is to enqueue nodes in order of estimate cost to goal, that's f(n) = g(n)+h(n). From the project prompt, if we set $h(n) \equiv 0$, then Uniform Cost Search is simply A*.

Observe the following pseud-code general search algorithm:

```
function general-search(problem, QUEUEING-FUNCTION)
nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-
STATE))
loop do
        if EMPTY(nodes) then return "failure"
        node = REMOVE-FRONT(nodes)
        if problem.GOAL-TEST(node.STATE) succeeds
then return node
        nodes = QUEUEING-FUNCTION(nodes,
EXPAND(node, problem.OPERATORS))
```

We can be inspired that when three algorithms: Uniform Cost Search, A* using the Misplaced Tile heuristic, and A* using the Manhattan Distance heuristic are implemented, most procedures are the same except queueing-function

Uniform Cost Search

As discussed above, Uniform Cost Search is just A* with h(n) hardcoded to equal 0, and it will expand the cheapest node, where the cost is the path cost g(n). It also should be noticed that since in this project there are no weights regards to expansion operators, and each expanded node has a cost of 1, Uniform Cost Search here becomes Breadth First search, where the path cost is just the depth.

A* with the Misplaced Tile heuristic

The Misplaced Tile heuristic $h_1(n)$= the number of misplaced tiles. For example, as for Figure 1, not counting the placeholder for the blank tile, all of the eight tiles are out of position, so the start state would have h_1(Start State) = 8. Because apparently any tile that is out of place must be moved at least once, h_1 is an admissible heuristic and its value is the lower the better when applied to the 8-puzzle

A* with the Manhattan Distance heuristic

Although Manhattan Distance Heuristic $h_2(n)$ like $h_1(n)$ also focuses on misplaced tiles, $h_2(n)$ further considers the number of tiles away from goal state position of misplaced tiles, that's h_2 = the sum of the distances of the tiles from their goal positions. Using the same example above again, not counting the position of 'blank', based on their positions in the start state and their goal state positions, g(n) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18. h_2 is also admissible because all any move can do is move one tile one step.

Comparison of Algorithms on Sample Puzzles

There were six puzzles of varying difficulty given to test provided by Instructor as shown in Figure 2.

Trivial	Easy	Oh Boy
1 2 3 4 5 6 7 8 *	1 2 * 4 5 3 7 8 6	8 7 1 6 * 2 5 4 3
Very Easy	**doable**	**IMPOSSIBLE**
1 2 3 4 5 6 7 * 8	* 1 2 4 5 3 7 8 6	1 2 3 4 5 6 8 7 *

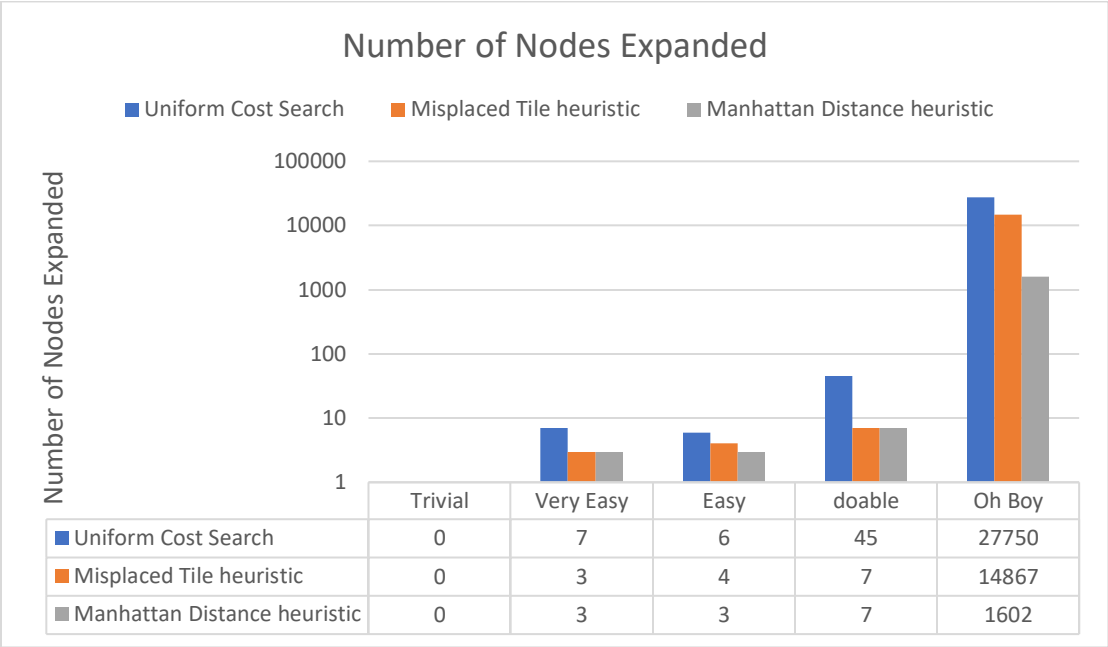
Figure 2: 6 Test cases with different difficulty

The easiest among the six is the trivial puzzle (the initial state being the goal state) and the hardest puzzle is even impossible to solve (the goal state except the position of tiles 7 and 8 swapped).

Since the only meaningful comparisons for the algorithms are time (number of nodes expanded) and space (the maximum size of the queue) from prompt, the Figure 3 and Figure 4 below provide a visual representation of algorithms time and space complexity using the number of nodes expanded and the maximum queue size, respectively.

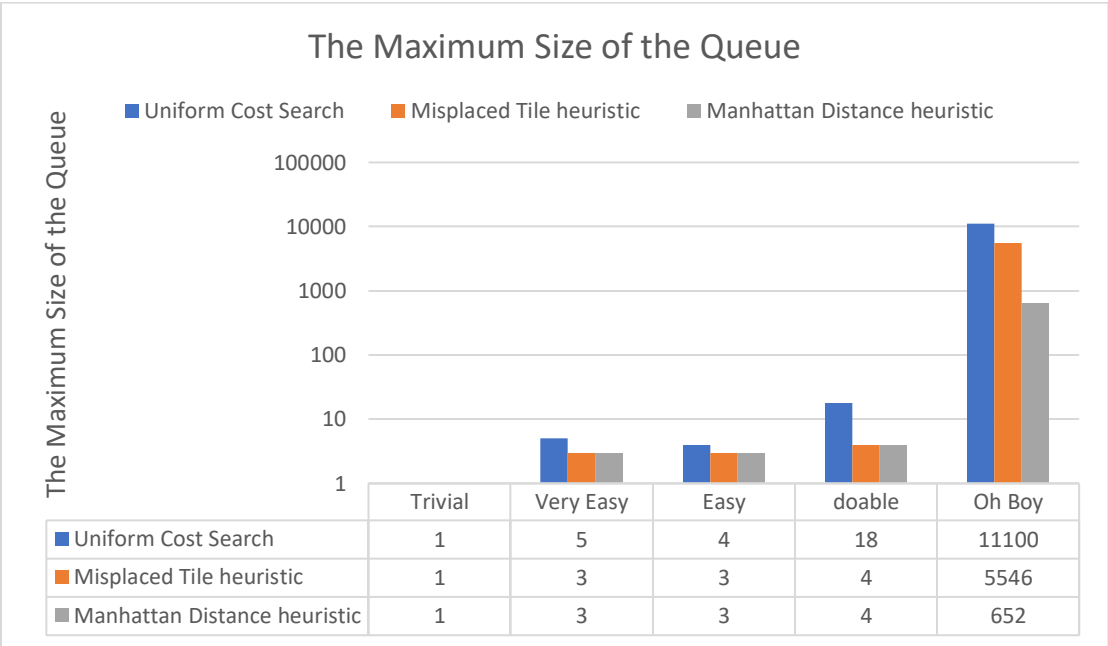
It should be pointed out that my program can know that puzzle impossible is unsolvable, so it does not search that puzzle.

It was found that the easier the puzzle, the difference between the three algorithms relatively more negligible. However, if the puzzle is still solvable but very difficult, then the existence and quality of heuristics make a significant difference in the time and space complexity.



	Trivial	Very Easy	Easy	doable	Oh Boy
Uniform Cost Search	0	7	6	45	27750
Misplaced Tile heuristic	0	3	4	7	14867
Manhattan Distance heuristic	0	3	3	7	1602

Figure 3: the number of nodes expanded of 6 puzzles



	Trivial	Very Easy	Easy	doable	Oh Boy
Uniform Cost Search	1	5	4	18	11100
Misplaced Tile heuristic	1	3	3	4	5546
Manhattan Distance heuristic	1	3	3	4	652

Figure 4: the maximum queue size of 6 puzzles

Conclusion

From what has been discussed and tested above, we can find some features of heuristics from the results:

1. For simple problems, having a (better) heuristic or not does not make a significant difference.
2. However, as the problems get harder, having a heuristic like Misplaced Tiles makes sense while having a better heuristic like Manhattan distance really improve the performance of the solver more.

Applying these features to the listed three algorithms, expectations are consistent with practical results: Among the three algorithms, the A* with Manhattan Distance Heuristic performed best, followed by the A* with Misplaced Tiles Heuristic, and Uniform Cost Search without Heuristic did worst. This can also be regarded as The Misplaced Tile and Manhattan Distance heuristics improve the performance of Uniform Cost Search, which has a time complexity $O(b^d)$ and a space complexity of $O(b^d)$, where b is the branching factor and d is the depth of the solution in the search tree. While both the Misplaced Tile Heuristic and Manhattan Distance Heuristic save the run time and space cost of Uniform Cost Search, it can be found that the Manhattan Distance Heuristic helped more. That's to say, while heuristics will improve the efficiency in both time and space of a blind search, a better heuristic should be chose for better assistance.

Reference:

1. https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/ (for an idea of how to judge whether a puzzle problem is solvable)
2. Russell, S. J., & Norvig, P. (2016). Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,. (for exact definition of the Misplaced Tile and Manhattan Distance heuristics, pseud-code of Uniform Cost Search)
3. Slides "Heuristic Search" and "Blind Search" by Dr Eamonn Keogh (for review of the three algorithms)
4. Sample project report provided by Dr Eamonn Keogh (for understanding high-quality work)
5. http://www.cplusplus.com/reference/queue/priority_queue/ (for reviewing the usage of priority_queue)
6. http://www.cplusplus.com/reference/stack/stack/ (for reviewing the usage of stack)
7. http://www.cplusplus.com/reference/cstring/memset/ (for reviewing the usage of memset)

All the important code is original. Unimportant subroutines that are not completely original are

1. "struct cmp { bool operator() (Node* &a, Node* &b)const { return a->priority > b->priority; } };"
2. "priority_queue<Node*, vector<Node*>, cmp> frontier;"

They are operator overloading which is available at https://stackoverflow.com/questions/14981590/priority-queue-declaration-and-bool-operator-declaration

Appendix: A trace of the Manhattan distance A* on an Example

Here is a trace of the Manhattan distance A* on the following problem:

```
Welcome to JIAQI XIONG's 8-puzzle solver!
Please type “1” to use a default puzzle, or “2” to enter your own puzzle.
1
Please enter your choice of algorithm:
1. Uniform Cost Search.
2. A* with the Misplaced Tile heuristic.
3. A* with the Manhattan distance heuristic.
3
Expanding state...
1 2 3
4 0 6
7 5 8

The best state to expand with a g(n) = 1 and h2(n)= 2 is
1 2 3
4 5 6
7 0 8     Expanding this node...

The best state to expand with a g(n) = 2 and h2(n)= 0 is
1 2 3
4 5 6
7 8 0     Expanding this node...

Congratulations! We succeed arriving the goal!
To solve this problem the search algorithm expanded a total of 6 nodes.
The maximum number of nodes in the queue at any one time was 5.
The depth of the goal node was 2.
```