Evaluating Search Algorithms for Solving n-Puzzle

Sumit Gupta, Sameedha Bairagi
Indiana University Bloomington
{guptasum, sbairagi}@indiana.edu

***Abstract*—Artificial game playing has gathered significant attention in past few decades. Several games are used to evaluate various algorithm. n-puzzle is a classical problem in computer science in evaluating search heuristics which is a central problem in artificial game playing and artificial intelligence in general. In this paper we formulate n-puzzle as an undirected graph problem and evaluate search algorithms with various heuristics. We find out relation between a random puzzle state and its time complexity and also provide evidence that finding a shortest path to an n-puzzle is an NP-Hard problem.**

I. Introduction

Puzzle and games have gathered huge interest of humans from ancient times. From Chess to the ancient game of Go, people have found different techniques and strategies to master these games. With development of computing power in 20th century, people have been trying to teach computers to play (or pretend to play) games. Few decades ago, IBMs Deep Blue mastered Chess when it beat Garry Kasparov [1]. It was estimated at that time that it will be another 100 years when a computer will be able to beat a human expert in Go. However, recent breakthrough in deep learning have had a huge impact on artificial game playing. Recently, AlphaGo [2] beat the world champion Lee Sedol in Go [3]. At the core of these intelligent computers, there are well designed and efficient algorithms which guide them to make moves and choose one optimum path out of several potential paths to achieve its target of winning the game. Motivated by the game playing capabilities of computer systems, we consider the problem of n-puzzle and evaluate its efficiency on various factors.

An n-puzzle is a sliding tile puzzle which is played on a matrix of n x n numbered tiles with one tile missing to allow tile shifts as shown in Figure 1. Figure 1(a) shows the target state of an 8-puzzle where all the number appear in order and empty tile occur at the bottom right corner. A non-target state would be the one where numbers are not in order as shown in Figure 1(b).

The complexity of an n-puzzle increases as n becomes large both in terms of running time complexity and space efficiency. The number of possible state for n-puzzle is n! which makes the search space enormous and intractable.

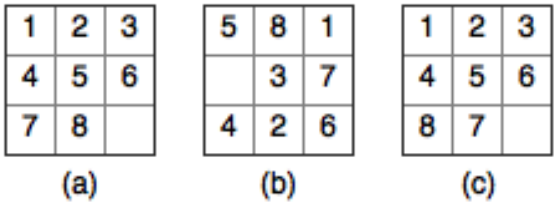

Fig. 1. (a) Target state of 8-puzzle, (b) A non-target state (c) An unsolvable state

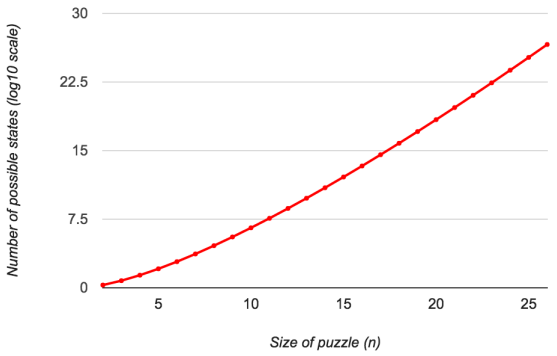

Fig. 2. Number of possible states are n!

Figure 2 shows the number of possible states of an n-puzzle for different n on a log scale. The motivation behind this paper is that despite being a simple puzzle, there is no known efficient algorithm for finding the shortest path between start and target state and the run time can increase significantly with changes in size and initial state.

By the nature of the problem, 8-puzzle can be generalized in smaller and larger puzzles of similar nature. For example, a matrix of 4x4 would be called a 15-puzzle and a matrix of 5x5 would be called a 24-puzzle and so on. In this paper, we try to evaluate various methods to solve a general n-puzzle. We formulate

the problem as a graph problem where every state of the n-puzzle is a vertex. Since there is a target state, we used graph traversal algorithms and evaluate their efficiency. We also implement A* search algorithm with different admissible heuristic functions like Hamming and Manhattan distances.

The contributions of this paper are as follows: (1) we explore relationship between run time complexity and level of randomization of an n-puzzle, size of n (2) we compare several A* search algorithm heuristics on their run time complexity, and (3) we provide evidence that finding the shortest path to an n-puzzle is NP-Hard. We discuss existing solutions in section 2, our approach in section 3, experiments results in section 4 followed by conclusion and future directions in section 5 & 6 respectively.

II. Related Works

n-puzzle is known for more than a century and was popularized by Sam Loyd [4]. n-puzzle has received quite an attention from research community because of its simplicity and power in testing search methods. Hart et al. [5] describes greedy A* best-first search algorithm in which time complexity of the algorithm largely depends on heuristics. For large search spaces, A* is not efficient if a bad heuristic function is chosen. E. F. Moore [6] proposes Breadth First Search and Trmauxs Depth First Search [7] are inefficient in case of huge search space as both of these graph traversal algorithms finishes some part of graph first before proceeding to next part irrespective of the likelihood of finding a solution. On the other hand, A* is a best-first search which intelligently selects the best choice at hand by using a heuristic function in hope to find the optimal solution. Korf [8] used the iterative deepening with depth-first A* (IDA*) to show better results than [5] and eliminating many difficulties in A*. To make [8] more efficient, [9, 10] proposes removing duplicate nodes during the search. [9] also proposes hybrid version of A* search in which they implement real-time search algorithm and linear-space best-first search algorithms. Drogoul et al. [14] proposes distributed approach to solve the puzzle. There are also numerous search heuristics testing literature. Pohl et al. [11] used the 15-puzzle for bi-directional search and dynamic weighting. Korf [12] used these puzzles for the Macro-Operators. Pearl [13] used the 8-puzzle in the Heuristics book as main example for testing heuristics.

III. Approach

We formulate the n-puzzle as an undirected dense graph structure. Consider any state of the n-puzzle, depending on the blank tile, there are at least 2 and at most 4 possible moves. If the blank tile is at any of the 4 corners, there are 2 possible moves. If the blank tile is at the border (except the corners), there are 3 possible moves. There are 4 possible moves in all other cases. If we consider a state of the puzzle as a vertex in a graph, every vertex will have at least 2 adjacent states (vertices) and at most 4. Every state is a vertex in the graph and adjacent vertices represents the possible achievable states from a particular state. Since the target state is unique, a unique vertex is assigned to it like all other states.

After representing the problem as a graph, we consider argument from [11] that not every n-puzzle is solvable. A n-puzzle is not solvable when the target state is unachievable irrespective of number of moves. Johnson et al. [11] used parity argument to show that half of the states in a n-puzzle are not solvable irrespective of number of moves. They used a function which is invariant to valid moves to classify puzzle states into solvable and unsolvable. An example of unsolvable 8-puzzle is given in figure 1 (c).

Since every state in n-puzzle is a vertex in our graph, it is simple to see that it is impossible to reach the target vertex from all other vertices. Hence, graph of all states is disconnected containing many sub-graphs. Only one of the sub-graph will contain all solvable states and the target state. Every other sub-graph will contain all unsolvable states. A simple visualization is given in Figure 3 where we show some sub-graphs of a 3-puzzle. It is important that we determine the solvability of a given state of n-puzzle beforehand otherwise we will waste a large amount of computation in searching for target state which remains unreachable. Figure 4 shows our representation of a sub-graph and its vertices. Every vertex has at least 2 outgoing edges and at most 4.

There are many approaches for traversing a graph particularly in case of n-puzzle. We focus on A* but brief explanations are provided below:

3.1 Brute Force Approach: One simple and naive way to solve the puzzle is to keep moving the blank tile randomly until the target state is achieved. This approach is highly inefficient as it is likely that the target state will never be achieved because of cycles in the graph.

3.2 Row-first Solution: Another naive way to solve the puzzle is to iteratively put tiles into their correct position starting from 1st tile. The problem with this approach is that the tile which are already in their correct position might needed to be moved to make it possible for the next tile to be positioned correctly.

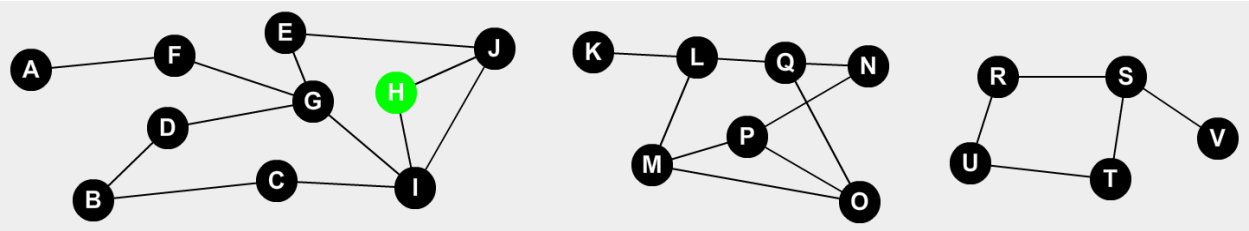

Fig. 3. Graph representation of all possible states. Each node represent a state and Green node represents target state. Note that target state is unachievable from non-solvable states.

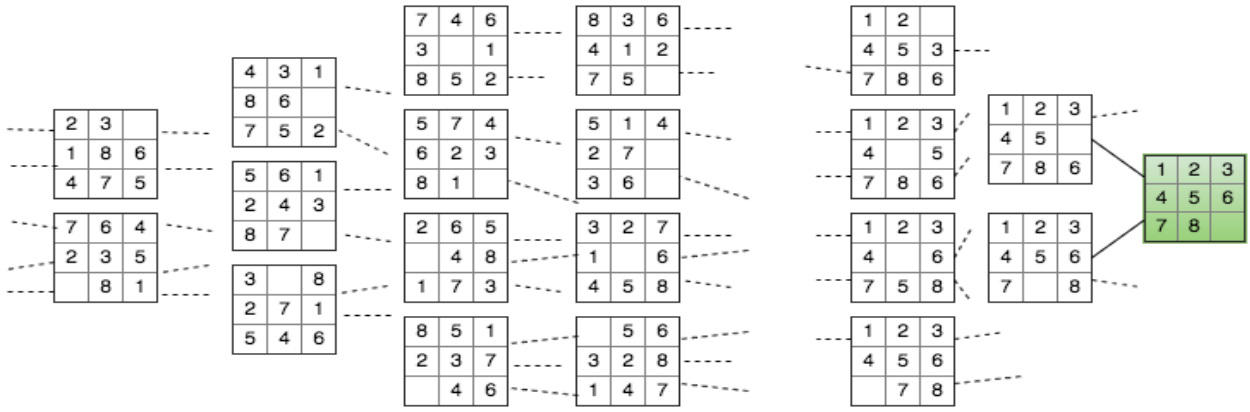

Fig. 4. In depth representation of the graph structure. Green state shows the target state.

3.3 Breadth Depth First Search: breadth first search unintelligently performs search operation in adjacent states first before moving to the next adjacent level of successor states. On the other hand, depth first search exhaustively performs search on a branch before going to the next branch. Both of these searches are inefficient because of their worst case time complexities.

While we are searching for the target state, we need to make sure that we are proceeding in the direction of the target state. Since there are many possible moves, it is difficult to find out which path would lead to the solution more quickly. To overcome this difficulty, we can use greedy algorithms to select which next move we should make in order to come a step closer to the target state.

3.4 A* Search: this approach is widely used in graph traversals where an intelligent decision is required to choose a possibly optimal option out of several other options. It is a type of greedy best-first search and chooses the option which incurs the least cost based on a heuristic function. We use two admissible heuristics: Manhattan distance and Hamming distance.

3.4.1 Manhattan Distance: for an n-puzzle, the Manhattan distance is the sum of distance between tiles and their correct position.

3.4.2 Hamming Distance: for an n-puzzle, hamming distance is the total number of tiles which are not in their correct position. We count the number of tiles which are not in their correct positions. Since we know the tiles and their correct positions from the target state, we can easily calculate this heuristic for each possible move. A* search will select the move which is likely to generate less number of misplaced tiles in future moves.

A* selects the successor state based on the successor function $f(n) = g(n) + h(n)$ where $g(n)$ is the total cost at node n and $h(n)$ is the heuristic function value at node n.

IV. Experiments

We implement our program in Java for experimenting with a large number of input n-puzzles. The program takes a range of n and a randomization number (see below). We create a n x n 2D matrices to store all n-puzzles. To create a good test input, we create an n-puzzle in its target state and move the blank tile for a fixed number of times (randomization number, see below) to generate a random solvable state. We then input this solvable random state to find out

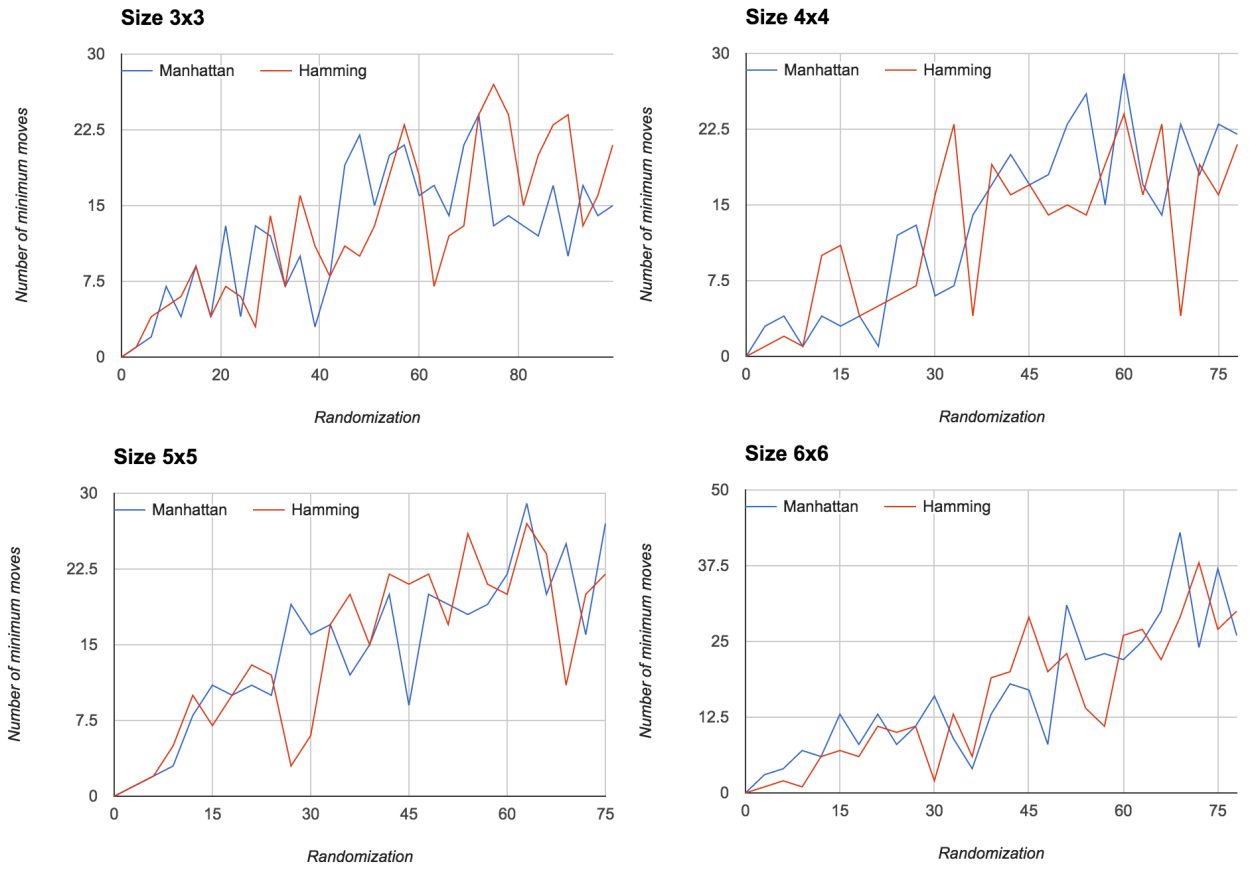

Fig. 5. Results comparing number of minimum moves to degree of randomization (i.e. distance from target state).

running times of various types of search algorithms and heuristics. We test our program by varying many different parameters like the the size and randomization number. We vary a randomization number (say, R) from 1 to 2000. R determines how many times the blank tile is moved randomly to create a random state. Note that a big matrix and smaller randomization will result in an almost solved state. In other words, a smaller randomization number will result in a state for which the corresponding vertex in the graph is near to the target state than the more randomized ones. We record the results in a text file for analysis.

We run our experiments on Macbook Pro with Intel i5 2.7 GHz processor, 8 GB of memory and 128 GB of disk space.

Our results are shown in figure 5, 6 and 7. As shown, we can observe that the number of minimum moves remains in a particular range for a fixed size of n-puzzle. The huge glitches in minimum number of moves are because of the randomness in the state generation. Some states are near even if randomly generated. Figure 6 shows the run time for n-puzzle of various sizes. We can see that upto the randomization factor of 75, the puzzles' solving time fluctuates but after it increases very rapidly. Which confirms that finding the minimum number of moves would be intractable for larger matrices. Figure 7 shows our longest run of a big matrix 99-puzzle. It took several hours on high performance Linux server for 10x10 puzzle to reach solutions for large randomization values.

V. Conclusion

We showed that there is deep relation between randomization of an n-puzzle and its run time complexity. With increase in the matrix size and its randomization, the run time drastically increases along with memory usage. We also showed experimentally that finding the shortest path is not feasible in case of large matrices.

VI. Future Works

Since the game can be generalized to a m x n matrix, it is quite interesting to see the behavior of run

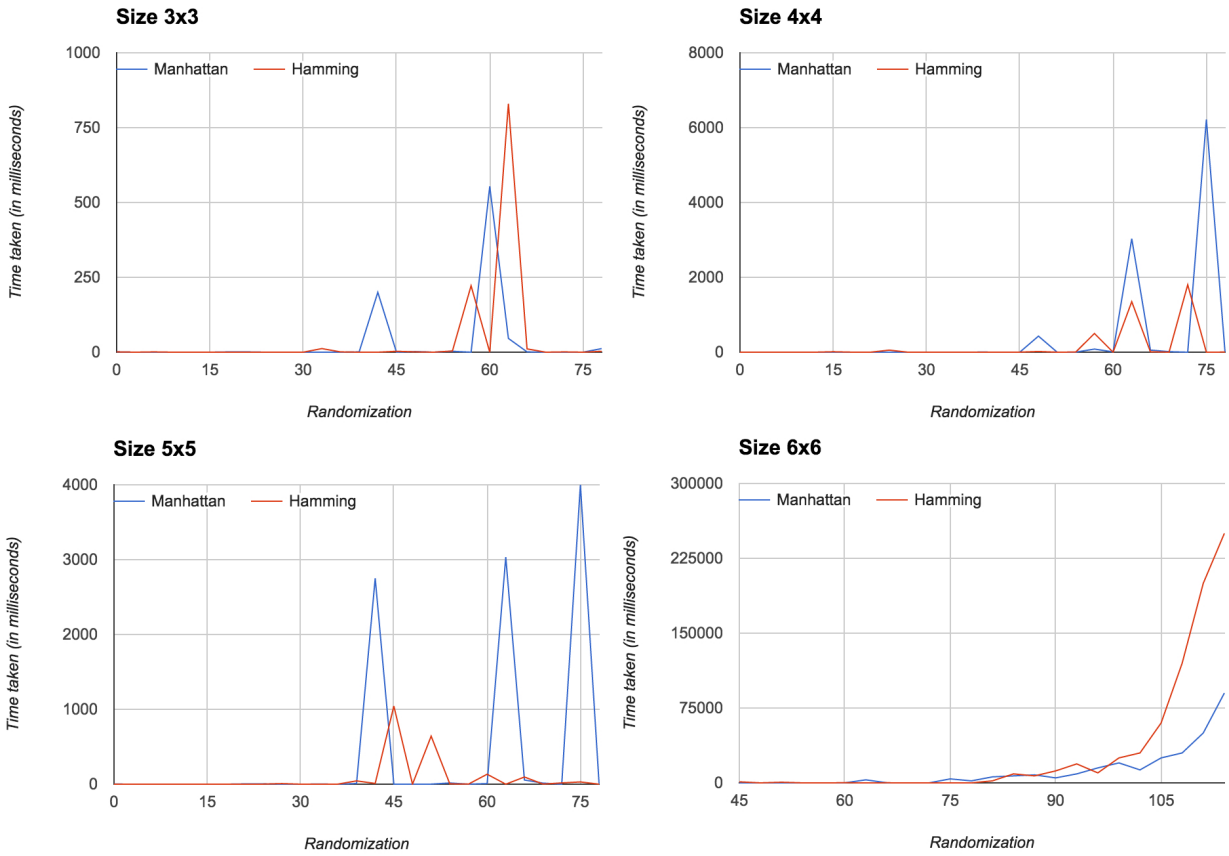

Fig. 6. Time taken to solve n-puzzle vs degree of randomization (i.e. distance from target state).

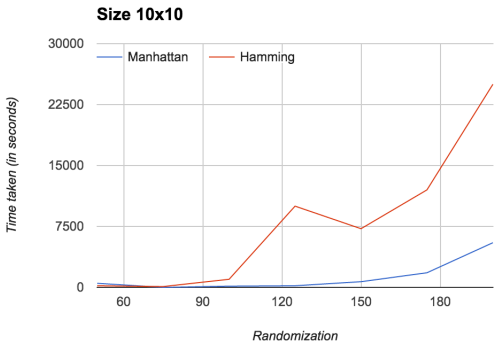

Fig. 7. Run time for 10x10 puzzle

time complexities and performance of various search algorithms in case of m x n where m is not equal to n. We can also explore more advanced but similar games like Rubiks cube and try more advanced techniques like Iterative Deepening A* (IDA*), Fringe Saving A* (FSA*) and Generalized Adaptive A* (GAA*).

We can also try to find the farthest node in the state graph from a solved state. The farthest node will give us the most random solvable state. In other words, we can explore the maximum number of moves required to solve an n-puzzle. We can use parallel programming to solve it more efficiently. For example, for all the possible moves, we can assign each of them to a thread which can efficiently run in parallel.

VII. References

References

[1] IBM. "Deep Blue Overview". IBM Research.
[2] AlphaGo, Google Deepmind https://deepmind.com/alpha-go
[3] AlphaGo, Google Deepmind http://googleasiapacific.blogspot.co.uk/2016/03/alphagos
[4] Sam Loyd, https://en.wikipedia.org/wiki/Sam_Loyd
[5] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics.
[6] Skiena, Steven (2008). The Algorithm Design Manual. Springer. p. 480.
[7] Depth First Search https://en.wikipedia.org/wiki/Depth-first_search
[8] Korf, R. E., Iterative-Deepening-A* : An Optimal Admissible Tree Search Proceedings of the Ninth International Joint Conference on Artijcial Intelligence, Vol. 2, pp. 1034-1035, 1985.
[9] A. Reinefeld and T. Marsland. Enhanced Iterative-Deepening Search, IEEE Transactions on Pattern Analysis and Machine Intelligence, to appear, 1994.

[10] L. Taylor and R. Korf. Pruning Duplicate Nodes in Depth-First Search, AAAI National Conference, pp. 756-761, 1993.
[11] Pohl, I., Practical and theoretical considerations in heuristic search algorithms, in Bernard Meltzer and Donald Michie (editors) ,Machine Intelligence, American Elsevier, New York, 1977.
[12] Korf, R. E., Learning to solve problems by searching for Macro-Operators. Research Notes in Artificial Intelligence 5, Pitman Advanced Publishing Program, 1985.
[13] Pearl, J., Heuristics. Intelligent search strategies for computer problem solving, Addison-Wesley Publishing Company, 1984.
[14] A Distributed Approach To n-puzzle solving. Alexis Drogoul and Christophe Dubreuil.
[15] 15-Puzzle Ideone http://ideone.com/DCirVQ
[16] R. Korf. Real-Time Heuristic Search. Artificial Intelligence, vol. 42, no. 2-3, pp. 189-211, 1990.
[17] R. Korf. Linear-Space Best-First Search. Artificial Intelligence, vol. 62, no. 1, pp. 41-78, 1993.
[18] Johnson, Wm. Woolsey; Story, William E. (1879), ”Notes on the ”15” Puzzle”, American Journal of Mathematics