N-Puzzle Problem

Shubhashis Roy Dipta, Shalima Binta Manir, Nilanjana Das [1]

Abstract

The N-Puzzle problem is a classical sliding puzzle problem. To solve this problem, the number must be rearranged in order. This work aims to explore and analyze different search strategies and compare the runtime, cost, search size, completeness, and optimality. Three categories of search strategies are implemented. They are Uninformed search, Informed Search, and Local Search. Under the Uninformed search, we have implemented Breadth-First Search, Depth-First Search, Iterative Deepening DFS, and Dijkstra's algorithm. Similarly, we have implemented Best-First Search, A* Algorithm, and Iterative Deepening A* as informed searches. Also, Hill Climbing Search is implemented as Local Search. Through analysis, we have shown Iterative Deepening A* algorithm works better for N-puzzle problem. The code has been made public [1].

1. Introduction

The N-puzzle is a sliding puzzle (sli) that consists of a frame made up of numbered square tiles in random order, with one tile open. The goal of the puzzle is to place the tiles in correct order by making sliding actions that use the empty space.

The puzzle was invented by Noyes Palmer Chapman. Sam Loyd popularized the puzzle in the 1870s and created many variations that increased its difficulty. The most common form of the N-puzzle has tiles numbered 1 through 15, with the blank tile represented by 0. The goal is to arrange the tiles in order from 1 to 15 with the empty space in the lower right corner. This can be accomplished by making sliding moves that use the empty space to move adjacent tiles into new positions. This is called a 15-puzzle problem (15p) due to the number of slides involved. The other variation we have explored in this work is the simpler version of 15-puzzle with 8 tiles, called 8-Puzzle problem.

There is an initial state and goal state of this puzzle. Different types of search strategies are used. These strategies are evaluated based on Completeness, Admissibility, Time complexity, and Space complexity. To understand the problem and the solving techniques, some background pieces of knowledge are described below:

Completeness: An algorithm is a well-defined process for converting a random input to a pre-defined goal state. An algorithm is said to be complete if it is guaranteed to produce the correct output for every possible input. That is, if there is a path from point A to point B, then it will always find such a path, regardless of what the actual layout of the points looks like. This property is important because it allows us to trust that the algorithm will always produce the correct results.

Admissibility: Admissibility means if a solution is found, it is guaranteed to be optimal. The admissibility of an algorithm refers to its ability to be used in practice. An algorithm is said to be admissible if it can find the most optimal solution in a finite time. The finite time constraint comes from the factor that it should generate the solution in a feasible time. The definition of feasible time differs from problem to problem.

Correctness: The correctness of an algorithm also needs to be taken into account when determining its admissibility. An algorithm is said to be correct if it produces the correct results for all inputs. However, proving correctness can be difficult and sometimes impossible. As such, most computer scientists rely on testing to verify correctness. Testing can never prove that an algorithm is correct, but it can show that it is correct with high probability.

Generality: An algorithm is said to be general if it can be applied to a wide range of inputs without modification. A specific algorithm is one that can only be applied to a limited range of inputs or requires modification for different inputs. Though there are some cases (i.e., NP Complete problems) where specific input algorithms work better and efficiently than the general ones.

Time Complexity: The time complexity of an algorithm is the amount of time required to run an algorithm on worst

[1]Department of Computer Science, University of Maryland Baltimore County, Baltimore, MD 21250 USA. Correspondence to: <>.

[1]https://github.com/dipta007/N-Puzzle-Problem

case. The time complexity can be measured in terms of the number of operations performed by the algorithm. The time complexity of an algorithm is often expressed as a function $T(n)$, where n is the input size.

Space Complexity: The space complexity is the amount of space it takes on memory to run an input and get the desired result. Like time complexity, space complexity is also measured on basis of input size, n.

2. Related Work

There had been a lot of attempts to solve the N-Puzzle Problem (Morazán, 2022a; Tuncer, 2021; Morazán, 2022a;b; Osaghae, 2018; Dirgová Luptáková & Pospíchal, 2021; Chu & Hough, 2019; Pizlo & Li, 2005; Lee & See, 2022; Igwe et al., 2013; Mathew et al., 2013). Specially, this paper (Tuncer, 2021) applies an algorithm which is meta-heuristic to solve the 15-puzzle problem. The author mentions that the function of linear conflict will produce even more efficacious results. From this we tried three other heuristics besides linear conflict. Markov et al. describes the work in the framework of the N-puzzle game. Their project employs the theme of machine learning to tie together fundamental AI concepts, and their paper also offers a number of educational applications for the N-puzzle problem and the challenges that may occur. Also, Gupta & Bairagi structured the N-puzzle problem as an undirected graph and assessed search strategies using a variety of different heuristics. They mentioned that searching for the shortest path in N-puzzle problem comes under the category of NP-hard. Also, Mathew et al. shows the efficiency of using uninformed search with different heuristics for solving an N-puzzle problem.

3. Experiment

3.1. Dataset

For the dataset, we have used an open source dataset of 8-puzzle-problem and 15-puzzle-problem, they were collected using crowdsource method. We just extracted enough data from the public dataset. We have also published our preprocessed dataset here. [2]

3.2. Method

3.2.1. Breadth-First Search (BFS):

BFS was invented in the late 1950s by Edward F. Moore, who used it to find the shortest paths in graphs. Since then BFS has been used in many problems such as path finding, map direction finding, grid search, flow problem, even in Artificial Intelligence.

In our problem, we have used BFS for finding the shortest path. We have maped our problem in a tree based problem, where the start node is the root of tree and the goal is the destination node. We need to find the shortest path from root to destination. As BFS does a level based traversal, in our experiments we have seen it takes a lot of time to get to the goal node as it has to explore many branches which has no solution at all.

3.2.2. Depth-First Search (DFS):

DFS describes this algorithm as it tries to find a particular node in a graph or a tree data structure. Similar to breadth-first search, this algorithm also begins at the root node of a tree or any random node in case of a graph. Before it can backtrack, this continues to explore down a particular branch till it reaches the node with the desired property. To maintain a track and not to over-explore the same node multiple times, a mapping has been used. DFS is not a complete or optimal algorithm, because it can goes to an infinite loop even after using a visited map.

3.2.3. Iterative Deepening Depth-First Search (IDDFS):

IDDFS is an algorithm that iteratively deepens the depth of the search tree, meaning it goes down each branch as far as possible until it reaches a leaf node. Once it reaches a leaf node, it backtracks to the last branch and continues to explore other branches at that level. It then goes one level deeper and repeats this process. IDDFS is guaranteed to find the shortest path if one exists, but it may not be the most efficient algorithm.

In short, IDDFS is a DFS algorithm, but just with a depth bound. It was introduced to prevent the infinity loop issue of original DFS algorithm.

3.2.4. Dijkstra's algorithm:

For a graph with non-negative edge weights, Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem. The algorithm was proposed by Dutch computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm creates a tree of shortest paths from the starting vertex, the source, to all other points in the graph. It does this by maintaining a set of nodes for which the shortest path from the source has already been found. The algorithm starts from source (root) and each time it takes the neighbor node that has the smallest weight. This algorithm is same as BFS but more efficient as it chooses its next step by the smallest weight. Also, this algorithm works with variable weight, which BFS doesn't. This algorithm is complete and optimal.

[2] https://github.com/dipta007/N-Puzzle-Problem/tree/main/data

3.2.5. GREEDY BEST FIRST SEARCH (GBFS):

GBFS algorithm makes locally optimal choices in the hope of getting a global optimum. An evaluation function $f(n) = h(n)$ is used. This search technique sorts the node in ascending order according to the value of f. The node with the smallest f value will be selected to expand, which means this algorithm selects the chosen node to expand that is appearing closest to the goal. When a child node is generated, its heuristic value is set using either an underestimate or overestimate of the actual cost to reach the goal. If an underestimate is used, then Greedy Best-First Search is guaranteed to find a path to the goal if one exists; however, it may not be optimal. If an overestimate is used, then Greedy Best-First Search is not guaranteed to find a path to the goal, but it will expand fewer nodes than if an underestimate was used and so it may be faster. This algorithm is not complete and also not admissible.

3.2.6. A* SEARCH ALGORITHM:

A* search algorithm is a pathfinding algorithm. It is an informed search algorithm, as it uses information about the environment and the goal to guide its search. The algorithm expands nodes in the tree by considering the cost of the path from the start node to the current node, as well as the estimated cost of the path from the current node to the goal node. Same as GBFS, A* also uses an evaluation function $f(n) = g(n) + h(n)$. Here, g is the cost from source to current node and h is the estimated cost from current node to goal. This h value is found using a heuristic function. This can be done using information about the environment, such as distance or terrain. The heuristic function we choose is usually related to the types of problems. This algorithm is complete and admissible.

3.2.7. ITERATIVE DEEPENING A* (IDA*):

The IDA* search algorithm is a variant of the A* search algorithm. It is more suitable for use in problems with large state spaces. The IDA* algorithm works by iteratively deepening the depth of the current search while keeping track of the best path found so far. The basic idea behind IDA* is to start from a shallow depth and gradually increase the depth limit until a goal state is reached. At each iteration, the algorithm first checks if the current state is a goal state. If it is, then IDA* has found a path and returns it. If not, then IDA* generates all the successor states of the current state and adds them to a list. It then sorts this list in order of increasing cost, so that the least costly successor state is expanded first. Finally, IDA* calls itself recursively on each successor state with an incremented depth limit until a goal state is found, or the maximum depth limit has been reached. There are two main benefits of using IDA*. These are:

1. IDA* can often find a path faster than A*, since it does not have to explore the entire state space.

2. IDA* uses much less memory than A*, since it only needs to keep track of states at each depth level, rather than all states in the entire state space.

There are also some disadvantages:

1. The time complexity of IDA* can be higher than A*, since it may need to iterate multiple times before finding a goal state.

2. The worst-case time complexity of IDA* is $O(b^d)$, where b is the branching factor and d is the depth of the goal state, whereas the worst-case time complexity for A* is only $O(b^d/2)$. Though in practice it finds the solution a lot earlier than the worst case.

3.2.8. HILL CLIMBING SEARCH:

Hill Climbing (HC) search is an algorithm that can be used to find the local optimum in a given function. The algorithm works by starting at a random point in the search space and then iteratively moving to the next point that is closer to the optimum. The algorithm terminates when it reaches a point where no further improvements can be made. The hill climbing search algorithm has a number of advantages over other optimization algorithms.

1. It is relatively simple to implement and understand. Second, it is efficient in terms of both time and space complexity.

2. It often converges to the global optimum, making it one of the most reliable optimization algorithms available.

There are a few disadvantages of hill climbing search as well.

1. It can get stuck in local optima, never finding the global optimum.

2. It can be slow to converge if the search space is large or complex.

3. It does not guarantee that an optimal solution will be found; it only guarantees that a locally optimal solution will be found. So this search algorithm is not complete.

Despite its disadvantages, hill climbing search remains one of the most popular optimization algorithms due to its simplicity and effectiveness.

Figure 1. Manhattan Distance

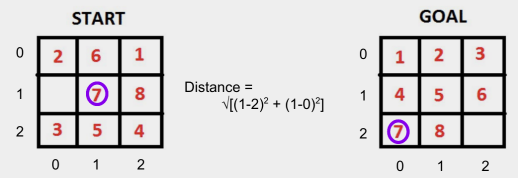

Figure 2. Euclidian Distance

Figure 3. Hamming Distance

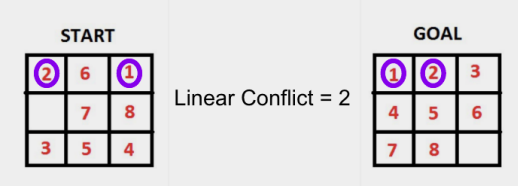

Figure 4. Linear Conflict

3.3. Heuristics

3.3.1. Manhattan Distance

The Manhattan distance is a metric in a normed vector space, where the distance between two vectors is the sum of the lengths of the projections of the vectors onto the coordinate axes. In other words, it is the distance between two points in a grid based on a strictly horizontal and/or vertical path, as opposed to the diagonal. In mathematical terms, Manhattan distance can be written as: $MD(x_i, y_i) = \sum_{i=1}^{n} |x_i - x_{i+1}| + |y_i - y_{i+1}|$, where x_i and y_i are coordinates in n-dimensional space and $|x|$ represents absolute value of x.

Figure 1 shows that the Manhattan distance for a random start node to goal node.

3.3.2. Euclidean Distance

The Euclidean distance between two points with coordinates (x_1, y_1) and (x_2, y_2) is given by: $ED = sqrt((x_2 - x_1)^2 + (y_2 - y_1)^2)$, where $sqrt(x)$ denotes square root of x. This formula can be extended to any number of dimensions n as follows: $distance = sqrt(\sum_i^n ((x_i - y_i)^2))$

Figure 2 shows that the Euclidean distance of a particular tile is given by the formula $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ where x_1 and x_2 are the tile's current and goal state row positions, respectively. y_1 and y_2 are the tile's current and goal state column positions, respectively.

3.3.3. Hamming Distance

In our problem, hamming distance is the number of tiles out of place. To find out of place, we just compare the current board with the goal board.

Figure 3 shows that the Hamming distance is calculated as the number of tiles out of places in the current node.

3.3.4. Linear Conflict

Linear Conflict was first termed by Korf & Taylor. Theoretically, Two tiles a and b are in linear conflict if there are on the target row or column, but they are on opposite side than the goal board configuration.

Figure 4 shows that on the current board tile 2 and 1 are on the same row as goal board but they are on the opposite side compared to the goal board. For each such configuration, 2 will be added to the total cost.

4. Result & Discussion

To analyze our results, we have used three evaluation metrics. The results for each of the metric has been discussed in the following subsections.

4.1. Time Complexity

In this metric, we have found out the time needed for an algorithm to execute. To prevent our algorithms to go into infinite loop, we have bounded the number of nodes explored to 50000. This value was chosen based on experimentation on different values and this value shown to work on most of the cases in a finite time. The result of this metric is reported on Table 1.

From the table we can see that, for 8-puzzle problem, IDA* works best with taking less than 0.01 seconds to find the solution and IDDFS works worst taking around 89 seconds to find the solution on average case.

On the other hand, for 15-puzzle problem, Hill Climbing HC fails to find any solution at all, whereas IDA* finds the optimal solution every time.

4.2. Node Expanded

This metric is somewhat relevant to the previous one, as the number of nodes expanded is proportional to time complexity in most of the cases. The result of this metric is reported on Table 2.

From the table, we infer that, for 8-puzzle problem, A* does best on average cases. But unlike time complexity, IDA* does a little worse than A*. This was assumed as in IDA*

Algorithm	Heuristic	Complete	Optimal	Median	Mean	Worst	Best	Standard Deviation
8 Puzzle Time								
BFS		Yes	Yes	6.59	6.64	15.35	0.03	3.84
DFS		No	No	None				
Dijkstra		Yes	Yes	6.27	6.39	14.39	0.69	3.29
GBFS	Manhattan	No	No	0.06	0.09	0.53	0.0	0.09
	Euclidean	No		0.11	0.22	1.58	0.0	0.26
	Hamming	No		0.04	0.06	0.44	0.0	0.07
	Linear Conflict	No		0.04	0.06	0.45	0.0	0.07
A*	Manhattan	Yes	Yes	0.05	0.08	0.59	0.0	0.09
	Euclidean	Yes		0.11	0.22	1.92	0.0	0.27
	Hamming	Yes		0.04	0.06	0.31	0.0	0.06
	Linear Conflict	Yes		0.04	0.06	0.37	0.0	0.06
IDA*	Manhattan	Yes	Yes	0.05	0.1	0.6	0.0	0.11
	Hamming	Yes		0.04	0.07	0.43	0.0	0.08
	Linear Conflict	Yes		0.04	0.07	0.54	0.0	0.08
IDDFS		Yes	Yes	24.47	82.92	1242.99	0.02	144.29
Hill Climbing	Manhattan	No	No	0.02	0.02	0.03	0.01	0.01
	Euclidean	No		0.03	0.03	0.04	0.01	0.01
	Hamming	No		0.03	0.03	0.06	0.01	0.01
	Linear Conflict	No		0.03	0.03	0.06	0.01	0.01
15 Puzzle Time								
IDA*	Hamming	Yes	Yes	45.95	59.5	195.32	5.52	55.45
	Linear Conflict	Yes		46.5	60.32	196.08	5.47	56.15
Hill Climbing	Manhattan	No	No	None				
	Euclidean	No		None				
	Hamming	No		None				
	Linear Conflict	No		None				

Table 1. Time Complexity of 8-Puzzle and 15-Puzzle solution

visits some initial nodes multiple times due to iterating through depth.

On the other hand, for 15-puzzle problem, hill climbing went to the maximum limit of the nodes but couldn't able to find a solution, whereas IDA* finds a solution though by exploring a lot of nodes.

4.3. Cost

In this metric, we find out if our algorithms can find out the optimal cost, if not whats the difference with the optimal cost. The result of this metric is reported on Table 3.

From the table, we infer that, for 8-puzzle problem, without hill climbing HC and DFS, other algorithms was able to find the solution. Hill climbing could find the solution, but not the optimal one. But DFS couldn't even find the solution due to its infinite loop.

On the other hand, for 15-puzzlem problem, even hill climbing couldnt find the solution at all. But IDA* could every time find the optimal solution.

5. Limitations

In 15-Puzzle, with more than 50 steps, is too long to solve even with pruning and heuristics. It will have $6.3 * 10^{58}$ states w/o any pruning. We were not able to get the cost of 15-puzzle problem for that long path.

6. Future Work

Recently there are some works (Igwe et al., 2013) where Artificial Intelligence based algorithm were used to solve N-puzzle problem. In future, we hope to try those algorithms to solve this problem for $n > 15$. Also, we wish to work on machine learning based heuristics to find out the cost depending on the current environment (board).

7. Conclusion

We have implemented eight and two algorithms for 8 and 15-puzzle problems, respectively. According to the cost tables, out of all the algorithms implemented BFS, Dijkstra, A*, IDA* and IDDFS have minimal cost and low standard deviation in comparison to the other algorithms that have been implemented for both of the variation. And IDA* is also both complete & optimal, we can conclude that IDA* is a promising approach for the n-puzzle prblem.

References

15-puzzle. https://en.wikipedia.org/wiki/15_puzzle.

Sliding puzzle. https://en.wikipedia.org/wiki/Sliding_puzzle.

A*. A* search algorithm.

BFS. Breadth-first search.

Chu, Y. and Hough, R. Solution of the 15 puzzle problem. *arXiv preprint arXiv:1908.07106*, 2019.

DFS. Depth-first search.

Dijkstra. Dijkstra's algorithm.

Dirgová Luptáková, I. and Pospíchal, J. Transition graph analysis of sliding tile puzzle heuristics. In *Recent Advances in Soft Computing and Cybernetics*, pp. 149–156. Springer, 2021.

GBFS. Best-first search.

Gupta, S. and Bairagi, S. Evaluating search algorithms for solving n-puzzle.

HC. Hill climbing.

IDA*. Iterative deepening a*.

IDDFS. Iterative deepening depth-first search.

Igwe, K., Pillay, N., and Rae, C. Solving the 8-puzzle problem using genetic programming. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, pp. 64–67, 2013.

Korf, R. E. and Taylor, L. A. Finding 0 al solutions to the twenty-four puzzle.

Lee, S. C. and See, T. H. Comparing the hamming and manhattan heuristics in solving the 8—puzzle by a* algorithm. In *Proceedings of the 7th International Conference on the Applications of Science and Mathematics 2021*, pp. 189–195. Springer, 2022.

Markov, Z., Russell, I., Neller, T., and Zlatareva, N. Pedagogical possibilities for the n-puzzle problem. In *Proceedings. Frontiers in Education. 36th Annual Conference*, pp. 1–6. IEEE, 2006.

Mathew, K., Tabassum, M., and Ramakrishnan, M. Experimental comparison of uninformed and heuristic ai algorithms for n puzzle solution. *Proceedings of the International Journal of Digital Information and Wireless Communications, Hongkong, China*, pp. 12–14, 2013.

Morazán, M. T. The n-puzzle problem. In *Animated Program Design*, pp. 21–45. Springer, 2022a.

Morazán, M. T. N-puzzle version 6. In *Animated Program Design*, pp. 303–320. Springer, 2022b.

Osaghae, E. An alternative solution to n-puzzle problem. *Journal of Applied Sciences and Environmental Management*, 22(8): 1199–1205, 2018.

Pizlo, Z. and Li, Z. Solving combinatorial problems: The 15-puzzle. *Memory & Cognition*, 33(6):1069–1084, 2005.

Tuncer, A. 15-puzzle problem solving with the artificial bee colony algorithm based on pattern database. *J. Univers. Comput. Sci.*, 27(6):635–645, 2021.

A. Appendix

A.1. Number of Node Expanded Table

Algorithm	Heuristic	Complete	Optimal	Median	Mean	Worst	Best	Standard Deviation
8 Puzzle Nodes Expanded								
BFS		Yes	Yes	93951	96874.65	180992	447	53208.7
DFS		No	No	None				
Dijkstra		Yes	Yes	94454	96406.45	181142	391	53030.4
GBFS	Manhattan	No	No	706	1182.07	6836	13	1238.63
	Euclidean	No		1372	2573.79	16019	13	2928.34
	Hamming	No		384	656.2	4283	13	705.91
	Linear Conflict	No		384	656.2	4283	13	705.91
A*	Manhattan	Yes	Yes	693	1172.08	6766	13	1215.57
	Euclidean	Yes		1386	2597.5	16760	13	2982.37
	Hamming	Yes		372	626.12	3023	13	604.4
	Linear Conflict	Yes		372	626.12	3023	13	604.4
IDA*	Manhattan	Yes	Yes	2646	4532.49	24840	20	5090.22
	Hamming	Yes		1311	2308.65	12408	20	2517.55
	Linear Conflict	Yes		1311	2308.65	12408	20	2517.55
IDDFS		Yes	Yes	1985063	6060769.26	80351599	1323	9951549.35
Hill Climbing	Manhattan	No	No	5	5.36	7	3	1.17
	Euclidean	No		5	5.57	9	3	1.14
	Hamming	No		6	5.84	9	3	1.19
	Linear Conflict	No		6	5.84	9	3	1.19
15 Puzzle Nodes Expanded								
IDA*	Hamming	Yes	Yes	914690	1238726.08	4125350	107975	1161134.13
	Linear Conflict	Yes		914690	1238726.08	4125350	107975	1161134.13
Hill Climbing	Manhattan	No	No	None				
	Euclidean	No		None				
	Hamming	No		None				
	Linear Conflict	No		None				

Table 2. Number of nodes expanded for 8-Puzzle and 15-Puzzle solution

A.2. Cost Table

Algorithm	Heuristic	Complete	Optimal	Median	Mean	Worst	Best	Standard Deviation
8 Puzzle Cost								
BFS		Yes	Yes	22	22.18	29	9	3.28
DFS		No	No	None				
Dijkstra		Yes	Yes	22	22.18	29	9	3.28
GBFS	Manhattan	No	No	22	22.31	30	9	3.35
	Euclidean	No		22	22.2	29	9	3.28
	Hamming	No		22	22.29	29	9	3.31
	Linear Conflict	No		22	22.29	29	9	3.31
A*	Manhattan	Yes	Yes	22	22.18	29	9	3.28
	Euclidean	Yes		22	22.18	29	9	3.28
	Hamming	Yes		22	22.18	29	9	3.28
	Linear Conflict	Yes		22	22.18	29	9	3.28
IDA*	Manhattan	Yes	Yes	22	22.18	29	9	3.28
	Hamming	Yes		22	22.18	29	9	3.28
	Linear Conflict	Yes		22	22.18	29	9	3.28
IDDFS		Yes	Yes	22	22.18	29	9	3.28
Hill Climbing	Manhattan	No	No	22	22.36	32	9	6.57
	Euclidean	No		23	23.39	40	9	5.93
	Hamming	No		25	24.3	38	9	6.02
	Linear Conflict	No		25	24.3	38	9	6.02
15 Puzzle Cost								
IDA*	Hamming	Yes	Yes	42	42.12	49	34	3.96
	Linear Conflict	Yes		42	42.12	49	34	3.96
Hill Climbing	Manhattan	No	No	None				
	Euclidean	No		None				
	Hamming	No		None				
	Linear Conflict	No		None				

Table 3. Cost of 8-Puzzle and 15-Puzzle solution