Laboratory Project for a search course in artificial intelligence - with Prof. Ariel Felner

IDA* vs. A*

8 Puzzle Game

Radwan Ganem
322509951
radwan@post.bgu.ac.il
054-7236777

Introduction

8 Puzzle Game: The 8 puzzle is a sliding puzzle. It has 8 square tiles numbered 1 to 8 in a frame that is 3 tile positions high and 3 tile positions wide, with one unoccupied position. Tiles in the same row or column of the open position can be moved by sliding them horizontally or vertically, respectively. The goal of the puzzle is to place the tiles in numerical order (from left to right, top to bottom).

For Example:

Initial State

1	2	3
8		4
7	6	5

Goal State

1	2	3
4	5	6
7	8	

More Explanation in this link:

https://www.almabetter.com/bytes/tutorials/artificial-intelligence/8-puzzle-problem-in-ai

You Can also try and play this game here: https://sliding.toys/mystic-square/8-puzzle/

Algorithms

1. **A*** is a graph traversal and pathfinding algorithm, which is used in many fields of computer science due to its completeness, optimality, and optimal efficiency.Given a weighted graph, a source node and a goal node, the algorithm finds the shortest path (with respect to the given

weights) from source to goal. Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Reasearch Institute published the algorithm in 1968. A* has better performance with the help of heuristics. it can be also seen as an extension of Dijkstra's algorithms.

A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until the goal node is reached.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes $f(n) = g(n) + h(n)$, where n is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal. The heuristic function is problem-specific. If the heuristic function is admissible – meaning that it never overestimates the actual cost to get to the goal – A* is guaranteed to return a least-cost path from start to goal.

Algorithm Complexity: let b – average branching factor, d – depth of graph from initial state to goal state

Time Complexity: $O(b^d)$

Space Complexity: $O(b^d)$

Completeness: Yes

Optimality: Yes

Source: https://en.wikipedia.org/wiki/A*_search_algorithm

2. **Iterative Deepening A* / IDA*** is a graph traversal and path search algorithm that can find the shortest path between a designated start node and any member of a set of goal nodes in a weighted graph. It is a variant of iterative deepening depth-first search that borrows the idea to use a heuristic function to conservatively estimate the remaining cost to get to the goal from the A* search algorithm. Since it is a depth-first search algorithm, its memory usage is lower than in A*, but unlike ordinary iterative deepening search, it concentrates on exploring the most promising nodes and thus does not go to the same depth everywhere in the search tree. Unlike A*, IDA* does not utilize dynamic programming and therefore often ends up exploring the same nodes many times. While the standard iterative deepening depth-first search uses search depth as the cutoff for each iteration, the IDA* uses the more informative $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to travel from the root to node n and $h(n)$ is a problem-specific heuristic estimate of the cost to travel from n to the goal. The algorithm was first described by Richard Korf in 1985.

   Iterative-deepening-A* works as follows: at each iteration, perform a depth-first search, cutting off a branch when its total cost $f(n) = g(n) + h(n)$, exceeds a given threshold. This threshold starts at the estimate of the cost at the initial state, and increases for each iteration of the algorithm. At each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold. Like A*, IDA* is guaranteed to find the shortest path leading from the given start node to any goal node in the problem graph, if the heuristic function h is admissible, that is $h(n) \le h*(n)$ for all nodes n, where h* is the true cost of the shortest path from n to the nearest goal (the "perfect heuristic").

IDA* is beneficial when the problem is memory constrained. A* search keeps a large queue of unexplored nodes that can quickly fill up memory. By contrast, because IDA* does not remember any node except the ones on the current path, it requires an amount of memory that is only linear in the length of the solution that it constructs. Its time complexity is analyzed by Korf et al. under the assumption that the heuristic cost estimate h is consistent, meaning that $h(n) \le \text{cost}(n,n') + h(n')$ for all nodes n and all neighbors n' of n; they conclude that compared to a brute-force tree search over an exponential-sized problem, IDA* achieves a smaller search depth (by a constant factor), but not a smaller branching factor.

Algorithm Complexity: let b – average branching factor, d – depth of graph from initial state to goal state

Time Complexity: $O(b^d)$

Space Complexity: O(b * d)

Completeness: Yes

Optimality: Yes

Source: https://en.wikipedia.org/wiki/Iterative_deepening_A*

3. **Misplaced Heuristic:** The number of tiles that are not in their goal position. **Except** the empty tile or zero tile.
4. **Manhattan Distance:** The Manhattan distance heuristic is used not only for its simplicity but also for its ability to estimate the number of moves required to bring a given puzzle state to the solution state. Manhattan distance is simply computed by the sum of the distances of each tile from where it should belong. or the sum of the absolute values of the horizontal and vertical distances of the tiles from their goal positions. **Except** the empty tile or zero tile.

The Purpose of the experiment

in this experiment I will test the hypothesis: on 8 puzzle game IDA* is more memory efficient than A* and A* is faster than IDA* and Heuristic Manhattan distance is better than misplaced. I will test this by comparing the number of vertices, vertices explored, time taken, number of duplicate vertices detected/undetected, and vertices saved in memory through the whole search.

Description of the experiment

Code overview: full code is at the end of the document it include a classes: Game_8_Puzzle, A_Star and IDA_Star and functions for executing and plotting graphs. Also full code with results can be found here: https://colab.research.google.com/drive/1c8Z37rOuRrfo8YOi0kYFRDmUjfZiMcQC?usp=sharing

Experimental Setup

Runs: Each experiment was run on at least 10 initial puzzle states generated randomly to ensure accuracy and few initial state that include easy, medium and hard levels of initial puzzle.

Environment: The experiment were conducted on Google Colab notebook, Python 3 Google Compute Engine Backend, Ram: 12.7 GB.

The 8 puzzle problem was solved using both A* and IDA* algorithms.

Important Note: Not all 8 puzzles are solvable, I will only use solvable puzzles

For an initial state, 8 puzzle is not possible to solve an instance of 8 puzzle if number of invertions is odd in the initial state input.

Source: https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/

Results

Figure 1:

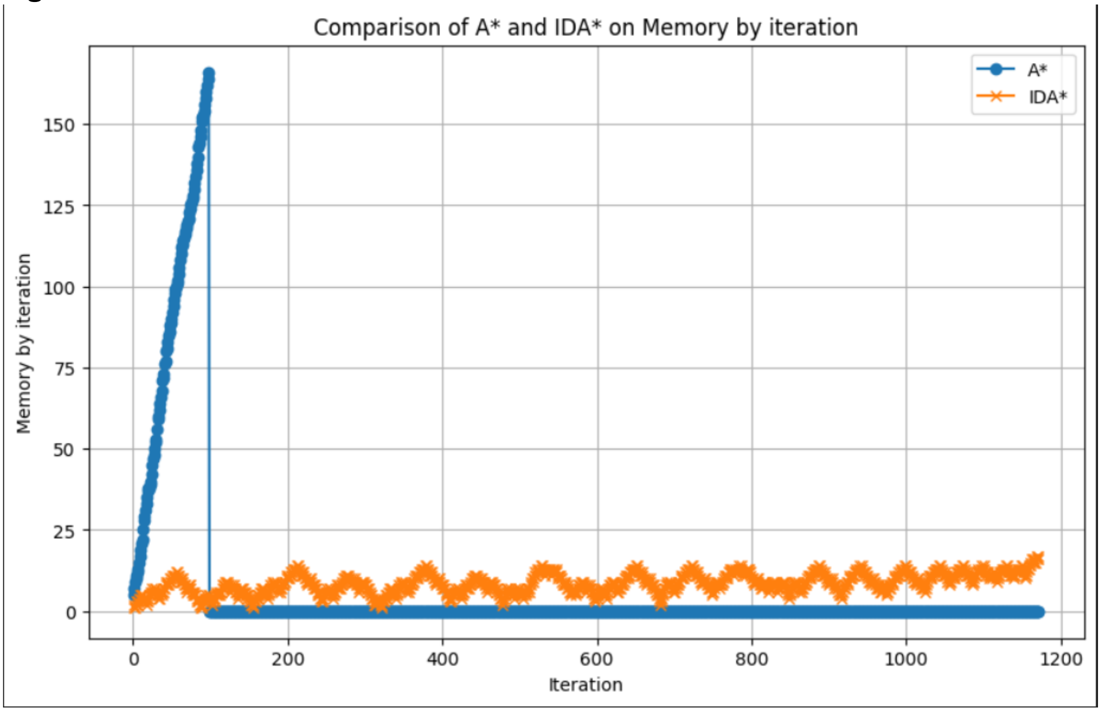


Figure 2:

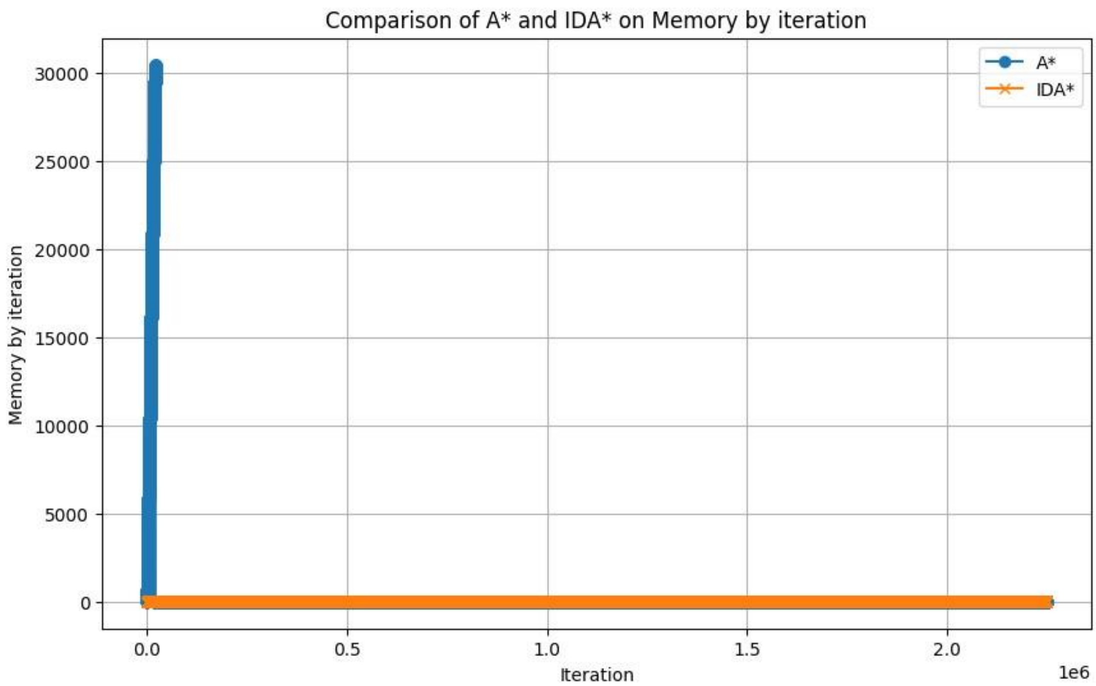


Figure 1 showcase the **behavior of memory usage** by iteration for both algorithms A* and IDA* with Manhattan distance on initial input: [[1,3,5], [7,0,8], [2,6,4]] that it's optimal solution is 16, it can be shown that A* took a lot less iterations to find optimal solution and used a lot more memory than IDA*, Figure 2 is the same but with more extreme initial input: [[6,4,7],[8,5,0],[3,2,1]] that takes the most moves to solve optimally with 31 move this initial input is taken from here: https://www.cs.princeton.edu/courses/archive/spring19/cos226/assignments/8puzzle/checklist.php , it can be shown from the two figures that A* is faster and using a lot more memory as it keeps running through the iterations than IDA* on their path to find the optimal solution for a given initial input.

Figure 3.a:

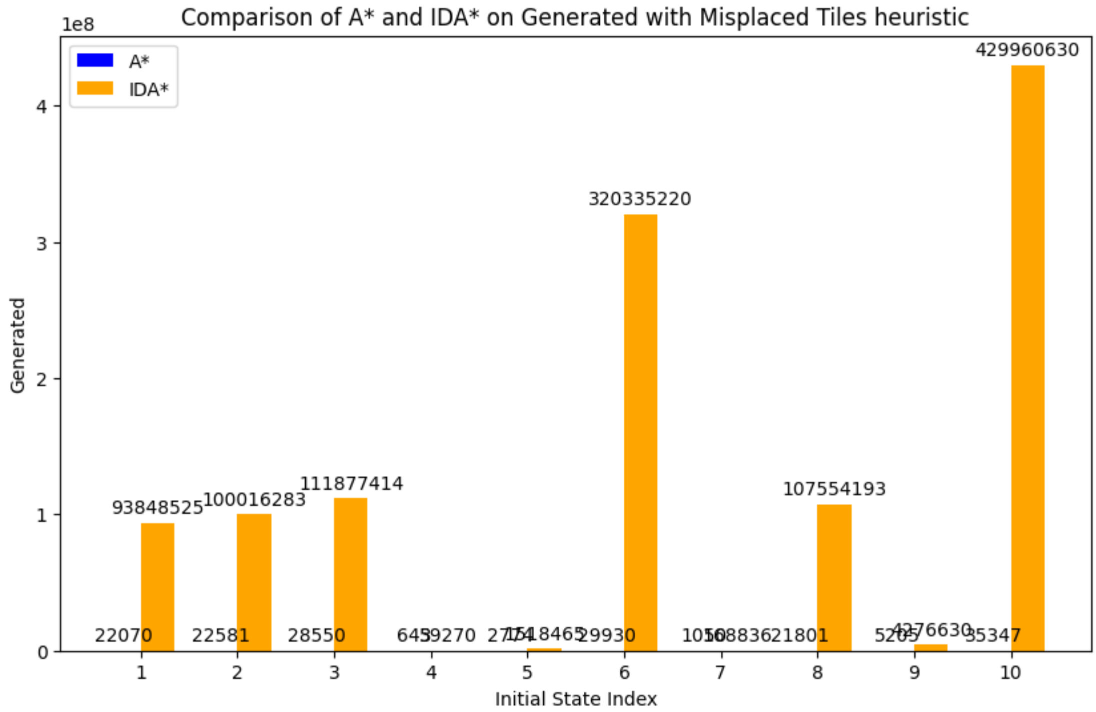


Figure 3.b:

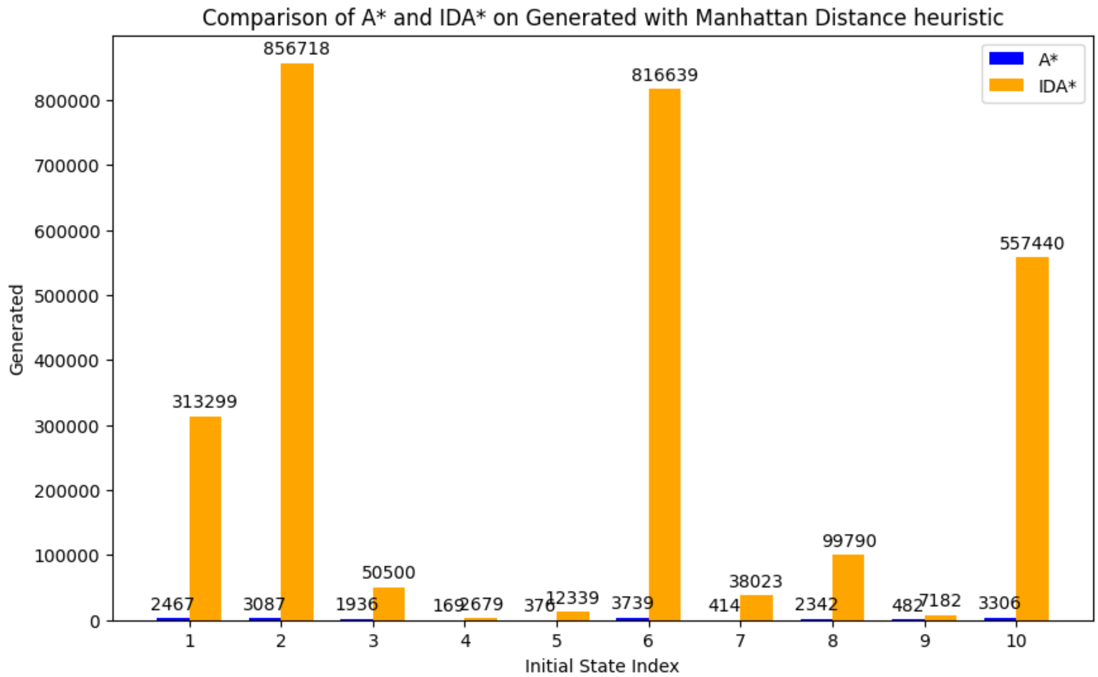


Figure 3.a show the number of Generated Vertices on given 10 initial states with the algorithms A* and IDA* using the misplaced heuristic, and Figure 3.b show the same but with Manhattan Distance Heuristic, both graphs showcase the same 10 initial states puzzle. it can be seen that with A* expand less vertices than IDA* and using Manhattan Distance both algorithms expand less vertices for all initial states.

Figure 4.a:

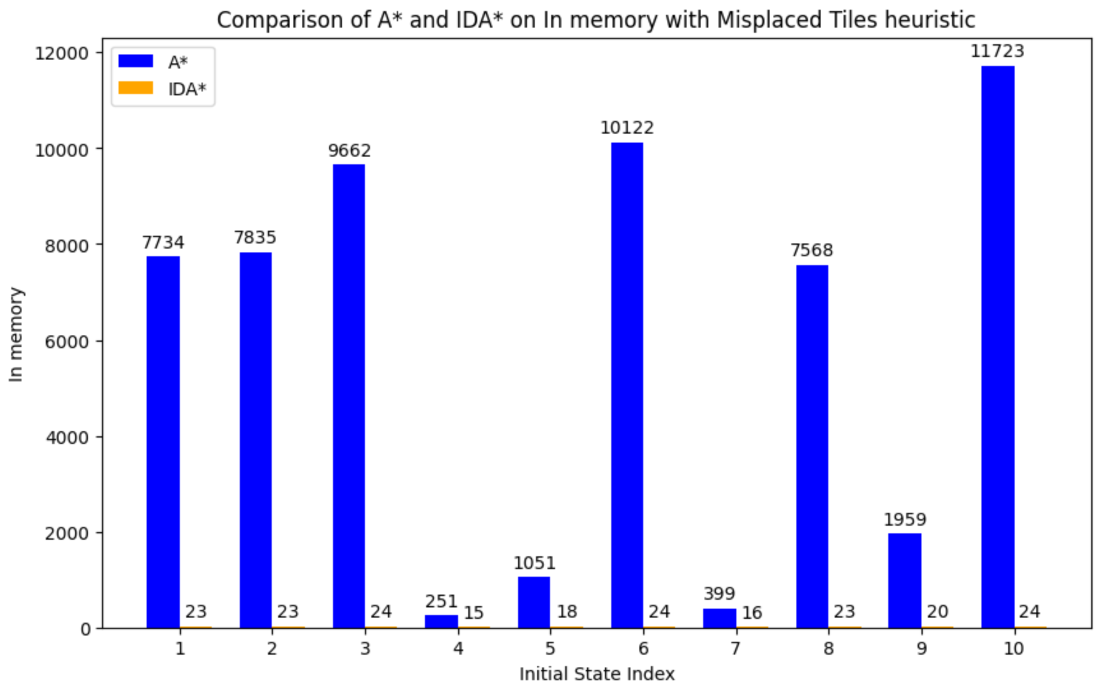


Figure 4.b:

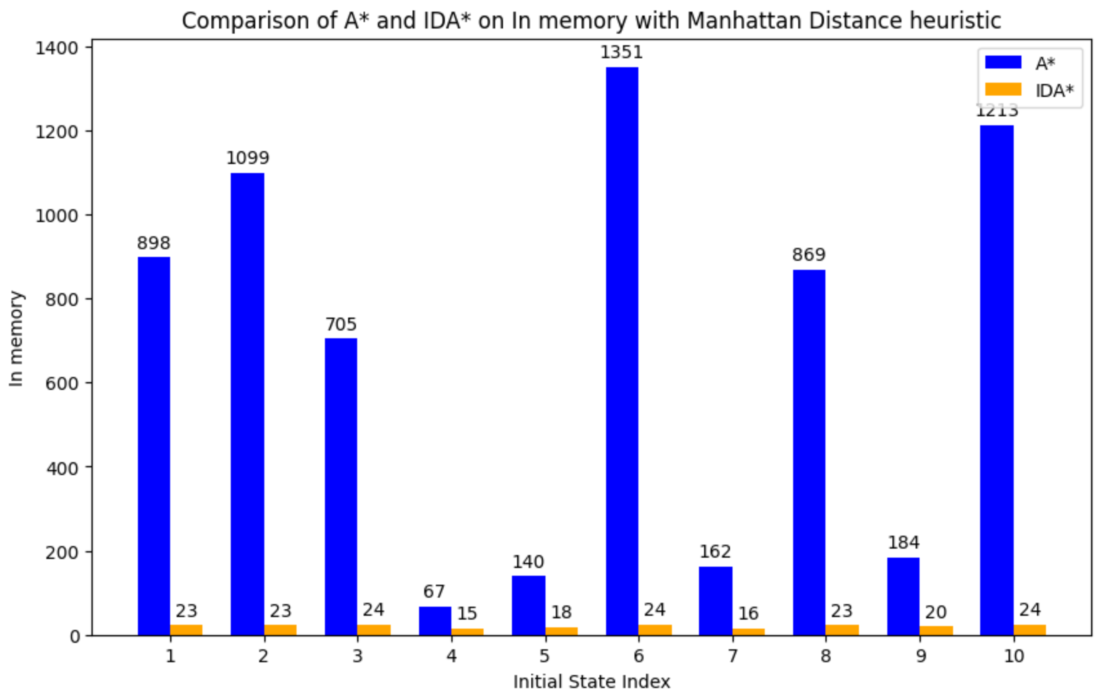


Figure 4.a show the number of in memory (number of states saved until reaching the optimal solution) Vertices on given 10 initial states with the algorithms A* and IDA* using the misplaced heuristic, and Figure 4.b show the same but with Manhattan Distance Heuristic, both graphs showcase the same 10 initial states

puzzle. it can be seen that with A* use more memory than IDA* ,and using Manhattan Distance A* used less memory than using Misplaced and IDA* memory usage stayed similar for both heuristics.

Figure 5.a:

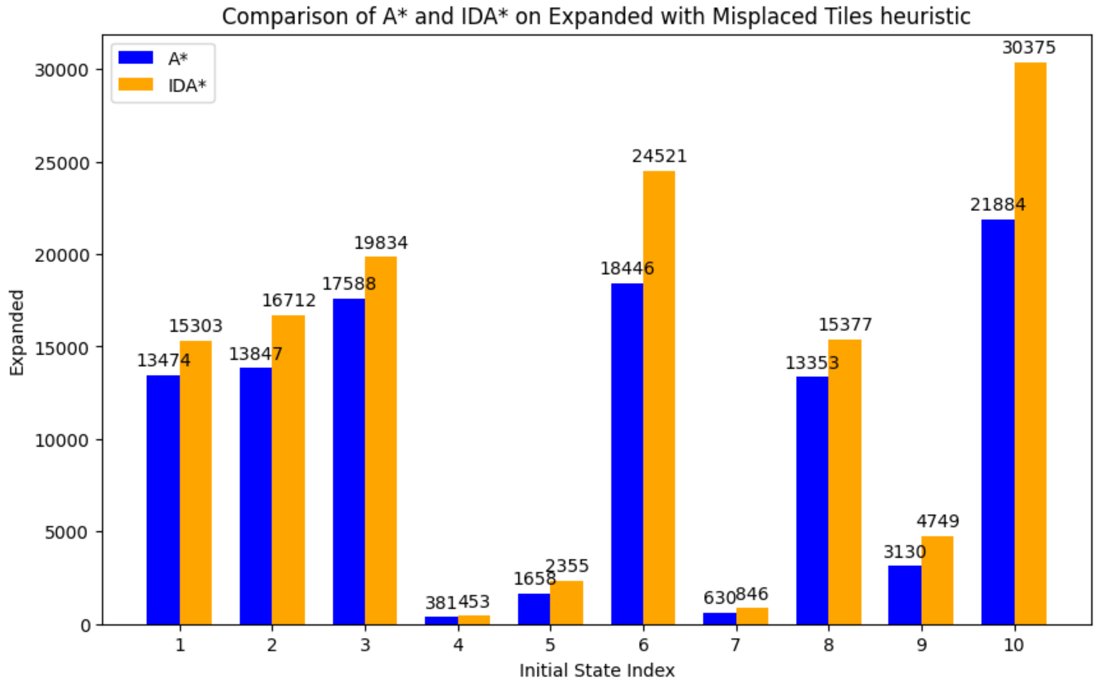


Figure 5.b:

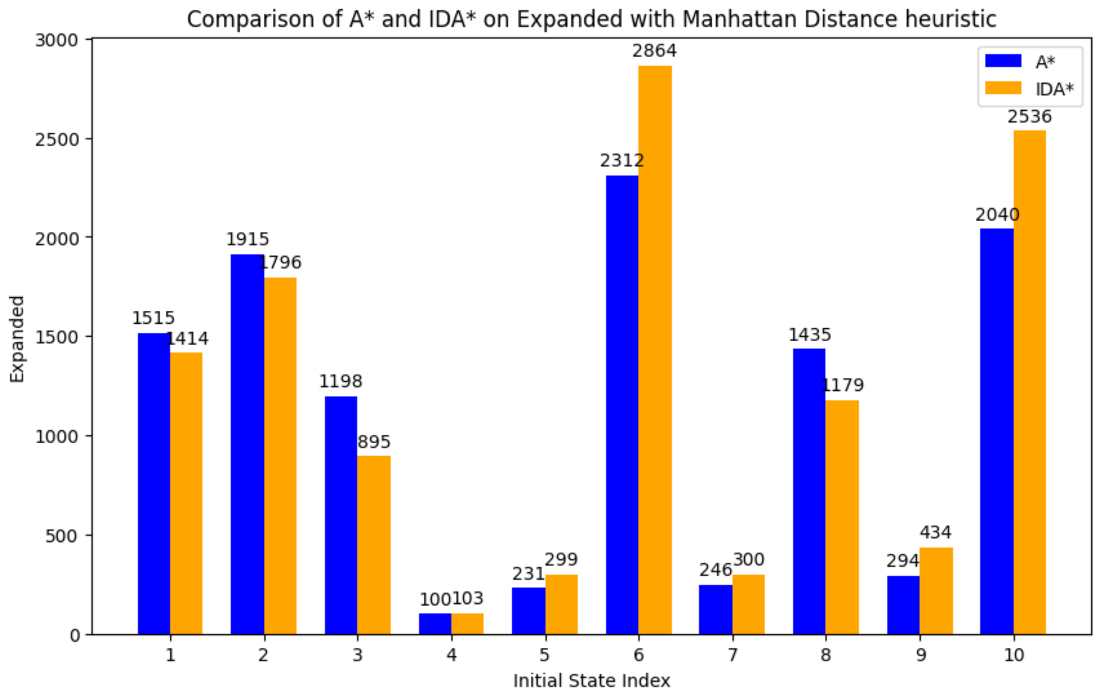


Figure 5.a show the number of Expanded Vertices on given 10 initial states with the algorithms A* and IDA* using the misplaced heuristic, and Figure 5.b show the same

but with Manhattan Distance Heuristic, both graphs showcase the same 10 initial states puzzle. it can be seen that with A* Expanded less vertices than IDA* with Misplaced ,and using Manhattan Distance A* and IDA* both expanded less than using Misplaced and for initial states 1, 2, 3, 8 A* expanded more than IDA* and for initial states 4,5,6,7,9,10 IDA* expanded more than A*.

Figure 6.a:

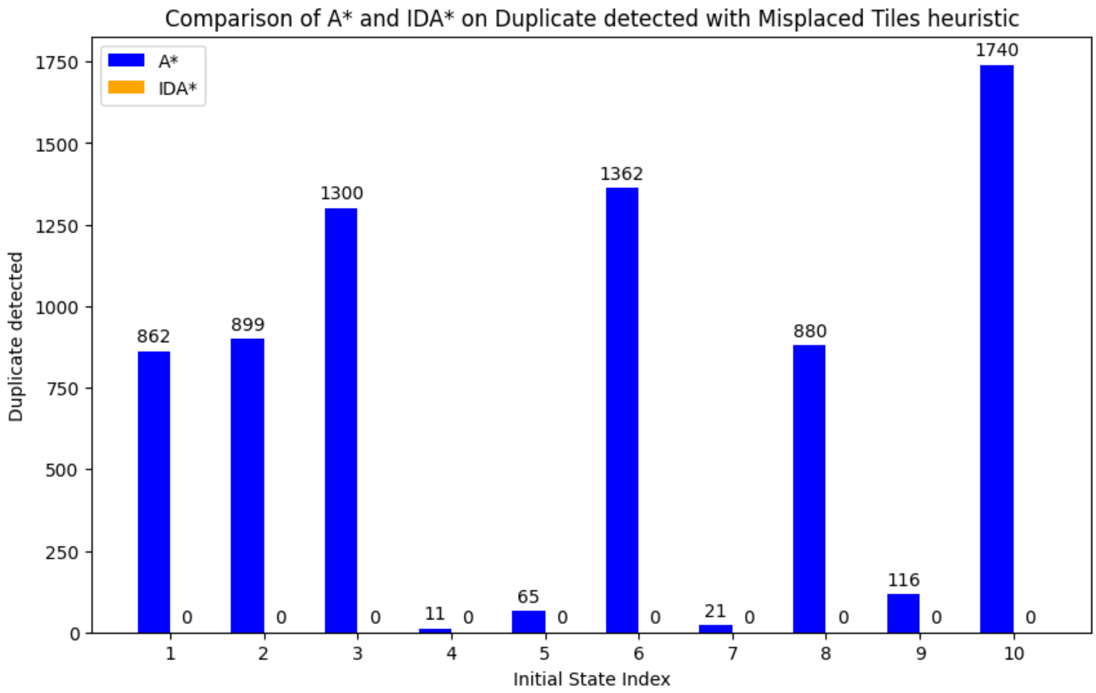


Figure 6.b:

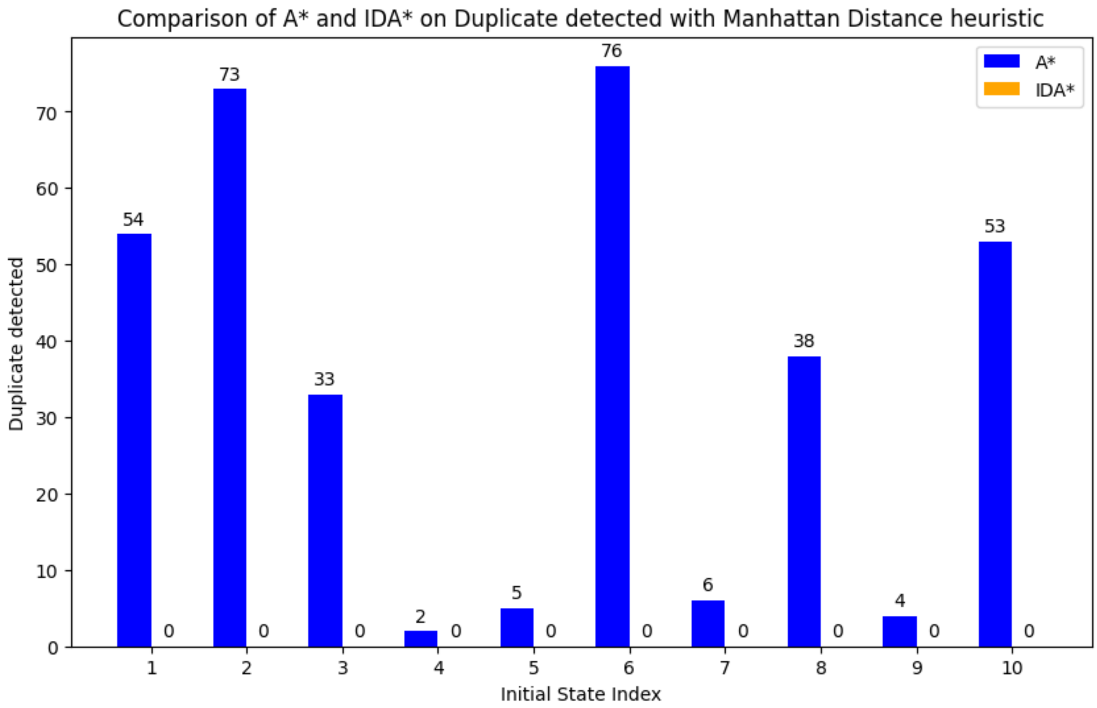


Figure 6.a show the number of duplicate Vertices detected and ignored on given 10 initial states with the algorithms A* and IDA* using the misplaced heuristic, and Figure 6.b show the same but with Manhattan Distance Heuristic, both graphs showcase the same 10 initial states puzzle. it can be seen that IDA* doesn't detect

any duplicates using both heuristics and A* detected on both heuristic ,and using Manhattan Distance A* detected less vertices (encountered less duplicates).

Figure 7.a:

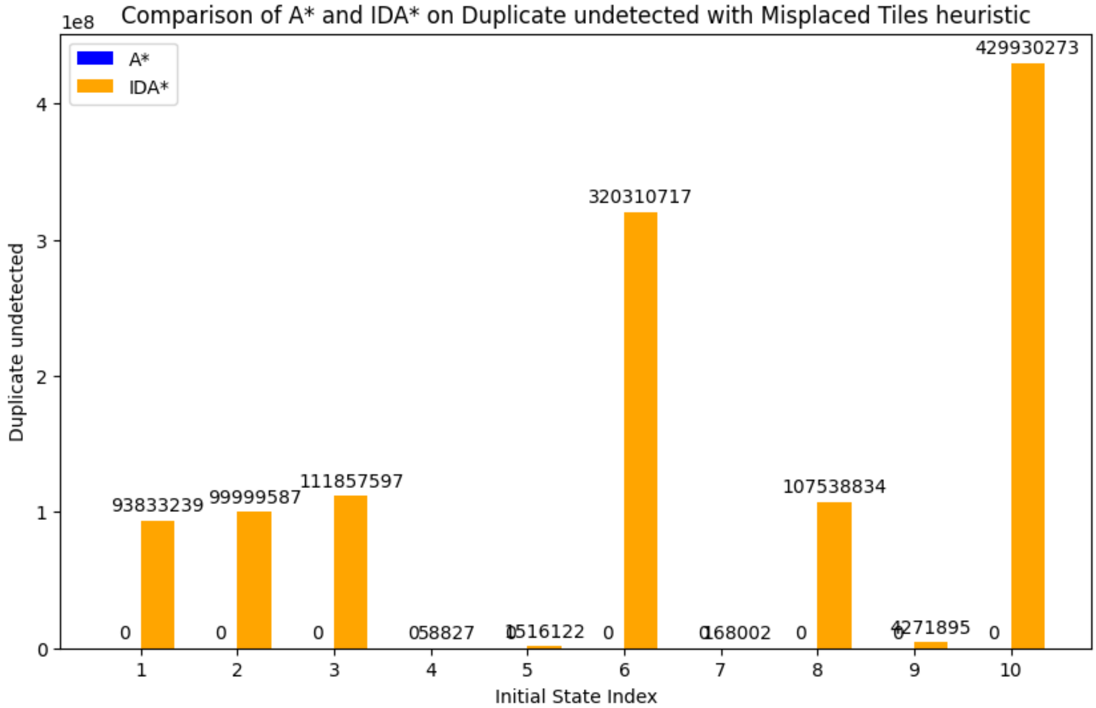


Figure 7.b:

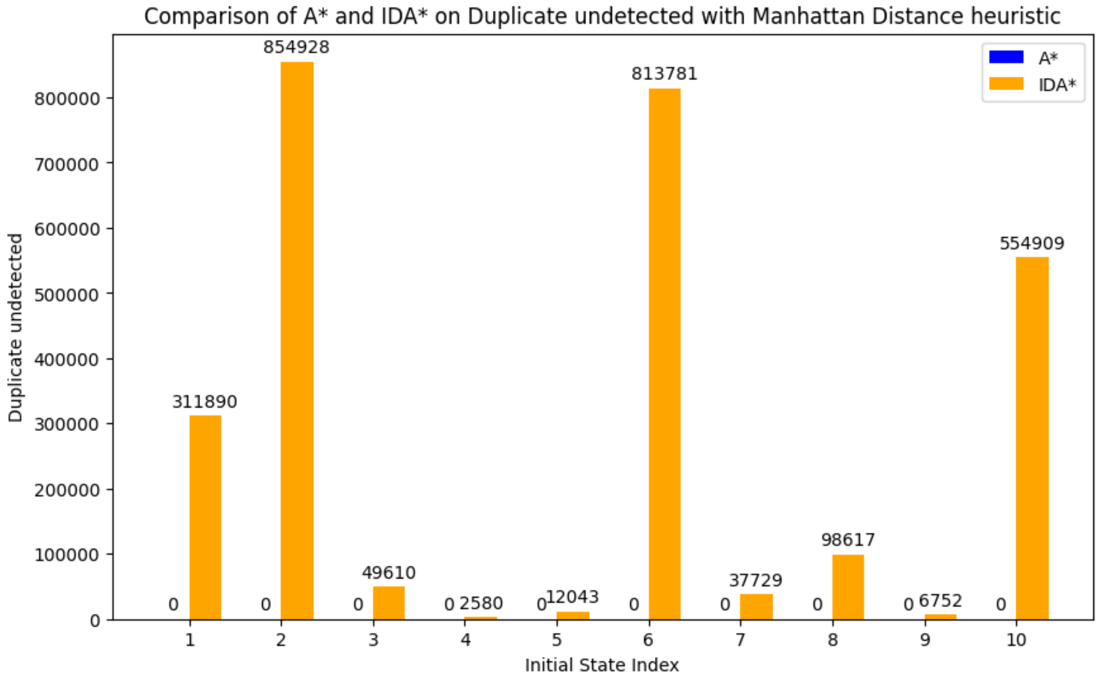


Figure 7.a show the number of duplicate Vertices undetected given 10 initial states with the algorithms A* and IDA* using the misplaced heuristic, and Figure 7.b show

the same but with Manhattan Distance Heuristic, both graphs showcase the same 10 initial states puzzle. it can be seen that IDA* doesn't detect any duplicates using both heuristics and IDA* undetected less vertices using Manhattan distance , A* doesn’t encounter any duplicate nodes that are undetected by the algorithm with both heuristics.

Figure 8.a:

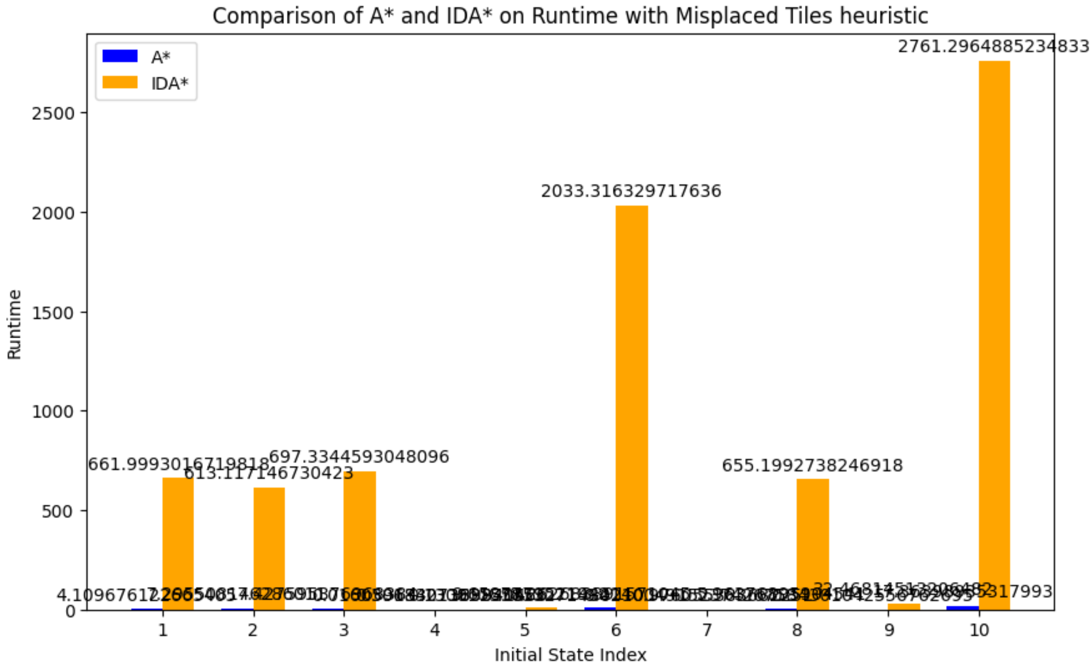


Figure 8.b:

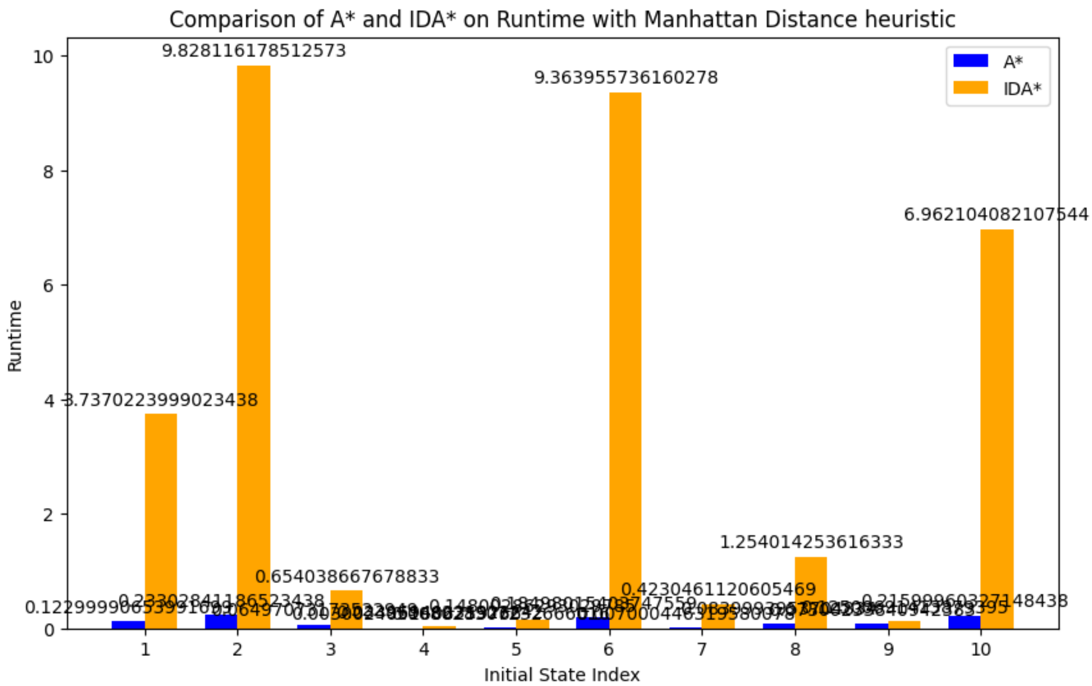
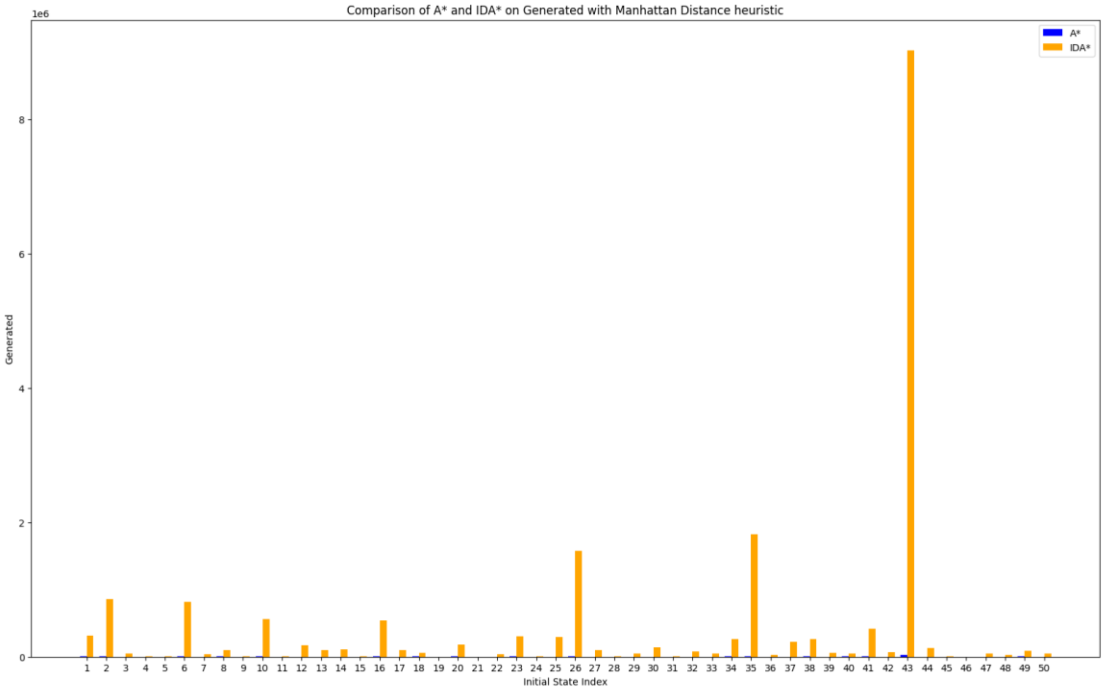


Figure 8.a show the runtime in ms on given 10 initial states with the algorithms A* and IDA* using the misplaced heuristic, and Figure 8.b show the same but with Manhattan Distance Heuristic, both graphs showcase the same 10 initial states puzzle. it can be seen that A* is faster than IDA* using both heuristics and that both A* and IDA* are faster using Manhattan Distance than using Misplaced.

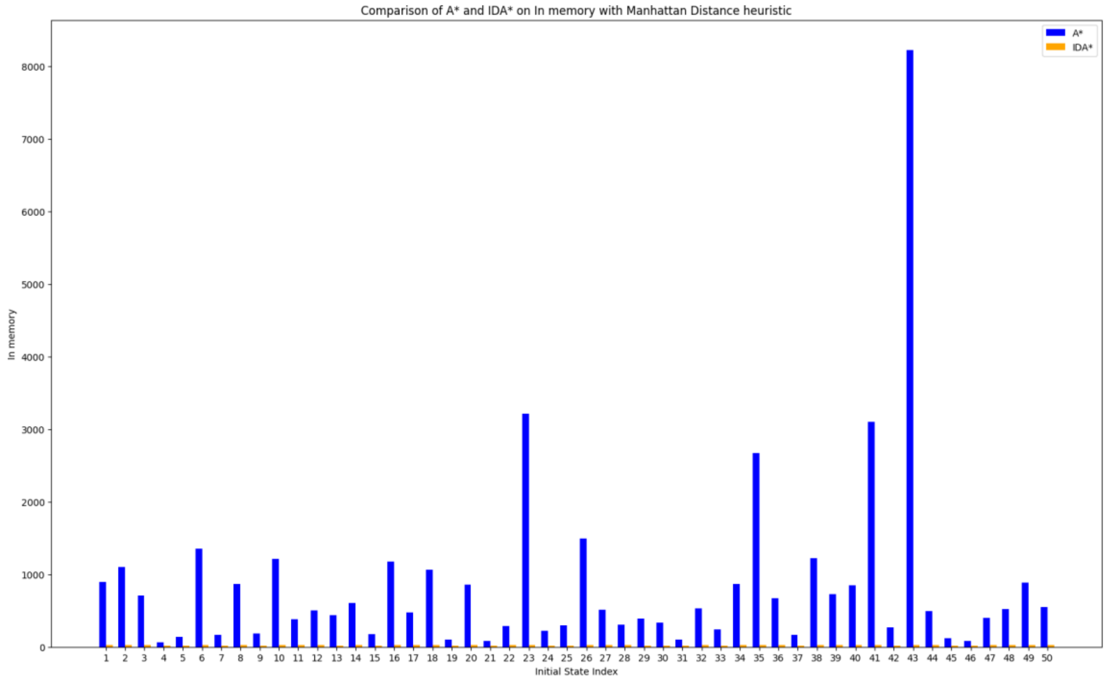
To ensure accuracy the following figures showcase IDA* vs A* on 50 different initial states with Manhattan distance heuristic

Figure 9:



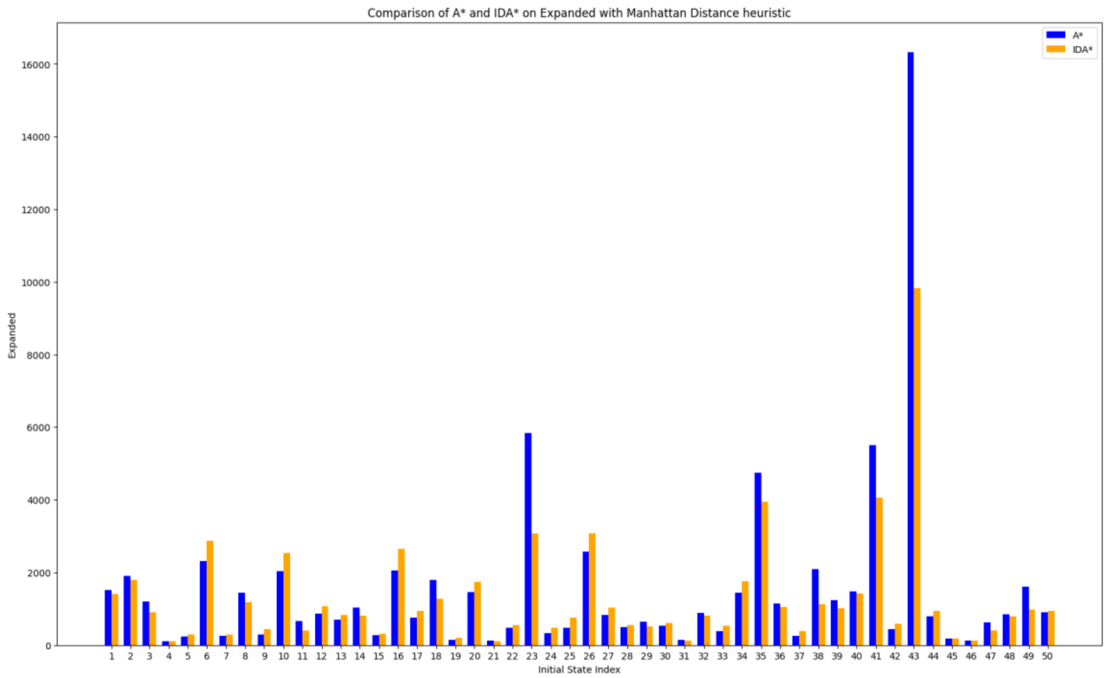
It is shown that IDA* generates more than A* with Manhattan Distance.

Figure 10:



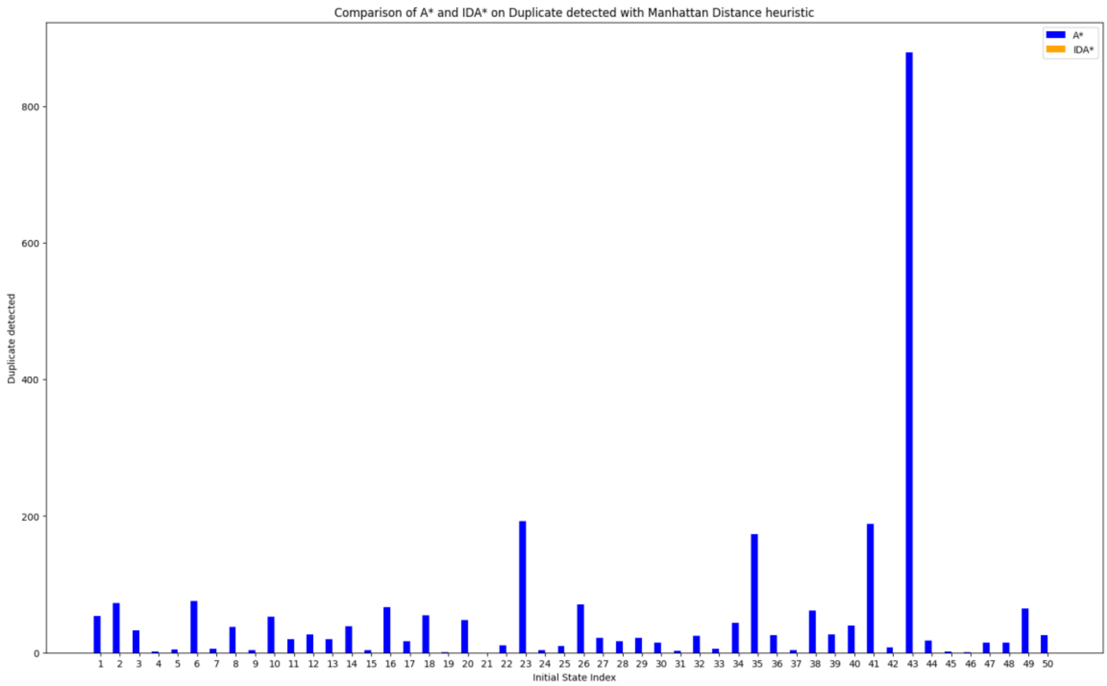
It is shown that A* uses more memory than IDA* with Manhattan Distance.

Figure 11:



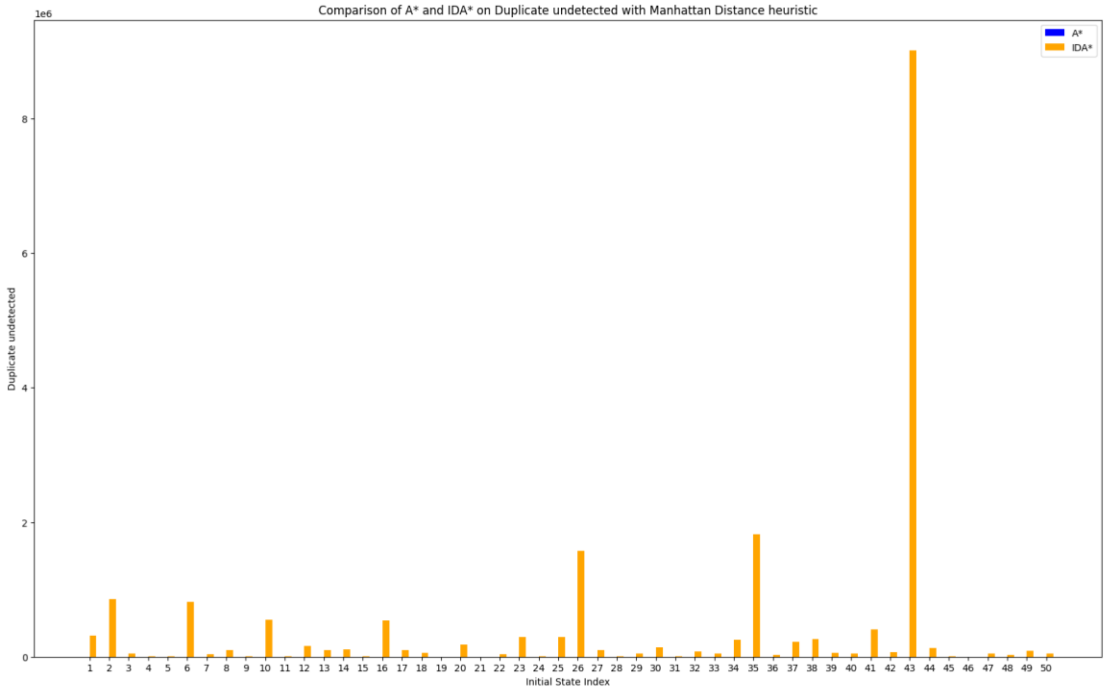
It is shown that A* expand more vertices than IDA* on initial input indexes 1,2,3,8,11,14,18,21,23,29,31,32,35,36,38,39,40,41,43,45,47,48,49. And the rest IDA* expanded more than A*.

Figure 12:



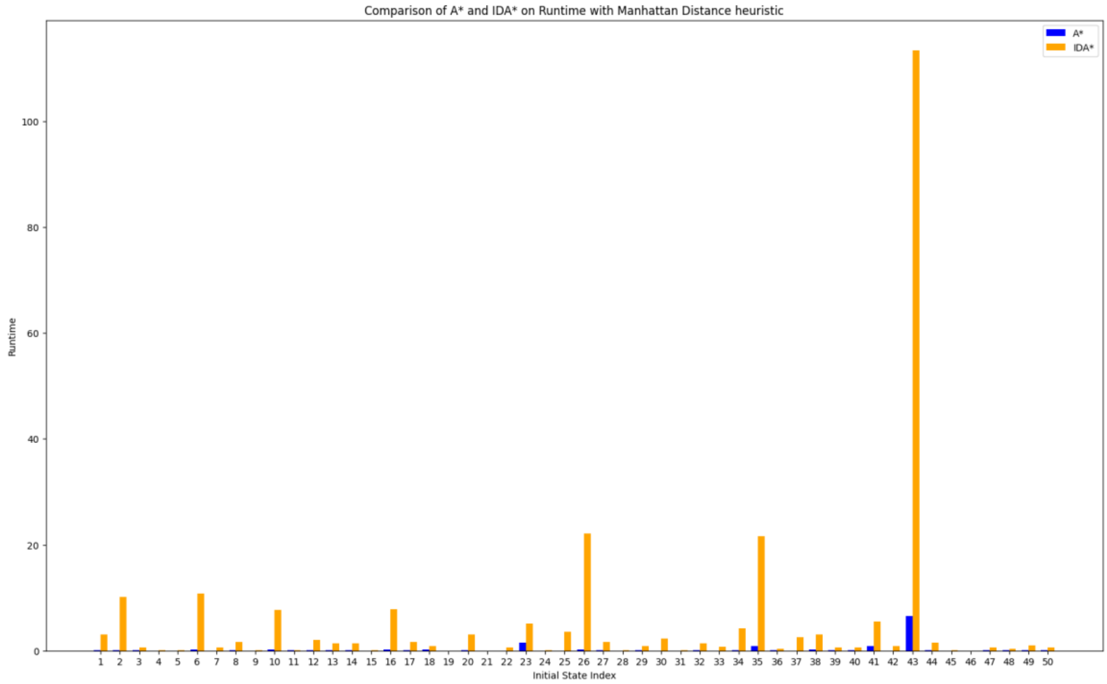
It is shown that A* detected duplicate and IDA* didn't detect any at all.

Figure 13:



It is shown that A* didn't encounter any undetected vertices at all and IDA* did.

Figure 14:



It is shown that IDA* took more time to find the solution in ms compared to A*.

Conclusions

After evaluating and comparing A* vs. IDA* with misplaced and Manhattan distance heuristics for solving 8 puzzle game. I can conclude from the results and graphs that my hypothesis was correct. Manhattan distance is more efficient than Misplaced (if we look at the number of states it explores and runtime before arriving at a solution), and The most efficient runtime algorithm is A* with Manhattan distance heuristic and the most efficient memory algorithm is IDA* with Manhattan distance heuristic. Also I avoided unnecessary runtime errors by checking first the 8 – puzzle initial state is solvable. Also A* outperforms IDA* because A* have open and closed lists so that a given state is not explored multiple times through the graph whereas in IDA* if there are multiple paths to a given state in the graph it will explore those states again and again.

References

A* - https://en.wikipedia.org/wiki/A*_search_algorithm

IDA* - https://en.wikipedia.org/wiki/Iterative_deepening_A*

8 puzzle solvable - https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/

8 puzzle – https://www.almabetter.com/bytes/tutorials/artificial-intelligence/8-puzzle-problem-in-ai

8 puzzle online game – https://sliding.toys/mystic-square/8-puzzle/

Examples of 8 puzzle initial states and their optimal solution – https://www.cs.princeton.edu/courses/archive/spring19/cos226/assignments/8puzzle/checklist.php

Appendex

Initial states Inputs: (this file was generated using the program)

inputs.xlsx

input
6 7 3 1 4 8 5 0 2
2 1 6 0 7 4 3 5 8
6 1 7 2 3 4 8 5 0
3 4 1 0 2 5 7 8 6
1 5 2 8 0 7 3 4 6
0 4 8 1 2 6 5 3 7
7 1 3 4 5 2 0 8 6

6 3 1 4 7 0 5 8 2
0 5 4 7 1 2 3 8 6
6 3 4 1 0 7 2 8 5
6 4 5 2 0 7 8 1 3
5 7 3 8 4 2 1 0 6
6 2 5 7 1 3 0 4 8
0 8 6 3 4 2 1 7 5
2 8 5 1 7 4 0 3 6
3 6 2 8 4 7 1 5 0

0 1 8 7 5 3 2 6 4
8 7 1 5 0 3 2 6 4
4 8 5 2 1 0 7 6 3
0 3 5 7 8 2 6 1 4
6 5 2 4 1 8 3 0 7
5 1 7 6 8 3 0 4 2
6 1 7 8 5 4 2 3 0
0 6 4 1 2 7 5 8 3
5 0 3 1 8 6 2 4 7

3 5 1 0 7 8 2 6 4
3 0 2 8 4 1 7 6 5
8 1 3 6 7 0 5 2 4
2 6 0 8 4 3 1 7 5
4 3 1 0 2 5 8 7 6
1 6 0 7 4 2 8 3 5
3 2 8 6 0 5 1 7 4
1 8 2 0 6 4 5 7 3
1 4 8 7 5 0 3 2 6

4 6 7 8 0 2 1 3 5
0 7 1 5 4 8 3 6 2
1 2 0 4 6 3 8 7 5
1 6 7 8 0 2 3 4 5
6 7 0 5 3 4 1 2 8
6 0 7 3 4 5 8 1 2
6 0 4 2 8 1 5 3 7
6 1 2 7 5 4 8 3 0
0 8 6 7 2 4 3 5 1

3 1 0 5 6 8 4 2 7
1 2 0 3 8 5 4 7 6
5 2 4 1 7 0 6 3 8
3 0 2 4 1 8 6 5 7
5 2 8 3 1 6 0 4 7
8 2 5 6 4 3 7 1 0
1 7 4 2 5 3 0 6 8

Output values for A* and IDA* with Misplaced, Manhatten Distance: (this file was generated using the program)

data_M_vs_MD.xlsx

Initial_State	Algorithm	Heuristic	Generated_Vertices	Vertices_In_Memory	Expanded_Vertices	Duplicate_Detected	Duplicate_Undetected	Run_Time
6 7 3 1 4 8 5 0 2	A*	Misplaced Tiles	22070	7734	13474	862	0	4.109676
6 7 3 1 4 8 5 0 2	IDA*	Misplaced Tiles	93848525	23	15303	0	93833239	661.9993
6 7 3 1 4 8 5 0 2	A*	Manhattan Distance	2467	898	1515	54	0	0.123
6 7 3 1 4 8 5 0 2	IDA*	Manhattan Distance	313299	23	1414	0	311890	3.737022
2 1 6 0 7 4 3 5 8	A*	Misplaced Tiles	22581	7835	13847	899	0	7.295508
2 1 6 0 7 4 3 5 8	IDA*	Misplaced Tiles	1E+08	23	16712	0	99999587	613.1171

2 1 6 0 7 4 3 5 8	A*	Manhattan Distance	3087	1099	1915	73	0	0.233028
2 1 6 0 7 4 3 5 8	IDA*	Manhattan Distance	856718	23	1796	0	854928	9.828116
6 1 7 2 3 4 8 5 0	A*	Misplaced Tiles	28550	9662	17588	1300	0	7.427595
6 1 7 2 3 4 8 5 0	IDA*	Misplaced Tiles	1.12E+08	24	19834	0	1.12E+08	697.3345
6 1 7 2 3 4 8 5 0	A*	Manhattan Distance	1936	705	1198	33	0	0.064971
6 1 7 2 3 4 8 5 0	IDA*	Manhattan Distance	50500	24	895	0	49610	0.654039
3 4 1 0 2 5 7 8 6	A*	Misplaced Tiles	643	251	381	11	0	0.016039

3 4 1 0 2 5 7 8 6	IDA*	Misplaced Tiles	59270	15	453	0	58827	0.50 6842
3 4 1 0 2 5 7 8 6	A*	Manhattan Distance	169	67	100	2	0	0.00 3002
3 4 1 0 2 5 7 8 6	IDA*	Manhattan Distance	2679	15	103	0	2580	0.03 3969
1 5 2 8 0 7 3 4 6	A*	Misplaced Tiles	2774	1051	1658	65	0	0.13 0012
1 5 2 8 0 7 3 4 6	IDA*	Misplaced Tiles	1518465	18	2355	0	1516122	9.65 8778
1 5 2 8 0 7 3 4 6	A*	Manhattan Distance	376	140	231	5	0	0.00 8029
1 5 2 8 0 7 3 4 6	IDA*	Manhattan Distance	12339	18	299	0	12043	0.14 8003

0 4 8 1 2 6 5 3 7	A*	Misplaced Tiles	29930	10122	18446	1362	0	8.262683
0 4 8 1 2 6 5 3 7	IDA*	Misplaced Tiles	3.2E+08	24	24521	0	3.2E+08	2033.316
0 4 8 1 2 6 5 3 7	A*	Manhattan Distance	3739	1351	2312	76	0	0.18498
0 4 8 1 2 6 5 3 7	IDA*	Manhattan Distance	816639	24	2864	0	813781	9.363956
7 1 3 4 5 2 0 8 6	A*	Misplaced Tiles	1050	399	630	21	0	0.021005
7 1 3 4 5 2 0 8 6	IDA*	Misplaced Tiles	168836	16	846	0	168002	0.976055
7 1 3 4 5 2 0 8 6	A*	Manhattan Distance	414	162	246	6	0	0.007
7 1 3 4 5 2 0 8 6	IDA*	Manhattan	38023	16	300	0	37729	0.423046

		Distance						
6 3 1 4 7 0 5 8 2	A*	Misplaced Tiles	21801	7568	13353	880	0	5.963769
6 3 1 4 7 0 5 8 2	IDA*	Misplaced Tiles	1.08E+08	23	15377	0	1.08E+08	655.1993
6 3 1 4 7 0 5 8 2	A*	Manhattan Distance	2342	869	1435	38	0	0.083999
6 3 1 4 7 0 5 8 2	IDA*	Manhattan Distance	99790	23	1179	0	98617	1.254014
0 5 4 7 1 2 3 8 6	A*	Misplaced Tiles	5205	1959	3130	116	0	0.34301
0 5 4 7 1 2 3 8 6	IDA*	Misplaced Tiles	4276630	20	4749	0	4271895	32.46815
0 5 4 7 1 2 3 8 6	A*	Manhattan Distance	482	184	294	4	0	0.073002

0 5 4 7 1 2 3 8 6	IDA*	Manhattan Distance	7182	20	434	0	6752	0.125007
6 3 4 1 0 7 2 8 5	A*	Misplaced Tiles	35347	11723	21884	1740	0	17.36399
6 3 4 1 0 7 2 8 5	IDA*	Misplaced Tiles	4.3E+08	24	30375	0	4.3E+08	2761.296
6 3 4 1 0 7 2 8 5	A*	Manhattan Distance	3306	1213	2040	53	0	0.216
6 3 4 1 0 7 2 8 5	IDA*	Manhattan Distance	557440	24	2536	0	554909	6.962104

Output values for A* vs IDA* with Manhattan Distance: (this file was generated using the program)

data_MD.xlsx

Initial_State	Algorithm	Heuristic	Generated_Vertices	Vertices_In_Memory	Expanded_Vertices	Duplicate_Detected	Duplicate_Undetected	Run_Time
6 7 3 1 4 8 5 0 2	A*	Manhattan Distance	2467	898	1515	54	0	0.082007
6 7 3 1 4 8 5 0 2	IDA*	Manhattan Distance	313299	23	1414	0	311890	3.042271
2 1 6 0 7 4 3 5 8	A*	Manhattan Distance	3087	1099	1915	73	0	0.113003
2 1 6 0 7 4 3 5 8	IDA*	Manhattan Distance	856718	23	1796	0	854928	10.18695
6 1 7 2 3 4 8 5 0	A*	Manhattan Distance	1936	705	1198	33	0	0.079034

6 1 7 2 3 4 8 5 0	IDA*	Manhattan Distance	50500	24	895	0	49610	0.617011
3 4 1 0 2 5 7 8 6	A*	Manhattan Distance	169	67	100	2	0	0.003006
3 4 1 0 2 5 7 8 6	IDA*	Manhattan Distance	2679	15	103	0	2580	0.031991
1 5 2 8 0 7 3 4 6	A*	Manhattan Distance	376	140	231	5	0	0.006997
1 5 2 8 0 7 3 4 6	IDA*	Manhattan Distance	12339	18	299	0	12043	0.143007
0 4 8 1 2 6 5 3 7	A*	Manhattan Distance	3739	1351	2312	76	0	0.240003
0 4 8 1 2 6 5 3 7	IDA*	Manhattan	816639	24	2864	0	813781	10.77711

		Distance						
7 1 3 4 5 2 0 8 6	A*	Manhattan Distance	414	162	246	6	0	0.010003
7 1 3 4 5 2 0 8 6	IDA*	Manhattan Distance	38023	16	300	0	37729	0.593037
6 3 1 4 7 0 5 8 2	A*	Manhattan Distance	2342	869	1435	38	0	0.115973
6 3 1 4 7 0 5 8 2	IDA*	Manhattan Distance	99790	23	1179	0	98617	1.583049
0 5 4 7 1 2 3 8 6	A*	Manhattan Distance	482	184	294	4	0	0.012001
0 5 4 7 1 2 3 8 6	IDA*	Manhattan Distance	7182	20	434	0	6752	0.132999

6 3 4 1 0 7 2 8 5	A*	Manhattan Distance	3306	1213	2040	53	0	0.217005
6 3 4 1 0 7 2 8 5	IDA*	Manhattan Distance	557440	24	2536	0	554909	7.656117
6 4 5 2 0 7 8 1 3	A*	Manhattan Distance	1057	379	658	20	0	0.030002
6 4 5 2 0 7 8 1 3	IDA*	Manhattan Distance	6116	22	398	0	5721	0.07597
5 7 3 8 4 2 1 0 6	A*	Manhattan Distance	1393	506	860	27	0	0.041029
5 7 3 8 4 2 1 0 6	IDA*	Manhattan Distance	167080	21	1063	0	166022	2.036995
6 2 5 7 1 3 0 4 8	A*	Manhattan	1153	436	697	20	0	0.097033

		Distance						
6 2 5 7 1 3 0 4 8	IDA*	Manhattan Distance	95781	20	828	0	94958	1.355014
0 8 6 3 4 2 1 7 5	A*	Manhattan Distance	1684	609	1036	39	0	0.062004
0 8 6 3 4 2 1 7 5	IDA*	Manhattan Distance	111518	22	812	0	110711	1.432019
2 8 5 1 7 4 0 3 6	A*	Manhattan Distance	459	177	278	4	0	0.009979
2 8 5 1 7 4 0 3 6	IDA*	Manhattan Distance	6416	20	306	0	6114	0.08002
3 6 2 8 4 7 1 5 0	A*	Manhattan Distance	3295	1177	2051	67	0	0.162969

3 6 2 8 4 7 1 5 0	IDA*	Manhattan Distance	543132	24	2649	0	540489	7.809096
0 1 8 7 5 3 2 6 4	A*	Manhattan Distance	1244	471	756	17	0	0.051999
0 1 8 7 5 3 2 6 4	IDA*	Manhattan Distance	103763	22	938	0	102830	1.696017
8 7 1 5 0 3 2 6 4	A*	Manhattan Distance	2921	1064	1802	55	0	0.235005
8 7 1 5 0 3 2 6 4	IDA*	Manhattan Distance	56347	24	1284	0	55067	0.851009
4 8 5 2 1 0 7 6 3	A*	Manhattan Distance	241	96	144	1	0	0.005
4 8 5 2 1 0 7 6 3	IDA*	Manhattan	1790	17	197	0	1596	0.026139

		Distance						
0 3 5 7 8 2 6 1 4	A*	Manhattan Distance	2361	856	1457	48	0	0.103867
0 3 5 7 8 2 6 1 4	IDA*	Manhattan Distance	183460	24	1739	0	181726	3.095037
6 5 2 4 1 8 3 0 7	A*	Manhattan Distance	211	84	127	0	0	0.006027
6 5 2 4 1 8 3 0 7	IDA*	Manhattan Distance	530	19	107	0	426	0.008972
5 1 7 6 8 3 0 4 2	A*	Manhattan Distance	775	286	478	11	0	0.027004
5 1 7 6 8 3 0 4 2	IDA*	Manhattan Distance	38942	22	555	0	38391	0.627005

6 1 7 8 5 4 2 3 0	A*	Manhattan Distance	9250	3216	5841	193	0		1.497045
6 1 7 8 5 4 2 3 0	IDA*	Manhattan Distance	300488	28	3076	0	297418		5.095663
0 6 4 1 2 7 5 8 3	A*	Manhattan Distance	554	219	331	4	0		0.023072
0 6 4 1 2 7 5 8 3	IDA*	Manhattan Distance	7156	20	475	0	6685		0.10036
5 0 3 1 8 6 2 4 7	A*	Manhattan Distance	787	301	476	10	0		0.020943
5 0 3 1 8 6 2 4 7	IDA*	Manhattan Distance	292666	19	749	0	291922		3.535346
3 5 1 0 7 8 2 6 4	A*	Manhattan	4140	1493	2576	71	0		0.225633

		Distance						
3 5 1 0 7 8 2 6 4	IDA*	Manhattan Distance	1579351	25	3084	0	1576272	22.07477
3 0 2 8 4 1 7 6 5	A*	Manhattan Distance	1355	509	824	22	0	0.057999
3 0 2 8 4 1 7 6 5	IDA*	Manhattan Distance	98347	21	1043	0	97309	1.672647
8 1 3 6 7 0 5 2 4	A*	Manhattan Distance	821	308	496	17	0	0.023998
8 1 3 6 7 0 5 2 4	IDA*	Manhattan Distance	8965	21	546	0	8423	0.152616
2 6 0 8 4 3 1 7 5	A*	Manhattan Distance	1056	394	640	22	0	0.104641

2 6 0 8 4 3 1 7 5	IDA*	Manhattan Distance	50531	20	507	0	50029	0.833173
4 3 1 0 2 5 8 7 6	A*	Manhattan Distance	882	330	537	15	0	0.025997
4 3 1 0 2 5 8 7 6	IDA*	Manhattan Distance	142433	19	604	0	141835	2.32623
1 6 0 7 4 2 8 3 5	A*	Manhattan Distance	254	101	150	3	0	0.006003
1 6 0 7 4 2 8 3 5	IDA*	Manhattan Distance	2815	16	126	0	2692	0.046999
3 2 8 6 0 5 1 7 4	A*	Manhattan Distance	1430	528	877	25	0	0.054004
3 2 8 6 0 5 1 7 4	IDA*	Manhattan	79028	22	803	0	78230	1.324016

		Distance						
1 8 2 0 6 4 5 7 3	A*	Manhattan Distance	624	241	377	6	0	0.016996
1 8 2 0 6 4 5 7 3	IDA*	Manhattan Distance	51095	19	529	0	50571	0.79904
1 4 8 7 5 0 3 2 6	A*	Manhattan Distance	2353	863	1446	44	0	0.109124
1 4 8 7 5 0 3 2 6	IDA*	Manhattan Distance	259050	23	1758	0	257298	4.194927
4 6 7 8 0 2 1 3 5	A*	Manhattan Distance	7589	2667	4748	174	0	0.86205
4 6 7 8 0 2 1 3 5	IDA*	Manhattan Distance	1825301	26	3954	0	1821352	21.57325

0 7 1 5 4 8 3 6 2	A*	Manh attan Dista nce	1832	668	1138	26	0	0.060 967
0 7 1 5 4 8 3 6 2	IDA*	Manh attan Dista nce	28745	24	1050	0	27699	0.331 054
1 2 0 4 6 3 8 7 5	A*	Manh attan Dista nce	414	161	249	4	0	0.007 984
1 2 0 4 6 3 8 7 5	IDA*	Manh attan Dista nce	218895	16	376	0	218525	2.534 03
1 6 7 8 0 2 3 4 5	A*	Manh attan Dista nce	3370	1223	2085	62	0	0.162 002
1 6 7 8 0 2 3 4 5	IDA*	Manh attan Dista nce	263531	24	1133	0	262402	2.995 002
6 7 0 5 3 4 1 2 8	A*	Manh attan	1993	731	1235	27	0	0.130 085

		Distance						
6 7 0 5 3 4 1 2 8	IDA*	Manhattan Distance	54912	24	1019	0	53898	0.635211
6 0 7 3 4 5 8 1 2	A*	Manhattan Distance	2370	845	1485	40	0	0.091002
6 0 7 3 4 5 8 1 2	IDA*	Manhattan Distance	50397	25	1426	0	48975	0.588006
6 0 4 2 8 1 5 3 7	A*	Manhattan Distance	8798	3103	5506	189	0	0.89801
6 0 4 2 8 1 5 3 7	IDA*	Manhattan Distance	415664	27	4057	0	411612	5.557198
6 1 2 7 5 4 8 3 0	A*	Manhattan Distance	718	273	437	8	0	0.015998

6 1 2 7 5 4 8 3 0	IDA*	Manhattan Distance	72981	20	582	0	72404	0.836015
0 8 6 7 2 4 3 5 1	A*	Manhattan Distance	25432	8225	16328	879	0	6.538996
0 8 6 7 2 4 3 5 1	IDA*	Manhattan Distance	9025262	30	9823	0	9015447	113.3925
3 1 0 5 6 8 4 2 7	A*	Manhattan Distance	1303	490	795	18	0	0.036008
3 1 0 5 6 8 4 2 7	IDA*	Manhattan Distance	127462	22	938	0	126530	1.491417
1 2 0 3 8 5 4 7 6	A*	Manhattan Distance	292	115	175	2	0	0.010003
1 2 0 3 8 5 4 7 6	IDA*	Manhattan	4395	16	173	0	4227	0.067999

		Distance						
5 2 4 1 7 0 6 3 8	A*	Manhattan Distance	215	85	129	1	0	0.004001
5 2 4 1 7 0 6 3 8	IDA*	Manhattan Distance	551	19	119	0	435	0.008
3 0 2 4 1 8 6 5 7	A*	Manhattan Distance	1039	397	627	15	0	0.029001
3 0 2 4 1 8 6 5 7	IDA*	Manhattan Distance	45542	21	412	0	45135	0.568039
5 2 8 3 1 6 0 4 7	A*	Manhattan Distance	1379	522	842	15	0	0.039967
5 2 8 3 1 6 0 4 7	IDA*	Manhattan Distance	31407	22	796	0	30616	0.368038

8 2 5 6 4 3 7 1 0	A*	Manhattan Distance	2569	888	1616	65	0	0.100001
8 2 5 6 4 3 7 1 0	IDA*	Manhattan Distance	90461	24	977	0	89491	1.036015
1 7 4 2 5 3 0 6 8	A*	Manhattan Distance	1474	548	900	26	0	0.099993
1 7 4 2 5 3 0 6 8	IDA*	Manhattan Distance	52703	22	936	0	51773	0.597289

Full Code with the results can be found in this link:

https://colab.research.google.com/drive/1c8Z37rOuRrfo8YOi0kYFRDmUjfZiMcQC?usp=sharing

Full Code text:

```
from cmath import inf
from collections import deque
import random
import time

class Game_8_Puzzle:
 def __init__(self, puzzle):
  # puzzle: a list of lists of 8 puzzle matrix, initial node
  self.puzzle = puzzle
  self.goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

 def __str__(self):
  s = ""
  for i in range(3):
   for j in range(3):
    s += str(self.puzzle[i][j]) + " "
   s += "\n"
  #s += "\n"
  return s

 def move(self, x1, y1, x2, y2):
  # swap two tiles on the puzzle
  new_puzzle = [list(ias) for ias in self.puzzle]
  tmp = new_puzzle[x1][y1]
  new_puzzle[x1][y1] = new_puzzle[x2][y2]
  new_puzzle[x2][y2] = tmp
  return new_puzzle

 def get_position_i_j(self, num, puzzle=None):
   # find number and return the i, j of the position
   if not puzzle:
     puzzle = self.puzzle

   for i in range(3):
     for j in range(3):
       if puzzle[i][j] == num:
         return i, j

 def options(self):
```

```
        # find all possible moves and return all options
        options = []
        x, y = self.get_position_i_j(0)
        if x > 0: # up
            options.append(Game_8_Puzzle(self.move(x, y, x - 1, y)))
        if x < 2: # down
            options.append(Game_8_Puzzle(self.move(x, y, x + 1, y)))
        if y > 0: # left
            options.append(Game_8_Puzzle(self.move(x, y, x, y - 1)))
        if y < 2: # right
            options.append(Game_8_Puzzle(self.move(x, y, x, y + 1)))
        return options

    def MD(self):
        # heuristic function manhattan distance
        h = 0
        for x in range(3):
            for y in range(3):
                if self.puzzle[x][y] != 0:
                    i, j = self.get_position_i_j(self.puzzle[x][y], self.goal)
                    h += abs(x - i) + abs(y - j)
        return h

    def M(self):
        # heuristic function misplaced tiles
        h = 0
        for x in range(3):
            for y in range(3):
                if self.puzzle[x][y] != 0 and self.puzzle[x][y] != self.goal[x][y]:
                    h += 1
        return h
    def NO(self):
        # heuristic function that return always 0 in case of A* then it behave exactly
like Uniform Cost Search
        h = 0
        return h

    def is_valid_puzzle_and_can_be_solved(self):
        # return true if puzzle is valid and solvable as explaned above
        cnt = 0
        puzzle_as_1d_list = [num for i in self.puzzle for num in i if not(num == 0)]
        for x in range(len(puzzle_as_1d_list)):
            for y in range(x + 1, len(puzzle_as_1d_list)):
                if puzzle_as_1d_list[x] > puzzle_as_1d_list[y]:
                    cnt += 1
        return cnt % 2 == 0
```

```
class A_Star:
  def __init__(self, initial_node, h):
    self.puzzle = initial_node
    self.h = h # heuristic function

  def run(self, get_memory_used = False):
    # array of memory usage: will contain len(closed set) + len(priority queue)
    memory_used = []
    # each item in the priority queue is (heuristic value, puzzle) it will help getting
the min path by its heuristic value faster
    priority_queue = [(self.h(self.puzzle) ,[self.puzzle])] # open_set
    closed_set = set() # expanded nodes
    # i will also calculate:
    number_of_expanded_vertices = 0
    number_of_duplicate_vertices_detected = 0
    number_of_generated_vertices = 0
    while priority_queue:

      # get lowest heurestic
      index = priority_queue.index(min(priority_queue, key=lambda x: x[0]))
      min_h, puzzle_path = priority_queue.pop(index)
      min_puzzle = puzzle_path[-1]
      if min_puzzle.puzzle == min_puzzle.goal:
        break
      if str(min_puzzle.puzzle) in closed_set:
        number_of_duplicate_vertices_detected += 1
        continue
      for option in min_puzzle.options():
        if str(option.puzzle) in closed_set:
            continue
        tmp = puzzle_path+[option]
        priority_queue.append((len(puzzle_path)-1 + self.h(option), tmp))
        number_of_generated_vertices += 1
      closed_set.add(str(min_puzzle.puzzle))
      if get_memory_used:
        memory_used.append(len(closed_set) + len(priority_queue))
      number_of_expanded_vertices += 1

    #number_of_vertices = len(closed_set)

    # print("len(open_list): "+str(len(priority_queue)))
    # print("len(closed_set): "+str(number_of_vertices))
    # print("num of expanded: "+str(number_of_expanded_vertices))

    number_of_vertices_in_queue = len(priority_queue)
    number_of_duplicate_vertices_undetected = 0
```

```
    return puzzle_path, number_of_generated_vertices,
number_of_vertices_in_queue, number_of_expanded_vertices,
number_of_duplicate_vertices_detected,
number_of_duplicate_vertices_undetected, memory_used


def print_path_to_sol(path):
  for x in path:
    print(x.__str__())

def play(game, print_boo=True, get_memory_used=False):
  assert game.puzzle.is_valid_puzzle_and_can_be_solved() == True
  start = time.time()
  path, num1, num2, num3, num4, num5, arr = game.run(get_memory_used)
  end = time.time()
  total_time =  end - start
  if print_boo:
    print("solution: ")
    print("num of moves: "+str(len(path)-1))
    print_path_to_sol(path)
    print("number of generated vertices: "+str(num1))
    print("number of vertices in (queue for A*/stack for IDA*): "+str(num2))
    print("number of expanded vertices: "+str(num3))
    print("amount of time(ms): "+str(total_time))
    print("number of duplicate vertices detected: "+str(num4))
    print("number of duplicate vertices undetected: "+ str(num5))
    if get_memory_used:
      print("memory used array:")
      print(arr)
  if get_memory_used:
    return arr
  return num1, num2, num3, num4, num5, total_time

class IDA_Star():
  def __init__(self, initial_node, h):
    self.puzzle = initial_node
    self.h = h # heuristic function
    self.expanded_nodes = set() # used this only to count undectected duplicates

  def run(self, get_memory_used=False):
    number_of_generated_vertices, number_of_vertices_in_stack,
number_of_expanded_vertices, number_of_duplicate_vertices_detected,
number_of_duplicate_vertices_undetected = 0, 0, 0, 0, 0
    # memory used array: will have the size of the path at each time
    memory_used = [1] # starting with 1 (only initial state)
    # here
    def search(path, g, bound):
```

```
    nonlocal number_of_generated_vertices, number_of_vertices_in_stack,
number_of_expanded_vertices, number_of_duplicate_vertices_detected,
number_of_duplicate_vertices_undetected, memory_used
    node = path[-1]
    if(str(node.puzzle) in self.expanded_nodes):
     number_of_duplicate_vertices_undetected += 1
    else:
     self.expanded_nodes.add(str(node.puzzle))

    f = g + self.h(node)
    if f > bound:
     return f
    if str(node.puzzle) == str(node.goal):
     return True
    min = inf
    for option in node.options():
      number_of_generated_vertices += 1
      if option not in path:
        number_of_vertices_in_stack += 1
        path.append(option)
        if get_memory_used:
         memory_used.append(len(path))
        t = search(path, g + 1, bound)
        if t == True:
         return True
        if t < min:
         min = t
        path.pop()
        if get_memory_used:
         memory_used.append(len(path))
        number_of_vertices_in_stack -= 1
      else:
        number_of_duplicate_vertices_detected += 1

    return min
   bound = self.h(self.puzzle)
   path = [self.puzzle]
   while 1:
    t = search(path, 0, bound)
    if t == True:
     return path, number_of_generated_vertices, number_of_vertices_in_stack,
len(self.expanded_nodes), number_of_duplicate_vertices_detected,
number_of_duplicate_vertices_undetected, memory_used
    if t == inf:
     return False
    bound = t
```

```
import matplotlib.pyplot as plt
import numpy as np

genrated_initial_states_number = 50
def generate_random_board():
  boo = True
  while boo:
   #Generate a random 8-puzzle board
   board = list(range(9))
   random.shuffle(board)
   board = [board[i:i + 3] for i in range(0, 9, 3)]
   board = Game_8_Puzzle(board)
   boo = not board.is_valid_puzzle_and_can_be_solved()
  return board

def generate_initial_states(n):
  #Generate n initial states for the 8-puzzle game.
  initial_states = []
  dups = set()
  while len(initial_states) < n:
    board = generate_random_board()
    if str(board) not in dups:
     initial_states.append(board)
     dups.add(str(initial_states))
  return initial_states

# example of random generated puzzle game
print(generate_random_board().__str__())

initial_states = [puzzle]
def run_experiment_MD_show_memory(initial_states): # run experiment with no
heuristic
  results = {
    "A*": {"memory_by_iteration": []},
    "IDA*": {"memory_by_iteration": []}
  }

  for state in initial_states:
    for algo in ["A*", "IDA*"]:
      game = None
      if algo == "A*":
       game = A_Star(state, Game_8_Puzzle.MD)
      elif algo == "IDA*":
       game = IDA_Star(state, Game_8_Puzzle.MD)
```

```
        generated, in_memory, expanded, duplicate_detected,
duplicate_undetected, runtime = 0, 0, 0, 0, 0, 0
        mem = play(game,True, True) # replace this later to false so no printing will
be involved
        results[algo]["memory_by_iteration"] = mem
  #return results
  metrics = ["memory_by_iteration"]
  num_states = max(len(results["A*"]["memory_by_iteration"]),
len(results["IDA*"]["memory_by_iteration"]))
  delta = abs(len(results["IDA*"]["memory_by_iteration"]) -
len(results["A*"]["memory_by_iteration"]))
  if len(results["A*"]["memory_by_iteration"]) <
len(results["IDA*"]["memory_by_iteration"]):
      results["A*"]["memory_by_iteration"] += [0] * delta
  elif len(results["A*"]["memory_by_iteration"]) >
len(results["IDA*"]["memory_by_iteration"]):
      results["IDA*"]["memory_by_iteration"] += [0] * delta
  x = np.arange(1, num_states + 1)
  for metric in metrics:
      plt.figure(figsize=(10, 6))
      plt.plot(x, results["A*"][metric], label="A*", marker='o')
      plt.plot(x, results["IDA*"][metric], label="IDA*", marker='x')
      plt.xlabel("Iteration")
      plt.ylabel(metric.replace("_", " ").capitalize())
      plt.title(f"Comparison of A* and IDA* on {metric.replace('_', ' ').capitalize()}")
      plt.legend()
      plt.grid(True)
      plt.show()

run_experiment_MD_show_memory(initial_states)

initial_states = generate_initial_states(1)
def run_experiment_MD_show_memory(initial_states): # run experiment with no
heuristic
  results = {
      "A*": {"memory_by_iteration": []},
      "IDA*": {"memory_by_iteration": []}
  }

  for state in initial_states:
      for algo in ["A*", "IDA*"]:
        game = None
        if algo == "A*":
          game = A_Star(state, Game_8_Puzzle.MD)
        elif algo == "IDA*":
          game = IDA_Star(state, Game_8_Puzzle.MD)
```

```
        generated, in_memory, expanded, duplicate_detected,
duplicate_undetected, runtime = 0, 0, 0, 0, 0, 0
        mem = play(game,True, True) # replace this later to false so no printing will
be involved
        results[algo]["memory_by_iteration"] = mem
  #return results
  metrics = ["memory_by_iteration"]
  num_states = max(len(results["A*"]["memory_by_iteration"]),
len(results["IDA*"]["memory_by_iteration"]))
  delta = abs(len(results["IDA*"]["memory_by_iteration"]) -
len(results["A*"]["memory_by_iteration"]))
  if len(results["A*"]["memory_by_iteration"]) <
len(results["IDA*"]["memory_by_iteration"]):
    results["A*"]["memory_by_iteration"] += [0] * delta
  elif len(results["A*"]["memory_by_iteration"]) >
len(results["IDA*"]["memory_by_iteration"]):
    results["IDA*"]["memory_by_iteration"] += [0] * delta
  x = np.arange(1, num_states + 1)
  for metric in metrics:
    plt.figure(figsize=(10, 6))
    plt.plot(x, results["A*"][metric], label="A*", marker='o')
    plt.plot(x, results["IDA*"][metric], label="IDA*", marker='x')
    plt.xlabel("Iteration")
    plt.ylabel(metric.replace("_", " ").capitalize())
    plt.title(f"Comparison of A* and IDA* on {metric.replace('_', ' ').capitalize()}")
    plt.legend()
    plt.grid(True)
    plt.show()

run_experiment_MD_show_memory(initial_states)

initial_states = generate_initial_states(genrated_initial_states_number)
import openpyxl
import pandas as pd

def write_rows_to_excel(file_name, sheet_name, rows):
  # Create a new workbook and select the active worksheet
  workbook = openpyxl.Workbook()
  sheet = workbook.active
  sheet.title = sheet_name

  # Write the rows to the worksheet
  for row in rows:
    sheet.append(row)

  # Save the workbook to a file
  workbook.save(file_name)
```

```
rows = [["input"]]
for x in initial_states:
   rows.append([x.__str__()])
print(rows)

write_rows_to_excel("inputs.xlsx", "Sheet1", rows)

initial_states = []
inputs_num = 10
# reading the inputs
df = pd.read_excel("inputs.xlsx", sheet_name='Sheet1', usecols="A:A",
nrows=inputs_num, skiprows=0, index_col=0)
#print(df)
for x in df.index:
   tmp = x.split('\n')[:3]
   new_input = []
   for t in tmp:
      tmp2 = t.split(' ')[:3]
      list_of_ints = [int(item) for item in tmp2]
      new_input.append(list_of_ints)
   initial_states.append(Game_8_Puzzle(new_input))
   #print(new_input)
#print(initial_states)
def run_experiment_M(initial_states): # run experiment with no heuristic
   results = {
      "A*": {"generated": [], "in_memory": [], "expanded": [], "duplicate_detected":
[], "duplicate_undetected": [], "runtime": []},
      "IDA*": {"generated": [], "in_memory": [], "expanded": [],
"duplicate_detected": [], "duplicate_undetected": [], "runtime": []}
   }

   for state in initial_states:
      for algo in ["A*", "IDA*"]:
         game = None
         if algo == "A*":
          game = A_Star(state, Game_8_Puzzle.M)
         elif algo == "IDA*":
          game = IDA_Star(state, Game_8_Puzzle.M)
         generated, in_memory, expanded, duplicate_detected,
duplicate_undetected, runtime = play(game,False)
         results[algo]["generated"].append(generated)
         results[algo]["in_memory"].append(in_memory)
         results[algo]["expanded"].append(expanded)
         results[algo]["duplicate_detected"].append(duplicate_detected)
         results[algo]["duplicate_undetected"].append(duplicate_undetected)
```

```
        results[algo]["runtime"].append(runtime)

    return results

def run_experiment_MD(initial_states): # run experiment with no heuristic
    results = {
        "A*": {"generated": [], "in_memory": [], "expanded": [], "duplicate_detected":
[], "duplicate_undetected": [], "runtime": []},
        "IDA*": {"generated": [], "in_memory": [], "expanded": [],
"duplicate_detected": [], "duplicate_undetected": [], "runtime": []}
    }

    for state in initial_states:
        for algo in ["A*", "IDA*"]:
            game = None
            if algo == "A*":
              game = A_Star(state, Game_8_Puzzle.MD)
            elif algo == "IDA*":
              game = IDA_Star(state, Game_8_Puzzle.MD)
            generated, in_memory, expanded, duplicate_detected,
duplicate_undetected, runtime = play(game,False)

            results[algo]["generated"].append(generated)
            results[algo]["in_memory"].append(in_memory)
            results[algo]["expanded"].append(expanded)
            results[algo]["duplicate_detected"].append(duplicate_detected)
            results[algo]["duplicate_undetected"].append(duplicate_undetected)
            results[algo]["runtime"].append(runtime)

    return results

def plot_results(results, h):
    metrics = ["generated", "in_memory", "expanded", "duplicate_detected",
"duplicate_undetected", "runtime"]
    num_states = len(initial_states)
    x = np.arange(1, num_states + 1)  # Label initial states from 1 to num_states
    for metric in metrics:
        a_star_values = results["A*"][metric]
        ida_star_values = results["IDA*"][metric]

        bar_width = 0.35  # Width of the bars
        fig, ax = plt.subplots(figsize=(10, 6))

        bar1 = ax.bar(x - bar_width/2, a_star_values, bar_width, label='A*',
color='blue')
        bar2 = ax.bar(x + bar_width/2, ida_star_values, bar_width, label='IDA*',
color='orange')
```

```
    ax.set_xlabel('Initial State Index')
    ax.set_ylabel(metric.replace("_", " ").capitalize())
    ax.set_title(f'Comparison of A* and IDA* on {metric.replace("_", "
").capitalize()} with {h} heuristic')
    ax.set_xticks(x)
    ax.legend()

    # Optional: Add labels above the bars
    def add_labels(bars):
      for bar in bars:
        height = bar.get_height()
        ax.annotate(f'{height}',
              xy=(bar.get_x() + bar.get_width() / 2, height),
              xytext=(0, 3),  # 3 points vertical offset
              textcoords="offset points",
              ha='center', va='bottom')

    add_labels(bar1)
    add_labels(bar2)

    plt.show()

for i in range(len(initial_states)):
  print("initial state #"+str(i+1)+" :")
  print(initial_states[i])
  print("-----------------------")

# Run the experiment
results_M = run_experiment_M(initial_states)

# Plot the results
plot_results(results_M, "Misplaced Tiles")

# Run the experiment
results_MD = run_experiment_MD(initial_states)

# Plot the results
plot_results(results_MD, "Manhattan Distance")

rows = [["Initial_State", "Algorithm", "Heuristic", "Generated_Vertices",
"Vertices_In_Memory", "Expanded_Vertices", "Duplicate_Detected",
"Duplicate_Undetected", "Run_Time"]]
# ["generated", "in_memory", "expanded", "duplicate_detected",
"duplicate_undetected", "runtime"]
for i in range(len(initial_states)):
```

```
  rows.append([initial_states[i].__str__(), "A*", "Misplaced Tiles",
results_M["A*"]["generated"][i], results_M["A*"]["in_memory"][i],
results_M["A*"]["expanded"][i], results_M["A*"]["duplicate_detected"][i],
results_M["A*"]["duplicate_undetected"][i], results_M["A*"]["runtime"][i]])
  rows.append([initial_states[i].__str__(), "IDA*", "Misplaced Tiles",
results_M["IDA*"]["generated"][i], results_M["IDA*"]["in_memory"][i],
results_M["IDA*"]["expanded"][i], results_M["IDA*"]["duplicate_detected"][i],
results_M["IDA*"]["duplicate_undetected"][i], results_M["IDA*"]["runtime"][i]])

  rows.append([initial_states[i].__str__(), "A*", "Manhattan Distance",
results_MD["A*"]["generated"][i], results_MD["A*"]["in_memory"][i],
results_MD["A*"]["expanded"][i], results_MD["A*"]["duplicate_detected"][i],
results_MD["A*"]["duplicate_undetected"][i], results_MD["A*"]["runtime"][i]])
  rows.append([initial_states[i].__str__(), "IDA*", "Manhattan Distance",
results_MD["IDA*"]["generated"][i], results_MD["IDA*"]["in_memory"][i],
results_MD["IDA*"]["expanded"][i], results_MD["IDA*"]["duplicate_detected"][i],
results_MD["IDA*"]["duplicate_undetected"][i], results_MD["IDA*"]["runtime"][i]])

#   results[algo]["generated"].append(generated)
#   results[algo]["in_memory"].append(in_memory)
#   results[algo]["expanded"].append(expanded)
#   results[algo]["duplicate_detected"].append(duplicate_detected)
#   results[algo]["duplicate_undetected"].append(duplicate_undetected)
#   results[algo]["runtime"].append(runtime)

print(rows)

write_rows_to_excel("data_M_vs_MD.xlsx", "Sheet1", rows)

initial_states = []
inputs_num = 50
# reading the inputs
df = pd.read_excel("inputs.xlsx", sheet_name='Sheet1', usecols="A:A",
nrows=inputs_num, skiprows=0, index_col=0)
#print(df)
for x in df.index:
   tmp = x.split('\n')[:3]
   new_input = []
   for t in tmp:
      tmp2 = t.split(' ')[:3]
      list_of_ints = [int(item) for item in tmp2]
      new_input.append(list_of_ints)
   initial_states.append(Game_8_Puzzle(new_input))
   #print(new_input)
#print(initial_states)

# Run the experiment
```

```
results_MD = run_experiment_MD(initial_states)

# Plot the results
plot_results(results_MD, "Manhattan Distance")


def plot_results_without_labels(results, h):
   metrics = ["generated", "in_memory", "expanded", "duplicate_detected",
"duplicate_undetected", "runtime"]
   num_states = len(initial_states)
   x = np.arange(1, num_states + 1)  # Label initial states from 1 to num_states
   for metric in metrics:
      a_star_values = results["A*"][metric]
      ida_star_values = results["IDA*"][metric]

      bar_width = 0.35  # Width of the bars
      fig, ax = plt.subplots(figsize=(20, 12))

      bar1 = ax.bar(x - bar_width/2, a_star_values, bar_width, label='A*',
color='blue')
      bar2 = ax.bar(x + bar_width/2, ida_star_values, bar_width, label='IDA*',
color='orange')

      ax.set_xlabel('Initial State Index')
      ax.set_ylabel(metric.replace("_", " ").capitalize())
      ax.set_title(f'Comparison of A* and IDA* on {metric.replace("_", "
").capitalize()} with {h} heuristic')
      ax.set_xticks(x)
      ax.legend()

      # # Optional: Add labels above the bars
      # def add_labels(bars):
      #    for bar in bars:
      #       height = bar.get_height()
      #       ax.annotate(f'{height}',
      #                xy=(bar.get_x() + bar.get_width() / 2, height),
      #                xytext=(0, 3),  # 3 points vertical offset
      #                textcoords="offset points",
      #                ha='center', va='bottom')

      # add_labels(bar1)
      # add_labels(bar2)

      plt.show()


plot_results_without_labels(results_MD, "Manhattan Distance")
```

```
rows = [["Initial_State", "Algorithm", "Heuristic", "Generated_Vertices",
"Vertices_In_Memory", "Expanded_Vertices", "Duplicate_Detected",
"Duplicate_Undetected", "Run_Time"]]
# ["generated", "in_memory", "expanded", "duplicate_detected",
"duplicate_undetected", "runtime"]
for i in range(len(initial_states)):
  rows.append([initial_states[i].__str__(), "A*", "Manhattan Distance",
results_MD["A*"]["generated"][i], results_MD["A*"]["in_memory"][i],
results_MD["A*"]["expanded"][i], results_MD["A*"]["duplicate_detected"][i],
results_MD["A*"]["duplicate_undetected"][i], results_MD["A*"]["runtime"][i]])
  rows.append([initial_states[i].__str__(), "IDA*", "Manhattan Distance",
results_MD["IDA*"]["generated"][i], results_MD["IDA*"]["in_memory"][i],
results_MD["IDA*"]["expanded"][i], results_MD["IDA*"]["duplicate_detected"][i],
results_MD["IDA*"]["duplicate_undetected"][i], results_MD["IDA*"]["runtime"][i]])

#   results[algo]["generated"].append(generated)
#   results[algo]["in_memory"].append(in_memory)
#   results[algo]["expanded"].append(expanded)
#   results[algo]["duplicate_detected"].append(duplicate_detected)
#   results[algo]["duplicate_undetected"].append(duplicate_undetected)
#   results[algo]["runtime"].append(runtime)

print(rows)
write_rows_to_excel("data_MD.xlsx", "Sheet1", rows)
```