Puzzle Solver

By Applying Problem Solving Techniques

Abdullah Al Raqibul Islam

University of North Carolina at Charlotte | Id# 801151189

Project Goal

The goal of this project is to build a generic interactive pluggable application for solving puzzles (i.e. 8-puzzle, 15-puzzle, 8-queen, Sudoku, etc.) using different problem-solving techniques (i.e. informed search, uninformed search, etc.). Here by word "pluggable" we mean in solving puzzles user can independently decide the search strategy along with the custom heuristic functions. If needed, user can extend this project to device their own solution with a very low effort. Besides, this application is designed in a way that, it will be an easy-going platform for benchmarking any puzzles w.r.t. different state space search strategy, optimization techniques and heuristic functions.

Project Structure

The project has several independent parts that we combine to work as a whole. Directory "core" contains two factory methods that produce the puzzle solver and heuristic instance based on the parameter passed. Puzzle solutions can be implemented in separate directory as we have done for "8-puzzle" here. All the heuristic implementations placed in the "heuristic" directory. Directory "utils" all the utility methods that help other functions to operate.

Please keep in mind, this is an active project and project architecture may change without any prior announcement.

How to Build

```
# go to project directory
$ cd puzzle-solver

# build the project
$ make clean && make
```

How to Run

General run command:

```
./puzzle -problem {PUZZLE_NAME} -algo {ALGORITHM_NAME} -mode {MODE} -heu
{HEURISTIC_METHOD} -initial {INITIAL_STATE_OF_THE_GAME} -goal
{GOAL_STATE_OF_THE_GAME} -print_path {PRINT_INITIAL_TO_GOAL}
```

Here is the parameter definition,

1. **-problem:** Specify the puzzle name to solve, for example, "8-puzzle".

2. **-algo:** Specify the search strategy to solve the puzzle, for example, "A*", "bfs", "dfs", etc.
3. **-mode:** Specify the inner methodology for the search strategy, for example, "bi-directional" bfs, "stack-based" dfs, etc.
4. **-heuristic:** Specify the heuristic function you want to use, for example, "hamming", "manhattan", "euclidean", etc. [default: manhattan]
5. **-initial:** Specify the initial board setup for your puzzle
6. **-goal:** Specify the goal state of your puzzle
7. **-print_path:** Flag to indicate printing path if solution exist, accepts boolean string, i.e., "true" or "false".

8-puzzle

Problem Formulation

8-puzzle problem is a classical state space search problem. Here, we device several search algorithms to solve 8-puzzle problem. 8-puzzle has an initial state, from where we explore possible search paths with a strategy to reach the goal state. Informed search strategies associate a path cost g(n) with a heuristic cost h(n) to find a optimal solution.

Implementation Domain

Currently we have implemented the following heuristic functions:

Heuristic Functions	Parameter token for this application
Hamming Distance	hamming
Manhattan Distance	manhattan
Euclidean Distance	euclidean

If you do not specify any particular heuristic function in the program parameter, manhattan will be considered as the default one.

We have used the following algorithms (with the modes) for solving 8-puzzle,

	bi-directional	greedy	recursive	stack-based	optimized
a_star	-	-	-	-	-
ida_star	-	-	-	-	-
bfs	y	y	-	-	y
dfs	-	-	y	y	-
dls	y	-	-	-	-
ids					

Please keep in mind that, for the current implementation all the names specified above is actually expected as the parameter name.

Motivation of Algorithm Mode

While implementing different algorithms for solving 8-puzzle problem, we found some optimization techniques help the algorithms' to be more efficient. These techniques can be rigid or independent w.r.t. the algorithm itself. For example, bi-directional search technique is very common in the domain of searching algorithms and can be applied to a large number of algorithms. Bi-directional search launch two searches, one from the initial state to the goal state and another one from the reverse direction. The key benefit here is it help reduce the search space, as we know the search trees grow exponentially by their depth. From table 1, we can find that the bi-directional strategy generates at least 2x less nodes comparing to the original algorithms (i.e. Optimized BFS vs. Bi-directional BFS). Fig 1 give a brief idea about the state space reduction by bi-directional strategy while applied in BFS. We can also observe in some cases within some constraints the original algorithm fails to reach the goal, whereas applying bi-directional approach may help reaching the goal (i.e. DLS vs. Bi-directional DLS for input 5). This become possible due to the lower search space generated by bi-directional approaches.

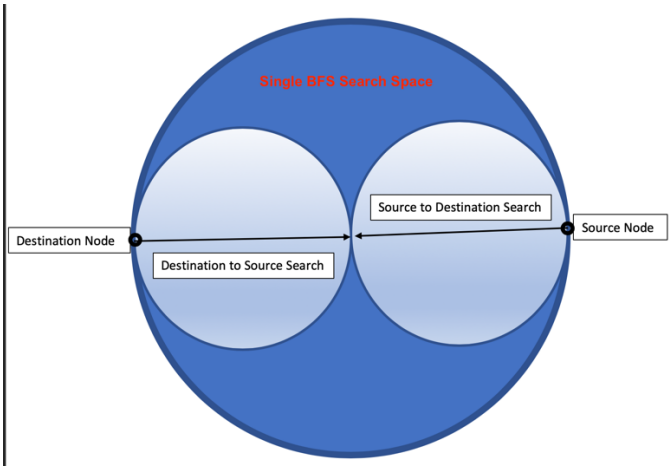


Figure 1: Showing search space reduction by bi-directional search

Some techniques can be applied to very selective algorithms. For example, DFS is a very well known algorithm in the domain of state space search and can be implemented in recursive and non-recursive (stack based) way. The benefit of using stack based DFS is avoiding stack overflow due to excessively deep recursions.

Basic Algorithm Description & Run Commands

For 8-puzzle, here is the list of run commands for different implemented algorithms along with the corresponding mode.

- A*

Implemented standard A* algorithm, used summation of heuristic cost and path cost to choose the node to explore. Used priority_queue from C++ Standard Template Library (STL).

Run command:

```
./puzzle -problem 8-puzzle -algo a_star -mode na -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- IDA*

Iterative deepening A* (IDA*), a variant of iterative deepening depth-first search (IDS), that borrows the idea of using heuristic function to evaluate the remaining cost to get to the goal from the A* search algorithm. As the working principle is depth-first search, you can consider IDA* as a memory constraint version of A* algorithm. Here for 8-puzzle while applying IDA*, we consider the constraint of the maximum cost (heuristic cost + path cost) as 200.

Run command:

```
./puzzle -problem 8-puzzle -algo ida_star -mode na -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- BFS

This one is the plain implementation of BFS, where the goal test is perform on the exploration phase.

Run command:

```
./puzzle -problem 8-puzzle -algo bfs -mode na -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- Optimized BFS

This is a slight variation of the previous one, where the goal test is perform on the new state generation phase. So the number of generated and expanded nodes has significantly reduced.

Run command:

```
./puzzle -problem 8-puzzle -algo bfs -mode optimized -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- BFS Greedy

This variation of BFS is also known as best-first search, where we consider the heuristic path cost to predict the goal and expand the node which has lowest such cost. For efficient selection of the current best node to expand, We used priority_queue from C++ Standard Template Library (STL). From table 2 we can observe that, BFS greedy performs the best in solving 8-puzzle. We will discuss about this later.

Run command:

```
./puzzle -problem 8-puzzle -algo bfs -mode greedy -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- Bi-directional BFS

Started 2 search simultaneously, one from initial state to the goal and another in the reverse direction. To avoid the complexity of multiple threads, we make an alternate strategy to expand the nodes by a single layer from both direction. By this strategy, the search from initial state to the goal state expand one depth and check if any solution found from the reverse direction (i.e. goal to the initial state). If no such solution found, then we moved to the reverse direction search and similarly expand by a single depth and perform the same operation.

Run command:

```
./puzzle -problem 8-puzzle -algo bfs -mode bi-directional -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- Recursive DFS

This is the plain recursive version of DFS.

Run command:

```
./puzzle -problem 8-puzzle -algo dfs -mode recursive -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- Stack Based DFS

To avoid the stack overflow by deep recursion, we introduce the stack based implementation of DFS. We found it particularly useful as from the table 2 we can observer recursive DFS caught segmentation fault in 4 out of 10 cases due to stack overflow. On the other hand, stack based DFS receive 0 segmentation fault.

Run command:

```
./puzzle -problem 8-puzzle -algo dfs -mode stack-based -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- DLS

This is the plain depth limit search implementation, where we consider maximum depth as 40.

Run command:

```
./puzzle -problem 8-puzzle -algo dls -mode na -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- Bi-directional DLS

This is the bi-directional version of the depth limit search. As we launch 2 search while applying bi-directional strategy, we halved the maximum depth of the search space comparing to the plain DLS.

Run command:

```
./puzzle -problem 8-puzzle -algo dls -mode bi-directional -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- IDS

Similar to DLS, maximum depth is set to 40.

Run command:

```
./puzzle -problem 8-puzzle -algo ids -mode na -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

- Bi-directional IDS

Similar to bi-directional DLS, halved the maximum depth of the search space comparing to the plain IDS.

Run command:

```
./puzzle -problem 8-puzzle -algo ids -mode bi-directional -heu manhattan -initial 1,2,3,7,4,5,6,8,0 -goal 1,2,3,8,6,4,7,5,0 -print_path false
```

Sample Board Configuration

All the test cases is listed in the following table. Input 6 is the base case, where the initial state and the goal state is actually same. Input 9 is the impossible case where the solution actually not exist. Input 7 and 8 is considered as the hardest eight-puzzle instances, as both require at least 31 moves to solve.

	initial state	goal state		initial state	goal state
input [1]	1 2 3 7 4 5 6 8 0	1 2 3 8 6 4 7 5 0	input [2]	2 8 1 3 4 6 7 5 0	3 2 1 8 0 4 7 5 6
input [3]	1 2 3 4 5 6 7 8 0	1 2 3 4 5 6 7 8 0	input [4]	2 8 1 3 4 6 7 5 0	2 1 6 3 8 0 7 4 5
input [5]	4 1 3 2 5 6 7 8 0	1 2 3 4 5 6 7 8 0	input [6]	1 2 3 4 5 6 7 8 0	1 2 3 4 8 5 7 6 0
input [7]	8 6 7 2 5 4 3 0 1	1 2 3 4 5 6 7 8 0	input [8]	6 4 7 8 5 0 3 2 1	1 2 3 4 5 6 7 8 0
input [9]	2 1 3 8 0 4 6 7 5	1 2 3 4 5 6 7 8 0	input [10]	1 3 4 8 6 2 7 0 5	1 2 3 8 0 4 7 6 5

Performance Characterization

Algorithm Performance

The following table contains the performance (w.r.t. the number of generated and expanded nodes) of the algorithms in solving 8-puzzle. If not stated otherwise, all the performance in this report is demonstrated as the format of {# of generated nodes} / {# of expanded nodes}. We plotted the performance metric for the three input cases (input 1, 4 and 8) in the Fig 2.

Algorithm\Input	input [1]	input [2]	input [3]	input [4]	input [5]	input [6]	input [7]	input [8]	input [9]	input [10]
a_star	17 / 9	12 / 6	1 / 0	10 / 5	32 / 16	8 / 4	11927 / 7535	13119 / 8296	Solution Not Found 241921 / 181440	10 / 5
ida_star	19 / 9	13 / 6	1 / 0	11 / 5	29 / 13	10 / 4	8549 / 5336	11427 / 7122	Solution Not Found 13822196 / 8491127	12 / 5
bfs	328 / 196	104 / 63	1 / 1	71 / 42	333 / 203	37 / 20	181440 / 181440	181440 / 181440	Solution Not Found 181440 / 181440	69 / 38
bfs [optimized]	195 / 113	62 / 37	1 / 0	41 / 24	202 / 116	19 / 10	181439 / 181385	181439 / 181399	Solution Not Found 181440 / 181440	37 / 22
bfs [greedy]	17 / 9	12 / 6	1 / 0	10 / 5	20 / 10	8 / 4	146 / 89	213 / 123	Solution Not Found 181440 / 181440	10 / 5
bfs [bi-directional]	59 / 29	34 / 19	2 / 0	20 / 10	49 / 24	12 / 6	16129 / 10240	16164 / 10261	Solution Not Found 362878 / 362657	24 / 11
dfs [recursive]	45902 / 25414	segfault	1 / 0	47 / 25	3302 / 1825	1539 / 849	segfault	segfault	segfault	58013 / 32132
dfs [stack-based]	172532 / 139521	44928 / 25911	0 / 0	132509 / 89866	106652 / 67853	53 / 30	111292 / 71403	112424 / 72473	Solution Not Found 181439 / 181440	98579 / 61635
dls	89990 / 54204	89991 / 54206	1 / 0	121282 / 73158	Solution Not Found 89991 / 54205	110642 / 66666	104943 / 58901	103680 / 58336	Solution Not Found 111711 / 67385	104943 / 58903

Algorithm\Input	input [1]	input [2]	input [3]	input [4]	input [5]	input [6]	input [7]	input [8]	input [9]	input [10]
dls [bi-directional]	2515 / 1517	2782 / 1677	1 / 0	4665 / 2802	6809 / 4072	2801 / 1687	16254 / 9094	7751 / 4319	Solution Not Found 29119 / 17439	1303 / 727
ids	720 / 423	662 / 390	1 / 0	47 / 25	segfault	669 / 394	segfault	92439 / 51938	segfault	52477 / 29331
ids [bi-directional]	2515 / 1517	2782 / 1677	1 / 0	4665 / 2802	6809 / 4072	2801 / 1687	16253 / 9094	7750 / 4319	segfault	1303 / 727

From our experiment, we observed several interesting facts regarding the performance of algorithms in solving 8-puzzle.

1. Bi-directional optimization is very effective in the state space search, improve the performance by a factor of 2x to 40x comparing with the single directional implementation of the corresponding algorithm.
2. Bi-directional optimization not only make the algorithm efficient but also help the algorithms achieving the completeness withing the comparable constraints. For example, previously we mentioned the depth of the bi-directional DLS and IDS halved the maximum depth of the search space. For input 5 bi-directional DLS and IDS is able to find the goal state whereas the single directional one not able to find the solution (IDS got segfault). We believe this is due to the boundary constraint of the algorithms.
3. IDA* performs better than A* with a factor of 1.3x, supports the previous statement of IDA* as a memory constraint version of A* algorithm.
4. Greedy BFS performs better than A* and IDA* and accordingly considered the best among all the algorithms. We believe this is due to the unweighted properties of the 8-puzzle problem, as greedy BFS only considered the heuristic cost to reach the goal while choosing node to expand.
5. For deep recursion like finding goal state of 8-puzzle, stack based DFS is a better choice comparing to the recursive one (0 vs 3 segmentation fault).
6. DLS perform the worst among all the algorithms, but it reaches to goal state more than IDS (two segmentation fault vs. one solution not found).

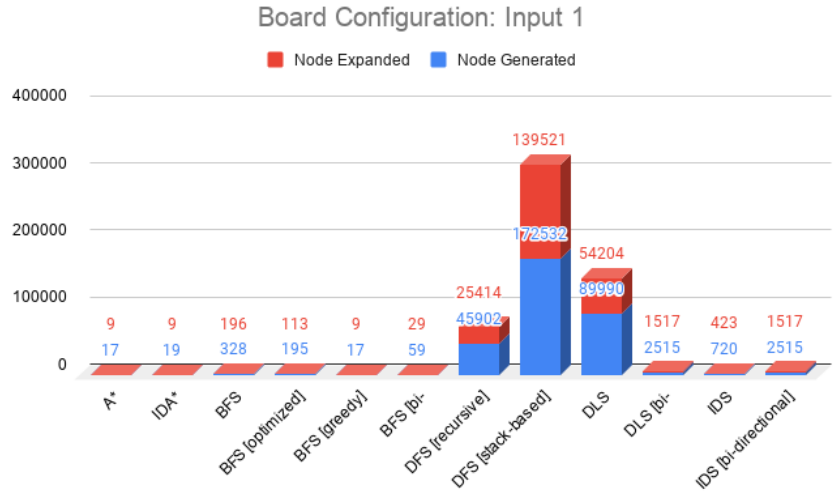
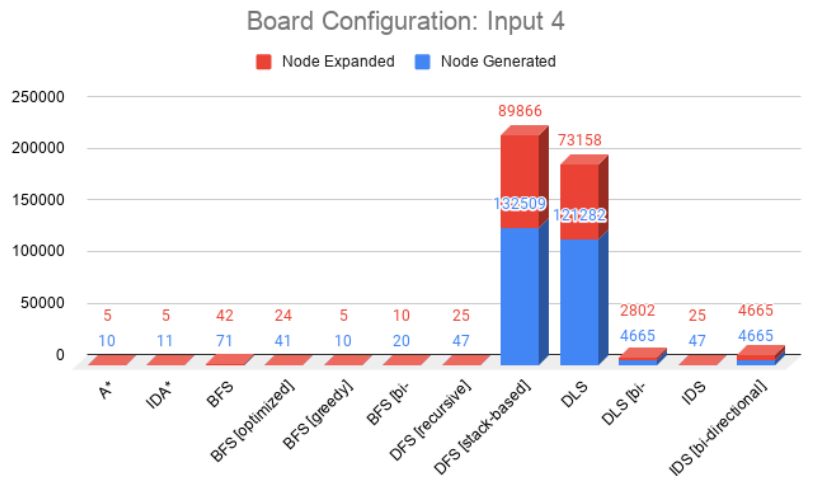
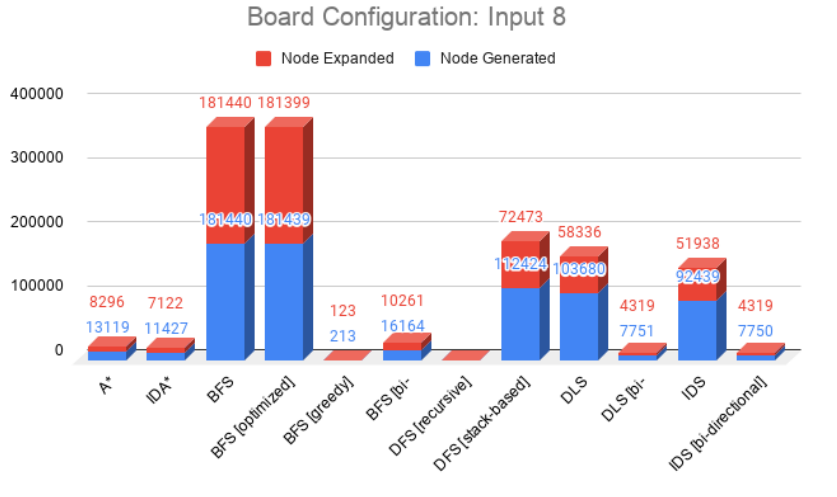
Figure 2 (a): Input 1	
Figure 2 (b): Input 4	
Figure 2 (c): Input 8	

Figure 2: Comparing performance of different algorithms for solving 8-puzzle

Heuristic Function Performance in A* Algorithm

The following table compares the performance of heuristic methods in A* algorithm. We found euclidean distance perform best among all the heuristic functions. Considering number of generated nodes, euclidean distance outperform manhattan distance by 108x and hamming distance by 1485x (input 8). Consequently hamming distance perform worst among all the heuristic functions. For impossible case, all the heuristic function perform same as expected. We also plotted the performance metric for the three input cases (input 1, 7 and 8) in the Fig 2.

Heuristic\Input	input [1]	input [2]	input [3]	input [4]	input [5]	input [6]	input [7]	input [8]	input [9]	input [10]
Manhattan Distance	17 / 9	12 / 6	1 / 0	10 / 5	32 / 16	8 / 4	11927 / 7535	13119 / 8296	Solution Not Found 241921 / 181440	10 / 5
Hamming Distance	40 / 21	14 / 7	1 / 0	10 / 5	39 / 21	10 / 5	186288 / 128467	179773 / 122814	Solution Not Found 241921 / 181440	12 / 6
Euclidean Distance	17 / 9	12 / 6	1 / 0	10 / 5	32 / 16	8 / 4	195 / 116	121 / 73	Solution Not Found 241921 / 181440	10 / 5

Figure 2 (a): Input 1

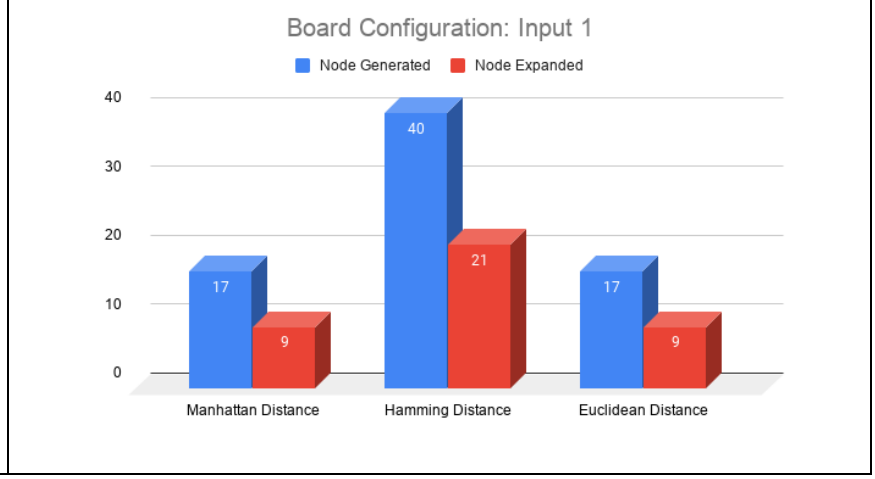
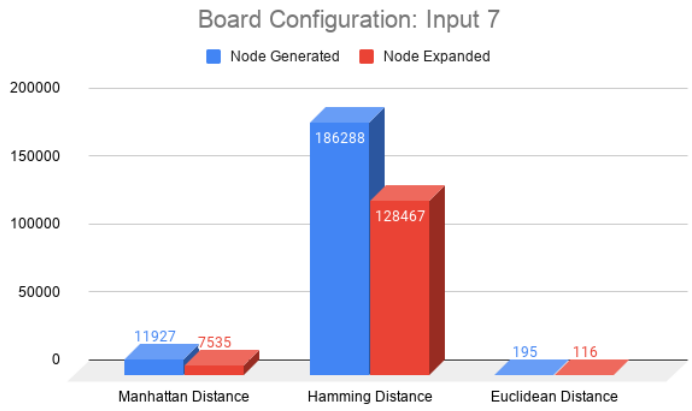
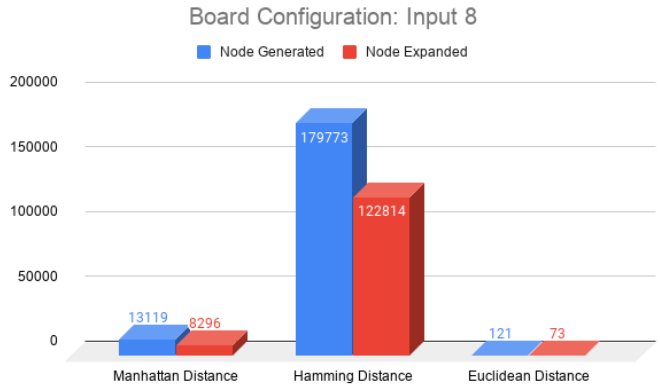


Figure 2 (a): Input 7	
Figure 2 (a): Input 8	
Figure 3: Comparing performance of different heuristic methods for solving 8-puzzle using A* algorithm	

Future Works [8-puzzle]

- Try with more advanced heuristic functions.
- Introduce different performance matrix (i.e. time taken to solve the puzzle, depth at which found the solution, etc.).
- For bi-directional mode, construct proper solution path.
- Try more optimization techniques (i.e. branch and bound).

Project Future Work

- Will solve 8-queen puzzle by different non-classical search strategy.
- Will solve 2 player game by applying minimax decisions and α–β pruning.

Resources

1. [Blog] Problem Solving Techniques
   part1: https://mhesham.wordpress.com/2010/04/08/problem-solving-techniques-part1/
2. [Blog] Problem Solving Techniques
   part2: https://mhesham.wordpress.com/tag/depth-limited-search/
3. [Blog] The hardest eight-puzzle instances: w01fe.com/blog/2009/01/the-hardest-eight-puzzle-instances-take-31-moves-to-solve/

A* Algorithm Path Construction

Heuristic Function: Manhattan Distance

Input 1	Input 2	Input 3	Input 4	Input 5	Input 6
1 2 3 7 4 5 6 8 0 --->` 1 2 3 7 4 0 6 8 5 --->` 1 2 3 7 0 4 6 8 5 --->` 1 2 3 7 8 4 6 0 5 --->` 1 2 3 7 8 4 0 6 5 --->` 1 2 3 0 8 4 7 6 5 --->` 1 2 3 8 0 4 7 6 5 --->` 1 2 3 8 6 4 7 0 5 --->` 1 2 3 8 6 4 7 5 0	2 8 1 3 4 6 7 5 0 --->` 2 8 1 3 4 0 7 5 6 --->` 2 8 1 3 0 4 7 5 6 --->` 2 0 1 3 8 4 7 5 6 --->` 0 2 1 3 8 4 7 5 6 --->` 3 2 1 0 8 4 7 5 6 --->` 3 2 1 8 0 4 7 5 6	1 2 3 4 5 6 7 8 0	2 8 1 3 4 6 7 5 0 --->` 2 8 1 3 4 6 7 0 5 --->` 2 8 1 3 0 6 7 4 5 --->` 2 0 1 3 8 6 7 4 5 --->` 2 1 0 3 8 6 7 4 5 --->` 2 1 6 3 8 0 7 4 5	4 1 3 2 5 6 7 8 0 --->` 4 1 3 2 5 0 7 8 6 --->` 4 1 3 2 0 5 7 8 6 --->` 4 1 3 0 2 5 7 8 6 --->` 0 1 3 4 2 5 7 8 6 --->` 1 0 3 4 2 5 7 8 6 --->` 1 2 3 4 0 5 7 8 6 --->` 1 2 3 4 5 0 7 8 6 --->` 1 2 3 4 5 6 7 8 0	1 2 3 4 5 6 7 8 0 --->` 1 2 3 4 5 0 7 8 6 --->` 1 2 3 4 0 5 7 8 6 --->` 1 2 3 4 8 5 7 0 6 --->` 1 2 3 4 8 5 7 6 0

Heuristic Function: Euclidean Distance

Input 1	Input 2	Input 3	Input 4	Input 5	Input 6
1 2 3 7 4 5 6 8 0 --->` 1 2 3 7 4 0 6 8 5 --->` 1 2 3 7 0 4 6 8 5 --->` 1 2 3 7 8 4 6 0 5 --->` 1 2 3 7 8 4 0 6 5 --->` 1 2 3 0 8 4 7 6 5 --->` 1 2 3 8 0 4 7 6 5 --->` 1 2 3 8 6 4 7 0 5 --->` 1 2 3 8 6 4 7 5 0	2 8 1 3 4 6 7 5 0 --->` 2 8 1 3 4 0 7 5 6 --->` 2 8 1 3 0 4 7 5 6 --->` 2 0 1 3 8 4 7 5 6 --->` 0 2 1 3 8 4 7 5 6 --->` 3 2 1 0 8 4 7 5 6 --->` 3 2 1 8 0 4 7 5 6	1 2 3 4 5 6 7 8 0	2 8 1 3 4 6 7 5 0 --->` 2 8 1 3 4 6 7 0 5 --->` 2 8 1 3 0 6 7 4 5 --->` 2 0 1 3 8 6 7 4 5 --->` 2 1 0 3 8 6 7 4 5 --->` 2 1 6 3 8 0 7 4 5	4 1 3 2 5 6 7 8 0 --->` 4 1 3 2 5 6 7 0 8 --->` 4 1 3 2 0 6 7 5 8 --->` 4 1 3 0 2 6 7 5 8 --->` 0 1 3 4 2 6 7 5 8 --->` 1 0 3 4 2 6 7 5 8 --->` 1 2 3 4 0 6 7 5 8 --->` 1 2 3 4 5 6 7 0 8 --->` 1 2 3 4 5 6 7 8 0	1 2 3 4 5 6 7 8 0 --->` 1 2 3 4 5 0 7 8 6 --->` 1 2 3 4 0 5 7 8 6 --->` 1 2 3 4 8 5 7 0 6 --->` 1 2 3 4 8 5 7 6 0