# **AI algorithms for solving 8-puzzle/L-puzzle**

In this assignment several AI algorithms were implemented to solve 8-puzzle and its generalized version L-puzzle. The algorithms that were used are -

1. Depth First Search

2. Iterative Deepening

3. Breadth First Search

4. A\* Search with Misplaced Heuristic

5. A\* Search with Manhattan Distance Heuristic

[5] Depth First Search is not a good fit to solve this problem, As it traverses a lots of states and doesn’t give an optimal move sequence to solve the puzzle. But using Iterative Deepening we can modify Depth First Search to ensure an optimal solution is produced.

Breadth First Search always gives an optimal solution but it should work slower in the cases where the depth of the optimal solution in higher.

Two heuristics were used with A\* search. Misplaced Heuristic and Manhattan Distance Heuristic. Both of them are Admissible Heuristic. So our A\* Search should always give the optimal solution.

[1] The data structure that was used in the task is a C++ struct. A bare-bone structure of the structure is as follows.

struct state{

vector < vector <int> > board;

int huristic\_value, depth;

unsigned long long hash\_val;

bool is\_terminal;

state() {}

state(vector < vector <int> > board, int depth): board(board), depth(depth){}

int huristic\_misplaced(){}

int huristic\_manhattan(){}

vector <state> successors(){}

void print\_state(){}

bool operator > (const state &s) const {}

};

Variables

|  |  |
| --- | --- |
| Variable Name | Description |
| board | A 2D vector that stores the board |
| huristice\_value | A variable that stores the heuristic value of the board |
| depth | Stores the depth of the state in the search tree |
| hash\_val | Stores a hash value to uniquely identify the board |
| is\_terminal | Stores True if the state is a goal state. Otherwise False |

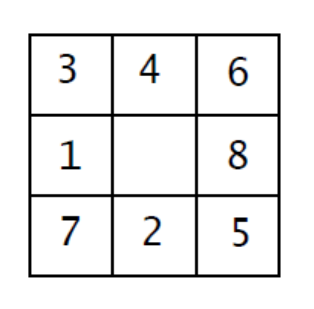
Functions

|  |  |
| --- | --- |
| Function Name | Description |
| huristic\_misplaced() | Calculates the Misplaced heuristic of the state |
| huristic\_manhattan() | Calculates the Manhattan heuristic of the state |
| successors() | Generates all the states that can be reachable from the state in on step. |
| print\_state() | Gives a visual representation of the state |

There is also a operator overloading of < for setting the priority in the priority queue.

**Experiments:**

[4][6][7] Solving using various algorithms:



Multiple simulations using different algorithms were ran on the state above. Here are the findings -

Using bfs,

1. Total visited states: 2513

2. Solution found at depth: 12

3. Time taken to solve: 0.05262

Using iterative deepening,

1. Total visited states: 1208

2. Solution found at depth: 12

3. Time taken to solve: 0.060236

Using A\* search with misplaced heuristic,

1. Total visited states: 176

2. Solution found at depth: 12

3. Time taken to solve: 0.003659

Using A\* search with manhattan heuristic,

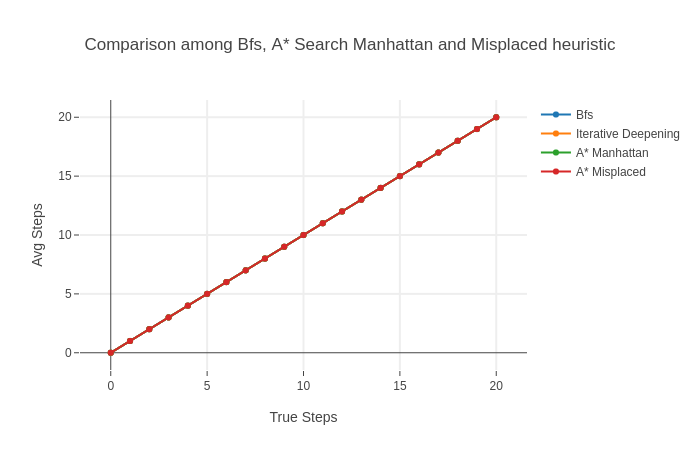
1. Total visited states: 42

2. Solution found at depth: 12

3. Time taken to solve: 0.002313

[8] Monte Carlo Simulation:

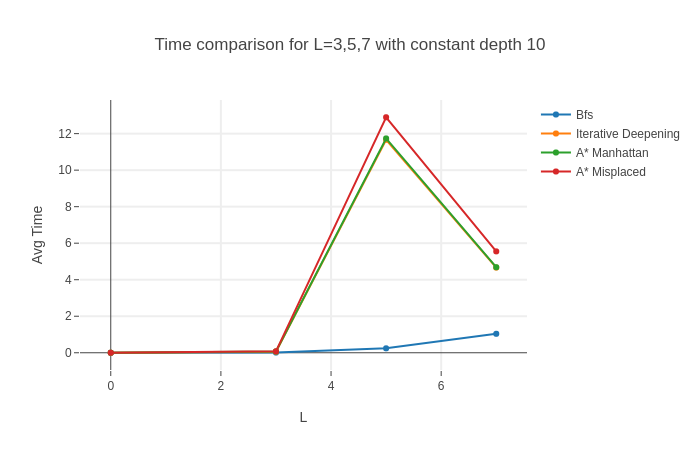
The output graph of the Monte Carlo Simulation came out as follows -



As all four algorithms are supposed to give optimal solution, Hence For each one of them Avg Steps = True Steps

[9] Size vs Time:

10 samples were generated for L = 3, 5, 7. Then average solving time for each L was calculated for each algorithm with constant True Steps = 10. Then they were plotted next to each another. The graph is as follows -

It gives an unexpected behavior. Usually A\* Search works faster than Breadth First Search. But in this case Breadth First Search worked faster. This happened due to lower value of True Steps. With increase of True Step, Breadth First Search will start to get slower and A\* Search will start to become comparatively faster.