SCHOOL OF SCIENCE & ENGINEERING FALL 2023/2024

CSC 4301 Artificial Intelligence

Term Project 1: Report

October 2nd, 2023

Realized by:

Adnane Ahroum 101142

Supervised by:

Professor RACHIDI, Tajjeddine

Teammate:

Anas Temouden 102929

Abstract:

This research project presents a comprehensive examination of enhancements made in the first protocol of the classic Eight Puzzle problem: a fundamental challenge in the field of artificial intelligence. The primary objective of the Eight Puzzle problem is to manipulate tiles within a 3*3 grid, constraining numbers from 1 to 8 with a space. The initial base provided is a solution that lacks efficiency and depth. The task required is to find the optimal solution in between the heuristics presented, to improve the performance of the agent, foster an understanding of the problem, and enhance the overall modularity.

I. Introduction:

Before Starting with the heuristic, we need to mark and clarify the components of our search problem project:

- State space: It would represent a configuration of the puzzle with the start state, goal test, and all the successor functions leading to the solution.
- Initial Space: Symbolizing the starting configuration of this puzzle, which is a 3*3 grid containing 8 numbers that are constantly shuffled and one blank tile. The initial state varies from one instance of a problem to another due to the randomized positions.
- Goal state: the optimal state that displays the final state where the agents stop transitioning. This state should display the tiles containing numbers arranged in an ascending order, and the empty tile on the top left position of the grid.
- Successor Function: the transition states are the successive states demonstrating the adequate link between the start state and the goal test. This would be shown as an order adjusting the blank tile to achieve the goal test, with the verb “Move” and the designated directions “Up, Down, Left, Right”.

- Solution: It is a path or a distance indicating the different successive states to reach the goal test indicating the strategy used.
- World State: All the possible combinations of the whole search problem.

A key focus of this research lies in the introduction of four heuristic functions, which play a pivotal role in informed search algorithms such as "A* Search". These heuristic functions provide estimates of the cost required to find the solution of the Eight Puzzle. Four heuristics will be introduced and analyzed with their implementation in this research as follows:

1. Misplaced Tiles count: This heuristic assesses the count of misplaced tiles at the end of the execution with an estimation of the state's distance from the solution. If the tile contains a number that is in its correct position, it is not added, but if it is misplaced, it is included.
2. Sum of the Euclidian Distances: This heuristic considers the Euclidian distance between each tile's current position and its correct position in the goal state. It provides a much more geometrically informed estimation of the state's distance to the goal by using this equation:

$$Euclidian\ Dist = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + (q_3 - p_3)^2 + \cdots . + (q_n - p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

3. Sum of Manhattan distances: This heuristic calculates the sum of the distances (horizontal and vertical) that each tile reaches its appropriate position that represents the goal position. Manhattan distance structure works with geometric

coordinates (x,y) representing the current tile position, as well as the intended position, or the goal coordinates designated with (goal_x, goal_y), by using a similar idea to the Euclidian states. The distance then can be calculated using the sum of the absolute value of the difference for each tile in both dimensions as follows:

$$Manhattan\ Dist = |x_2 - x_1| + |y_2 - y_1|$$

With the current tile A with coordinates (x_1, y_1), and the goal tile B with goal coordinates (x_2, y_2). The resultant "$Manhattan\ Dist$" is an estimate of number of moves to provide a view of how misaligned the entire grid is.

4. Number of Tiles Out of Row and Column: also called "Row and Column Deviation". The fourth heuristic evaluates the respective deviation of tiles for each row and column from their adequate positions. It provides a unique estimation of the state evaluation, as it considers both row-wise and column-wise during the algorithm.

Through the research, we will go through both exploration and analysis of these heuristics, by figuring out the implementation of each one of them, as well as comparing and contrasting their relative effectiveness to solve our Eight Puzzle problem.

II. Implementation:

1. H1: Misplaced Tiles:

Using Python, we define the variable “a” to hold the result which is the number of misplaced tiles. By using a double loop, we would constantly compare the current state with the goal test, and we would increment by one in case the tiles are misplaced. The higher the count, the farther we are from the goal.

```
def h1(state, problem=None):
    """
    This heuristic caclulate misplaced tiles
    """
    #a is the number of misplaced tiles
    a=0
    #actual_arr is the current state of the board
    actual_arr = (state.cells[:])
    #goal_arr is the goal state of the board
    goal_arr =[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
    #we compare the current state with the goal state and increment a if the tiles are misplaced
    for i in range(3):
        for j in range(3):
            if(goal_arr[i][j] != actual_arr[i][j] and state.cells[i][j]!=0):
                a+=1
    return a
```

Figure1: First Heuristic Implementation (Misplaced Tiles)

2. H2: Euclidian Distance:

This equation can be used as a method to determine the indices that a number should occupy based on its value:

$$(x_i\ , y_i) = (\left\lfloor \frac{value_i}{N} \right\rfloor, value_i, mod\ N)$$

With:

- x: the goal column index in the 0-index system of the number
- y: the goal row index in the 0-index system of the number
- value: the value of that number
- N: the row or the column length of the NxN Puzzle

  With i: 0,1, 2, ..., NxN-1.

Then, we iterate through each number in the current state and calculate the sum using the formulas mentioned earlier. Finally, we return the rounded result to two decimal places as shown below:

```
def h2(state, problem =None):
    """
    This heuristic caclulate euclidian distance
    """
    euclidian_sum=0
    # I calculate the euclidian distance (0 if the number in right place) for each cell and sum them up
    for i in range(3):
        for j in range(3):
            if (state.cells[i][j]!=0):
                euclidian_sum += ((i - (state.cells[i][j] // 3))**2 + (j -  (state.cells[i][j] % 3))**2)**0.5
    return round(euclidian_sum, 2)
```

Figure2: Second Heuristic Implementation (Euclidian Distance)

3. H3: Manhattan Distance:

This heuristic is known to be used in algorithms like A* Search to guide the search process by prioritizing states with lower heuristic values because they are generally expected to be closer to the goal state. The algorithm is similar to the second heuristic, and it goes as follows: Initializing “Manhattan_sum” to be 0, then iterating column and row indices of the 3x3 grid, and checking if the tile is not empty as it represents our base case, then it calculates the absolute value of both vertical and horizontal distances between the current column and row indexes ([i] [j]), and the desired position of the tile.

```python
def h3(state, problem=None):
    """
    This heuristic caclulate manhattan distance
    """

    manhattan_sum=0
    # I calculate the manhattan distance (0 if the number in right place) for each cell and sum them up
    for i in range(3):
        for j in range(3):
            if(state.cells[i][j]!=0):
                manhattan_sum += abs(i - (state.cells[i][j] // 3)) + abs(j -  (state.cells[i][j] % 3))
    return manhattan_sum
```

Figure 3: Third Heuristic Implementation (Manhattan Distance)

4. H4: Out of Row and Column:

The implementation is designed to iterate over each tile and compare the actual row-index, and column-index with the correct index. Otherwise, we increment “total_misplaced” containing the number of tiles that deviate from their suitable position.

```python
def h4(state, problem=None):
    """
    This heuristic caclulate number of tiles out of row and column
    """
    total_misplaced=0
    for i in range(3):
        for j in range(3):
            if(state.cells[i][j]//3 != i and state.cells[i][j]!=0):
                total_misplaced+=1
            if(state.cells[i][j]%3 != j and state.cells[i][j]!=0):
                total_misplaced+=1
    return total_misplaced
```

Figure 4: Fourth Heuristic Implementation (Number of Tiles)

III. Comparing Heuristics:

When solving the 8-puzzle problem, the choice of search algorithm is crucial for achieving efficient results. The 8-puzzle problem is not only a classic puzzle but also represents a broader class of combinatorial search problems. Its solution requires navigating a complex state space by making informed decisions and minimizing the number of moves needed to reach the goal state. Different search algorithms approach this task with varying levels of efficiency and optimality.

1. Solution Quality:

   The best heuristic should provide accurate estimates of the number of moves required to reach the goal state. Heuristics that consistently produce lower values for the number of misplaced tiles or Manhattan distances tend to guide the search more effectively. Heuristics that yield towards lower metric values such as H1 or H3, tend to guide the search more effectively. Thus, the quest for a superior heuristic involves evaluating its ability to offer precise predictions regarding the effort required to reach the desired goal configuration.

2. Search Efficiency:

   The best heuristic should lead to a more efficient search process by expanding to fewer nodes. Heuristics that guide the search towards the goal state with fewer expanded nodes can significantly reduce the computational effort needed to find a solution. A heuristic's efficiency is closely linked to its admissibility: the property of never overestimating the true cost to reach the goal. Therefore, the selection of a heuristic should be guided by its ability to optimize search efficiency without sacrificing solution quality.

3. Computational Complexity:

   Heuristics that require less computational effort to compute their values are generally preferred. Complex heuristics may introduce significant overhead, impacting the overall performance of the search algorithm. Complex heuristics may be impractical for solving large-scale instances of the 8-puzzle problem due to their computational demands. Consequently, the choice of heuristic should strike a balance between accuracy and computational efficiency to ensure a practically viable search algorithm.

4. Empirical Evaluation:

   Conducting experiments and analyzing the performance of the heuristics on a set of representative problem instances can provide valuable insights. Comparing their execution times, number of expanded nodes, and solution quality metrics can help identify the best heuristic for the 8-puzzle problem. Such evaluations are paramount in guiding the choice of heuristics for solving the 8-puzzle problem effectively.

Now, we will move on to analyze each heuristic alone, proposing into the table their characteristics and potential effectiveness, considering the four key points mentioned above to choose the most reliable heuristic for our problem.

1) H1

   - This admissible heuristic is characterized by its simplicity as it never overestimates the number of moves required to reach the goal.
   - However, it may not be very informative as it treats all misplaced tiles equally and does not consider their position and the distance from their goal state.

2) H2:

- This heuristic considers the spatial relationship between tiles and provides a more accurate geometric estimate of the distance, as well as being admissible as the first one.
- It is effective, but not as effective in our search problem, as the Euclidian heuristic assumes a straight-line path as a solution from the starting state to reach the goal state, and it doesn't consider the blocked paths and obstacles that can be found during the testing period. This may lead to suboptimal or inefficient search path.
- In addition to that, in our case, the tiles can only move in one direction at a time, but this function treats the movement in any direction with a uniform cost, disregarding the constraints in the rules of the 8-puzzle problem.

3) H3:

- Considered as the most optimistic and admissible heuristic between all the others, guaranteeing the optimal path.
- Manhattan distance considers obstacles and recognizes when the tile impedes its movement from a straight-line path, making the estimations more accurate and not overestimated.
- The Computation of Manhattan is the quickest of them to implement, involving simple arithmetic operations, making it lightweight to be processed computationally compared to more complex heuristics.

- This Heuristic doesn't take into consideration the order of arrangement of the tiles, it works to consider the individual positions of each tile, and their distances from the goal. Thus, this can lead to sensitivity to certain patterns or configurations of tiles that require specific movement.

4) H4:

- Captures the extent to which the tiles deviate from their desired row and column positions, making it completely unique than the other worked heuristics.
- Admissibility.
- Insufficient Information computed, as H4 doesn't consider the intermediate steps or interactions between tiles, nor the spatial relationships, nor the distances. It only checks the row and column position, which leads to suboptimal estimations and underestimating the number of moves for complex situations.

By analyzing all the available information above, and comparing all the possible pros and cons, we can clearly decide that the third heuristic "Sum of Manhattan Distances" is the most suitable and efficient heuristic to be implemented in the eight-puzzle problem. After analyzing the mathematical difference between the H2 and H3, we can see that the Manhattan offers many more possibilities.

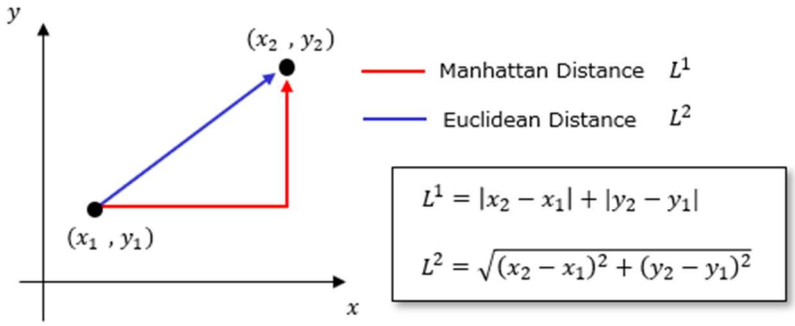

Figure 5: Graph showing the difference between Manhattan and Euclidian Distance


result_task2.csv

Index	Expanded	Fringe	Depth	Path
s1:h1	15	29	7	["up', 'left', 'down', 'right', 'up', 'up', 'left']"
s1:h2	9	20	7	["up', 'left', 'down', 'right', 'up', 'up', 'left']"
s1:h3	9	20	7	["up', 'left', 'down', 'right', 'up', 'up', 'left']"
s1:h4	9	20	7	["up', 'left', 'down', 'right', 'up', 'up', 'left']"
s2:h1	157	268	13	["right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'down', 'left', 'up', 'left', 'up']"
s2:h2	39	68	13	["right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'down', 'left', 'up', 'left', 'up']"
s2:h3	34	58	13	["right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'down', 'left', 'up', 'left', 'up']"
s2:h4	40	71	13	["right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'down', 'left', 'up', 'left', 'up']"
s3:h1	1047	1698	17	["down', 'left', 'up', 'right', 'down', 'left', 'left', 'up', 'up', 'right', 'down', 'down', 'right', 'up', 'up', 'left', 'left']"
s3:h2	323	522	17	["down', 'left', 'up', 'right', 'down', 'left', 'left', 'up', 'up', 'right', 'down', 'down', 'right', 'up', 'up', 'left', 'left']"
s3:h3	308	493	17	["down', 'left', 'up', 'right', 'down', 'left', 'left', 'up', 'up', 'right', 'down', 'down', 'right', 'up', 'up', 'left', 'left']"
s3:h4	440	728	17	["down', 'left', 'up', 'right', 'down', 'left', 'left', 'up', 'up', 'right', 'down', 'down', 'right', 'up', 'up', 'left', 'left']"
s4:h1	39	69	9	["down', 'right', 'up', 'up', 'left', 'down', 'right', 'up', 'left']"
s4:h2	17	35	9	["down', 'right', 'up', 'up', 'left', 'down', 'right', 'up', 'left']"
s4:h3	16	32	9	["down', 'right', 'up', 'up', 'left', 'down', 'right', 'up', 'left']"
s4:h4	19	38	9	["down', 'right', 'up', 'up', 'left', 'down', 'right', 'up', 'left']"
s5:h1	7	15	7	["left', 'up', 'right', 'right', 'up', 'left', 'left']"
s5:h2	7	15	7	["left', 'up', 'right', 'right', 'up', 'left', 'left']"
s5:h3	7	15	7	["left', 'up', 'right', 'right', 'up', 'left', 'left']"
s5:h4	7	15	7	["left', 'up', 'right', 'right', 'up', 'left', 'left']"
s6:h1	12	25	7	["down', 'left', 'up', 'right', 'up', 'left', 'left']"
s6:h2	10	19	7	["down', 'left', 'up', 'right', 'up', 'left', 'left']"
s6:h3	10	19	7	["down', 'left', 'up', 'right', 'up', 'left', 'left']"
s6:h4	10	19	7	["down', 'left', 'up', 'right', 'up', 'left', 'left']"
s7:h1	3	9	3	["down', 'left', 'up']"
s7:h2	3	9	3	["down', 'left', 'up']"
s7:h3	3	9	3	["down', 'left', 'up']"
s7:h4	3	9	3	["down', 'left', 'up']"
s8:h1	7	16	5	["down', 'right', 'up', 'left', 'up']"
s8:h2	6	13	5	["down', 'right', 'up', 'left', 'up']"
s8:h3	6	13	5	["down', 'right', 'up', 'left', 'up']"
s8:h4	6	13	5	["down', 'right', 'up', 'left', 'up']"
s9:h1	7	13	7	["down', 'right', 'right', 'up', 'up', 'left', 'left']"
s9:h2	7	13	7	["down', 'right', 'right', 'up', 'up', 'left', 'left']"
s9:h3	7	13	7	["down', 'right', 'right', 'up', 'up', 'left', 'left']"
s9:h4	7	13	7	["down', 'right', 'right', 'up', 'up', 'left', 'left']"
s10:h1	149	252	13	["right', 'right', 'up', 'left', 'down', 'right', 'down', 'left', 'left', 'up', 'right', 'up', 'left']"
s10:h2	30	55	13	["right', 'right', 'up', 'left', 'down', 'right', 'down', 'left', 'left', 'up', 'right', 'up', 'left']"
s10:h3	30	55	13	["right', 'right', 'up', 'left', 'down', 'right', 'down', 'left', 'left', 'up', 'right', 'up', 'left']"
s10:h4	49	89	13	["right', 'right', 'up', 'left', 'down', 'right', 'down', 'left', 'left', 'up', 'right', 'up', 'left']"
s11:h1	72	125	11	["down', 'left', 'up', 'left', 'down', 'right', 'right', 'up', 'up', 'left', 'left']"
s11:h2	39	72	11	["down', 'left', 'up', 'left', 'down', 'right', 'right', 'up', 'up', 'left', 'left']"
s11:h3	39	72	11	["down', 'left', 'up', 'left', 'down', 'right', 'right', 'up', 'up', 'left', 'left']"
s11:h4	40	74	11	["down', 'left', 'up', 'left', 'down', 'right', 'right', 'up', 'up', 'left', 'left']"
s12:h1	62	114	11	["left', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'left', 'up']"
s12:h2	33	62	11	["left', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'left', 'up']"
s12:h3	33	62	11	["left', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'left', 'up']"
s12:h4	35	67	11	["left', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'left', 'up']"
s13:h1	12	25	7	["right', 'right', 'down', 'left', 'up', 'left', 'up']"
s13:h2	10	21	7	["right', 'right', 'down', 'left', 'up', 'left', 'up']"
s13:h3	10	21	7	["right', 'right', 'down', 'left', 'up', 'left', 'up']"
s13:h4	10	21	7	["right', 'right', 'down', 'left', 'up', 'left', 'up']"
s14:h1	7	15	7	["right', 'down', 'left', 'down', 'left', 'up', 'up']"
s14:h2	7	15	7	["right', 'down', 'left', 'down', 'left', 'up', 'up']"
s14:h3	7	15	7	["right', 'down', 'left', 'down', 'left', 'up', 'up']"
s14:h4	7	15	7	["right', 'down', 'left', 'down', 'left', 'up', 'up']"
s15:h1	25	49	9	["down', 'left', 'down', 'right', 'right', 'up', 'left', 'up', 'left']"

Figure 6: Results of the Second task (100 tests)

We can conclude that manipulating Manhattan distances is generally useful in grid-like environments such as the 8 puzzle here.

IV. Comparing Strategies:

After we managed to pick the adequacy in terms of heuristic, we needed to apply the Manhattan algorithm with all the used search strategies to see which is the perfect combination that will guarantee the most optimal solution. We worked with three uninformed and one informed search techniques:

1. Uninformed search:

.) DFS (Depth First Search):

By applying the LIFO strategy (Last in First Out), the implementation can be explained as:

Creating a stack to store the states, then pushing the initial state onto the stack. Then pop, and check if it is goal state, which will be our base case. Else, the algorithm generated all the possible successive functions transitioning from the initial state to the next one and updated the fringe length and the expanded node(s). This lacks optimality because it is exploring all the nodes and it may encounter a loop which may lead to an infinite path that doesn't guarantee a solution, which makes the depth and the fringe go into unexpected huge numbers as shown below:

result_task3.csv

```
Index  ,Expanded ,Fringe ,Depth ,Path
s1:BFS ,       0 ,     0 ,    0 ,[]
s1:DFS ,       0 ,     0 ,    0 ,[]
s1:UCS ,       0 ,     0 ,    0 ,[]
s1:A*  ,       0 ,     1 ,    0 ,[]
s2:BFS ,       3 ,     6 ,    2 ,"['up', 'up']"
s2:DFS ,      30 ,    42 ,   30 ,"['right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up',
'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'up']"
s2:UCS ,       3 ,     9 ,    2 ,"['up', 'up']"
s2:A*  ,       2 ,     3 ,    2 ,"['up', 'up']"
s3:BFS ,       6 ,    12 ,    3 ,"['left', 'left']"
s3:DFS ,       2 ,     4 ,    2 ,"['left', 'left']"
s3:UCS ,       6 ,    19 ,    3 ,"['left', 'left']"
s3:A*  ,       2 ,     3 ,    2 ,"['left', 'left']"
s4:BFS ,       5 ,    12 ,    2 ,"['up', 'left']"
s4:DFS ,    2655 ,  3316 , 2602 ,"['right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down',
'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'up', 'right', 'right', 'down',
'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left',
'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'up', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left',
'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left',
'up', 'up', 'right', 'right', 'down', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left',
'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'up', 'right', 'down', 'left', 'left',
'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up',
'right', 'right', 'down', 'left', 'left', 'up', 'up', 'right', 'right', 'down', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down',
'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right',
'up', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right',
'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'up', 'right', 'right', 'down', 'left', 'left', 'down', 'right',
'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right',
'up', 'left', 'left', 'down', 'right', 'up', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right',
'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'up', 'right', 'right',
'down', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up',
'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'up', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down',
'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left',
'left', 'up', 'up', 'right', 'right', 'down', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left',
'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'up', 'right', 'down', 'left',
'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'left',
'up', 'right', 'right', 'down', 'left', 'left', 'up', 'up', 'right', 'right', 'down', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left',
'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'up',
'right', 'right', 'down', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right',
'right', 'up', 'left', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'right', 'up', 'right', 'down', 'left', 'left', 'up', 'right',
```

Figure 6: Result showing the issue with DFS Search (line 15 in VSC)

.) BFS (Breath First Search):

By Applying FIFO strategy (First in First Out), the implementation can be explained as:

Creating a queue to store the states, as well as a list to store the visited states. Then, loop if the queue is not empty, and check the dequeued state if it is the goal test (base case). If it is not the case, enqueue all the possible successive function states into the queue and add the visited one into the list. BFS's time complexity is a huge number that varies in the exponential, which makes the solution take a very long time to find the set of successive moves to generate the solutions, but it performs much better than DFS. As well as that the fringe space is much bigger than DFS, making memory usage a much bigger challenge.

.) UCS (Uniformed Cost Search):

Initialize a priority queue “frontier” with the start node containing the start state, and an empty list of actions with a cost initialized to 0. Loop until the queue is empty, and by each iteration, the node with the lowest cost is popped. If the current node has not been visited and its cost is lower than what is previously recorded, then it is directly added to the list. The algorithm finishes when the goal state is reached, giving the sequence of actions taking, the number of expanded nodes, and the maximum length of the fringe. In addition to all the Uninformed Search downsides, UCS is also inefficient because it doesn’t incorporate heuristics to guide the search projects, but it is for sure the best one in between the two others, because it takes into consideration the cost factor. It is a good searching technique, but it needs a complementary idea.

2. Informed Search:

.) A* Search:

By Switching into Informed Search, we are more focused on the factors that would find the most optimal solution, the algorithm states:

Initializing a priority queue like UCS, but the priority is determined by the sum of the path cost so far and a heuristic estimate of the remaining cost, combining past results, and future expectations, by using both functions g(n) and h(n). Overall, the code is really like UCS, but the heuristic inclusion makes a huge difference in the process of the result, making A* Search much more flexible than the other techniques. This is exactly what the UCS technique needed to be complete: An informed search with an effective heuristic.

The answer is clear: A* search with the Manhattan is definitely the best way to solve the 8-puzzle problem, striking a balance between informed exploration and efficiency, completeness, optimality, and low space complexity to guide the search towards our wanted solution.

V. Conclusion:

In conclusion, this research project has examined four heuristic functions for improving the performance of the Eight Puzzle problem: Misplaced Tiles count, Sum of Euclidean Distances, Sum of Manhattan distances, and Number of Tiles Out of Row and Column. Each heuristic has its own strengths and limitations, and further research is needed to determine the most effective approach. This research contributes to the field of artificial intelligence and informs the development of informed search algorithms for combinatorial search problems.