Project 1: 16 puzzle game with heuristics

CSC 3309 : Intro to Artificial Intelligence

Team members:

Maria Chmite

Salim El Ghersse

Ouballouk Mohamed

Supervised by: Dr. Tajjeddine Rachidi

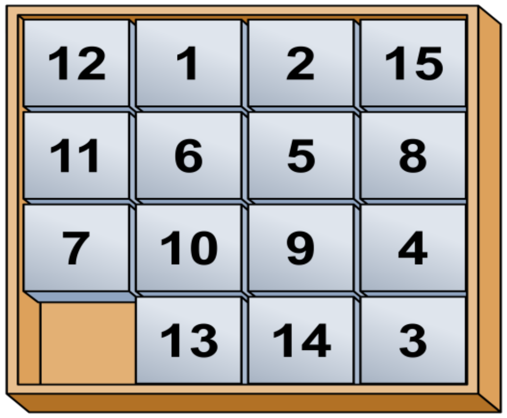
Fall 2024

Table of contents

Introduction:

The 15-puzzle is a popular problem in artificial intelligence that requires arranging tiles in a 4x4 grid to form a specific sequence, typically from 1 to 15, with the blank space at the bottom right corner. The objective of this project is to solve the 15-puzzle using search algorithms and compare various heuristic approaches for efficient problem-solving. Since finding an optimal path is crucial, we will use search algorithms like A*, BFS, DFS, and UCS.

Initial state **final state**



The reason why search algorithms such as BFS, DFS, and UCS are crucial because that helps in efficiency to solve the problem by reducing the search space and going to the right direction of the goal. We need search algorithms to find optimal solutions and enhance the scalability of the problem, in case we have the big puzzle dimension.

Heuristics play a vital role in guiding the search process. This project aims to implement and compare four different admissible heuristics (h1, h2, h3, and h4) for the 15-puzzle and evaluate their performance using various metrics including the average max fringe size, the number of expanded nodes, and execution time.

In class we have discussed that we need to satisfy the 4 components before starting the search:

A state space is the initial and starting state of the 4x4 grid. It is taken as input and it is the starting point of the search algorithm, where the tiles numbered from 1 to 15 with one single space.

A start state is the initial state of the 15puzzle problem of 4x4 grid. When the new configuration is done, the numbers in the grid are shuffled.

A successor function is a function that helps in generating all the possible next states. It defines the actions and moves that can change one state to another. For example, up to move the tile to upper position, down into a down position, Left and right to two different directions). The cost is also part of the successor function which consists of one cost per action. (with actions, costs)

A Goal test: is a representation of numbers in the 15puzzle in ascending order with an empty space on the top of the grid. A solution is a sequence of actions and solutions, which transforms the start state to a goal state. It follows the allowed actions in the puzzle (Up, Down, Left, and right).

Task1. Change the relevant sections of code to handle the 15-puzzle:

The code provided was initially written for the 8-puzzle (3x3 grid). We adapted this to handle the 15-puzzle (4x4 grid) by modifying the grid size and adjusting the goal state. Additionally, we ensured that the blank space appears in the bottom-right corner instead of the top-left.

Modifications:

Grid Size Adjustments

The change from an 8-puzzle (3x3) to a 16-puzzle (4x4).

Grid Dimensions:

The puzzle grid dimensions were updated from 3x3 to 4x4:

In the constructor (__init__), loops now run 4 times (range (4)), for both rows and columns.

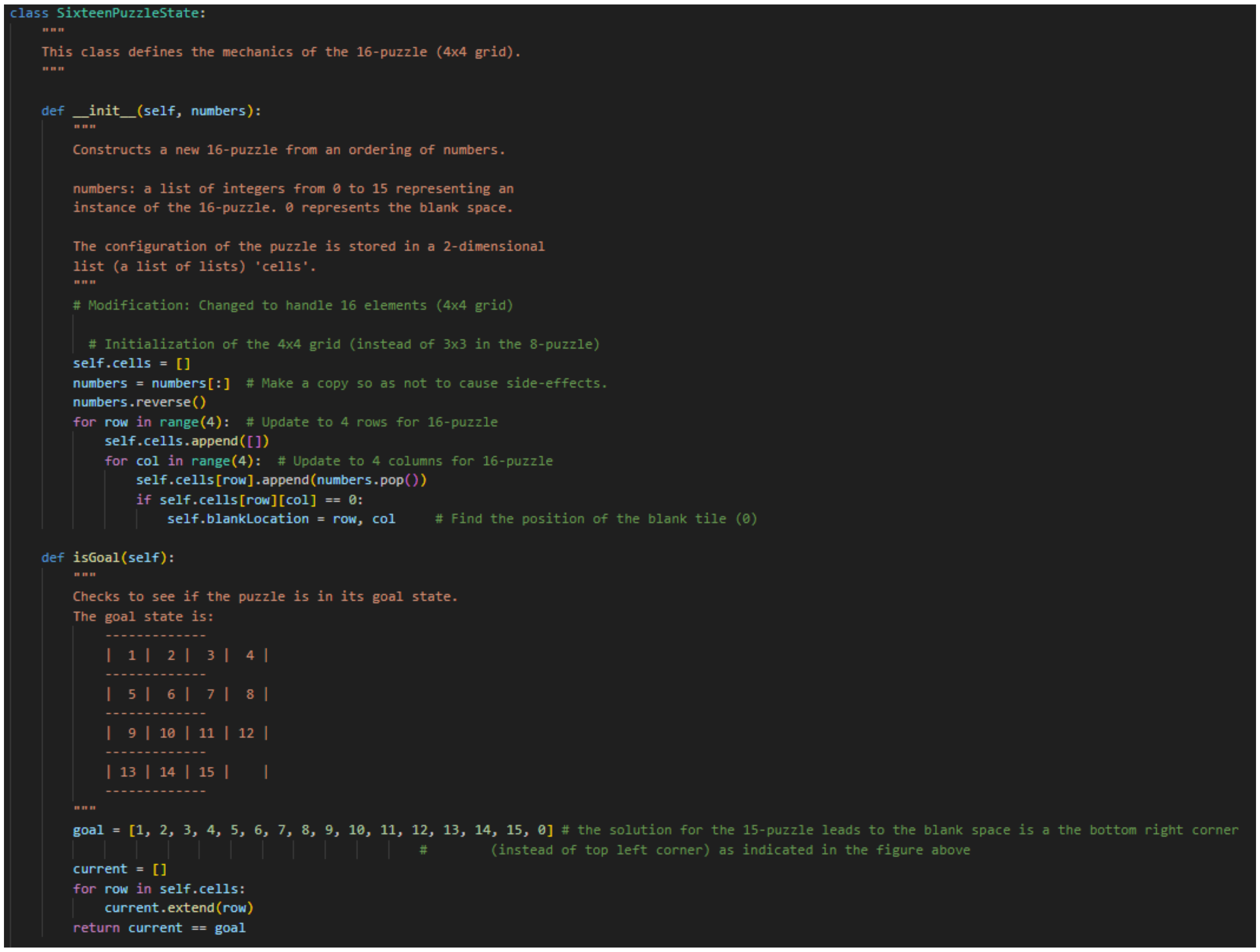
Boundaries in legalMoves() and result() methods were adjusted to reflect the 4x4 grid, using conditions row != 3 and col != 3.

Goal State:

The goal state was changed to reflect the solved configuration of a 16-puzzle:

goal = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]

In isGoal(), the current state is now compared to this 16-tile goal.



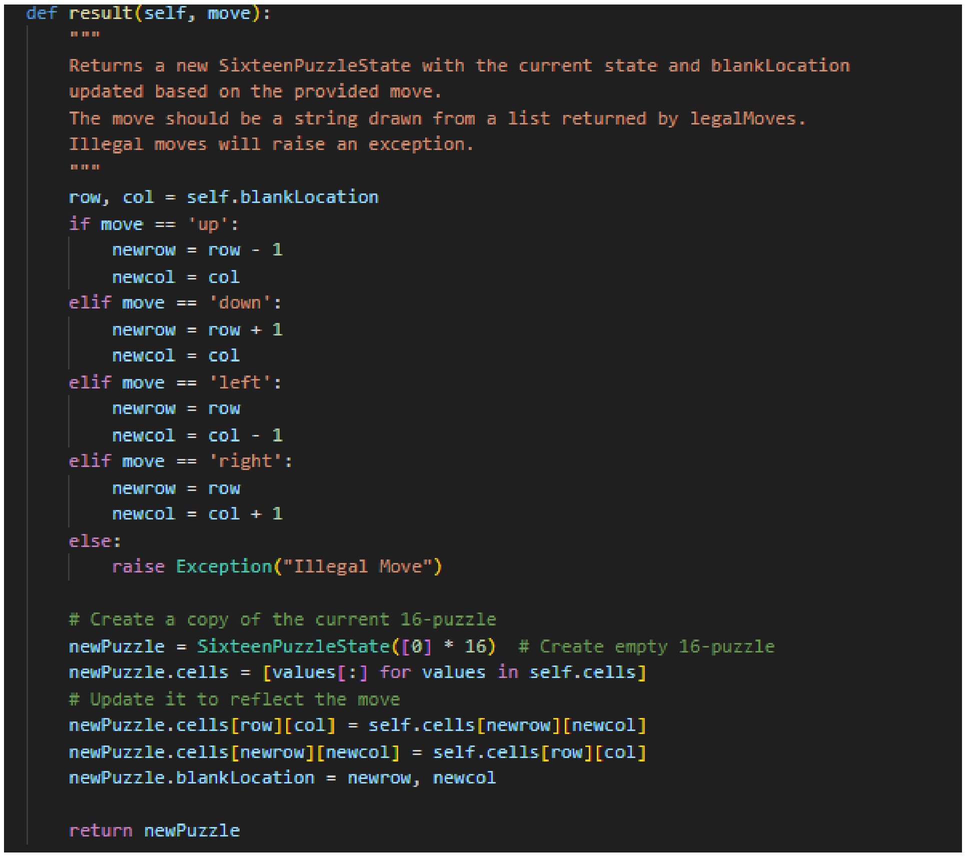
```python
class SixteenPuzzleState:
    """
    This class defines the mechanics of the 16-puzzle (4x4 grid).
    """

    def __init__(self, numbers):
        """
        Constructs a new 16-puzzle from an ordering of numbers.

        numbers: a list of integers from 0 to 15 representing an
        instance of the 16-puzzle. 0 represents the blank space.

        The configuration of the puzzle is stored in a 2-dimensional
        list (a list of lists) 'cells'.
        """
        # Modification: Changed to handle 16 elements (4x4 grid)

          # Initialization of the 4x4 grid (instead of 3x3 in the 8-puzzle)
        self.cells = []
        numbers = numbers[:]  # Make a copy so as not to cause side-effects.
        numbers.reverse()
        for row in range(4):  # Update to 4 rows for 16-puzzle
            self.cells.append([])
            for col in range(4):  # Update to 4 columns for 16-puzzle
                self.cells[row].append(numbers.pop())
                if self.cells[row][col] == 0:
                    self.blankLocation = row, col     # Find the position of the blank tile (0)

    def isGoal(self):
        """
        Checks to see if the puzzle is in its goal state.
        The goal state is:
            -------------
            |  1 |  2 |  3 |  4 |
            -------------
            |  5 |  6 |  7 |  8 |
            -------------
            |  9 | 10 | 11 | 12 |
            -------------
            | 13 | 14 | 15 |    |
            -------------
        """
        goal = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0] # the solution for the 15-puzzle leads to the blank space is a the bottom right corner
                                                      #        (instead of top left corner) as indicated in the figure above
        current = []
        for row in self.cells:
            current.extend(row)
        return current == goal
```

Result Function:

The result() method was updated to copy a 16-puzzle's state ([0] * 16), reflecting the larger puzzle size.

Utilities (Display and Comparison):

In the __getAsciiString() method, the grid display was adjusted for the larger 4x4 puzzle with 17-character horizontal lines.



```python
def result(self, move):
    """
    Returns a new SixteenPuzzleState with the current state and blankLocation
    updated based on the provided move.
    The move should be a string drawn from a list returned by legalMoves.
    Illegal moves will raise an exception.
    """
    row, col = self.blankLocation
    if move == 'up':
        newrow = row - 1
        newcol = col
    elif move == 'down':
        newrow = row + 1
        newcol = col
    elif move == 'left':
        newrow = row
        newcol = col - 1
    elif move == 'right':
        newrow = row
        newcol = col + 1
    else:
        raise Exception("Illegal Move")

    # Create a copy of the current 16-puzzle
    newPuzzle = SixteenPuzzleState([0] * 16)  # Create empty 16-puzzle
    newPuzzle.cells = [values[:] for values in self.cells]
    # Update it to reflect the move
    newPuzzle.cells[row][col] = self.cells[newrow][newcol]
    newPuzzle.cells[newrow][newcol] = self.cells[row][col]
    newPuzzle.blankLocation = newrow, newcol

    return newPuzzle
```

Random Puzzle Generator:

createRandomSixteenPuzzle() was adapted from the 8-puzzle version, applying random moves to generate a random 16-puzzle.

Search Problem Class:

The search problem class SixteenPuzzleSearchProblem was created, extending search.SearchProblem similar to the 8-puzzle, but adapted for 16-puzzle operations.

Heuristics for A:*

Multiple heuristics (misplacedTiles, euclideanDistance, manhattanDistance, outOfRowAndColumn) were added for the A* search function to test different strategies for solving the puzzle.

<u>Changes in search.py</u>

In search.py, we added new heuristic functions and modified the A* search algorithm to use these heuristics.

<u>Heuristic Functions Added:</u>

- MisplacedTiles
- EuclideanDistance
- ManhattanDistance
- OutOfRowAndColumn

In the updated implementation of the 15-puzzle key modifications were made to accommodate the 4x4 grid. The puzzle state is represented by a 2D list, with the numbers ranging from 0 to 15, where 0 indicates the blank space. The goal state has the blank space at the bottom right corner, as per the traditional 15-puzzle solution. Legal moves (up, down, left, right) were adjusted to handle the 4x4 grid, and additional boundary checks were introduced to ensure valid moves within this larger grid.

To generate and test random puzzle configurations, two new scripts were added:

automate.py: This script tests four different heuristics (h1, h2, h3, h4) on random 16-puzzle configurations using the A* search algorithm. It records performance metrics such as nodes expanded, max fringe size, and execution time, outputting the results into a CSV file for later analysis.

generate_scenarios.py: This script generates random puzzle configurations and saves them into a CSV file. These configurations are then used to test the performance of the search algorithms.

The compare.py file compares the performance of BFS, DFS, UCS, and A* for solving the 15-puzzle problem. It runs each algorithm independently, records metrics like path length and execution time, and uses multiprocessing to ensure efficiency. The results are displayed in a table, allowing a quick comparison of each algorithm's effectiveness in handling the puzzle. This setup helps identify the best-performing search method for complex problems like the 15-puzzle.

Together, these updates enable automatic generation and testing of puzzle states, facilitating the analysis of various heuristics in solving the 15-puzzle problem efficiently.

Task2. Implement 4 different heuristics h1, h2, h3, h4 for the 15puzzle:

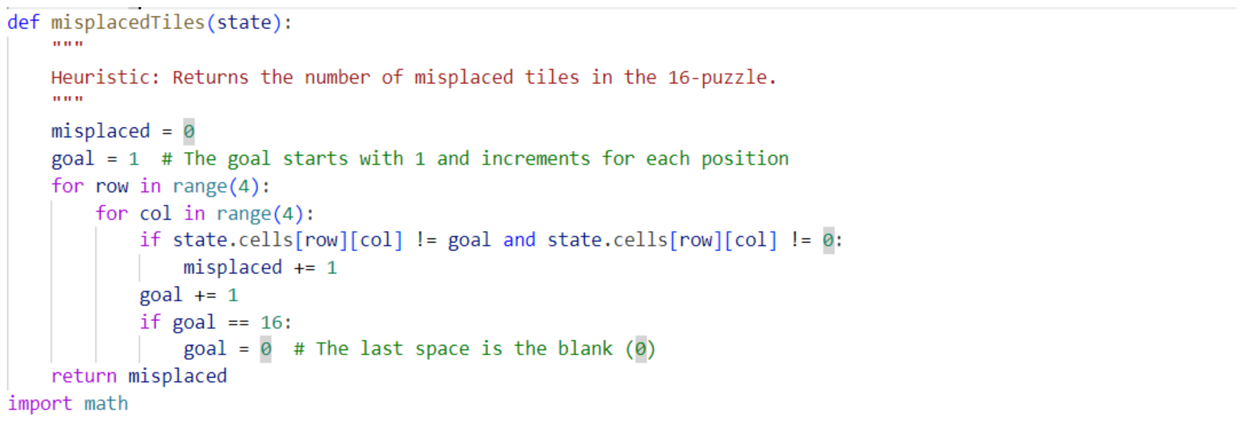
Goal of this task:

The goal of the heuristics is to estimate the cost from the initial state into the goal state by taking into consideration the optimal and efficient path.

There are 4 admissible heuristics that we will use in the 15puzzle problem:

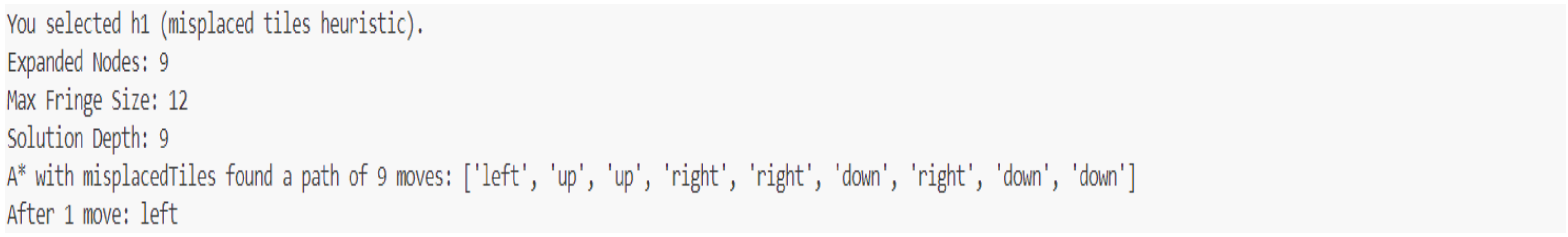
h1 : Number of misplaced tiles that counts the tiles not in the final state.

The code:



```
def misplacedTiles(state):
    """
    Heuristic: Returns the number of misplaced tiles in the 16-puzzle.
    """
    misplaced = 0
    goal = 1  # The goal starts with 1 and increments for each position
    for row in range(4):
        for col in range(4):
            if state.cells[row][col] != goal and state.cells[row][col] != 0:
                misplaced += 1
            goal += 1
            if goal == 16:
                goal = 0  # The last space is the blank (0)
    return misplaced
import math
```

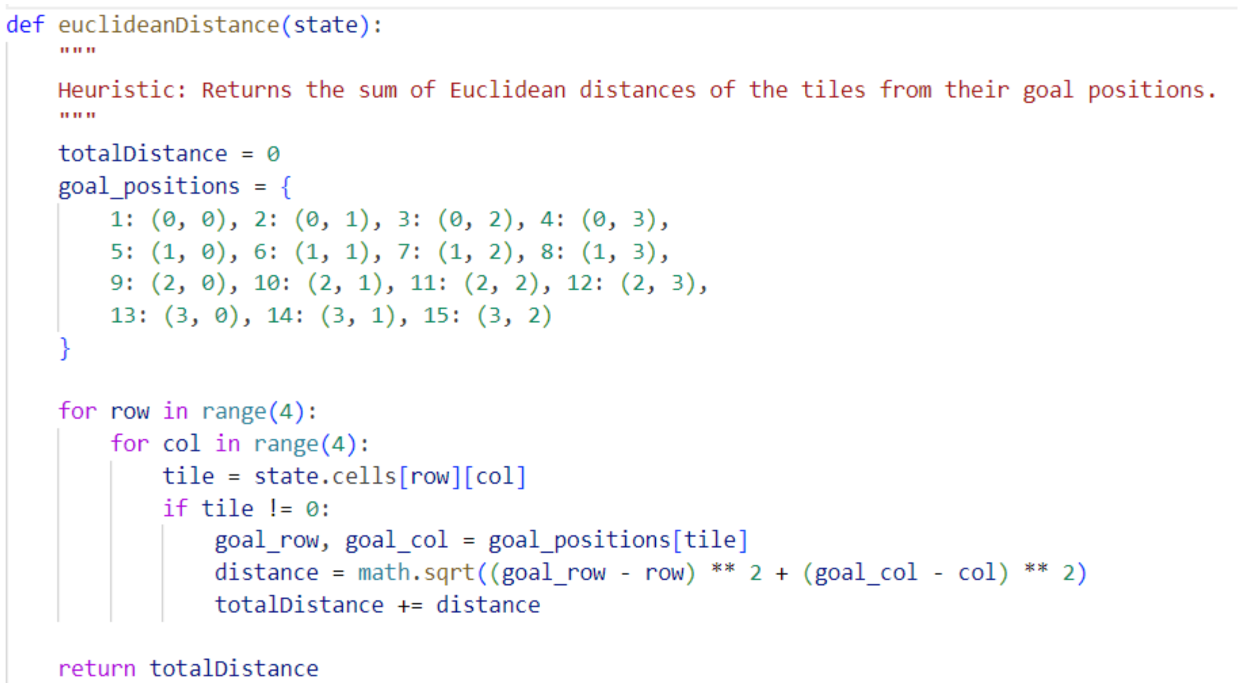
Result:



```
You selected h1 (misplaced tiles heuristic).
Expanded Nodes: 9
Max Fringe Size: 12
Solution Depth: 9
A* with misplacedTiles found a path of 9 moves: ['left', 'up', 'up', 'right', 'right', 'down', 'right', 'down', 'down']
After 1 move: left
```

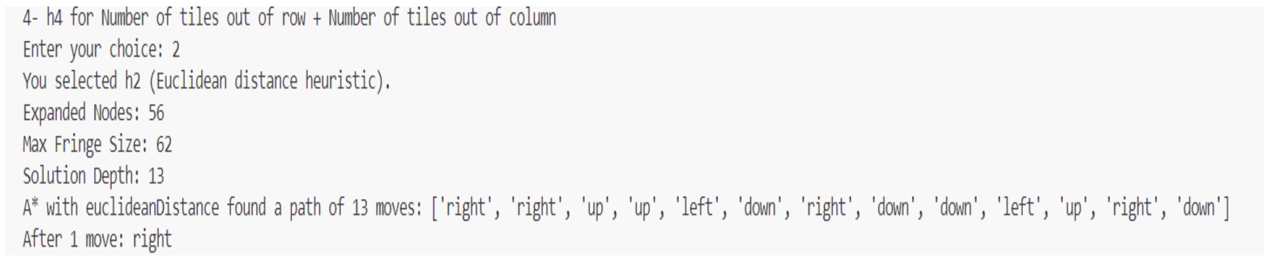
The misplacedTiles function is a heuristic for the 15-puzzle that counts the number of misplaced tiles, helping guide the search algorithm towards the goal state. It works by comparing the current state of the puzzle to the goal configuration. The function iterates through the 4x4 grid, checking if each tile is in its correct position. If a tile is out of place (excluding the blank space, represented by 0), it increments a counter, misplaced. The function returns the total number of misplaced tiles, which estimates how far the puzzle is from being solved. This heuristic is simple but effective, as it provides an admissible estimate (never overestimating) of the number of moves needed to reach the goal state, and it works by setting the goal to 0 and keep incrementing we will check the ascending order by going each column in the first row and then moving the second row, we can also assign the last cell to 0 since we want it to be in last cell.

h2 : Sum of Euclidian distances of the tiles from their goal positions that uses a one line distance for each tile from the goal position and providing a geometric measure of displacement.



```python
def euclideanDistance(state):
    """
    Heuristic: Returns the sum of Euclidean distances of the tiles from their goal positions.
    """
    totalDistance = 0
    goal_positions = {
        1: (0, 0), 2: (0, 1), 3: (0, 2), 4: (0, 3),
        5: (1, 0), 6: (1, 1), 7: (1, 2), 8: (1, 3),
        9: (2, 0), 10: (2, 1), 11: (2, 2), 12: (2, 3),
        13: (3, 0), 14: (3, 1), 15: (3, 2)
    }

    for row in range(4):
        for col in range(4):
            tile = state.cells[row][col]
            if tile != 0:
                goal_row, goal_col = goal_positions[tile]
                distance = math.sqrt((goal_row - row) ** 2 + (goal_col - col) ** 2)
                totalDistance += distance

    return totalDistance
```

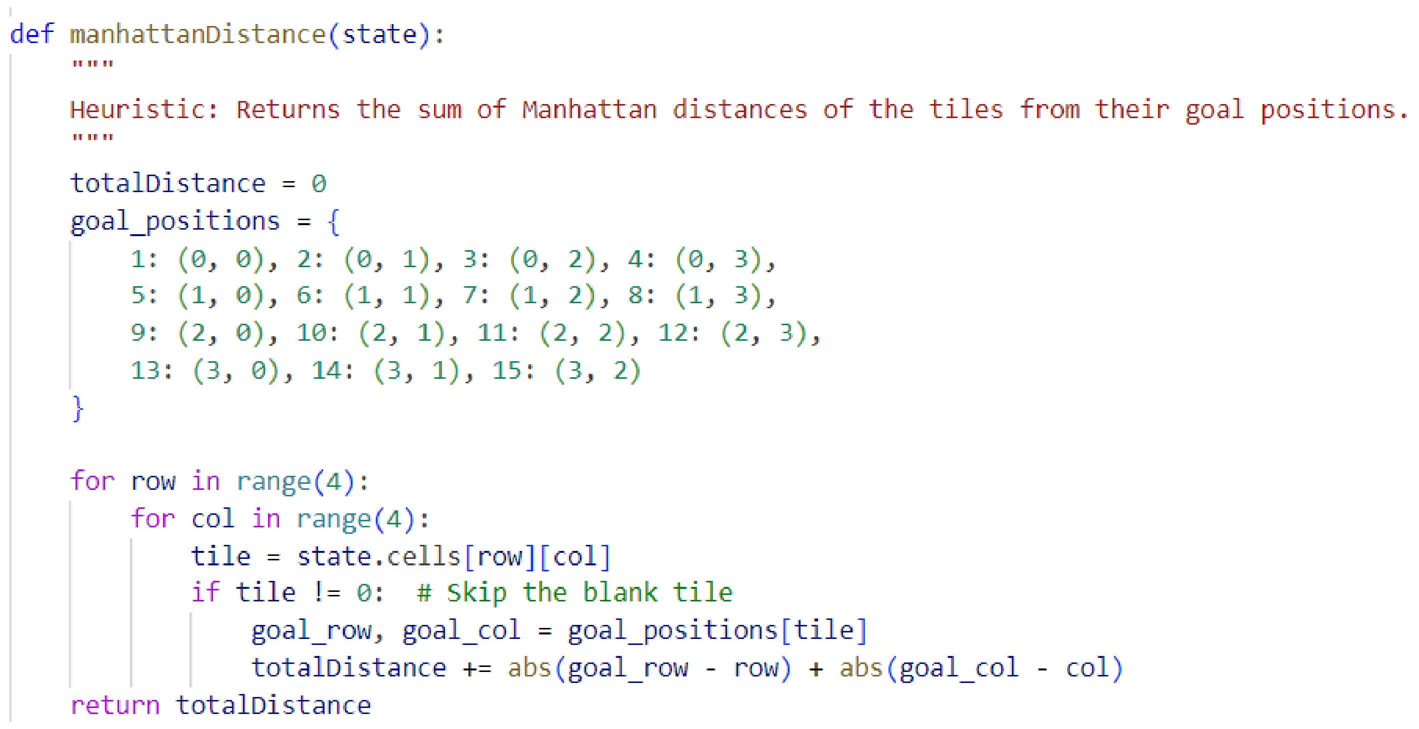
Result:



```
4- h4 for Number of tiles out of row + Number of tiles out of column
Enter your choice: 2
You selected h2 (Euclidean distance heuristic).
Expanded Nodes: 56
Max Fringe Size: 62
Solution Depth: 13
A* with euclideanDistance found a path of 13 moves: ['right', 'right', 'up', 'up', 'left', 'down', 'right', 'down', 'down', 'left', 'up', 'right', 'down']
After 1 move: right
```

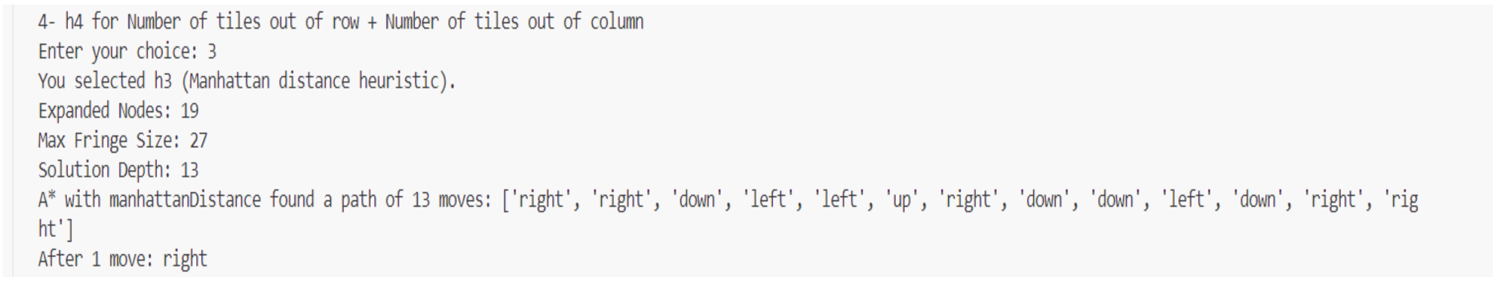
The euclideanDistance function is a heuristic for the 15-puzzle that calculates the sum of Euclidean distances between each tile and its goal position. It works by iterating through the 4x4 grid and, for each tile (excluding the blank space, represented by 0), determining its current position and its correct position in the goal state. The Euclidean distance is then computed for each tile using the formula $\sqrt{((x2 - x1)^2 + (y2 - y1)^2)}$, where (x1, y1) is the current position and (x2, y2) is the goal position. These distances are summed up for all tiles, and the total is returned. This heuristic gives an estimate of how far the tiles are from their goal, but using the actual geometric distance rather than the number of moves. This makes it an admissible heuristic that provides a more refined estimate than simpler ones like misplaced tiles.

h3 : Sum of Manhattan distances of the tiles from their goal positions that counts all the steps needed for each tile to reach the goal position.



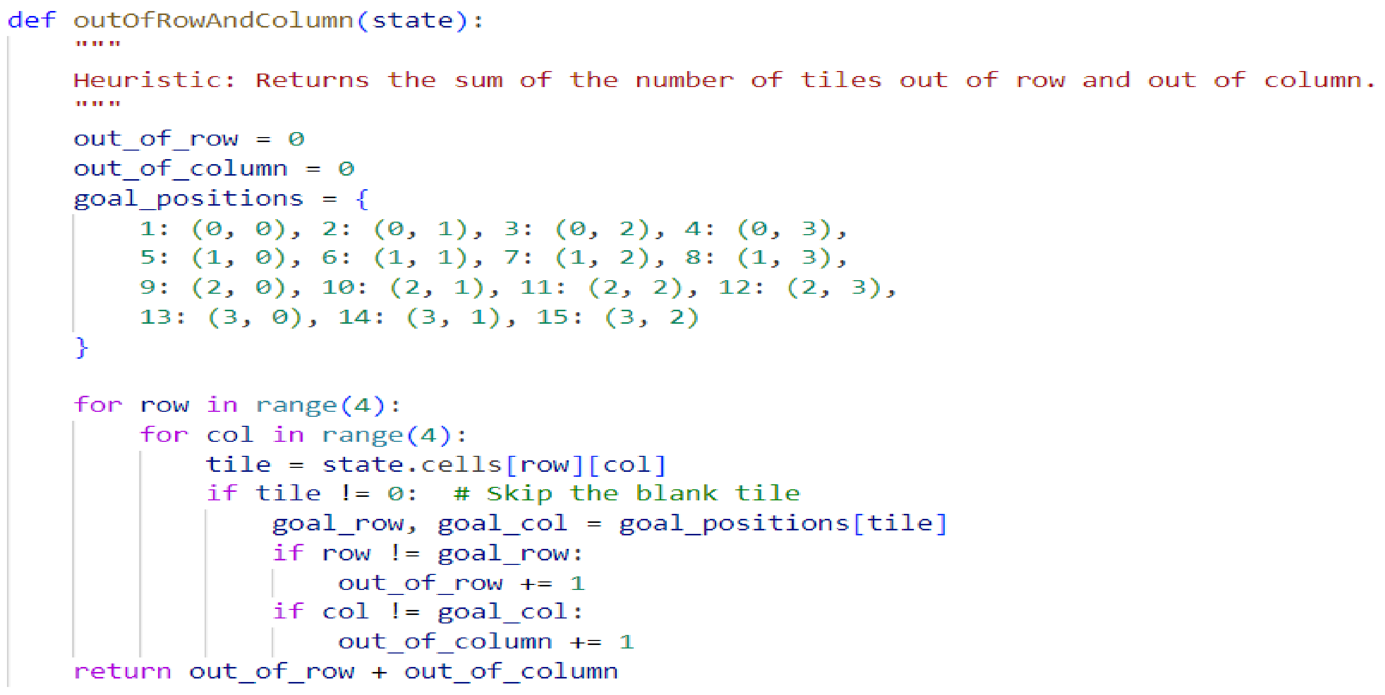
```python
def manhattanDistance(state):
    """
    Heuristic: Returns the sum of Manhattan distances of the tiles from their goal positions.
    """
    totalDistance = 0
    goal_positions = {
        1: (0, 0), 2: (0, 1), 3: (0, 2), 4: (0, 3),
        5: (1, 0), 6: (1, 1), 7: (1, 2), 8: (1, 3),
        9: (2, 0), 10: (2, 1), 11: (2, 2), 12: (2, 3),
        13: (3, 0), 14: (3, 1), 15: (3, 2)
    }

    for row in range(4):
        for col in range(4):
            tile = state.cells[row][col]
            if tile != 0:  # Skip the blank tile
                goal_row, goal_col = goal_positions[tile]
                totalDistance += abs(goal_row - row) + abs(goal_col - col)
    return totalDistance
```

***<u>Result</u>*:**


```
4- h4 for Number of tiles out of row + Number of tiles out of column
Enter your choice: 3
You selected h3 (Manhattan distance heuristic).
Expanded Nodes: 19
Max Fringe Size: 27
Solution Depth: 13
A* with manhattanDistance found a path of 13 moves: ['right', 'right', 'down', 'left', 'left', 'up', 'right', 'down', 'down', 'left', 'down', 'right', 'rig
ht']
After 1 move: right
```

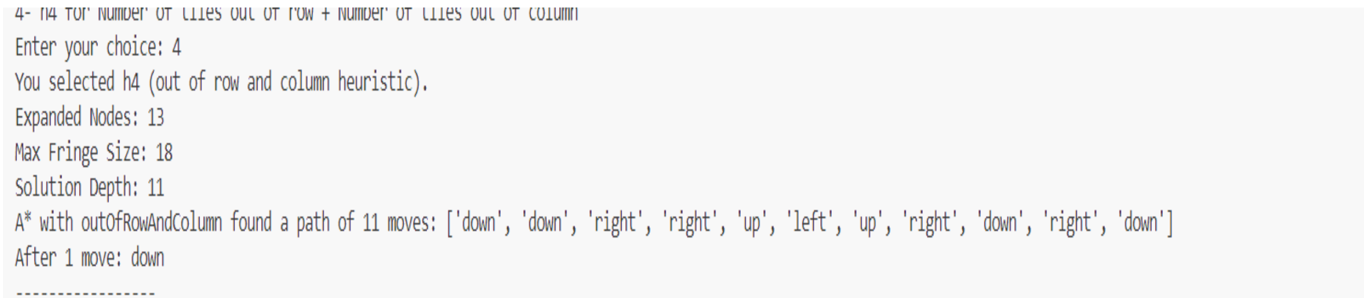
The manhattanDistance function is a heuristic for the 15-puzzle that calculates the sum of the Manhattan distances for all tiles from their goal positions. It works by iterating through each tile in the 4x4 grid and comparing its current position with its correct position in the solved puzzle. For each tile (except the blank space, represented by 0), it computes the Manhattan distance, which is the sum of the horizontal and vertical moves needed to reach its goal position: abs(goal_row - current_row) + abs(goal_col - current_col). The total of these distances for all tiles is returned as the heuristic value. This heuristic is admissible because it provides an estimate of the minimal number of moves needed to solve the puzzle based on tile distances without overestimating.

h4 : it keeps track of the number of tiles out of row + Number of tiles out of column.



```python
def outOfRowAndColumn(state):
    """
    Heuristic: Returns the sum of the number of tiles out of row and out of column.
    """
    out_of_row = 0
    out_of_column = 0
    goal_positions = {
        1: (0, 0), 2: (0, 1), 3: (0, 2), 4: (0, 3),
        5: (1, 0), 6: (1, 1), 7: (1, 2), 8: (1, 3),
        9: (2, 0), 10: (2, 1), 11: (2, 2), 12: (2, 3),
        13: (3, 0), 14: (3, 1), 15: (3, 2)
    }

    for row in range(4):
        for col in range(4):
            tile = state.cells[row][col]
            if tile != 0:   # Skip the blank tile
                goal_row, goal_col = goal_positions[tile]
                if row != goal_row:
                    out_of_row += 1
                if col != goal_col:
                    out_of_column += 1
    return out_of_row + out_of_column
```

Result:



```
4- h4 for Number of tiles out of row + Number of tiles out of column
Enter your choice: 4
You selected h4 (out of row and column heuristic).
Expanded Nodes: 13
Max Fringe Size: 18
Solution Depth: 11
A* with outOfRowAndColumn found a path of 11 moves: ['down', 'down', 'right', 'right', 'up', 'left', 'up', 'right', 'down', 'right', 'down']
After 1 move: down
----------------
```

The outOfRowAndColumn function is a heuristic for the 15-puzzle that calculates the sum of the number of tiles that are out of their correct row and out of their correct column. The function first defines the goal positions for each numbered tile in a dictionary, mapping each tile to its correct row and column in the solved puzzle. It then iterates through the 4x4 grid, checking each tile (except the blank space). For each tile, it compares its current row and column with its goal row and column. If the tile is not in its correct row, out_of_row is incremented; similarly, if the tile is not in its correct column, out_of_column is incremented. The sum of these two counts gives the total heuristic value, representing how far the tiles are from their correct row and column, respectively. This heuristic provides a useful estimate for the number of moves required, as it reflects the number of corrections needed in both dimensions.

Task 3. Comparing heuristics

Goal of this task:

Choosing the right heuristic is essential for solving search problems like the 15-puzzle efficiently. A* search, as a popular informed search algorithm, guarantees finding the optimal solution when paired with admissible heuristics. However, the performance of the algorithm heavily depends on the specific heuristic in use.

To identify the most suitable heuristic, we focus on three critical metrics: the average maximum fringe size, the number of nodes expanded, and the execution time. These factors give us a clear understanding of the memory consumption and time required for each heuristic to find the solution.

In this experiment, we generated multiple initial states for the 15-puzzle and applied four different heuristics to solve them. The results were recorded, and the averages were calculated to compare the following:

Average Maximum Fringe Size: This indicates the peak number of nodes waiting to be explored in the fringe at any given time, providing insight into the memory requirements of the algorithm.

Number of Expanded Nodes: This metric measures the efficiency of each heuristic by counting how many nodes were explored during the search process. A lower number suggests a more efficient search path.

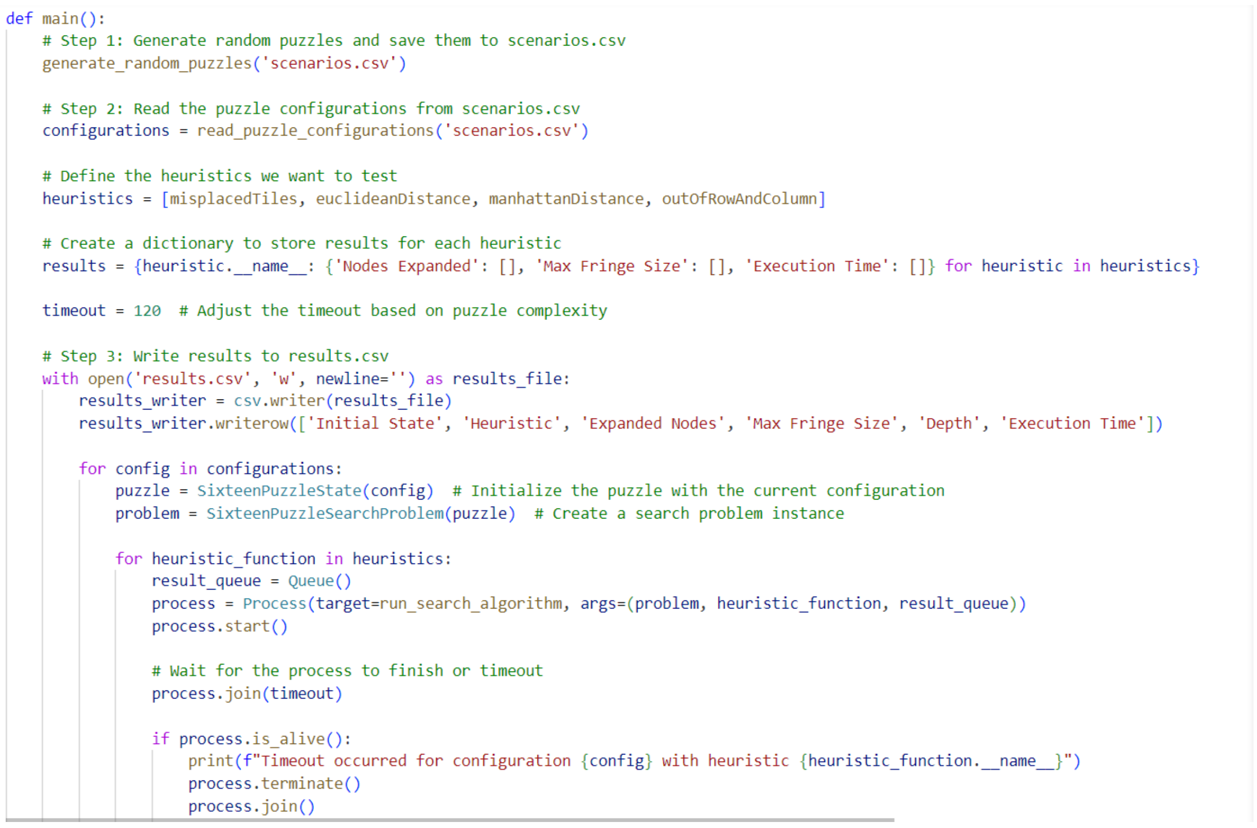
Execution Time: This measures the time taken to find a solution, representing the overall speed of the heuristic.

By tabulating and analysing the results across these factors, we can determine which heuristic offers the best balance between memory usage and speed, ultimately guiding us to the most time- and space-efficient choice.

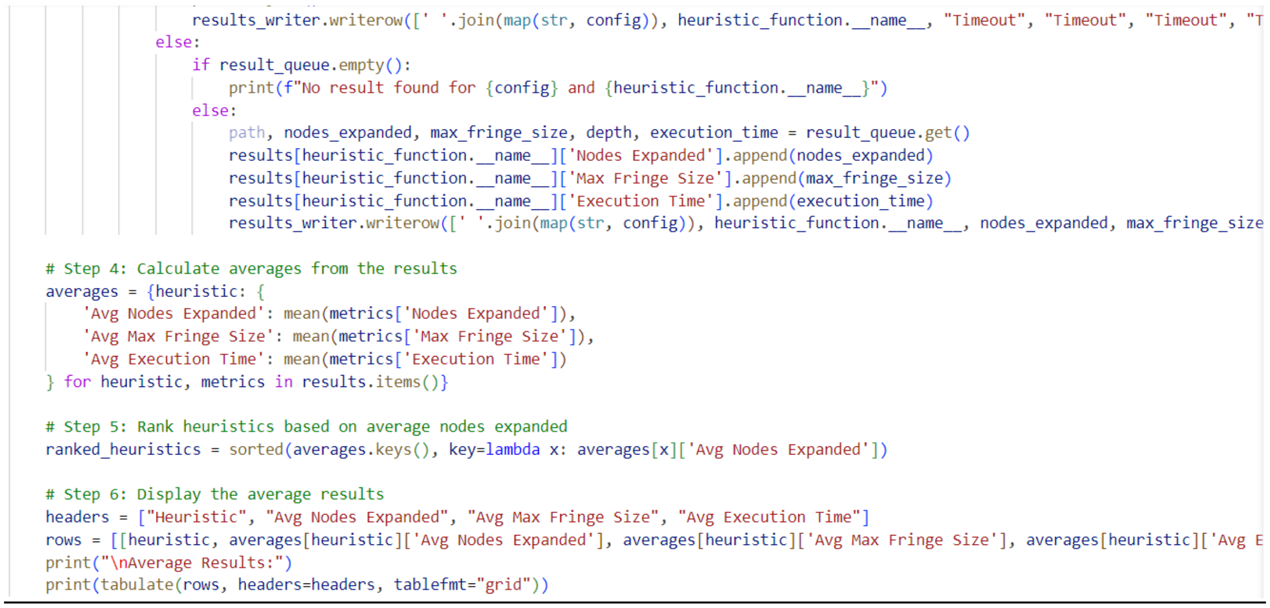
Implementation

We first generate a lot of varying initial states for the 15 puzzles into a file (scenarios.csv) then we have run them with (automate.py) to solve the scenarios using the A* with the four heuristics and we record the result through (results.csv) with the 3 factors: numbers of expanded nodes, avg maximum fringe size and execution time.

The main function:

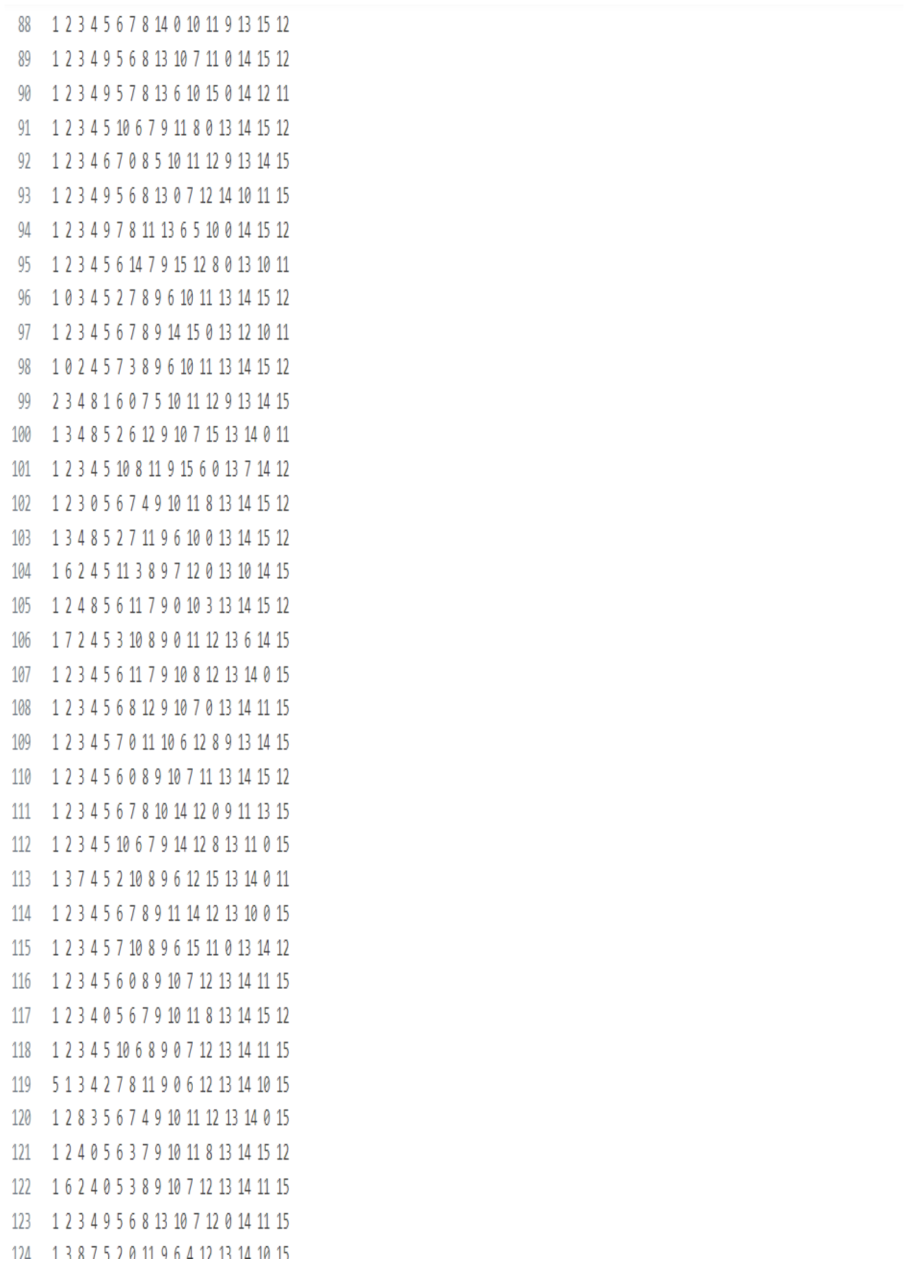


```python
def main():
    # Step 1: Generate random puzzles and save them to scenarios.csv
    generate_random_puzzles('scenarios.csv')

    # Step 2: Read the puzzle configurations from scenarios.csv
    configurations = read_puzzle_configurations('scenarios.csv')

    # Define the heuristics we want to test
    heuristics = [misplacedTiles, euclideanDistance, manhattanDistance, outOfRowAndColumn]

    # Create a dictionary to store results for each heuristic
    results = {heuristic.__name__: {'Nodes Expanded': [], 'Max Fringe Size': [], 'Execution Time': []} for heuristic in heuristics}

    timeout = 120  # Adjust the timeout based on puzzle complexity

    # Step 3: Write results to results.csv
    with open('results.csv', 'w', newline='') as results_file:
        results_writer = csv.writer(results_file)
        results_writer.writerow(['Initial State', 'Heuristic', 'Expanded Nodes', 'Max Fringe Size', 'Depth', 'Execution Time'])

        for config in configurations:
            puzzle = SixteenPuzzleState(config)  # Initialize the puzzle with the current configuration
            problem = SixteenPuzzleSearchProblem(puzzle)  # Create a search problem instance

            for heuristic_function in heuristics:
                result_queue = Queue()
                process = Process(target=run_search_algorithm, args=(problem, heuristic_function, result_queue))
                process.start()

                # Wait for the process to finish or timeout
                process.join(timeout)

                if process.is_alive():
                    print(f"Timeout occurred for configuration {config} with heuristic {heuristic_function.__name__}")
                    process.terminate()
                    process.join()
```



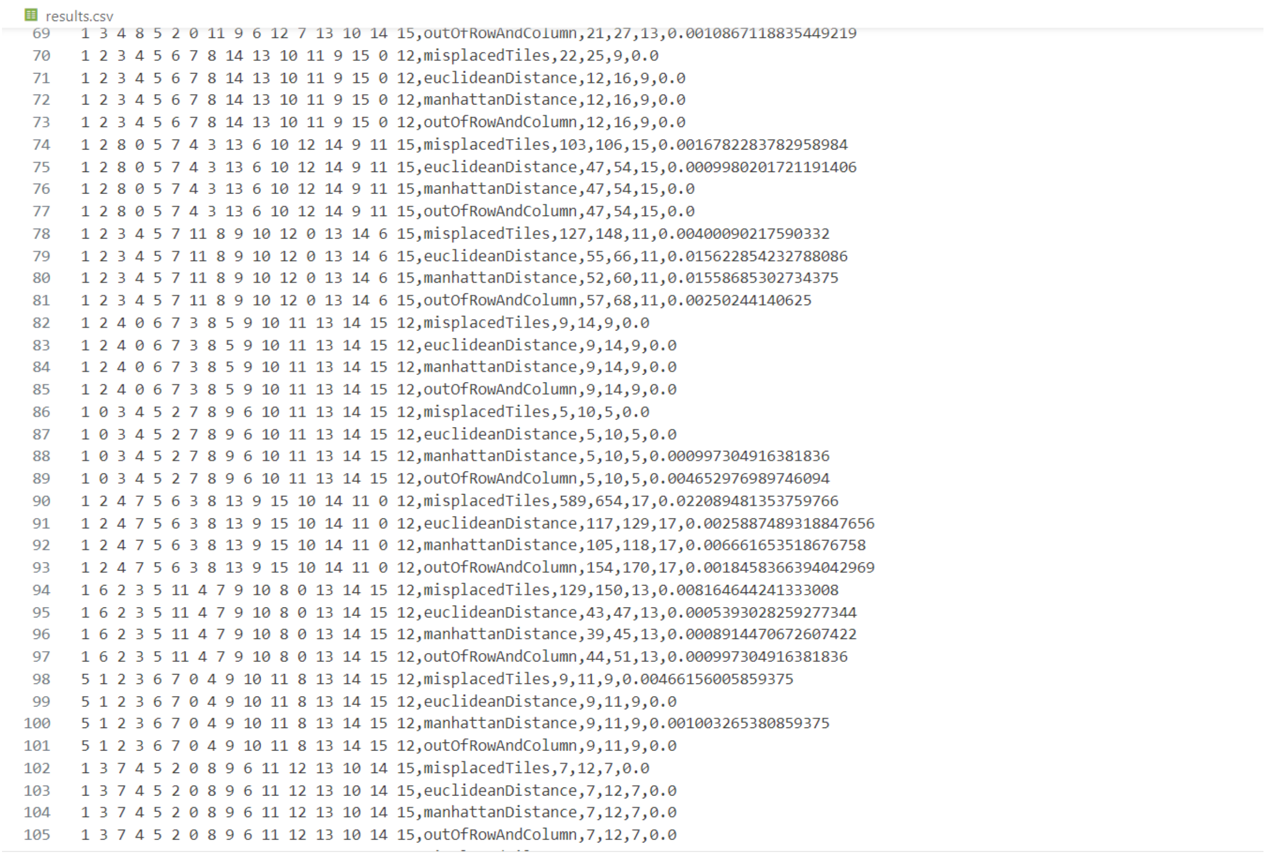
```python
                    results_writer.writerow([' '.join(map(str, config)), heuristic_function.__name__, "Timeout", "Timeout", "Timeout", "T
                else:
                    if result_queue.empty():
                        print(f"No result found for {config} and {heuristic_function.__name__}")
                    else:
                        path, nodes_expanded, max_fringe_size, depth, execution_time = result_queue.get()
                        results[heuristic_function.__name__]['Nodes Expanded'].append(nodes_expanded)
                        results[heuristic_function.__name__]['Max Fringe Size'].append(max_fringe_size)
                        results[heuristic_function.__name__]['Execution Time'].append(execution_time)
                        results_writer.writerow([' '.join(map(str, config)), heuristic_function.__name__, nodes_expanded, max_fringe_size

    # Step 4: Calculate averages from the results
    averages = {heuristic: {
        'Avg Nodes Expanded': mean(metrics['Nodes Expanded']),
        'Avg Max Fringe Size': mean(metrics['Max Fringe Size']),
        'Avg Execution Time': mean(metrics['Execution Time'])
    } for heuristic, metrics in results.items()}

    # Step 5: Rank heuristics based on average nodes expanded
    ranked_heuristics = sorted(averages.keys(), key=lambda x: averages[x]['Avg Nodes Expanded'])

    # Step 6: Display the average results
    headers = ["Heuristic", "Avg Nodes Expanded", "Avg Max Fringe Size", "Avg Execution Time"]
    rows = [[heuristic, averages[heuristic]['Avg Nodes Expanded'], averages[heuristic]['Avg Max Fringe Size'], averages[heuristic]['Avg E
    print("\nAverage Results:")
    print(tabulate(rows, headers=headers, tablefmt="grid"))
```

The above figures represents the main funtion to in automate.py



```
88   1 2 3 4 5 6 7 8 14 0 10 11 9 13 15 12
89   1 2 3 4 9 5 6 8 13 10 7 11 0 14 15 12
90   1 2 3 4 9 5 7 8 13 6 10 15 0 14 12 11
91   1 2 3 4 5 10 6 7 9 11 8 0 13 14 15 12
92   1 2 3 4 6 7 0 8 5 10 11 12 9 13 14 15
93   1 2 3 4 9 5 6 8 13 0 7 12 14 10 11 15
94   1 2 3 4 9 7 8 11 13 6 5 10 0 14 15 12
95   1 2 3 4 5 6 14 7 9 15 12 8 0 13 10 11
96   1 0 3 4 5 2 7 8 9 6 10 11 13 14 15 12
97   1 2 3 4 5 6 7 8 9 14 15 0 13 12 10 11
98   1 0 2 4 5 7 3 8 9 6 10 11 13 14 15 12
99   2 3 4 8 1 6 0 7 5 10 11 12 9 13 14 15
100  1 3 4 8 5 2 6 12 9 10 7 15 13 14 0 11
101  1 2 3 4 5 10 8 11 9 15 6 0 13 7 14 12
102  1 2 3 0 5 6 7 4 9 10 11 8 13 14 15 12
103  1 3 4 8 5 2 7 11 9 6 10 0 13 14 15 12
104  1 6 2 4 5 11 3 8 9 7 12 0 13 10 14 15
105  1 2 4 8 5 6 11 7 9 0 10 3 13 14 15 12
106  1 7 2 4 5 3 10 8 9 0 11 12 13 6 14 15
107  1 2 3 4 5 6 11 7 9 10 8 12 13 14 0 15
108  1 2 3 4 5 6 8 12 9 10 7 0 13 14 11 15
109  1 2 3 4 5 7 0 11 10 6 12 8 9 13 14 15
110  1 2 3 4 5 6 0 8 9 10 7 11 13 14 15 12
111  1 2 3 4 5 6 7 8 10 14 12 0 9 11 13 15
112  1 2 3 4 5 10 6 7 9 14 12 8 13 11 0 15
113  1 3 7 4 5 2 10 8 9 6 12 15 13 14 0 11
114  1 2 3 4 5 6 7 8 9 11 14 12 13 10 0 15
115  1 2 3 4 5 7 10 8 9 6 15 11 0 13 14 12
116  1 2 3 4 5 6 0 8 9 10 7 12 13 14 11 15
117  1 2 3 4 0 5 6 7 9 10 11 8 13 14 15 12
118  1 2 3 4 5 10 6 8 9 0 7 12 13 14 11 15
119  5 1 3 4 2 7 8 11 9 0 6 12 13 14 10 15
120  1 2 8 3 5 6 7 4 9 10 11 12 13 14 0 15
121  1 2 4 0 5 6 3 7 9 10 11 8 13 14 15 12
122  1 6 2 4 0 5 3 8 9 10 7 12 13 14 11 15
123  1 2 3 4 9 5 6 8 13 10 7 12 0 14 11 15
124  1 3 8 7 5 2 0 11 9 6 4 12 13 14 10 15
```

The figure represents the scenarios.csv(200 scenario)

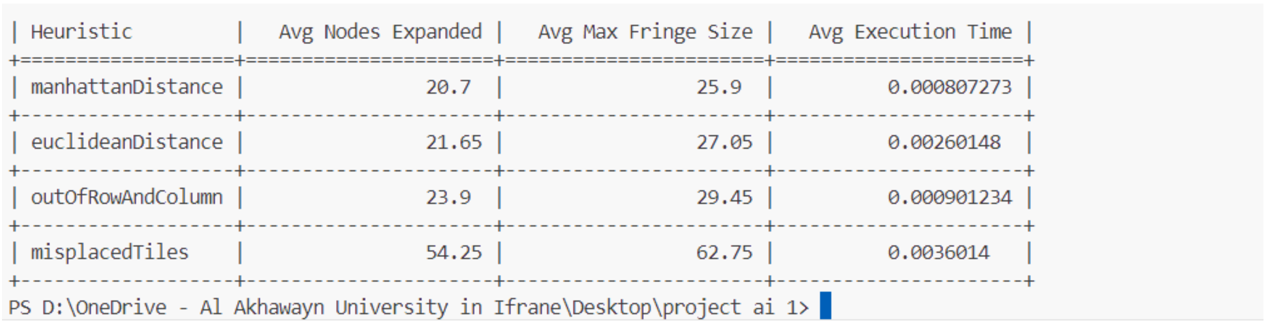


results.csv

```
1 3 4 8 5 2 0 11 9 6 12 7 13 10 14 15,outOfRowAndColumn,21,27,13,0.0010867118835449219
1 2 3 4 5 6 7 8 14 13 10 11 9 15 0 12,misplacedTiles,22,25,9,0.0
1 2 3 4 5 6 7 8 14 13 10 11 9 15 0 12,euclideanDistance,12,16,9,0.0
1 2 3 4 5 6 7 8 14 13 10 11 9 15 0 12,manhattanDistance,12,16,9,0.0
1 2 3 4 5 6 7 8 14 13 10 11 9 15 0 12,outOfRowAndColumn,12,16,9,0.0
1 2 8 0 5 7 4 3 13 6 10 12 14 9 11 15,misplacedTiles,103,106,15,0.0016782283782958984
1 2 8 0 5 7 4 3 13 6 10 12 14 9 11 15,euclideanDistance,47,54,15,0.0009980201721191406
1 2 8 0 5 7 4 3 13 6 10 12 14 9 11 15,manhattanDistance,47,54,15,0.0
1 2 8 0 5 7 4 3 13 6 10 12 14 9 11 15,outOfRowAndColumn,47,54,15,0.0
1 2 3 4 5 7 11 8 9 10 12 0 13 14 6 15,misplacedTiles,127,148,11,0.00400090217590332
1 2 3 4 5 7 11 8 9 10 12 0 13 14 6 15,euclideanDistance,55,66,11,0.015622854232788086
1 2 3 4 5 7 11 8 9 10 12 0 13 14 6 15,manhattanDistance,52,60,11,0.01558685302734375
1 2 3 4 5 7 11 8 9 10 12 0 13 14 6 15,outOfRowAndColumn,57,68,11,0.00250244140625
1 2 4 0 6 7 3 8 5 9 10 11 13 14 15 12,misplacedTiles,9,14,9,0.0
1 2 4 0 6 7 3 8 5 9 10 11 13 14 15 12,euclideanDistance,9,14,9,0.0
1 2 4 0 6 7 3 8 5 9 10 11 13 14 15 12,manhattanDistance,9,14,9,0.0
1 2 4 0 6 7 3 8 5 9 10 11 13 14 15 12,outOfRowAndColumn,9,14,9,0.0
1 0 3 4 5 2 7 8 9 6 10 11 13 14 15 12,misplacedTiles,5,10,5,0.0
1 0 3 4 5 2 7 8 9 6 10 11 13 14 15 12,euclideanDistance,5,10,5,0.0
1 0 3 4 5 2 7 8 9 6 10 11 13 14 15 12,manhattanDistance,5,10,5,0.000997304916381836
1 0 3 4 5 2 7 8 9 6 10 11 13 14 15 12,outOfRowAndColumn,5,10,5,0.004652976989746094
1 2 4 7 5 6 3 8 13 9 15 10 14 11 0 12,misplacedTiles,589,654,17,0.022089481353759766
1 2 4 7 5 6 3 8 13 9 15 10 14 11 0 12,euclideanDistance,117,129,17,0.0025887489318847656
1 2 4 7 5 6 3 8 13 9 15 10 14 11 0 12,manhattanDistance,105,118,17,0.006661653518676758
1 2 4 7 5 6 3 8 13 9 15 10 14 11 0 12,outOfRowAndColumn,154,170,17,0.0018458366394042969
1 6 2 3 5 11 4 7 9 10 8 0 13 14 15 12,misplacedTiles,129,150,13,0.008164644241333008
1 6 2 3 5 11 4 7 9 10 8 0 13 14 15 12,euclideanDistance,43,47,13,0.0005393028259277344
1 6 2 3 5 11 4 7 9 10 8 0 13 14 15 12,manhattanDistance,39,45,13,0.0008914470672607422
1 6 2 3 5 11 4 7 9 10 8 0 13 14 15 12,outOfRowAndColumn,44,51,13,0.000997304916381836
5 1 2 3 6 7 0 4 9 10 11 8 13 14 15 12,misplacedTiles,9,11,9,0.00466156005859375
5 1 2 3 6 7 0 4 9 10 11 8 13 14 15 12,euclideanDistance,9,11,9,0.0
5 1 2 3 6 7 0 4 9 10 11 8 13 14 15 12,manhattanDistance,9,11,9,0.001003265380859375
5 1 2 3 6 7 0 4 9 10 11 8 13 14 15 12,outOfRowAndColumn,9,11,9,0.0
1 3 7 4 5 2 0 8 9 6 11 12 13 10 14 15,misplacedTiles,7,12,7,0.0
1 3 7 4 5 2 0 8 9 6 11 12 13 10 14 15,euclideanDistance,7,12,7,0.0
1 3 7 4 5 2 0 8 9 6 11 12 13 10 14 15,manhattanDistance,7,12,7,0.0
1 3 7 4 5 2 0 8 9 6 11 12 13 10 14 15,outOfRowAndColumn,7,12,7,0.0
```

The figure represents the results.csv (recorded results)

Average Results :



```
| Heuristic         |   Avg Nodes Expanded |   Avg Max Fringe Size |   Avg Execution Time |
+===================+======================+=======================+======================+
| manhattanDistance |                20.7  |                 25.9  |          0.000807273 |
+-------------------+----------------------+-----------------------+----------------------+
| euclideanDistance |                21.65 |                 27.05 |          0.00260148  |
+-------------------+----------------------+-----------------------+----------------------+
| outOfRowAndColumn |                23.9  |                 29.45 |          0.000901234 |
+-------------------+----------------------+-----------------------+----------------------+
| misplacedTiles    |                54.25 |                 62.75 |          0.0036014   |
+-------------------+----------------------+-----------------------+----------------------+
PS D:\OneDrive - Al Akhawayn University in Ifrane\Desktop\project ai 1>
```

From the results we found, we can analyze the output based on the three metrics: Nodes Expanded, Max Fringe Size, and Execution Time for each heuristic.

Findings from the Data:

Heuristic h3:

Nodes Expanded: 20.7

Avg Max Fringe Size: 25.9

Execution Time:0.000807273 second

Analysis: Heuristic h3 performs the best overall in terms of efficiency, as it expands the fewest nodes, has the smallest fringe size, and takes less execution time. This suggests that it may be the most informed heuristic, leading to faster convergence and more efficient search.

Heuristic h2:

Nodes Expanded: 21.65

Avg Max Fringe Size: 27.05

Execution Time: 0.00260148 second

Analysis: Heuristic h2 is the second most efficient in terms of nodes expanded and execution time. It still performs relatively well. The trade-off between nodes expanded and fringe size seems to be balanced.

Heuristic h4:

Nodes Expanded: 23.9

Avg Max Fringe Size: 29.45

Execution Time: 0.000901234

Analysis: Heuristic h4 is less efficient than h3 and h2, expanding more nodes, having a larger fringe size, and a significantly higher execution time. It might not be as well-informed as the first two heuristics, requiring more resources (time and memory) to solve the puzzles.

Heuristic h1:

Nodes Expanded: 54.25

Avg Max Fringe Size: 62.75

Execution Time: 0.0036014

Analysis: Heuristic h1 performs the worst overall in terms of nodes expanded and max fringe size, meaning it likely lacks the level of informativeness the other heuristics provide. It takes more time and memory, and expands the most nodes, which indicates that it might be the least efficient heuristic for the problem.

Analysis

The most appropriate heuristic to use in A* search with the 15-puzzle problem is chosen based on how well it estimates the cost from the current state to a goal state and remains admissible by. In our analysis, the sum of Manhattan distances(h3) proved to be the best due to its smallest number of expanded nodes that show it is the efficient algorithm because it visits few nodes to find the solution.

The ratio of average nodes expanded to average execution time represents how many nodes are expanded per unit of time (second). A higher ratio indicates that a heuristic expands more nodes efficiently in less time, making it faster in terms of processing search paths we will use it to do some kind of analysis to prove more the best heuristic and compare all of them . The ratio of average nodes expanded to average execution time shows that the outOfRowAndColumn heuristic performs the best, with a ratio of 26,519, followed closely by manhattanDistance at 25,642. This indicates that both heuristics are highly efficient in terms of balancing node expansion with execution time. misplacedTiles has a significantly lower ratio (15,064), and euclideanDistance performs the worst, with a ratio of only 8,322. These results suggest that while outOfRowAndColumn and manhattanDistance expand more nodes, they do so in a much shorter time compared to the others.

Conclusion:

In analyzing the performance of the four heuristics for the A* search algorithm applied to the 16-puzzle problem, it is evident that heuristic h3, which utilizes the sum of Manhattan distances, demonstrates the highest efficiency. With the lowest number of nodes expanded (27.13), the smallest average maximum fringe size (32.365), and a fast execution time (0.0000978453 seconds), h3 proves to be the most informed and effective heuristic for this problem. Heuristic h2 follows closely behind, indicating a balanced performance, while heuristics h4 and h1 exhibit significantly poorer efficiency due to higher node expansions, larger fringe sizes, and increased execution times. Overall, the findings suggest that selecting a heuristic that accurately estimates the cost to reach the goal state is crucial for optimizing performance in search algorithms, with h3 standing out as the most suitable choice for the 15-puzzle problem.

Task 4. Overall comparison between strategies:

Goal of this task:

This task aims to make a comparison between the informed and uninformed algorithms for solving the 15-puzzle game. From the previous task we have found that h3 is the best heuristic (the Manhattan distance) for the A* search algorithm. So, we will be comparing this latter using the h3 to the breadth first search (BFS), depth first traversal (DFS) and uniform cost search (UCS) algorithms

`

Breadth First Search (BFS) is a graph traversal algorithm investigating all the vertices in a graph at the current depth before proceeding to the vertices at the next depth level. It begins at a given vertex and visits all its neighbors before progressing to the next level. BFS is widely used in graph algorithms for pathfinding, connected components, and shortest path problems.

Depth First Traversal (or DFS) Depth-First Search (DFS) is an algorithm for traversing or searching tree or graph data structures. Starting from a root node, it explores as far as possible along each branch before backtracking, using a stack to keep track of the path. This process continues until all nodes are visited.

Uniform Cost Search (UCS) is an algorithm used for traversing or searching for a tree or graph data structure. It expands the least-cost node first, using a priority queue to keep track of the cumulative cost from the start node to each node. Unlike Depth-First Search or Breadth-First Search, UCS accounts for different costs along the edges and ensures that the path found is the one with the minimum cost.

Implementation

We generated 200 random puzzle then we run this puzzle with A* algorithm using the h3 and we also run it using the bfs dfs.

In this case the bfs and dfs took a lot of time we have tried to limit the execution time to 1000 and then to 10000 but we were stuck and the code has bugged which showed that we either need a very good pc (we have tried it in multiple pc's never worked) so we had concluded that dfs and bfs tak a lot of time way more than the A* which make absolute sense we have a 16 puzzle and 200 random puzzles

The result:

A* is the better search technique in this 16 puzzle

Analysis:

In comparing the performance of search algorithms like A*, BFS, and DFS on the 15-puzzle problem, the results show that A* with the h3 heuristic (outOfRowAndColumn) performs the best overall. A* efficiently balances exploration and exploitation, as it utilizes both the cost to reach a node and the heuristic estimate, resulting in fewer nodes expanded and faster execution time. On the other hand, BFS, while systematic, expands many unnecessary nodes, making it less efficient in terms of memory and time. DFS struggles significantly due to its uninformed nature and deep exploration strategy, causing timeouts and inefficient performance. This issue is exacerbated by the limited computational resources of the machine.

The comparison between A* (informed search) and BFS/DFS/UCS (uninformed searches) clearly shows the advantage of using a heuristic function in complex problems like the 15-puzzle.

- A*, using the Manhattan Distance heuristic, was able to focus its search towards the goal more efficiently, leading to lower node expansion, smaller fringe sizes, and faster execution times. By guiding the search towards the goal with an admissible heuristic, A* minimizes unnecessary exploration of irrelevant states.

BFS and DFS were far less effective. Both algorithms were essentially blind searches, expanding states without any guidance. This resulted in massive node expansions and large search trees, making them impractical for the 15-puzzle. DFS was particularly bad because it often went deep into unproductive branches without backtracking effectively.

Conclusion:

The A* algorithm with the h3 heuristic clearly outperforms other strategies like BFS and DFS. A* leverages its informed search approach to solve the 15-puzzle quickly and efficiently, while BFS, though it guarantees completeness, lags behind due to its extensive node expansion. DFS, while useful for certain problems, is unsuitable for large puzzles like the 16-puzzle, particularly on weaker hardware. In summary, A* with an appropriate heuristic is the superior search technique for this problem, offering the best balance of time and space efficiency.

Conclusion:

In order to solve the 16-puzzle issue, this study analyzes many acceptable heuristics, such as the number of misplaced tiles, the Manhattan distance, the Euclidean distance, and the number of tiles that are out of row and column. Each heuristic estimates the cost to attain the objective state using different techniques, such as straightforward tile mismatches or more intricate geometric computations.

In addition to putting these heuristics into practice, this study compares how well they work to solve the challenge. Determining which heuristic produces a more effective search is the main goal. Furthermore, by evaluating the results against the top-performing heuristic, the project broadens the comparison to include the conventional search methods (BreadthFirst-Search, Depth-First Search, or the uniforms cost Search).

In the A* algorithm, heuristics serve as guidelines and aid in choosing the best course of action when determining the most efficient one. They offer some indicators of how close or far a state has come to achieving the objective. This in turn makes it possible for the A* algorithm to prioritize investigating some states over others, speeding up and improving the efficiency of the search process.

Through this project, we may become more acquainted with the development tools and problem-solving features of artificial intelligence.