

**SSE:** Fall 2024

**CSC 4301:** Intro to AI

**Project 1:** Report

**Team:** Amine Lahnin, Omar Ismail,

Mohammed Harouak

Table of Contents:

1. **Introduction:**

This project aims to study and comprehend the use of heuristic-informed search algorithms to solve search problems. The project revolves around the eight-puzzle problem in transforming it to a fifteen-puzzle problem, and then applying various search algorithms to determine which works best, more efficient, and why. Both puzzles share similar criteria that classify them as search problems; they both consist of:

-State Space: Confined in a 3x3 grid for the eight puzzles, and a 4x4 grid for the fifteen puzzles, where the blank tile can move in.

-Successor Function: Which allows the blank tile to move from one state to another within the bounds of the state space.

-Start State: The randomly generated beginning state for each puzzle.

-Goal Test: a test function to assess whether the puzzle is solved. This occurs for the eight puzzle only when the cells are ordered in an ascending order:

|  |  |  |
| --- | --- | --- |
|  | **1** | **2** |
| **3** | **4** | **5** |
| **6** | **7** | **8** |

As for the fifteen puzzle, the solution must have the blank space at the bottom right of the table, with the cells also being ordered in an ascending order:

|  |  |  |  |
| --- | --- | --- | --- |
| **1** | **2** | **3** | **4** |
| **5** | **6** | **7** | **8** |
| **9** | **10** | **11** | **12** |
| **13** | **14** | **15** |  |

The project will therefore cover the following tasks:

-Task 1: Transforming an eight puzzle into a fifteen puzzle.

-Task 2: Implementing four different heuristics for the A\* algorithm in the fifteen puzzle.

-Task 3: Comparing the heuristics to determine the most performant.

-Task 4: Comparing the winning heuristic with other uninformed search algorithms (BFS, DFS, and Uniform Cost Search).

1. **Task 1:** *Transforming 8-puzzle to 15-puzzle.*

The 8-puzzle operates in a 3x3 grid; 1 blank cell, and 8 other cells each containing a number from 1 to 8. The 15-puzzle on the other hand, should instead operate in a 4x4 grid; 1 blank cell and 15 other cells, hence the name 15-puzzle. To support a 4x4 table, the following code had to modified as so:

-The row & col count changed both to 4 in the FifteenPuzzleState constructor where the table cells are initialized, and the isGoal(self) function where it handles goal checking of the current state:

A computer screen shot of a program

Description automatically generated

Figure 1: FifteenPuzzleState constructor in **fifteenpuzzle.py**

A screenshot of a computer program

Description automatically generated

Figure 2: isGoal() function in **fifteenpuzzle.py**

-The legalMoves(self) function has been changed to not allow movement beyond the newly extended table borders (row < 3 for down, col < 3 for right):

A screen shot of a computer program

Description automatically generated

Figure 3: legalMoves() function in **fifteenpuzzle,py**

-The ASCII drawing function getAsciiString(self) has been modified to handle drawing a larger 4x4 grid:

A screen shot of a computer program

Description automatically generated

Figure 4: getAsciiString function in **fifteenpuzzle.py**

-The rest of the changes in the fifteenpuzzle.py file represent name refactoring of classes and functions to better suit the FifteenPuzzle context, as well as commenting out the unused loadFifteenPuzzle:

A screenshot of a computer program

Description automatically generated

Figure 5: FifteenPuzzleState constructor in **fifeenpuzzle.py**

-Execution Trace:  
Running the fifteenpuzzle.py file:

A screenshot of a computer program

Description automatically generated

Figure 6: Running the fifteenpuzzle.py

-Reaching the goal state:

A screenshot of a computer

Description automatically generated

Figure 7: Puzzle solution execution trace

1. **Task 2:** *Implementing Heuristics.*

**a). Heuristic 1:** *Number of Misplaced Tiles.*

The first heuristic consists of calculating the number of misplaced tiles of the current state compared to the goal state.

This is achievable by looping through each cell of the grid and checking whether the value of that current cell matches the intended value in the goal state. The current variable increments in each iteration of the loop, and then the function performs a check whether the goal state cell value matches with the current cell value ; if not true, then the number of misplaced tiles should increment, which would represent the value of the heuristic.

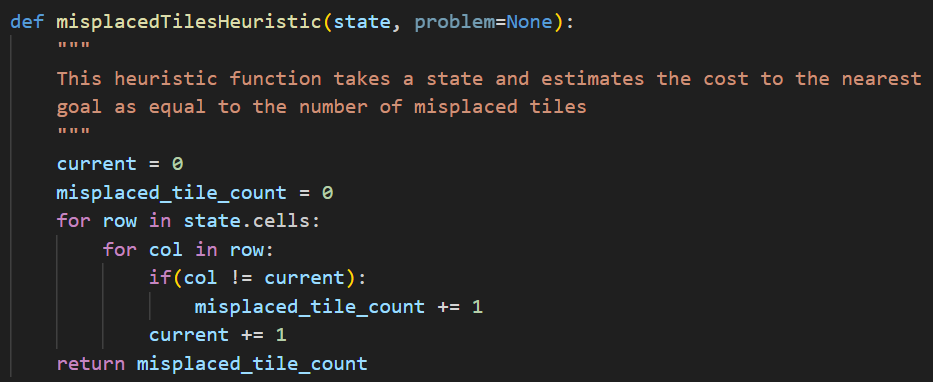


Figure 8: Heuristic 1 function implementation in **search.py**

Usage in the A\* function:

For all the heuristics, replacing the heuristic call with the intended one shall suffice:

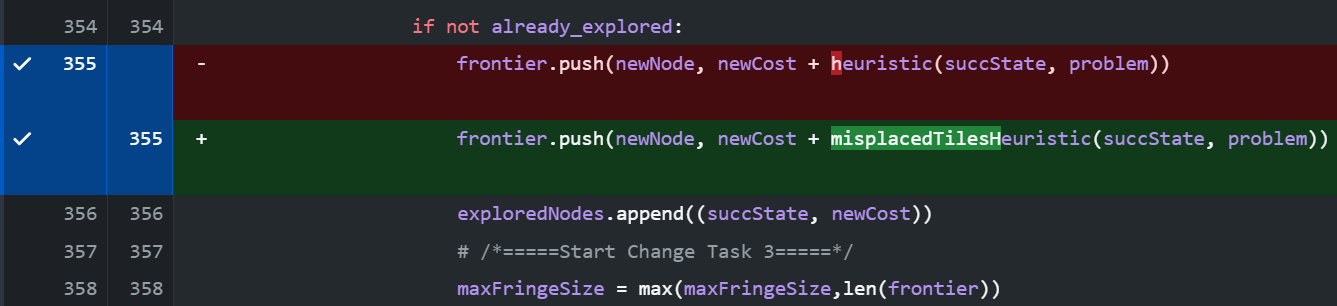


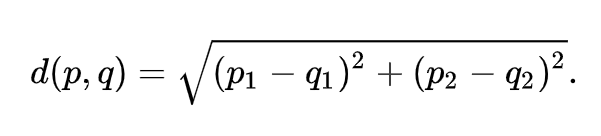
Figure 9: Heuristic integration in aStarSearch() function of **search.py**

**-Execution Trace:**

**b). Heuristic 2:** *Euclidean Distance.*

The second heuristic utilizes Euclidean distance to measure the distance between each cell in the current state, and its goal destination in the goal state. The value of the heuristic is the sum of all the latter values measured using Euclidean distance. To achieve such results, we implemented two functions:

-euclideanDistance function which takes in two coordinates of points (in this case the cells in the 2D plane which is the tab represented by the puzzle’s grid), and calculates the distance between the two points using the following formula:



With (p, q) representing a coordinate system.

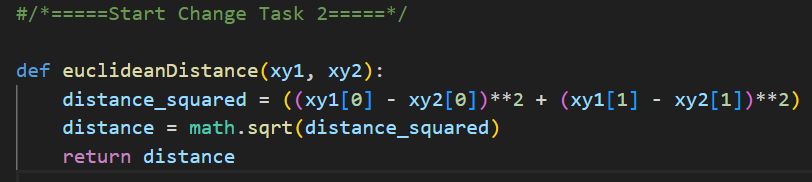


Figure 10: euclideanDistance function in **util.py**

-getFinalPosition function which transforms a particular cell into two coordinates used for the euclideanDistance function:

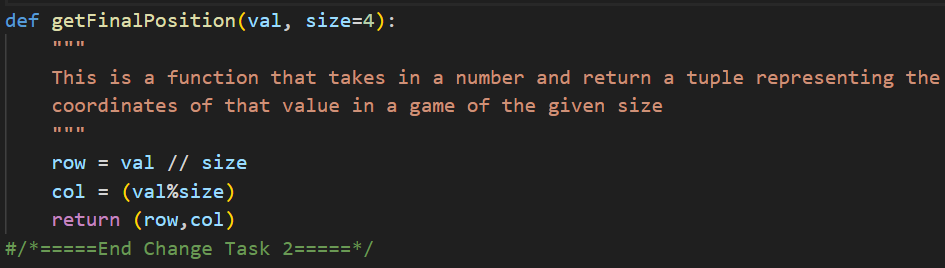
**

Figure 11: getFinalPosition function in **util.py**

-We use both functions to implement the euclideanHeuristic. Looping through each cell of the current state, and calculating its euclidean distance with the cell of the goal state, then summing the values iteratively for all cells in that table.s

A screen shot of a computer code

Description automatically generated

Figure 12: euclideanHeuristic function in **search.py**

**-Execution Trace:**

**c). Heuristic 3:** *Manhattan Distance.*

The third heuristic shares a similar aproach to that of the second heuristic. The only difference being the use of manhattan distance rather than the euclidean one. Which calculates the number of increments (i.e. movements) of a given cell in the current state necessary to reach the final position of the cell in the goal state. To do this we can leverage the coordinate system we have used to create the manhattan distance function following this formula:



With (x1, y1) and (x2, y2) representing two points in the coordinate system.

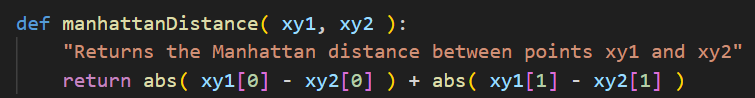


Figure 13: manhattanDistance function in **util.py**

-The implementation of the Manhattan distance heuristic thus becomes almost identical to that of the Euclidian one, with the only difference being the use of the manhattanDistance function.

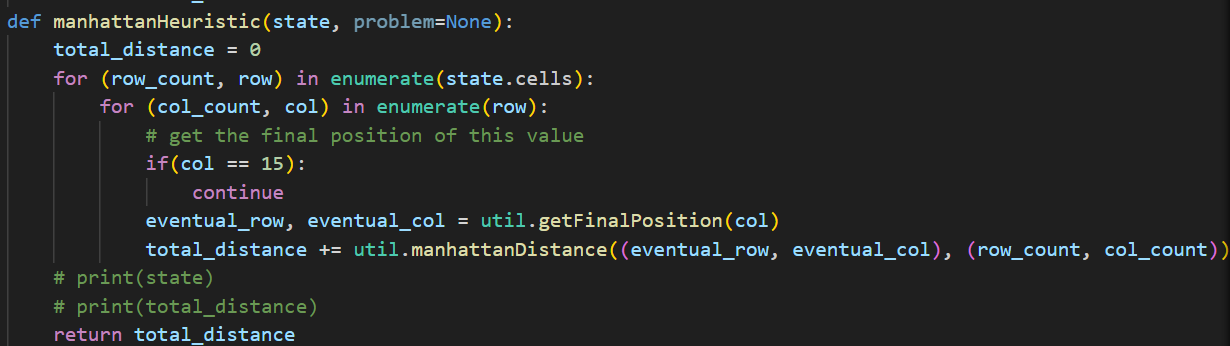


Figure 14: manhattanHeuristic function in search.py

**-Execution Trace:**

**e). Heuristic 4:** *Tiles out of row and column.*

The fourth heuristic function calculates the sum of misplaced tiles in terms of the row and column. For each difference in the row and column placement of the tiles compared to their final position, the total estimated cost is incremented by 1. We can do so by looping through each cell of the current state

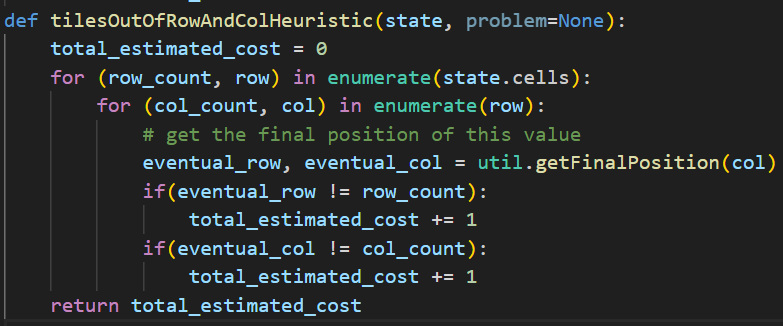


Figure 15: tilesOutOfRowAndColHeuristic function in **search.py**