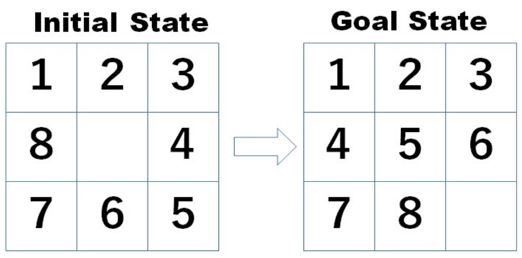
Sara Dejkoski, Muhamed Fajic, Ishak Sijamhodzic, Greta Berdynaj

https://github.com/saradejkoski/Lab01v2

8-PUZZLE



Short task description

The task is to implement the Eight Puzzle using two different heuristic functions (Hamming (misplaced tiles) and Manhattan) to solve the puzzle. In addition, we should compare these two heuristics (Computation Time). For a better understanding of the code, we should comment on the code.

Eight puzzle problem

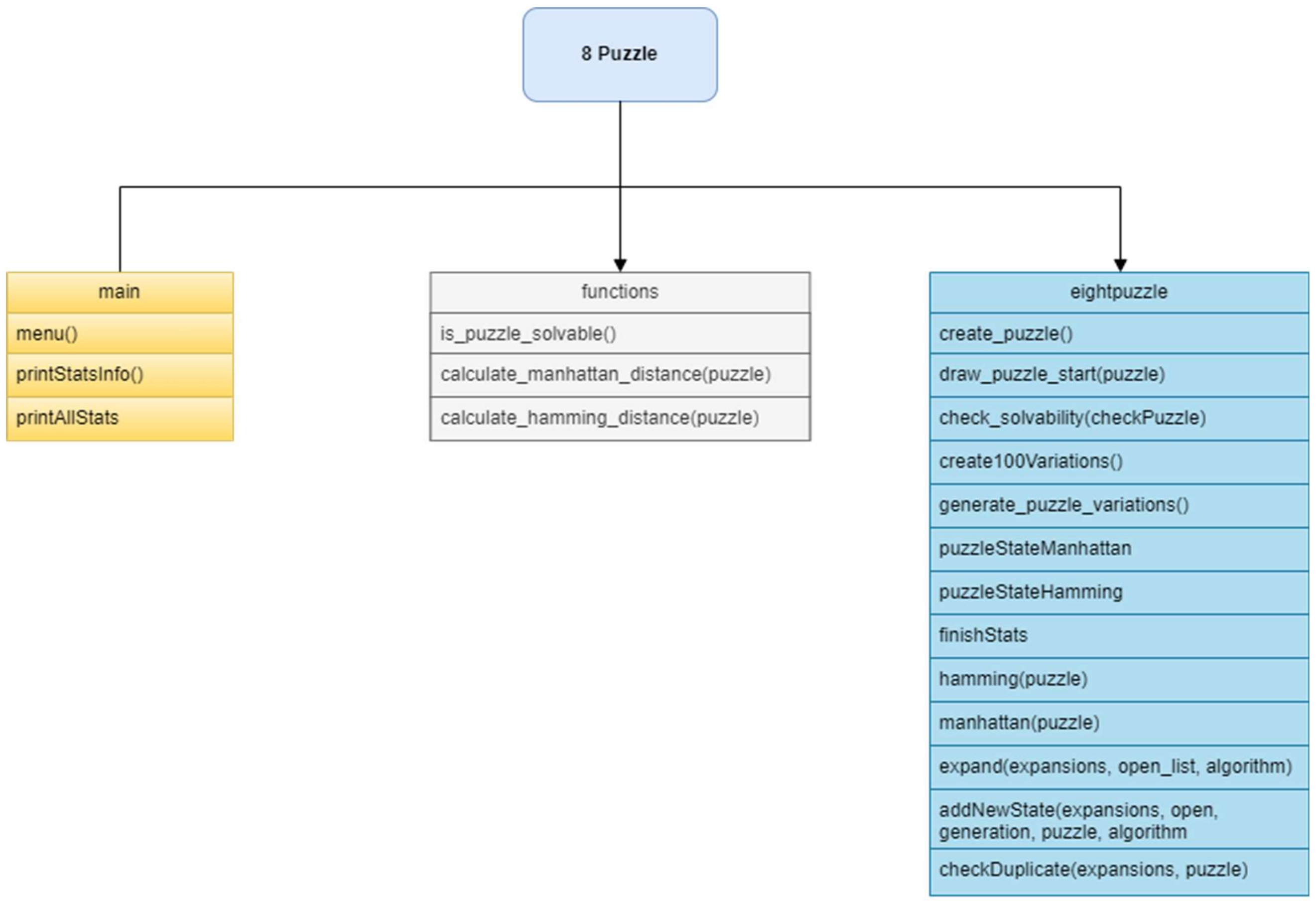
The Eight puzzle problem is a sliding puzzle that consists of a frame of numbered square tiles in random order with one tile missing. It was invented and popularized by Noyes Palmer Chapman in the 1870s. It is a smaller version of the N puzzle problem which is a classical problem for modeling algorithms involving heuristics. If the square frame size is 3×3 tiles, the puzzle is called the 8-puzzle. Starting from an initial, random state, a goal state has to be reached (Figure above).

The Goal of the Problem

The goal of the problem is to reach the destination state by sliding the empty tile vertically or horizontally in each move. This should be done in minimum numbers of moves possible.

The goal state exists as a reachable or unreachable state from the source state. No matter how many moves are made some states are unreachable from the source state. It is not solvable, if the number of inversions is odd in the input state. A pair of tiles form an inversion if the values on tiles are in reverse order of their appearance in the goal state.

Software architecture diagram


8 Puzzle
main
menu()
printStatsInfo()
printAllStats
functions
is_puzzle_solvable()
calculate_manhattan_distance(puzzle)
calculate_hamming_distance(puzzle)
eightpuzzle
create_puzzle()
draw_puzzle_start(puzzle)
check_solvability(checkPuzzle)
create100Variations()
generate_puzzle_variations()
puzzleStateManhattan
puzzleStateHamming
finishStats
hamming(puzzle)
manhattan(puzzle)
expand(expansions, open_list, algorithm)
addNewState(expansions, open, generation, puzzle, algorithm
checkDuplicate(expansions, puzzle)

Short Description of modules and interfaces

Architecture: Modules and Interfaces (Functions, Inputs/Outputs)

In eight_puzzle.py contains the functions which create the random puzzles, check if the puzzles are solvable, calculate the statistical information and find the next blank tiles.

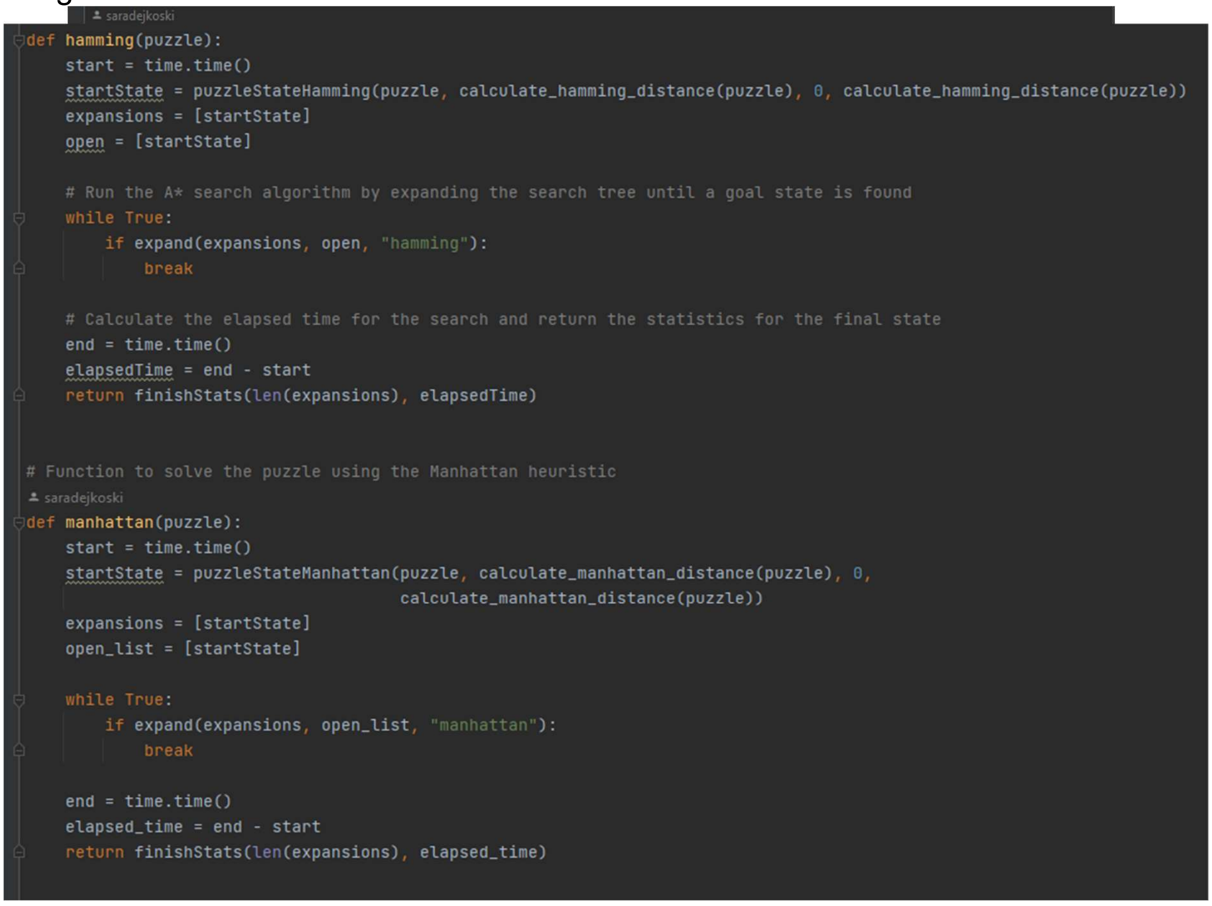
Then in functions.py there is the heuristic template function, that takes a puzzle object, and a function that checks if the puzzle is solvable, as well as the 2 other heuristics:

Manhattan and Hamming. The functions calculate_manhattan_distance(puzzle) and calculate_manhattan_distance (puzzle) calculate the heuristics' distances.

Finally in main.py we have a function for displaying a UI. In addition to the functions printStatsInfo() and printAllStats() the user can choose an option of 7 from the menu. The functions printStatsInfo() and printAllStats() prints statistical information about the algorithm's performances.

Define Heuristics you are going to use?

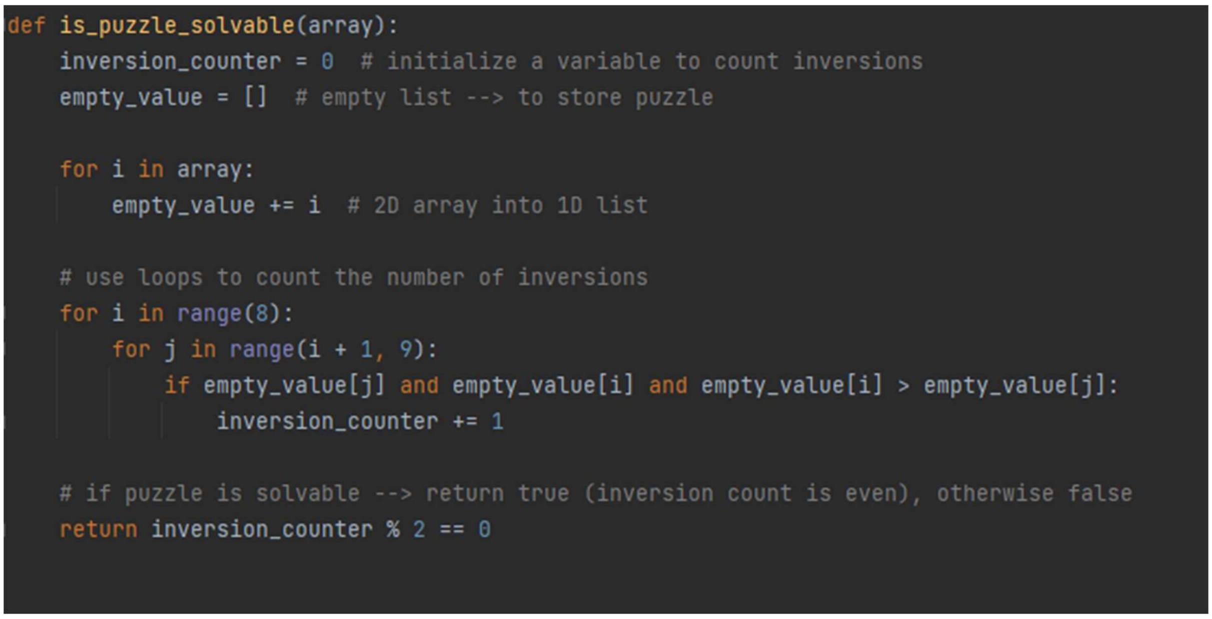
We used the Manhattan and Hamming heuristics. The methods can be seen in the image below.



```
saradejkoski
def hamming(puzzle):
    start = time.time()
    startState = puzzleStateHamming(puzzle, calculate_hamming_distance(puzzle), 0, calculate_hamming_distance(puzzle))
    expansions = [startState]
    open = [startState]

    # Run the A* search algorithm by expanding the search tree until a goal state is found
    while True:
        if expand(expansions, open, "hamming"):
            break

    # Calculate the elapsed time for the search and return the statistics for the final state
    end = time.time()
    elapsedTime = end - start
    return finishStats(len(expansions), elapsedTime)


# Function to solve the puzzle using the Manhattan heuristic
saradejkoski
def manhattan(puzzle):
    start = time.time()
    startState = puzzleStateManhattan(puzzle, calculate_manhattan_distance(puzzle), 0,
                                      calculate_manhattan_distance(puzzle))
    expansions = [startState]
    open_list = [startState]

    while True:
        if expand(expansions, open_list, "manhattan"):
            break

    end = time.time()
    elapsed_time = end - start
    return finishStats(len(expansions), elapsed_time)
```

How would you test your code?

By finding out if the puzzle is solvable or not. This is checked by calculating the number of inversions. If the number of inversions is odd, the puzzle is not solvable.
If the number of inversions is even, the puzzle is solvable.



```python
def is_puzzle_solvable(array):
    inversion_counter = 0  # initialize a variable to count inversions
    empty_value = []  # empty list --> to store puzzle

    for i in array:
        empty_value += i  # 2D array into 1D list

    # use loops to count the number of inversions
    for i in range(8):
        for j in range(i + 1, 9):
            if empty_value[j] and empty_value[i] and empty_value[i] > empty_value[j]:
                inversion_counter += 1

    # if puzzle is solvable --> return true (inversion count is even), otherwise false
    return inversion_counter % 2 == 0
```

Explain design decisions

The reason for selecting Python as the programming language for this exercise was primarily to expand our knowledge of programming with Python, which we had only briefly explored in the first semester. Furthermore, it allowed us to work directly in the realm of AI and data science. Our team opted for PyCharm as we were already familiar with it, having previously used other IDEs developed by JetBrains.

As for the project structure, we separated the program into three parts: main.py, functions.py and eightpuzzle.py. In main.py we included the functions for the output in the console. In functions.py we included the calculation of the Hamming and Manhattan distance and the function for checking if the puzzle is solvable. We thought that it would make sense to separate the calculations from the generation of the puzzles. Because of that we included the generation and run through into eightpuzzle.py.

Which data structure would you use?

The data structure that was used is the 3x3 2d array of integers, (eigth_puzzle.py -> create_puzzle), where we generate our table, draw the initial numbers as well. Also, arrays were used in functions.py -> is_puzzle_solvable, where we check if the puzzle is solvable, as explained previously by checking the number of inversions.

Which functions/ submodules would you use?

- function for checking the solvability from a start state:
    - is_puzzle_solvable(array) - if the number of inversions is odd, it's not solvable. If it is even, it is solvable. Our program prints a random startState that is always solvable.
- Heuristic() function which implements the A* Algorithm:
    - lambda function passed to it for the heuristic
    - expand(expansions, open_list, algorithm) → a function to expand the search tree by generating new states from the current state
- functions for calculating the distance of the algorithms → calculate_manhattan_distance(puzzle), calculate_hamming_distance(puzzle):

Discussion and conclusion

Describe your experience

It was not an easy task to implement an 8 Puzzle game as our first project in AI but we are satisfied with the outcome we produced. 8 Puzzle was a great start to AI as it was not hard to understand the logic but also not easy to solve the problem. We had to do a lot of research to implement the code for the different tasks and at times it would help to get to the solution by trying and then fixing the error.

Provide a table with complexity comparisons of different heuristics

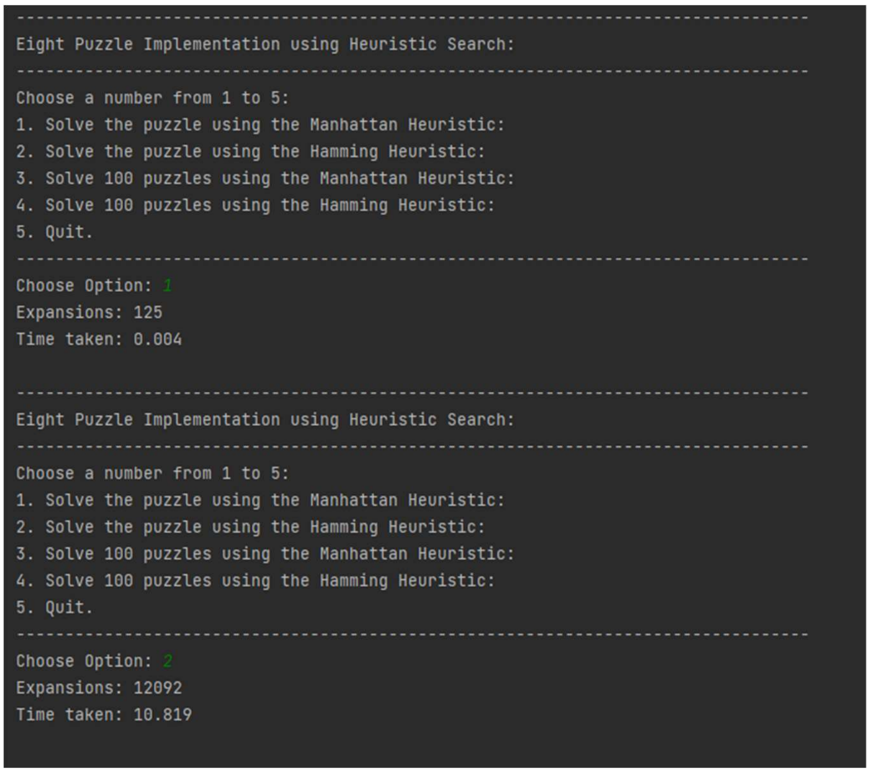
The 8-puzzle problem is an exponential search problem that requires an A* algorithm with a time and space complexity of O(bd), where b is the branching factor and d is the depth of the solution. The branching factor of an 8-puzzle varies from 2 to 4 based on the blank tile's position. When evaluating time and space complexity, finding an admissible heuristic is critical. Instead of searching for the shortest path to the goal state, it's better to find a path that expands as few nodes as possible. One way to achieve this is by relaxing the problem definition, such as using the Manhattan distance calculation to remove the constraint that tiles can't move to occupied fields or using the Hamming distance calculation that disregards the fact that tiles can only move to adjacent fields.

How would you estimate complexity? (Compare both Heuristics: Hamming and Manhattan)

By measuring the time needed for each heuristic algorithm to solve the puzzle.
The quantity of states, meaning the length of the path, can also be used to estimate the complexity and compare both heuristics - Manhattan and Hamming.

Time comparison and Expansion comparison

After running the code, we can determine how long it takes for the Algorithm to solve the puzzle. With the usage of the Manhattan heuristics, we can get the results within 0.004 seconds. With the Hamming heuristics it takes longer with the time to solve being 10.819 seconds. The times can change for different puzzle line-ups.

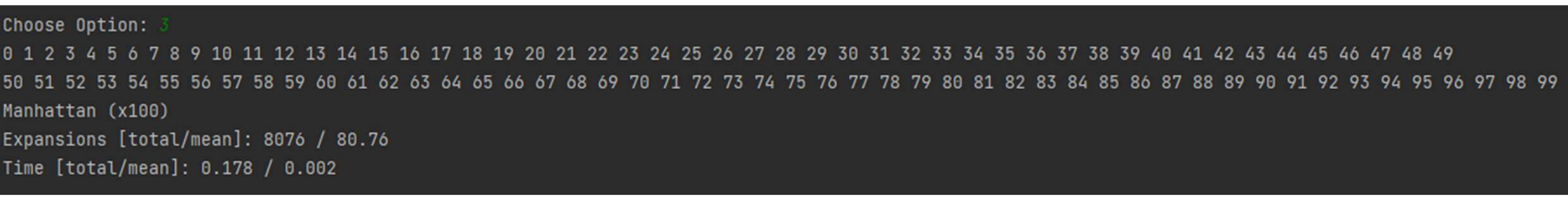


```
--------------------------------------------------------------------------------
Eight Puzzle Implementation using Heuristic Search:
--------------------------------------------------------------------------------
Choose a number from 1 to 5:
1. Solve the puzzle using the Manhattan Heuristic:
2. Solve the puzzle using the Hamming Heuristic:
3. Solve 100 puzzles using the Manhattan Heuristic:
4. Solve 100 puzzles using the Hamming Heuristic:
5. Quit.
--------------------------------------------------------------------------------
Choose Option: 1
Expansions: 125
Time taken: 0.004

--------------------------------------------------------------------------------
Eight Puzzle Implementation using Heuristic Search:
--------------------------------------------------------------------------------
Choose a number from 1 to 5:
1. Solve the puzzle using the Manhattan Heuristic:
2. Solve the puzzle using the Hamming Heuristic:
3. Solve 100 puzzles using the Manhattan Heuristic:
4. Solve 100 puzzles using the Hamming Heuristic:
5. Quit.
--------------------------------------------------------------------------------
Choose Option: 2
Expansions: 12092
Time taken: 10.819
```

Seeing the results, the Manhattan heuristic is the preferred way in terms of time needed to solve the puzzle and the steps needed.

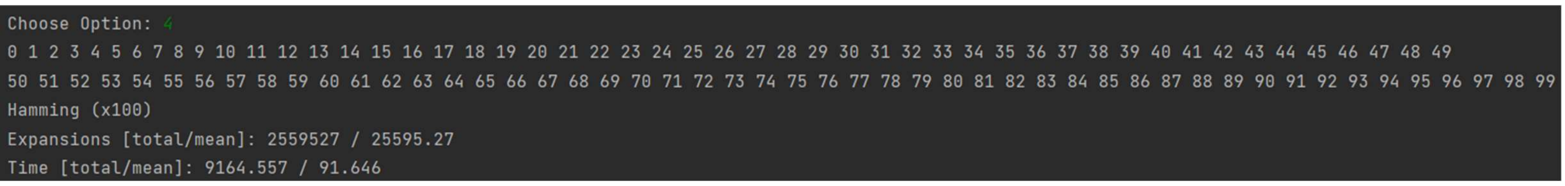
Another example:

For 100 puzzles, Manhattan needs 0.178 seconds, as shown below.



```
Choose Option: 3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
Manhattan (x100)
Expansions [total/mean]: 8076 / 80.76
Time [total/mean]: 0.178 / 0.002
```

Another example:
For 100 puzzles, Hamming needs 9164.557 seconds, as shown below.



```
Choose Option: 4
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
Hamming (x100)
Expansions [total/mean]: 2559527 / 25595.27
Time [total/mean]: 9164.557 / 91.646
```

Heuristic	**1 Puzzle**	**100 Puzzles**
Manhattan	0.004 seconds (125 expansions)	0.178 seconds (8076 expansions)
Hamming	10.819 seconds (12092 expansions)	9164.557 (2559527 expansions)

As you can see in the table above, in both cases Manhattan needs less states than the Hamming heuristic.

Possible improvements in the future

Coding a proper Graphical User Interface would probably provide a better understanding of the problem for people who have never come in touch with it. Also, one would then have the chance to play the game with pictures like the 8-puzzle games known from the internet.