8 Puzzle – Python3

Kogard Valentin
Kominek Samuel
Klaus Guntram

version 1.0

30.12.2022

Table of contents

Kogard Valentin, Kominek Samuel, Klaus Guntram
CSDC24VZ

1. Task description

The 8-puzzle is to be implemented as an introductory project in Python or Java as part of the course "Introduction to AI". It is a two-dimensional game, which has a base area of 3x3 fields.

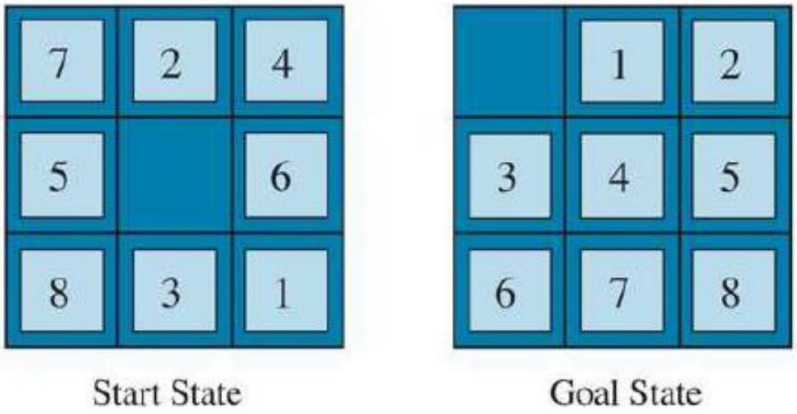

Figure 1 8 Puzzle

The goal is to move the individual fields in such a way that the target state is reached at the end. Only fields directly adjacent to the empty field may be moved. A diagonal shift is not allowed.

The realization by means of a brute force algorithm is possible, however, this needs in comparison to the realization with a heuristic function very long and is therefore not purposeful. Therefore, the participants in this project deal with the implementation of two different heuristic functions - Hemming and Manhatten.

Furthermore, the software technique “pair programming” was used with only one laptop. This has resulted in git commits not being evenly distributed among the participants in the group. The code, which was written in the context of this project can be found at the following link: https://github.com/valentinkogard/8Puzzle.git

Kogard Valentin, Kominek Samuel, Klaus Guntram
CSDC24VZ

2. Design Decisions

The SW was implemented in Python3. Thereby it was paid attention to the fact that the SW design was realized object-oriented. To ensure this, several files were created, each with a class. Furthermore, most classes have a constructor, so that an instance of the respective class can be generated. Those classes, which do not have a constructor, contain methods, which are meant for instance for writing text files. Furthermore, the class "heuristic" does not contain a constructor, since in this the computation of the distance between current play condition and desired play condition is computed.

2.1. SW Architecture Diagram

Figure 2 shows the activity diagram for the designed eighth puzzle. In the first step the game is initialized and afterwards the first randomly generated game board is created. Then it is checked whether it is solvable. In case it is not solvable, a new board is generated and again checked for solvability. If it is solvable, the algorithm Hamming is used in the first step to solve the puzzle and the algorithm Manhatten in the second step. Then, the number of iterations needed and the amount of time to successfully solve the puzzle of each algorithm is stored in a text file for subsequent evaluation.

This process is repeated several times and ends when the number of boards to be created has been reached.

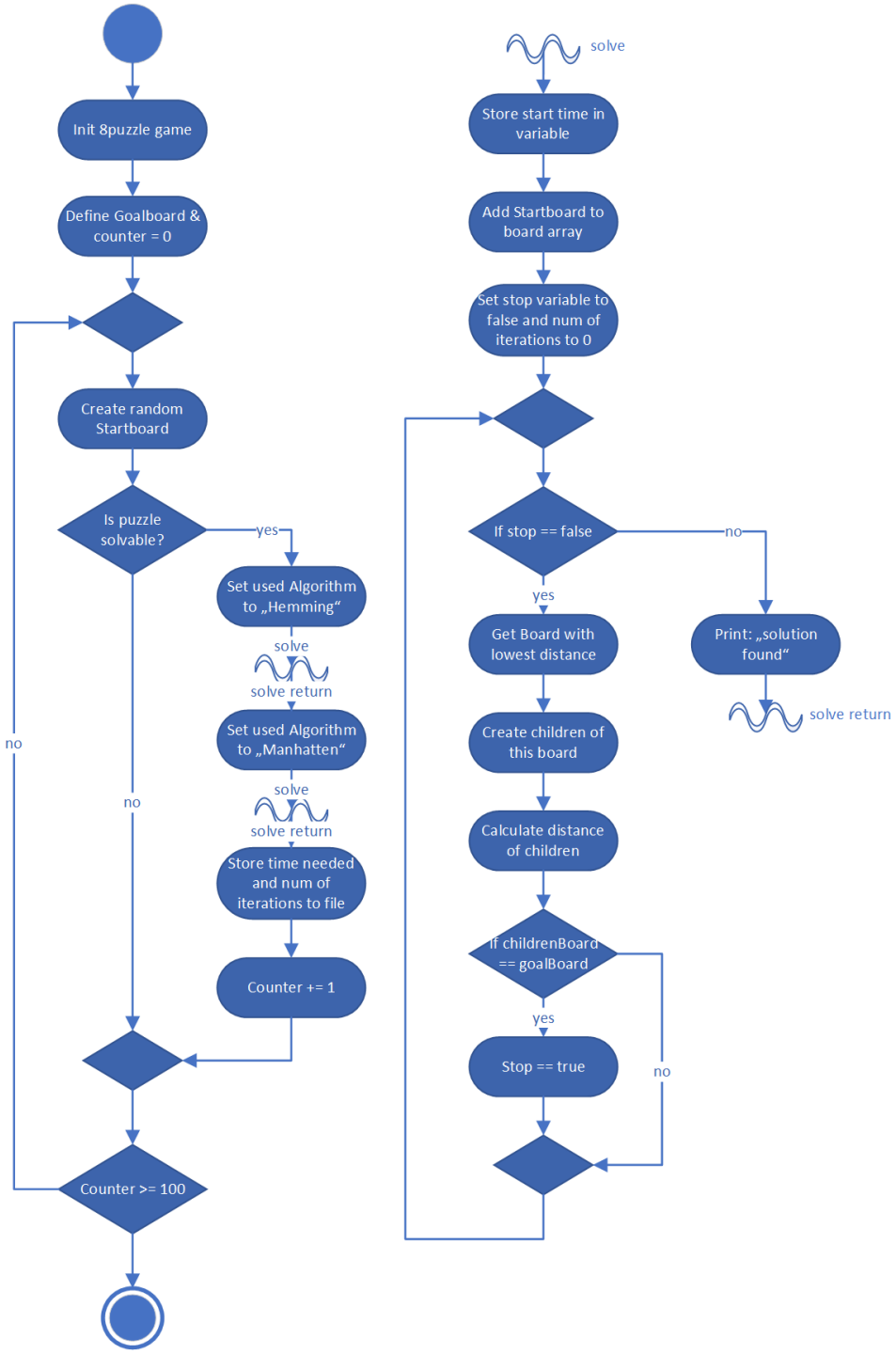

Figure 2 Activity Diagram

Kogard Valentin, Kominek Samuel, Klaus Guntram

2.2. Check for Solvability

The solvability of an eighth puzzle depends on the arrangement of the individual numbers in the board. To check this, the following code was implemented.

```
    def isSolvable(self, board):
        """
        counts the inversions - if inversion is odd the puzzle is NOT
solveable / if the inversion is even the puzzle is solveable

        Parameters:
            self (Solve): solve object.
            board (Array): a 2D matrix.

        Returns:
            solvable (Bool): is puzzle solvable

        """
        counter = 0

        for i in range(len(board)*len(board[0])-1):
            row1 = math.floor(i/len(board))
            col1 = i % len(board)
            for j in range(i+1, len(board)*len(board[0])):
                row2 = math.floor(j/len(board))
                col2 = j % len(board)
                if(board[row1][col1] != "_" and board[row2][col2] != "_"):
                    if(int(board[row1][col1]) > int(board[row2][col2])):
                        counter += 1
        if(counter % 2 == 0):
            return True
        return False
```

The function isSolvable() is passed a two-dimensional game board in the first step. Since each field must be checked, it is most convenient to use a for loop. However, since the game board is not a one-dimensional list, it cannot be iterated through with the counter of the loop. To implement this anyway, the simple mathematical formulas *math.floor(i/len(board))* and *i % len(board)* were used. Afterwards it is checked that the empty field - in this case marked with an underscore - is not considered in the calculation of the *counter*. This would otherwise lead to a wrong result. After the complete check whether the first number is larger or smaller than the following number in the array, the *counter* modulo two is calculated to determine whether this is an even or odd number. If the *counter* is even, the individual fields are only twisted to each other and the puzzle can be solved. If the *counter* is odd, there is no solution even after an infinite number of attempts to move the puzzle pieces.

2.3. Implementation of heuristic functions

The eighth puzzle can be implemented using a brute force algorithm. However, since such algorithms try out all possible combinations to reach the desired goal, this algorithm is not useful (at least in this example). It is therefore recommended to choose an algorithm that works with so-called distances. If this distance is very close to zero, then the target is not far from the current point. If the distance is zero, the target has been reached. These algorithms are called heuristic algorithms.

To implement this in the eighth puzzle, the algorithms of Hemming and Manhatten are used and then compared.

2.3.1. Hamming

Hamming's algorithm compares, in the case of this example, the current two-dimensional game board with the target result of the game board. In the first step, two boards are passed to the function hemming(). In this particular example it is irrelevant whether the parameter board1 or the parameter board2 contains the target game board. The other parameter contains the board to be matched with the target board. Then the two boards are compared field by field. If the considered fields are the same, the counter, which was initialized with 0 before, is not increased. If the fields are different, the counter is increased by one.

```
    def hemming(board1, board2):
        """
        heuristic function which counts wrong placed fields.

        Parameters:
            board1 (Array): a 2D matrix.
            board2 (Array): a 2D matrix.

        Returns:
            counter (int): number of missmatches.
        """
        counter = 0

        for i in range(len(board1)):
            for j in range(len(board1[0])):
                if(board1[i][j] != board2[i][j]):
                    counter += 1
        return counter
```

Through this method, two different game boards, both of them not yet reached the target state, can be compared and the game board with the lower counter or distance to the target game board can be considered first and the child game boards can be calculated.

2.3.2. Manhatten

Mahatten's algorithm has the same goal as Hamming's algorithm. However, the approach is different.

This algorithm does not compare whether a single number is in the right or wrong place in the board but calculates the distance between the actual and the target position. For this it determines the horizontal and the vertical distance and then adds them together. This is done for each square. The result is therefore the sum of all individual distances of all numbers.

```
    def __calcDistManhatten(board1, board2, row, col):
        """
        calculates the manhatten distance between should place and current
place
        Parameters:
            board1 (Array): a 2D matrix.
            board2 (Array): a 2D matrix.
            row (int): row of current field.
            col (int): column of current field.
        Returns:
            horizontalDist + verticalDist (int): distance
        """
        symbol = board1[row][col]
        for i in range(len(board1)):
            for j in range(len(board1[0])):
                if(symbol == board2[i][j]):
                    horizontalDist = abs(row - i)
                    verticalDist = abs(col - j)
                    return horizontalDist + verticalDist

    def manhatten(board1, board2):
        """
        distance between solution and start point
        Parameters:
            board1 (Array): a 2D matrix.
            board2 (Array): a 2D matrix.
        Returns:
            distance (int): this distance includes the whole gameboard
        """
        distance = 0
        for i in range(len(board1)):
            for j in range(len(board1[0])):
                distance += heuristic.__calcDistManhatten(board1, board2, i,
j)
        return distance
```


Kogard Valentin, Kominek Samuel, Klaus Guntram

3. Modules and Interfaces

For better readability of the code, the different methods were written in different classes, which in turn can be found in different files.

3.1. main.py

This file is executed first. An object of Puzzle is created and then a method of this object is executed. The Puzzle object contains methods to initialize the target game board and the start game board. The random creation of a start game board is also defined in this class.

3.2. solve.py

In this file the class solve of the same name is present. This has methods which are used to set the algorithm to be used. Also implemented are methods for executing the selected algorithm, for checking the solvability of a puzzle and the actual solve function.

3.3. boardtree.py

This file contains - like the solve.py file - a class, which has the same name as the file itself - boardtree. This is needed to contain all the different boards and to provide different methods to retrieve them.

All boards are stored in a linear list. Each board has the additional attributes “expanded” and “distance”. These are needed to know the status of the boards later without repeated loop passes.

The class also has methods which, for instance, return the board with the smallest distance or the number of all stored boards.

3.4. heuristic.py

This file also contains a class (heuristic) but has no constructor to be able to create an object. The reason for this decision is that there is no object-oriented data to be stored here. Instead, in this class is the actual implementation of both heuristic functions – Hamming and Manhatten.

4. Discussion and Conclusion

In this chapter, the two different heuristic algorithms are compared and then personal experiences with this project are described.

4.1. Comparison between different heuristic functions

Hamming's algorithm and Manhattan's algorithm both proved their correctness and, more importantly, their functionality in the project. Nevertheless, the team was able to identify significant performance differences between both of them.

In order to establish a valid comparison, over 200 random game boards, which were defined as solvable according to chapter 2.2, were generated and given to both of them to solve. The required time and the required iterations were then stored in a text file for evaluation.

Figure 3 shows the required iterations to solve the puzzles. On the x-axis the different puzzles are plotted and on the y-axis the number of iterations needed. If the outliers of Hamming are not considered, it could be claimed that Manhattan and Hamming are about equally good. However, if the same data is plotted in a box plot diagram (Figure 4), it is clear that the algorithm requires much fewer iterations on average than Hamming's algorithm.

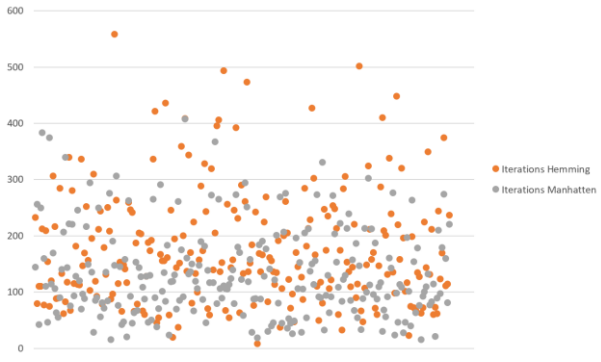

Figure 3 Iteration comparison - dot diagram

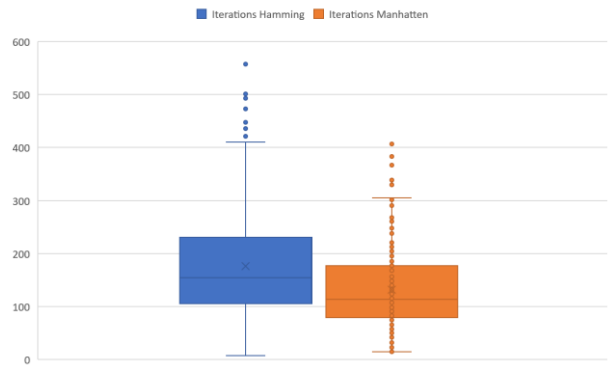

Figure 4 Iteration comparison - boxplot

Considering the time each algorithm needs to solve the puzzles, it can already be seen in the dot plot (Figure 5) that the Manhattan algorithm is more effective. This is also proven by the box plot diagram in Figure 6.

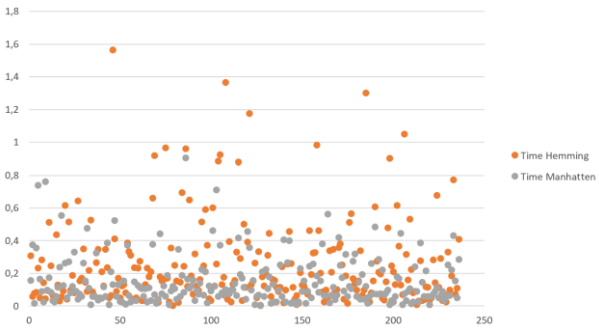

Figure 5 time comparioson - dot diagram

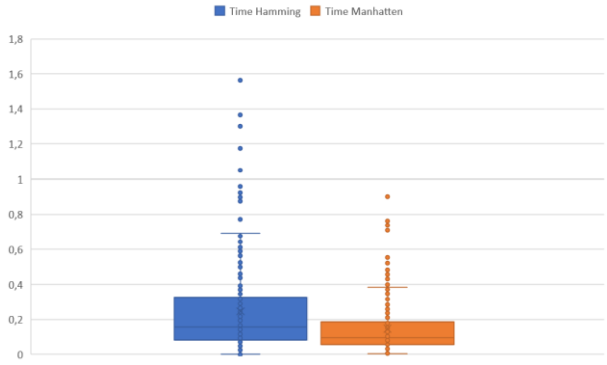

Figure 6 time comparison - boxplot

In conclusion, it can be said that Manhattan's algorithm is better than Hamming's algorithm for the 8-puzzle problem.

4.2. Describe your experience

The 8-puzzle is a good task to get a sense of heuristic functions. Also, it makes sense to realize this task in Python, as this is a language that is gaining more and more importance in the world of computer science and thus prepares students well for their future career paths.