Method to solve the 15-Block Puzzle using C++

Tanay Rangasamy: 2544162

School of Electrical and Information Engineering (Digital Arts)

Abstract

The 15-Block puzzle is a 4x4 matrix which can be solved by organizing the digits in ascending order using the block with no digit as a reference for movement. The puzzle can hence be solved by implementing heuristic functions (Ooman & Rueda, 2005) as the 15-Block puzzle is solved by getting the current state of the puzzle to the "optimal state". The heuristic function implemented is by combining the Manhattan Distance Algorithm with another heuristic to get the final state of the puzzle (Chiu et al, 2016). The algorithm used is therefore a combination of the Manhattan Distance Algorithm and the IDA search methods (Bu & Korf, 2019).

1.Introduction

The 15-Block puzzle is solved by moving the digits until they are in the goal state (ascending order). The puzzles can hence be solved by implementing the heuristic functions in C++.

The C++ code to solve this puzzle will utilize classes and functions to implement the Manhattan Distance Algorithm to solve the puzzle in as few moves as possible. The only constraint to the program is that no global variables may be used and only 1000 moves or less may be made.

The design stages for the project are therefore split into multiple sections, namely: implementing movement of digits into the block with no digit, implementing the Manhattan Algorithm, and finally solving the puzzles.

2.Design

2.1 Implementation of the movement of digits

Digits are moved around the puzzle by moving digits adjacent to the blank block into this block. The initial puzzle state is stored in a two - dimensional array and a function is used to find the position of the blank space in terms of the row and column index.

Simple movements centered around the blank position are created, namely an "Up" function which moves the digit above the blank position into the blank position, a "Down" function which moves the digit below into the blank position, a "Left" function which moves the digit to the left into the blank position and a "Right" function which moves the digit to the right into the blank position. These functions simply swap the two numbers in the array, therefore the blank space swaps with the digit above, below, to the right or to the left, depending on the function used. These functions use the column and row index of the blank space to determine which digits to swap.

The region of the array where the blank space is will determine which movement functions can be performed ("Up", "Down", "Left" or "Right") as some movements will not be allowed in certain parts of the array as shown in Figure 1. Multiple "if statements" are used to accommodate these movement restrictions.

Down Right	Down Left Right	Down Left Right	Down Left
Up Down Right	Every direction allowed	Every direction allowed	Down Up Left
Up Down Right	Every direction allowed	Every direction allowed	Down Up Left
Up Right	Up Left Right	Up Left Right	Up Left

Figure 1: Grid displaying movement allowed depending on position of blank space

An additional two-dimensional Boolean array is implemented to track which digit moved into the blank space. This array will be used to prevent moves from looping as the movement will be restricted if the array shows that the same digit was used in the prior move.

2.2 Implementing the Manhattan Distance Algorithm

The Manhattan Distance Algorithm is given by Equation 1 (Singh, 2022). This algorithm will be used to generate a numerical value for each of the possible moves. The lowest value will determine which move the program will make as the Manhattan Distance approaches zero the closer it gets to the completed puzzle state (Chiu et al, 2016). The algorithm gets the sum of the absolute values of the vertical distance and the horizontal distance between a digit's current position and its correct position. These values are summed to provide the total Manhattan Distance for the puzzle's configuration when a move is made.

(1) $\sum_{i=0}^{n}|xgoali - xcurrenti| + |ygoali - ycurrenti|$ = Manhattan distance

3.Results

3.1 Testing the movement of digits

The functions for moving the digits (“Up’, “Down”, “Left’ and “Right”) worked well in moving the digits accurately in conjunction with the restrictions highlighted in Figure 1. The two-dimensional array prevented moves from looping minimizing unnecessary moves.

3.2 Solving the puzzle using the Manhattan Algorithm

The puzzle used to test the program is given by Figure 2. The Manhattan Algorithm initially worked for the first 15 moves but then the moves would begin bypassing the restrictions set by the two-dimensional array and the moves would begin looping. This was due to some Manhattan distances being equivalent as shown in Figure 3 and Table 1. Therefore, it was observed that logical errors were causing the moves to alternate between two Manhattan distances and hence moves were looping.

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

Figure 2: Puzzle used to test program

Table 1: Manhattan Distances

Move	1-15	20-30	40+
MD	25	20	18/25

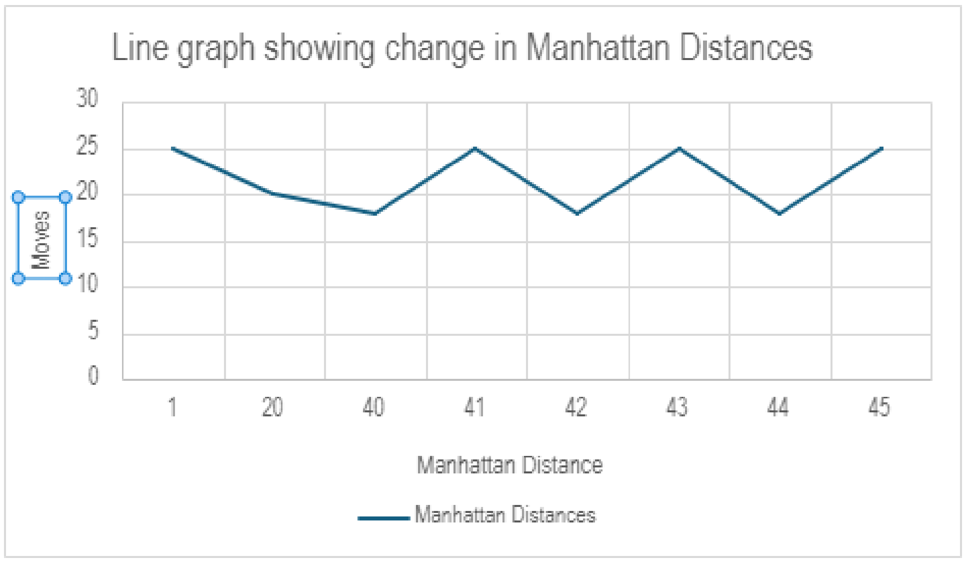

Figure 3: Line graph showing the changes in Manhattan distances

4.Discussion and analysis of results

Despite the precautions to prevent moves from looping the equivalent Manhattan Distances for certain moves caused looping to occur.

The program was altered to reinforce the prevention of looping moves. The primary restriction utilizing the two-dimensional array is now the initial condition. Manhattan Distances are therefore only implemented on moves that are allowed and previous moves are no longer considered in determining the next move.

After reinforcing the restrictions, the Manhattan Distance solves most of the rows in the puzzle, but there are still a few digits not in the correct positions. The Manhattan Distance algorithm solves the puzzle; however, this algorithm cannot usually solve it within the constraint of only 1000 moves. Therefore, the Manhattan Distance algorithm alone is not optimal in solving the 15-Block Puzzle.

To improve the program so it can solve the puzzle within the constraints, another heuristic function must be used with the Manhattan Distance algorithm such as the breadth-first search or the depth-first search (Russel et al, 2019).

Conclusion

The Manhattan Distance Algorithm alone is not optimal for solving the 15-Block Puzzle. To solve the puzzle optimally with as few moves as possible another heuristic needs to be introduced in conjunction with the Manhattan Distance Algorithm.

References

[1] BJ Oommen, LG Rueda - Artificial Intelligence, 2005 – Elsevier.

[2] W. -Y. Chiu, G. G. Yen and T. -K. Juan, "Minimum Manhattan Distance Approach to Multiple Criteria Decision Making in Multi-objective Optimization Problems," in *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 6, pp. 972-985, Dec. 2016.

[3] V Singh, “All about Manhattan Distance”, Shiksha online, December 2022.

[4] Z Bu, RE Korf - IJCAI, 2019 - ijcai.org

[5] S Russel, P Nordvig. “Artificial Intelligence-A Modern Approach”. December 2019.

Appendix

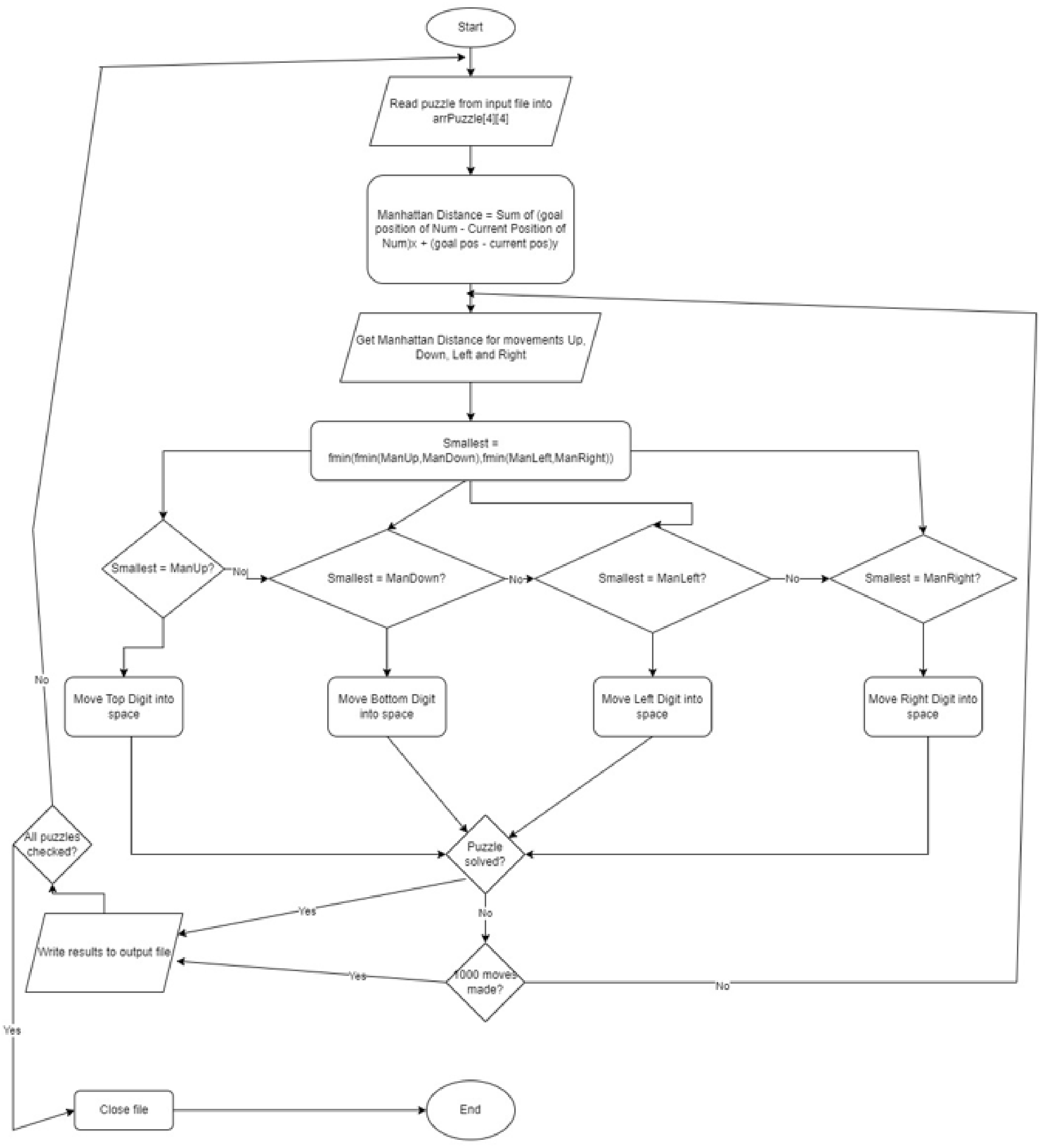
Start
Read puzzle from input file into arrPuzzle[4][4]
Manhattan Distance = Sum of (goal position of Num - Current Position of Num)x + (goal pos - current pos)y
Get Manhattan Distance for movements Up, Down, Left and Right
Smallest = fmin(fmin(ManUp,ManDown),fmin(ManLeft,ManRight))
Smallest = ManUp?
No
Smallest = ManDown?
No
Smallest = ManLeft?
No
Smallest = ManRight?
Move Top Digit into space
Move Bottom Digit into space
Move Left Digit into space
Move Right Digit into space
Puzzle solved?
Yes
No
1000 moves made?
Yes
No
Write results to output file
All puzzles checked?
No
Yes
Close file
End

Figure 4: A Flowchart of the program

Table 2: Planned schedule for project

Date	**Activity**
22 April 2024	Research algorithms
26 April 2024	Begin coding basic movement functions
27 April 2024	Code solving algorithm
28 April 2024	Finish coding and check for errors
29 April 2024	Write report
30 April 2024	Make flowchart

Table 3: Actual date of activities

Date	**Activity**
27 April 2024	Begin coding basic movements
28 April 2024	Research Algorithms
29 April 2024	Code algorithms
30 April 2024	Fix looping moves issue
3 May 2024	Fixed looping move issue
4 May 2024	Write report, make flowchart, fix Manhattan Algorithm
5 May 2024	Fix classes