Eight Puzzle

Richard Cox, Andreas Dworscak

Short Task Description

The goal of this exercise is to solve an Eight Puzzle with two different implementations of the A* algorithm. For this task, *Manhattan* and *Hamming* distances were used for the algorithm. Each solution starts with a random state of an Eight Puzzle board. This means, every piece (0 - 9) is in a random place in the 3 by 3 matrix of the board, where 0 represents the empty tile. After generating, one should check whether the puzzle is solvable. Since we generate a board by first generating a solved one, and then shuffling it up, the puzzle is guaranteed to be solvable, thus, no check has to be done in this implementation. As already discussed, the puzzle shall be solved using two different heuristic functions (Manhattan, Hamming), as well as estimating the time complexity of each. Measurements of run time and nodes expanded are to be implemented, too. A console based GUI for choosing the type of the algorithm is displayed at the start of the program.

The program can be downloaded from GitHub through the following link: https://github.com/DworshiFH/EightPuzzle

Heuristics

Hamming

Hamming Distance is straight forward. It counts the given elements of an object out of place when compared to another object. In our case we count the number of tiles not in place when compared to a solved puzzle. Consider the following boards:

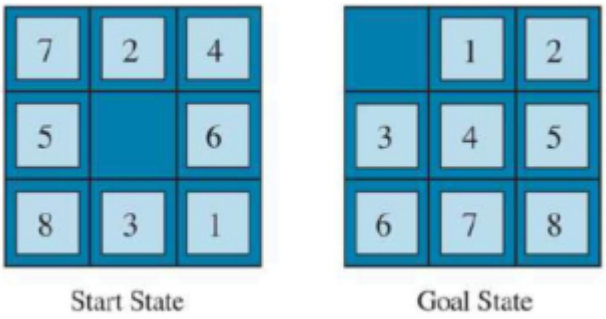

Figure 1: Start and goal states

When both boards are compared using hamming distance, the result will be eight. This means that eight tiles are not in the place they should be.

Manhattan

Manhattan Distance is a bit more complicated than Hamming. It is the sum of the lengths of the projections of the line segment between the points onto the coordinate

axes. In simple words, it is the sum of the distances in all dimensions between two points. For the eight puzzle, the sum of all distances are accumulated into a single value.
Consider *figure 1* again, in the case of Manhattan distance the result will be 18. For each tile, we add up the distance from its current place to the place where it should be, for the x and y axes respectively.

Software Architecture

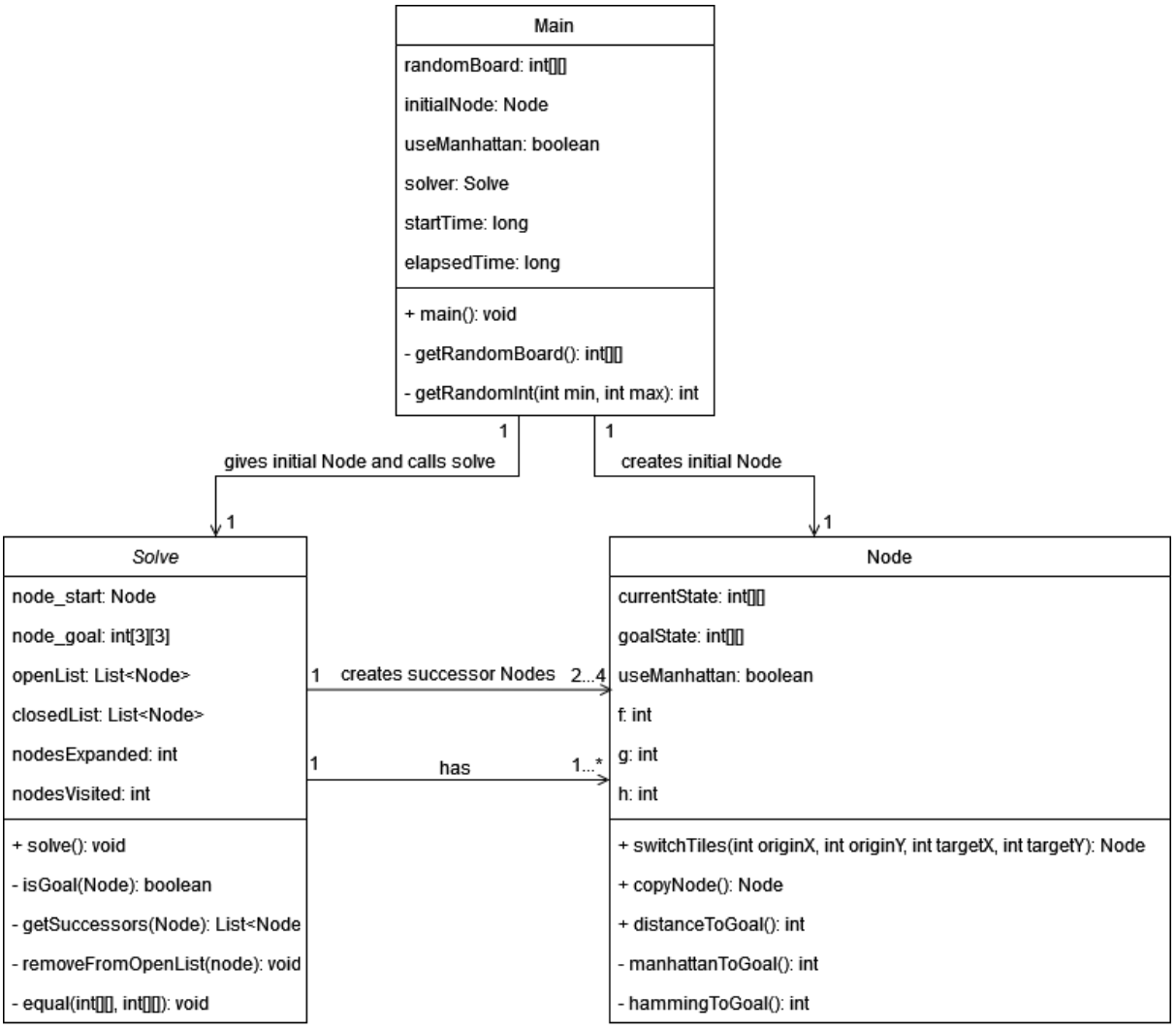

Figure 2: Class Diagram

Main

The class Main controls all aspects of the program. It contains a simple UI, the initial node as well as an object of class Solver. When the heuristic function, or a comparison mode has been chosen by the user, it constructs the variable solver by giving it the initial node, which has previously been constructed using the randomBoard variable. After which it logs the current time, prints the initial Node to

the console, and calls the solve() method. When solve has finished, Main() logs the elapsedTime and prints it to the console.

In the case of a statistically relevant comparison, Main() will solve 100 puzzles witch each puzzle being solved both using Manhattan and Hamming distance. The logged data consisting of of the initial node, elapsed time and nodes expanded will then be saved into a pair of arrays of simple “Statistics” objects. One array contains data for Manhattan distance, the other contains data for Hamming distance. When finished solving all 100 puzzles, the content of both arrays will then be saved to .csv files for later analysis.

Solve

Solve contains variables related to the A* algorithm [1], as well as two measuring variables , nodesExpanded and nodesVisited. When the constructor is called, it will first copy the goal state from the initial node, after which it sets the start node to be the initial node and initializes nodesExpanded and nodesVisited to zero. When solve() is called, the pathfinding to the goal state will be executed. A number of private functions are also in this class, they are all related to the functionality of solve(). The A* algorithm has been designed using pseudo code [2] from a website.

Node

Each node represents a current state of the puzzle as a 2-dimensional int array. Also, each node has a goal state as a 2-dimensional int array, which is used to check whether the puzzle has been solved or not. Of course, each A* algorithm needs its own values for f, g, and h, which are represented as ints in this class. The main method in this class is the switchTiles() one. As the name says, it switches a tile with one of its neighbors. Its parameters are the X and Y coordinates of the origin and the X and Y coordinates of the targeted place. First, it checks if a move is possible or not, if it is possible, it returns a new node where the tiles have been switched, if it is not possible, it returns null. Another method is the distanceToGoal() and its only purpose is to return the manhattanToGoal() or hammingToGoal() depending on the value of the useManhattan boolean.

Design decisions

For the sake of refreshing our Java coding knowledge, we decided to use Java when implementing our project, even though Python might have been a better choice for implementing algorithms. For ease of use (and because we are used to it) we decided to implement our nodes as class objects.

Before solving an eight puzzle, it is advisable (required as per assignment) to check whether the puzzle is solvable. In our case it is NOT necessary to check for solvability, as the function getRandomBoard actually simulates the way a real board

would be shuffled. If there is a path to the shuffled puzzle, there is a path to a solved puzzle.

Discussion and conclusion

Our experience

The goal of this exercise was to solve the Eight Puzzle using two different Heuristic search algorithms. This is a very good way to get in touch with informed search algorithms, and generally implementing code from a pseudo code source.

Complexity comparison

In order to compare Manhattan and Hamming distance, first the initial nodes, of a set of 100 solved puzzles, with the maximum as well as minimum number of nodes expanded, has been found. The following tables show the results:

	Value	Unit	Initial Node
Minimal number of nodes expanded	61	[n]	[[014][857][362]]
Minimal amout of time expended	0,1143	[ms]	[[014][857][362]]
Maximal number of nodes expanded	16702	[n]	[[678][431][205]]
Maximal amout of time expended	14162,4105	[ms]	[[678][431][205]]

Figure 3: Maxima and Minima when using Manhattan Distance

	Value	Unit	Initial Node
Minimal number of nodes expanded	303	[n]	[[452][318][670]]
Minimal amout of time expended	1,5581	[ms]	[[452][318][670]]
Maximal number of nodes expanded	104759	[n]	[[546][027][831]]
Maximal amout of time expended	608746,8483	[ms]	[[703][186][254]]

Figure 4: Maxima and Minima when using Hamming Distance

The following table represents an overview of Figure 3 and 4. It considers the best and worst cases of both heuristics and compares how they fared mutually to each other.

Minimal	Manhattan		Hamming	
Initial Node	Nodes expanded [n]	Time expended [ms]	Nodes expanded [n]	Time expended [ms]
Minimum Number of Nodes Expanded				
[[014][857][362]]	61	0,1143	526	3,4986
[[452][318][670]]	130	0,5029	303	1,5581
Maximum Number of Nodes Expanded				
[[678][431][205]]	16702	14162,4105	103887	608460,4133
[[546][027][831]]	12658	5639,1211	104759	604769,6293
Minimum Amount of Time Expended				
[[014][857][362]]	61	0,1143	526	3,4986
[[452][318][670]]	130	0,5029	303	1,5581
Maximum Amount of Time Expended				
[[678][431][205]]	16702	14162,4105	103887	608460,4133
[[703][186][254]]	10093	2358,3272	104166	608746,8483

Figure 5: Time and space complexity table

It can easily be seen that using Manhattan distance as a heuristic is the superior approach in the case of an eight puzzle. The time it takes to solve each puzzle is around two orders of magnitude lower, when compared to Hamming distance. The number of nodes expanded is around an order of magnitude lower, when compared to Hamming distance.

The full set of data and evaluations can be found in Appendix 1.
Appendix 1 also contains a few interesting graphics.

The median of nodes expanded and time expended can be found in the following table:

Medians	Manhattan	Hamming
Median Nodes Expanded	820	8718 [n]
Median Time Expended	10,27535	1561,767 [ms]

Figure 6: Time and Space complexity medians

Possible improvements

One possible improvement could be to include more heuristics for comparison. Such as Euclidean distance, Nilssons's Sequence Score, Hamming for tiles out of row + Hamming for tiles out of column.

The presentation "Solving the 8-Puzzle using A* Heuristic Search" [3] goes into detail for a number of admissible heuristics.

Appendices

1. Evaluation And Data Sheet

References

[1] A* search algorithm. Accessed on 5th June 2022. [On-line]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm

[2] A* algorithm pseudo code. Accessed on 5th June 2022. [On-line]. Available: https://www.geeksforgeeks.org/a-search-algorithm/

[3] Solving the 8-Puzzle using A* Heuristic Search. Accessed on 5th June 2022. [On-line]. Available: https://cse.iitk.ac.in/users/cs365/2009/ppt/13jan_Aman.pdf