8-PUZZLE EXERCISE DOCUMENTATION

Introduction to AI and Data Science

WS 2022/23

CSDC24 VZ

Lena Gross, Magdalena Keller, Susanne Kloss

Inhaltsverzeichnis

1. 8-Puzzle Task Description

A program has to be written that solves the classical 8-puzzle - 8 numbered sliding tiles arranged in random order on a 3 x 3 board - by using the A* algorithm and implementing two different heuristics, Hamming distance and Manhattan distance. Starting from a random start state, a goal state has to be generated, where the tiles are aligned in ascending order starting with the blank tile (s. figure 1). All randomly generated puzzles have to be checked for solvability.

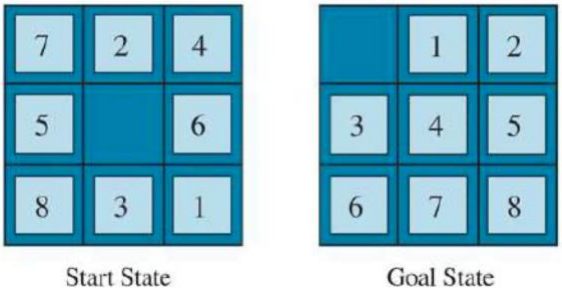

Figure 1: start state (random) and goal state of the 8-puzzle

An estimation of the algorithm's complexity has to be provided and the time and space complexity of the two heuristics have to be compared. For this, run time and the number of nodes expanded have to be measured for each of 100 random states and each heuristic. Then, mean, and standard deviation of execution time and memory usage for each heuristics have to be calculated. The implementation can be done in Python or Java.

After a short introduction on A* algorithm, Hamming and Manhattan distances this documentation will proceed with an activity diagram to illustrate the implemented software architecture. This is followed by a short description of the code components and an explanation of the design decisions made. Finally, in the discussion and conclusion part an analysis of the complexity of the A* algorithm and the two implemented heuristics, Hamming distance and Manhattan distance, is provided. The documentation concludes with an evaluation of the experience doing this exercise and the insights gained for future projects.

2. A* Algorithm, Hamming and Manhattan Distances

The goal of the exercise is solving a random 8-puzzle by sliding the tiles horizontally or vertically into the empty space. An important feature of the 8-puzzle is the fact, that the goal state cannot be reached from every randomly generated start state. Solvability can be checked by counting the number of inversions in the initial state, with an inversion being a pair of tiles in reverse order compared to the goal state. For an odd numbered board, like the 3 x 3 board used in this exercise, only an even number of inversions in the initial state will lead to a solved 8-puzzle.

Different algorithms can be used for finding the solution of an 8-puzzle. Among them is A* algorithm, an extended and improved Dijkstra algorithm performing an informed search. For calculating h(n), the distance from a node to the goal state, different heuristic approaches can be used. Hamming distance simply counts the number of misplaced tiles. Manhattan distance is the sum of the vertical and horizontal distances of the tiles to their position in the goal state. A* algorithm takes also into account g(n), the path cost from the initial state to a node, and then always chooses the next node n with minimal f(n) = g(n) + h(n).

3. Software Architecture - Activity Diagram

4. Code Components and Interfaces

The implementation was done in Python using PyCharm as IDE. The user interaction happens via console, the user can choose from 4 options and exit with option 5 (s. figures 3 and 4). Figure 2 shows the project structure, the main components are explained below.

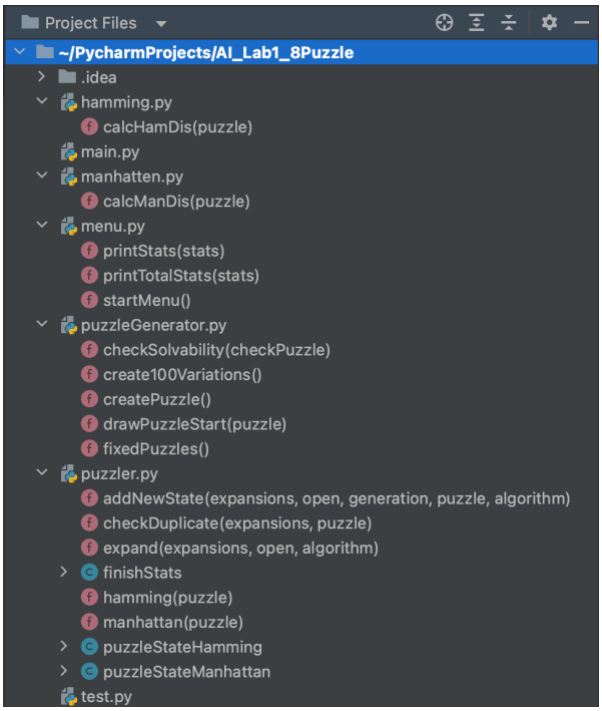

Figure 2: 8-puzzle program project structure

main.py: just calls *startMenu()*

hamming.py and manhattan.py: contain the heuristics with just one function each for the calculation of the Hamming and Manhattan distances of a puzzle

puzzleGenerator.py: for the puzzle generation, with *createPuzzle()* and *checkSolvability()* a solvable random puzzle - an array from 0 to 8 in random order - is generated, *create100Variations()* returns a list of 100 puzzles

puzzler.py: contains the main part of the A* algorithm. There are the classes *puzzleStateHamming* and *puzzleStateManhattan* with parameters f(n), g(n), h(n) and a closed Boolean for already expanded puzzle states. The class *finishStats* has the parameters time (run time), steps (count of tile movements), expansions (count of expanded nodes). For a puzzle run *hamming()* or *manhattan()* are needed, they call *expand()* and return a finishStats object. *expand()* first searches for the state with minimal f(n), then calculates the state's index of the blank tile and expands from there up, down, to the right or to the left. After expansion the

parent generation is set to closed. Through *addNewState()* the new expanded state is generated, with *checkDuplicate()* trashing a potential already existing state. If Manhattan or Hamming distance equals zero, then the goal state has been reached. *addNewState()* returns a Boolean true, causing a break in *hamming()* or *manhattan()*.

menu.py: main menu of the program, via the console *startMenu()* lets the user choose among the different options to solve the 8-puzzle, only accepting values from 1 to 5. It calls *printStats()*, which then either calls *manhattan()* or *hamming()* from **puzzler.py** when only 1 puzzle is to be solved (options 1 or 2). For running 100 puzzles (options 3 or 4) first a puzzles list is created by calling *create100Variations()* from **puzzleGenerator.py**. In the end *printStats()* is called for 1 puzzle or *printTotalStats()* for 100 puzzles to print statistics to the console (s. figures 3 and 4).

```
(1)  Solve 1 Puzzle using Manhattan
(2)  Solve 1 Puzzle using Hamming
(3)  Solve 100 Puzzles using Manhattan
(4)  Solve 100 Puzzles using Hamming
(5)  Exit

Choose Option: 2

Hamming (x1)
Expansions: 3640
Steps: 355
Time taken: 0.705

Press enter to continue.
```

Figure 2: view of console with options for 8-puzzle and stats for 1 Hamming run

```
(1)  Solve 1 Puzzle using Manhattan
(2)  Solve 1 Puzzle using Hamming
(3)  Solve 100 Puzzles using Manhattan
(4)  Solve 100 Puzzles using Hamming
(5)  Exit

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]: 121569 / 1215.69
Steps [total/mean]: 12322 / 123.22
Time [total/mean]: 14.22 / 0.142

Expansions standard deviation: 857.308
Steps standard deviation: 85.103
Time standard deviation: 0.205

Press enter to continue.
```

Figure 3: stats for a 100 puzzles Hamming run, the list of numbers from 0 to 99 indicating the progress of calculation

5. Explanation of Design Decisions

We decided to use Python as programming language for this exercise mainly because this exercise gave us the chance to learn more about programming with Python, which we got to know only briefly in the first semester. It also gave us the opportunity to do so directly in the field of AI and data science, an area in which it is especially important for students of computer science to get a mastery of Python. PyCharm was chosen because the team members had already worked with it and with other IDEs from JetBrains.

As for the project structure, we wanted to logically separate the different parts of the program and the two different heuristics, Manhattan and Hamming distances, where it made sense. At the same time the program should be as compact as possible and avoid unnecessary code reuse. Therefore, the calculation of Hamming and Manhattan distances is done in separate files, and it also made sense to create a separate file for the generation of the puzzles. Thus puzzler.py, the file containing the main part of the A* algorithm, was not overloaded but stayed neat and clear. The creation of three main classes brought the benefit of object oriented programming, again, where it made sense to us. Classes *puzzleStateHamming* and *puzzleStateManhattan* allow caching and access to the puzzle states, class *finishStats* does so regarding the gathering of the statistical values.

Finally, another important design aspect was the usability of the program. On one hand, this regards code comprehensibility and involved the naming of the code components and writing clean and sufficiently commented code. On the other hand, we wanted the program to be user friendly. So, there is checking for invalid input and the results are printed to the console in a neatly arranged way. In addition, when running puzzles 100 times, the user is not left waiting alone in front of an empty console, but can follow the calculation progress, which might take some time.

6. Discussion and Conclusion

6.1. Complexity Analysis of A*, Hamming and Manhattan Distances

8-puzzles is an exponential search problem. Time and space complexity for using an A* algorithm will be $O(b^d)$, with b being the branching factor and d the depth of the solution when picturing the algorithm as a search tree. For an 8-puzzle the branching factor will be 2 to 4, depending on the location of the blank tile. But the crucial factor here is the depth of the solution, as it increases the number of nodes expanded exponentially.

So, for the evaluation of time and space complexity the used heuristic is the decisive factor. Rather than finding the shortest path to the goal state, like the Dijkstra algorithm does, it is much better to find a path with expanding as few nodes as possible. This can be achieved by finding a heuristic that is as admissible as possible, meaning it is near to the true costs.

One way of doing this, can be by first relaxing the problem definition. For the 8-puzzle the calculation of the Manhattan distance does so by deleting the constraint that it is not possible to move a tile to an occupied field. The calculation of the Hamming distance ignores the fact that a tile can only be moved to an adjacent field.

6.2. Comparison of Hamming and Manhattan Distances

For the comparison of the two heuristics, Manhattan and Hamming distances, the run time and the number of nodes expanded were measured for each of 100 random searches and each heuristic. From these values mean and standard deviation were calculated. The results are listed in table 1.

Table 1: time and space complexity calculated from 100 Hamming and Manhattan runs

	sample size = 100	**Hamming**	**Manhattan**
Run time [sec]	mean	0.142	0.026
	standard deviation	0.205	0.024
Nodes expanded	mean	1215.69	504.38
	standard deviation	857.308	303.114
Steps??	mean	123.22	42.49
	standard deviation	85.103	17.357

Comparing the values for time and space complexity when solving an 8-puzzle with different heuristics, it is obvious that the heuristic using the Manhattan distance clearly outperforms the approach relying on the Hamming distance. Regarding the run time, the mean of 100 Manhattan runs is lower by the factor of 5.5 compared to the mean of 100 Hamming runs, moreover Manhattan's mean memory usage (number of nodes expanded) is lower by the factor of 2.4. Regarding the standard deviation, the values for run time and number of expanded nodes for the 100 Hamming runs are scattered more than those for the 100 Manhattan runs.

6.3 Evaluation of exercise experience - possible future improvements

First, we are quite happy with the outcome of our first computer science project in the field of AI. Still, for the next time we plan to first take time and think in more detail about the software architecture and the design of our project, applying the different skills and techniques we have learned in this field especially in this semester at the FH. Also, the testing of the code surely could be expanded, and unit testing should be implemented.

Regarding the A* algorithm, a possible improvement would be the implementation of a priority queue instead of always searching for the node with the least cost. Also, it would be a nice additional feature that the user can choose the number of runs freely.