COMP 472 – Mini project 1

Heuristic search - Analytical report

October 13th, 2018

Victor Debray (ID: 40102554)
Theo Penavaire (ID: 40102474)

This report provides a description of the researches made for the COMP 472 course's first project. During this project, we had to resolve a puzzle, using various algorithms. This puzzle is a variation of the 8-tiles puzzle, with a 4x3 tiles board and diagonal moves allowed. Depth-first search (DFS), Best-first search (BFS) and A* were to be implemented to resolve this puzzle.

In this report we will cover the description of our heuristics, as well as the difficulties we faced and how we addressed them. We will finally conclude with an analysis of our experiments, to challenge our algorithms.

For its simplicity, rapidity and robustness, we chose to use the Python language to code our program.

We used the 2nd definition of the following sentence in the definition of the moves: *"UP means "move the blank tile UP" (and exchange it with the numbered tile right above it)"*

I. Heuristics description

A. H1 – Misplaced tiles

We wanted to have a very basic heuristic to have the opportunity to test and compare it with a more performant one. That is why we chose to first implement a misplaced tiles heuristic. It counts the number of tiles that are not on their correct position on the board.

H1 is therefore defined as follow:

h1 = 1 if the tile is misplaced

h1 = 0 if the tile is correctly placed

B. H2 – Manhattan distance

Our second heuristic is a Manhattan distance. It is the sum of the horizontal and vertical distances between points on a grid, here, the puzzle board.

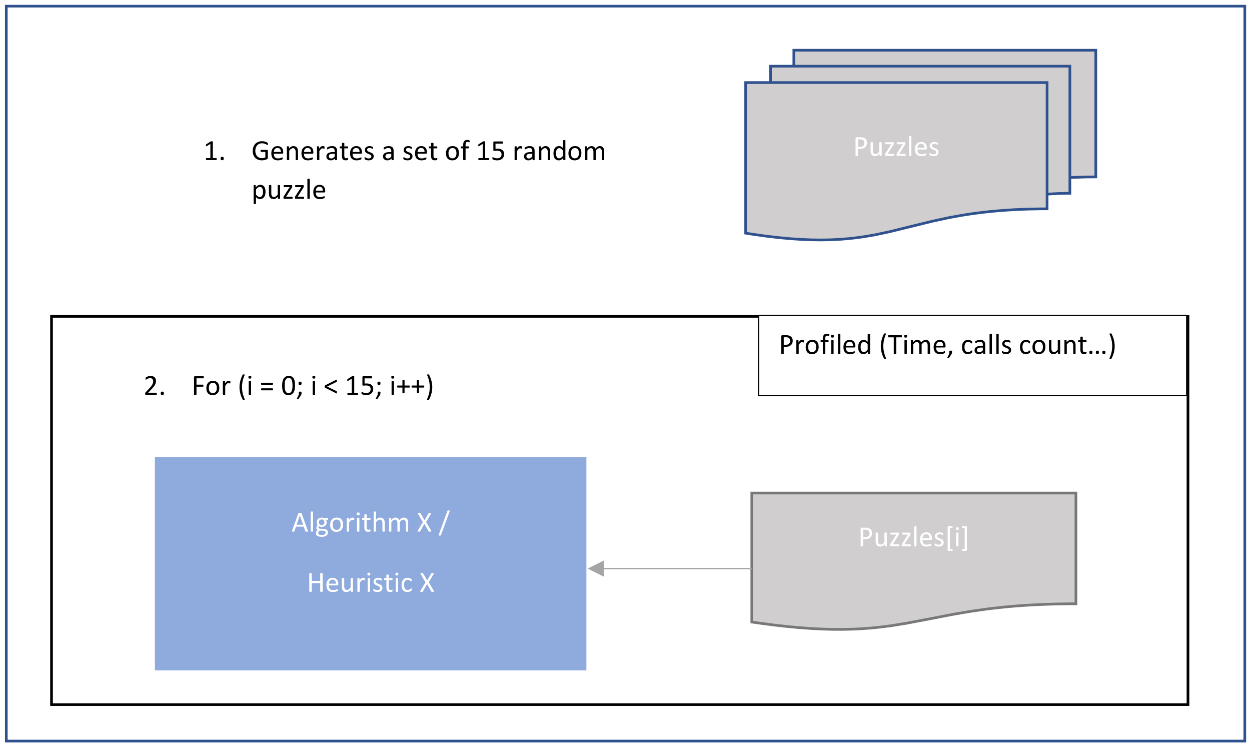
H2 is therefore defined as follow:

h2 = SquareRoot((x – goal_x) ^ 2 + (y – goal_y) ^ 2)

We chose to implement the Manhattan distance because it is an admissible heuristic. It means that it guarantees to find the shortest solution path to the goal, if it exists.

II. Experiments and their results

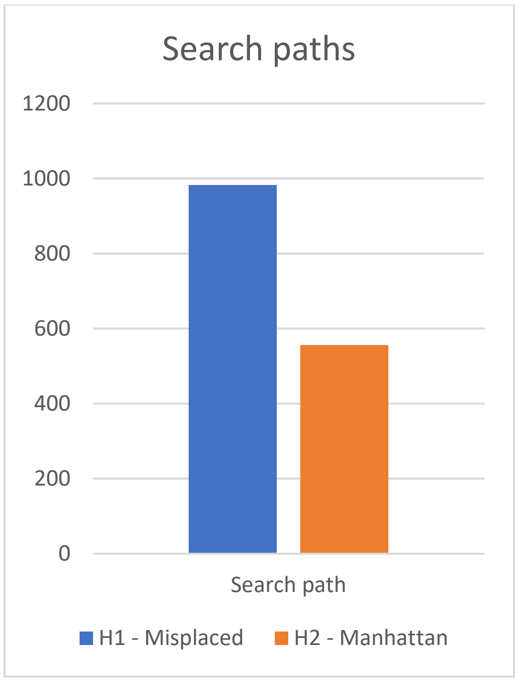
To correctly test our heuristics and algorithms, we first had to think about a "stress system". The responsibility of this system is to generate a random set of puzzles, and to call the different heuristics or algorithms with the same parameters. To be more precise and have more significant results, all heuristics and algorithms calls are made 15 times in a row. Such a system can be represented by the following diagram:

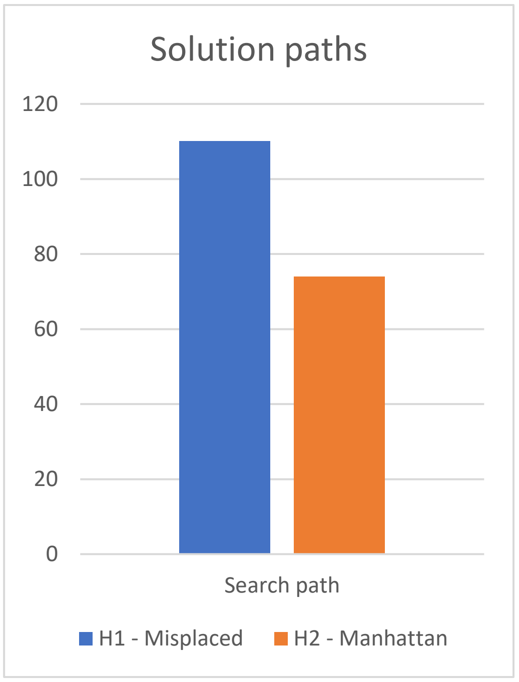

1. Generates a set of 15 random puzzle
Puzzles
Profiled (Time, calls count…)
2. For (i = 0; i < 15; i++)
Algorithm X / Heuristic X
Puzzles[i]

The tests were made on Windows 10 with an i7-8750 processor and 16gb of RAM.

A. Comparison of H1 and H2

To compare our heuristics, we first carefully compared the average length of the search and solution paths associated to them. As said before, the tests were made 15 times in a row and these are the average values on random puzzles. The same puzzles were givent to the heuristics, and the search was performed using a best-first algrithm.





By comparing these results, we can say that H2 is more informed than H1 (it searches a smaller space). Indeed, we figured out that the results given by H2 were bigger than the ones of H1. It is because H2 counts the number of misplaced tiles but also give them a score points, depending on how far they are from their goal position. Therefore H2 is always bigger than H1.

This is also visible through their calling frequence. During our tests, the function handling H1 was called 54 948 times, whereas the one handling H2 was called 35 492 times. H2 is obviously better than H1 in terms of informedness, as it needs to be called less times to solve the puzzle.

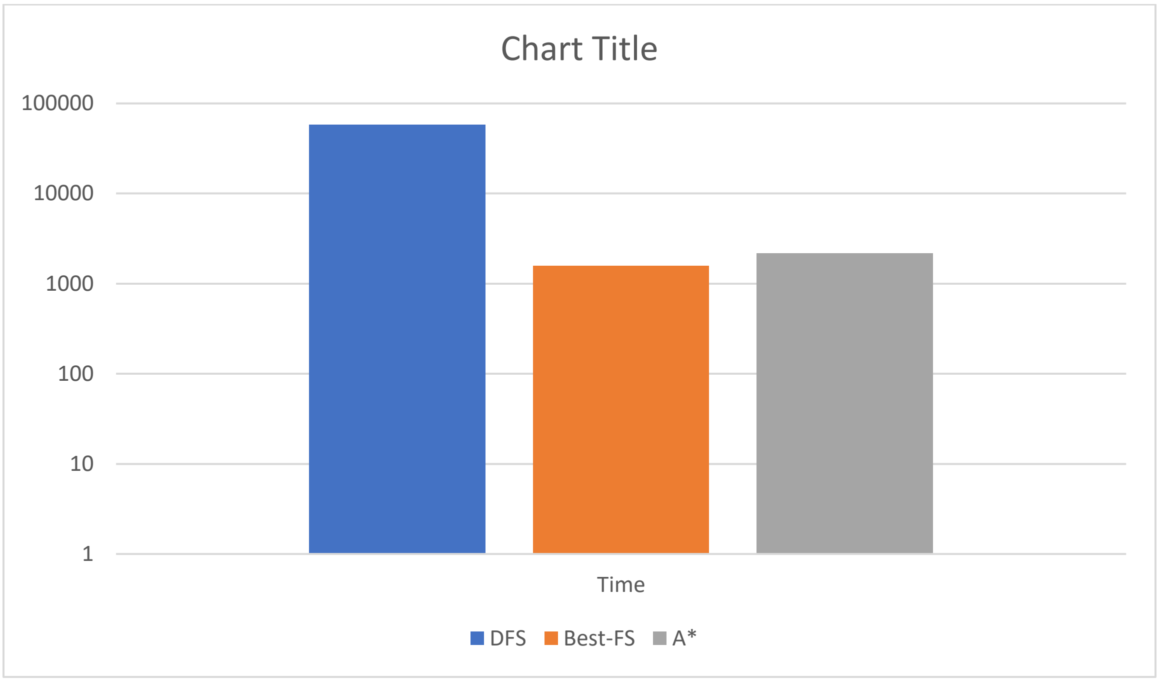
Finally we can say that both heuristics are admissible, as they indicate a number of moves to reach the goal state that will always be smaller or equal to the true cost. The following statement is true for these heuristics, with h(n) being the estimate cost to reach the goal and h*(n) the actual cost.

$$h(n) <= h^*(n)$$

B. Comparison of the different algorithms

1. Time

The first intersting information was the time taken by the different algorithms. Here is a diagram showing it.



The values are scaled with a logarithm algorithm, for an easier reading. We can confirm that the DFS algorithm is the one which takes most of the time (about 57 seconds). It is important to say that we fixed a cutoff to limit the time of the search. This is because the depth-first algorithm takes random branches and explore them until it reaches the cutoff. It has a lot of branches to explore this way.

The second most time consuming algorithm is the A*. This is because it has a lot more computation to do than the BFS, for a single state examination.

When we consider the time factor, the best algorithm is therefore the best-first search (BFS).

2. Search space and solution paths

To compare our algorithms, the second interesting results were the size of their search space and solution paths.

Algorithm	**Search path size**	**Solution path size**
DFS	160 162	154
Best-First Search	1752	115
A*	1233	103

Again, these are the results after having processed the 15 same puzzles for each algorithm (with h1 and h2). We calculated the average value for each comparison point.

As expected, the DFS is the one with the biggest search space size, and also the biggest solution path.

The Best-First search and A* are respectively better. They handle the cost of the path during their search, so they can find the best one.

III. Feedback on the project

A. Python

The first difficulty that we faced during this project was learning to code in Python. We've had very limited experience with it during our studies. We decided to learn Python to be able to implement AI concepts very quickly in the future. It is indeed a "high-level" language, that allows developers to focus more on the concepts instead of their actual implementation.

It was not easy to understand the syntax at first, because we both come from a lower-level knowledge. In particular, we struggled with learning how to pass variables by copy, and not by reference, as it is the default way in Python. The variable types were also difficult to read because they are implicit in Python. However we quickly managed to overcome this issue, as Python was made to be simple and easy to learn.

B. State IDs

The second issue that we faced was a practical one: how to represent a state in the computer memory, and make sure that we never processed the same state twice ? It was a real problem for a few hours, because our algorithms kept looping. Debugging a great number of states was very hard.

So we thought of giving each state a serialized ID of the following form: "bcaefhgijlk". These letters are the alphabetical values of each tile in the state put next to each other. We then stored these IDs as keys in a dictionnary (or map), and compared them with the current state being examined. This way, we ensured that no node was being processed twice.

C. Depth first

Finally we had to think about our depth-first search algorithm a little bit. Its first implementation could not find a solution easily because of the very big depth of the search path. That is why we added a depth cutoff, configurable by adding an argument to the program. We also optimized our methods by profiling the program.

IV. Acknowledgements

General explanations

- AI course slides, presented by Leila Kosseim for the COMP 472 course at Concordia University.

A* Algorithm

- For an explanation of how A* and an implementation: https://www.gamedev.net/articles/programming/artificial-intelligence/a-pathfinding-for-beginners-r2003
- For A* pseudo-code: https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7

Best- first search

- For Best-first search pseudo-code: https://www.researchgate.net/figure/Pseudocode-for-Best-First-Search-algorithm_fig1_315347498