AI Project 1 : Puzzle solver

Yann Kerichard

15th October 2018

1 Introduction

During this first project, I was asked to implement a solution to solve an 11d-puzzle using different algorithms (uninformed and informed) combined with two heuristics of my choice. However, this puzzle also had to be able to handle diagonal moves (so a tile could have 8 possible moves instead of 4). I also had to demonstrate some creativity making different experimentations. The final goal was to compare and analyze the different way to solve the puzzle.

I decided to work with the definition 2 that was suggested (we move the blank tile). So, the program will generate the moves in a clockwise direction, starting from UP. That means that we will try to move the blank tile UP, then UP-RIGHT, then RIGHT, etc...

This report tries to bring an answer to the following questions: How to solve an 11d-puzzle? What are some of the different solutions that can be applicable to a such problem? And how does each of the solutions I used to solve the puzzle compete between each other (Which one is the best)?

To bring an answer to those questions, I will detail exactly what I realised, so you can get an overview of my whole work. Then, you will find a section focussing on the 2 heuristics that I chose. They will be described in detail, so you will be able to understand how they work, and the way they help to solve the puzzle. In a third part, I will let you know the difficulties I encountered and how I addressed them. Finally, the last section will be dedicated to the analysis of the different solving methods.

2 Realization

2.1 Description of the project

My solution is a Java project composed of 3 packages: grids, heuristics and utilities. The first package, grids contains all the algorithms that I implement. There are 3 algorithms: Depth-First Search, Best-First Search and A star. However, because the computing time of the Depth-First Search was way too long, I also added an option allowing to set a cut-off and to execute the Depth-Limited Search. In that way, the DFS algorithm was able to solve more grids. All the algorithms in the program (named by grid) inherits from the methods of a parent class Grid that contains a lot of very useful methods.

The second package, heuristics stores the different heuristics I used. You will be able to find 5 heuristics here. Each heuristic will be detailed in next section.

Finally, the third package contains all the utility functions such as the module allowing to generate text files and the class that start the program.

When I finished implemented all those algorithms, I was able to run and solve a puzzle using 12 different methods: 2 for the two version of DFS (with and without cut-off), 5 for BFS (one with each heuristic), and 5 else for A star (also with the 5 heuristics). For each of those 12 executions, my solution is displaying the elapsed time and the number of moves of the solution found. So, by curiosity and also because I had to provide some analysis for this report, I created an Excel sheet (it is included in the project in the documents folder) where I realised some benchmarks. It allowed me to draw conclusions about the efficiency of each algorithm and heuristic. The details will be given later in the report.

2.2 About creativity

During the project, I tried to be creative and to test new things. This is the reason why I decided to implement 5 heuristics instead of the 2 asked. In fact, I implemented the 3 heuristics seen in the class because I wanted to test their performance and I was curious to see how they would compete between each other, but also with the 2 other custom heuristics that I created. Also, because the DFS algorithm was not usable at all, I decided to find a solution to handle this problem, and I implemented an option to set a cut-off. Finally, in order to be able to draw more accurate solutions I wanted to have real data, so I created a system to get the execution time and the number of moves found by each of the algorithms.

3 Heuristics

The project contains the 5 following heuristics that are executed by the 2 informed algorithms (BFS and A star):

- **Wrong neighbours**: By looking for a new heuristic, I found out that on a website, [1] someone was solving the puzzle by counting the number of good neighbours of each tile. By summing the neighbours of each tile, we obtain the value of the grid. However, with this version, the biggest heuristic value was the best, and my program was considering the lowest value to be the best. So, I modified it a bit and the heuristic count the number of wrong neighbours for each tile and sum it. In the heuristic, I just count the direct neighbours (only up, right, down and left, not diagonals). I thought it would be interesting to see how it would compete with other heuristics, and if this information was enough to be able to find a solution to the puzzle.

- **Manhattan distance with diagonals**: In the original Manhattan distance algorithm, we sum the distance of each tile to its final position, but we only consider 4 types of moves. However, because in our puzzle we could also move tiles in diagonals, I was very curious to see what would happen if we also applicate this change to the Manhattan distance. In my opinion, it made sense because it was a bit like counting for each tile, the minimum number of moves necessary to place the tile to the good position.

- **Manhattan distance**: As seen in class, this heuristic sums the distance of each tile to its final position. I added it in order to compare it with the Manhattan custom heuristic (see above) and with the other seen in the class.

- **Hamming distance**: This heuristic was also seen in class. The principle is to count the number of misplaced tiles.

- **Permutation inversions**: Seen in class. For each numbered tile, sum of how many tiles on its right should be on its left.

4 Difficulties

To be honest, I did not face a lot of difficulties implementing the algorithms and heuristics, but I found that my code was not well organised because I did not take into account that I would implement up to 5 heuristics and I had to refactor my code so that it is easy to plug in new algorithms and heuristics. When my program was finished, I though I got a bug with the cut-off on DFS because sometimes, the algorithm was not finding any solutions, but with a lower cut-off, a solution was founded. But finally, I realised that it was an expected behaviour and I will discuss about it in the next section. Also, this is probably not a difficulty, but I was really surprised that the original DFS takes so much time to solve some grids. I really thought that it would not take more than something like 1 hour, so when I implemented the DFS, I though that I still had a bug, but it was definitely something expected because the number of combinations is way too large.

5 Analysis

In order to explain clearly my analysis for each of the features I implemented, I will first give an analysis of the heuristics, then I will compare the algorithms. After that, I will discuss about which one of the combinations Heuristic/Algorithm is the best. Concerning the process of benchmarks, I took many different grids, that I classified among difficulties (easy, medium, hard and insane). I evaluated the difficulty just by looking at the grid and if it was almost solved, I said it was easy for example. I generated many random grids, trying to get a balance between the difficulties, and I calculated average time and moves for each heuristic and for BFS and A* algorithm. It was not relevant for DFS because DFS is not able to solve almost solved grids.

5.1 Heuristics analysis

In term of time, the 2 versions of Manhattan distance and hamming distance seems to be very close and to be the fastest heuristics. We notice a little advantage for my custom version of Manhattan distance. Permutation inversions is the one that require the most time and my second custom heuristic (wrong neighbours) is pretty good. I would say that the Manhattan heuristic and Hamming distance are particularly fast because it provides more accurate results the grid than the others. Those 3 heuristics are directly evaluating each tile comparing with their real final position, whereas in the 2 others, the result of each tile is based on the position of other tiles, no matter if they are at the good position or not. As a consequence, Manhattan and Hamming tend to provide more accurate indications that lead the software faster on the good path.

However, in terms of number of moves, Hamming distance is a bit less efficient and both Manhattan versions still stand at the top. Permutation inversions seems to stay viable

though. But, even though my custom heuristic on the wrong neighbours was able to compete in term of time, concerning the number of moves, it is completely outperformed. It seems logic that Manhattan outperform all the other on this point. The secret is that Manhattan distance directly evaluates the distance of each tile to its final position. So, by constantly taking the lowest remaining distance to the goal for the sum of the tiles, it seems logical that the goal is encountered earlier: that means a shorter path. The custom Manhattan is a bit better than the original one both in term of time and moves because it also considers more possibilities, then it can have more choice to select a better path: the time spent for each mode may be a bit longer, but the path is still a bit shorter, so finally, it is worth. Permutations, Hamming and more especially the Neighbour heuristic are behind because they do not have any notion of distance or number of moves from their position to the goal. As an example, in the case of the neighbours, we only consider the tiles around a current tile. But we have no idea if our current tile is at the good or wrong position. So, in the case where the current tile is at the wrong position, but neighbours will be good, it can be counter-productive because, the value will be lower, but the tiles around will also be at the wrong position. However, I noticed that sometime, my custom Manhattan heuristic can be a bit behind original Manhattan because more grids can have equals value due to the fact it is a bit less informed than original Manhattan (value of h is lower due to the diagonal moves). In a such case, the custom Manhattan does not really know which grid it should favorize.

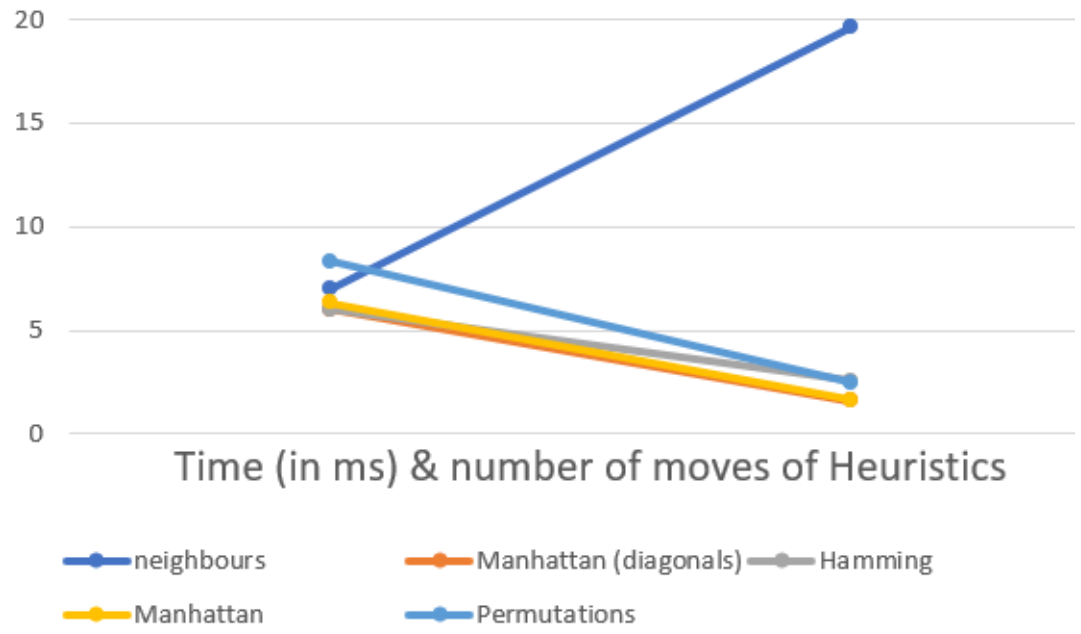

Figure 1: Comparing heuristics in execution time and in number of moves

5.2 Algorithms analysis

To talk about the DFS algorithm, I found out that the algorithm is only able to solve only certain types of grids, which are almost finished and where there is not a lot of solutions. This algorithm is very inefficient to solve our puzzle. I tried to execute the DFS algorithm on many types of grids, but when the solution is not evident, it never ends. I even tried to let DFS run during one whole night (almost 12 hours), and it still did not finish to compute. DFS is not working because we have a branching factor of 8 (8 successors for each

node). Then, when DFS is executed, it tries to go the deeper it can through one of those branches, but it never comes back to test another branch: there are just too much possible combinations to test. The more it tries to verify a combination, the more it adds new child. So finally, when the DFS starts and choose the wrong way (for example if it tries up first, but if the first solution child should be down), it will always try to go deeper and to look for other combinations on this road. So, DFS will take many hours/days to realize its first move was probably wrong. So, it will take too much time to come back to the first nodes, and so, other roads will be explored very late. In order to solver this, I implemented a cut-off. This works pretty well when cut-off is lower than 10 (so only for easy grids), because after that, the branching factor is too high, and we come back to the problem of DFS. However, there is also another problem with that. Sometimes, we just do not find any solution, whereas there is a solution. For example, let say I take a cut-off of 6, and there is a goal state is at depth 5. It may happen that one of the nodes leading to the solution path be explored before, let say at depth 6 for example. Then it is added in the closed list. That means that when we will reach this node later, that leads to the solution path, it will not be inspected. So, the solution path will not be discovered!

However, DFS can not compete with BFS and A star both in time and number of moves. From my benchmarks, I found that BFS is almost always faster than A star, but the number of moves is generally higher. It is easily explainable. Since BFS will always take the grid with the lowest heuristic, and because heuristics are made to get closer to the solution, it means that BFS will always look for a next grid that brings him as close as possible to the solution, no matter the longer of the solution path. With this behaviour, BFS clearly aims to get a solution as fastest as possible. However, in the case of A star, it reproduces the same behaviour as BFS, but instead of only taking into account the remaining estimated cost to get to the solution (the heuristic), it also takes into account the distance travelled from the root node. So, with such a behaviour, the algorithm, will not only be concerned about getting the solution as fastest as possible, but it will also be concerned about getting the shortest path. By mixing up those 2 factors, we lose few milliseconds, but we generally earn a lot of moves in the final solution. Isnt that worth it? Well, not always! I found out that in some cases, A star struggles. This happens when we give a grid that is very complex (hard or insane difficulty) to the algorithm. Whereas BFS is still only concerned about getting the solution as fastest as possible and returns a result very quickly, A star struggles and can takes a lot of time. This fact is due to 2 things. The first thing is that A star computes a bit more calculation for each node comparing with BFS. It calculates the heuristic for the grid such as BFS, but also has to go through all the ancestors to calculate the cost of the actual path (distance from the root). So, for the grids that have a long solution path, the time spent by A star to evaluate this cost rockets. The second thing is related to the first one. When the solution path is long, the last good nodes of the solution path will be checked after all the bad first nodes, because the evaluation function is the sum of the heuristic and the actual distance from the root. So, first bad nodes will have big heuristic and low distance, and good last nodes of the solution path will have low heuristic and high distance. However, the more the solution path will be long, the deeper we will go through the nodes, the more the value of the actual distance will be important comparing with the heuristic cost. So, this is why bad first nodes will be tested before good last nodes of the solution path. To resume, whereas BFS is the fastest and more stable, A star gets the optimal solution path.

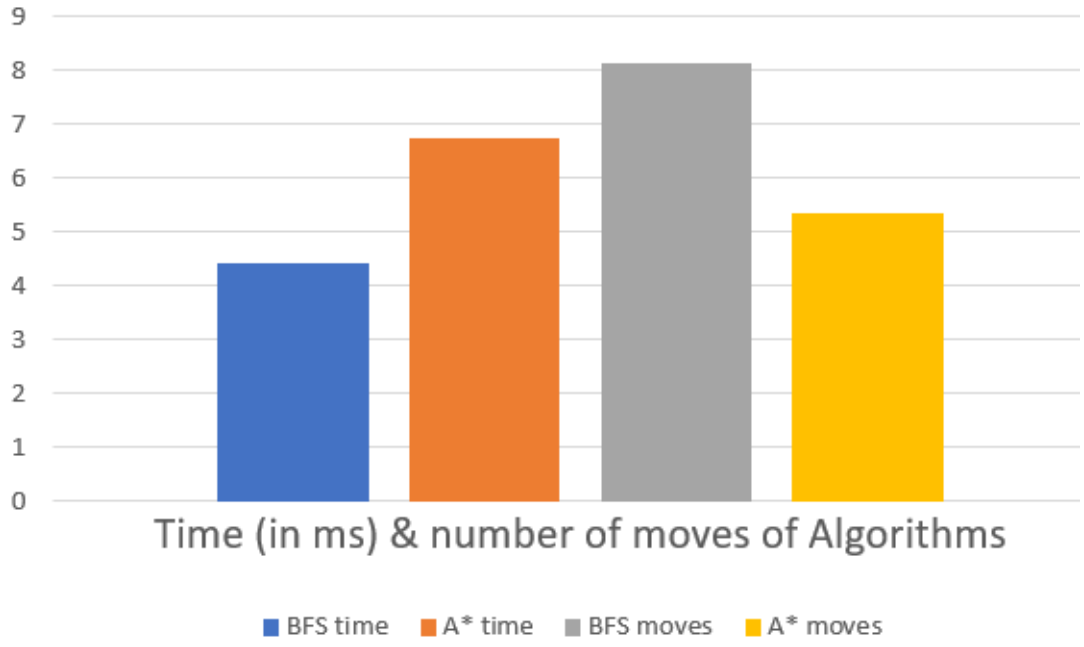

Figure 2: Comparing BFS and A* in execution time and in number of moves

5.3 So, what is the best option?

Finally, here we are! So, whereas regarding heuristics, we dont have any doubt that Manhattan and custom Manhattan outperform all the other in almost any case, its not the case for algorithms. I would say that if we have an easy or medium difficulty grid (if the tiles are not too mixed up), we should go with the A star and custom Manhattan. However, for very hard or insane difficulty of grids, A star will struggle, so we need to use a BFS and custom Manhattan!

References

[1] Pierre-Edouard Portier. Heuristiques pour la recherche dans un espace d'tats, 2016.