University of Liege

Pacman - Search agent

Introduction to artificial intelligence

Maxime Meurisse (20161278)
Valentin Vermeylen (20162864)

3rd year of Bachelor Civil Engineer
Academic year 2018-2019

1 Formalization of the problem

1.1 Formalization as a search problem

First, we define the agent. In this problem, the Pacman player will be a search agent, implemented as a `PacmanAgent` class.

Second, we define the state of the agent. This state consists of a series of information, including Pacman's position in the labyrinth and the boolean state of all food dots.

A search problem consists of an initial state of the agent, a description of the possible actions for an agent state, a transition model that returns the new state of the agent after execution of an action, a goal test and a path cost.

Let us specify these components in the case of our problem :

- initial state of the agent : the starting position of Pacman in the grid (defined by the layout, i.e. the game grid with food dots) and a boolean matrix of food dots ;
- possible actions for a given state : Pacman can move in 4 directions : North, South, West and East or may not move. If a move leads Pacman into a wall, it is said to be an illegal move, otherwise it is a legal move ;
- transition model : thanks to one of the four previous actions, Pacman updates its position in the labyrinth and possibly updates the boolean state of a food dot. The new state of Pacman is a successor of his previous state if Pacman does a legal move ;
- goal test : check if all the food dots have been eaten, i.e. if all the food dots matrix is composed of false boolean value.
- path cost : each single movement has the same cost but a sequence of movements can lead to different scores. The path cost is defined by Pacman's score.

2 Performance of algorithms

The algorithms were run [1] on the different layouts to study and compare their performance. All reported data is an average of 10 similar data.

1. The algorithms were run on a machine equipped with an 3.1 GHz Intel Core i7 processor with 16 GB of RAM.

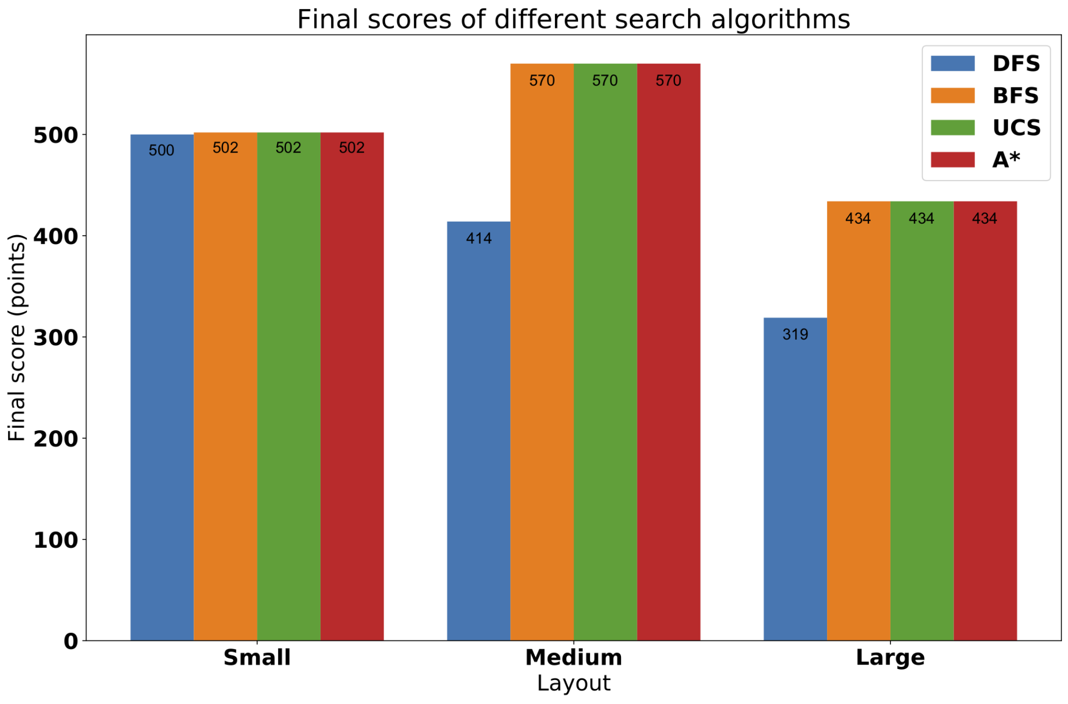

Figure 1 – Comparison of the final scores of the algorithms on the different layouts.

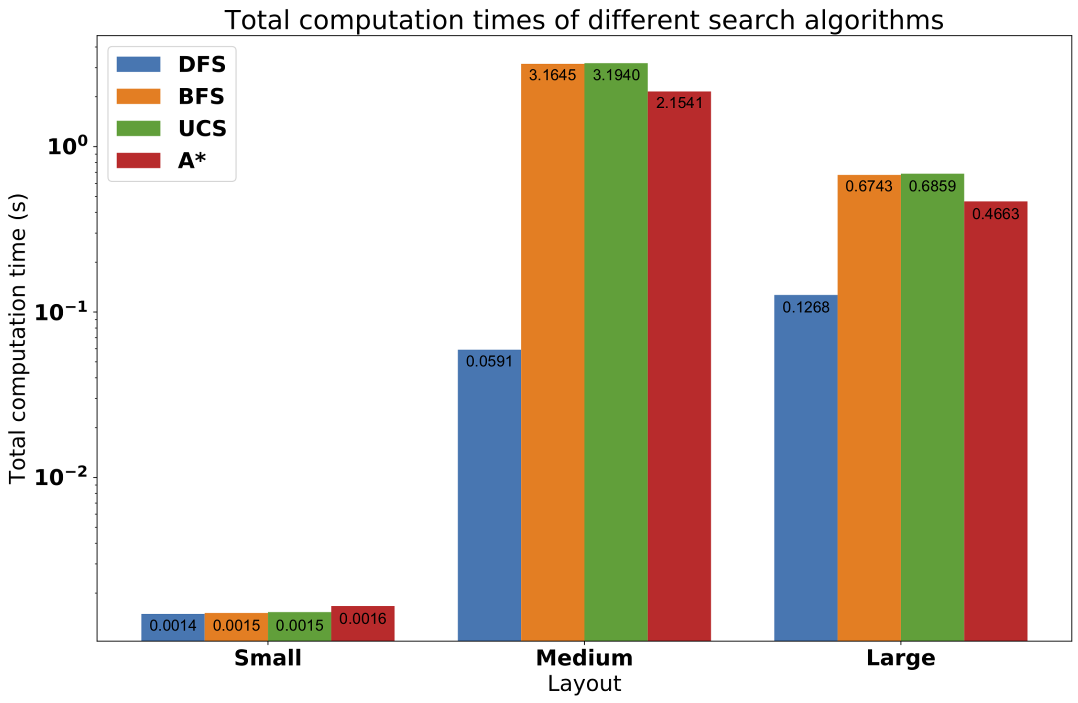

Figure 2 – Comparison of the total computation time of the algorithms on the different layouts.

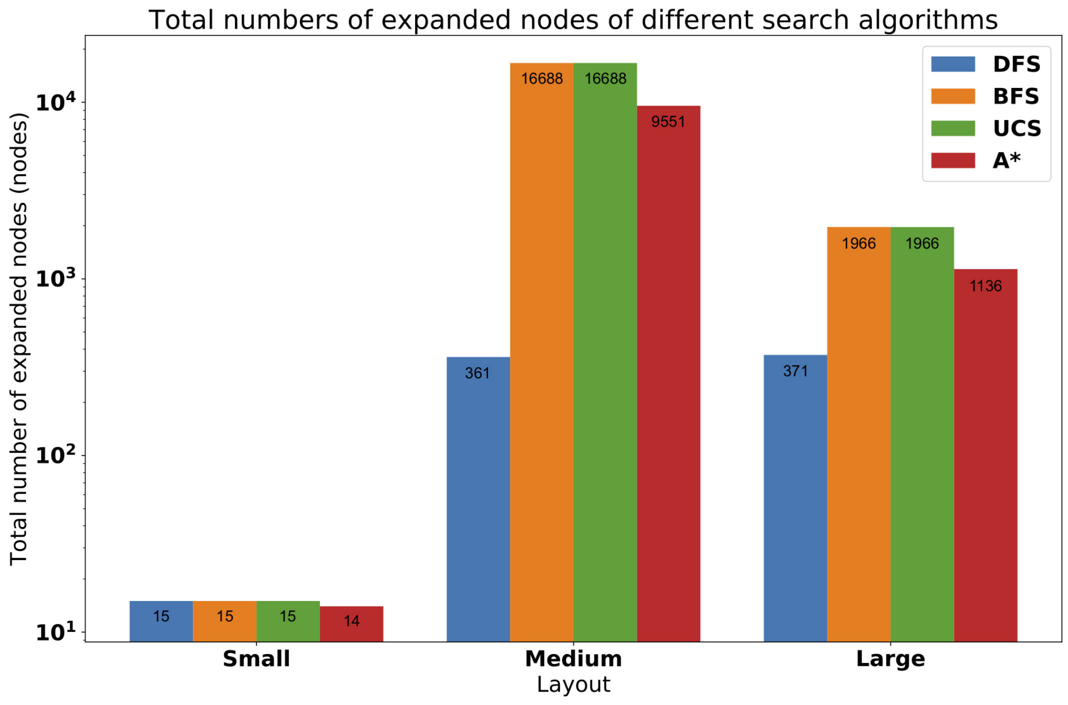

Figure 3 – Comparison of the total number of expanded nodes of the algorithms on the different layouts.

3 Discussion

As a general observation for all agents, the medium layout requires the most computation time, and that is due to the fact that there are a lot of food dots to eat, so the solutions are less direct, and Pacman needs to explore almost all cells, and sometimes more than once.

The large layout, on the other hand, only have 4 dots. So, while it has larger dimensions than the medium one, the computation time is smaller because the path to the solution is more direct (less convoluted).

3.1 Depth-first search

As can be seen in figure 1, `DFS` is not optimal in any of the layout on which it has been tested. We can easily understand why : `DFS` goes as deep as it can in the state tree and, therefore, finds the leftmost deepest solution, regardless of its optimality. And, as each branch corresponds to an action taken in the maze, a deep solution corresponds to a long sequence of actions. Thus, the score decreasing with the number of actions of the sequence, a deep solution means a non-optimal score.

However, as it concentrates its search on the left part of the tree, it finds a solution quite quickly, thanks to the fact that an infinite number of solutions exist in this project (so we can be assured that `DFS` will find one in the left part of the search tree). That provides us with an algorithm that is significantly faster, computationally-wise, than all the other ones, but that provides a worst solution, and these two aspects are amplified as the layout gets larger.

The number of expanded nodes being directly related to the computation time, the analysis conducted just above holds for that aspect too, and is not mentioned for the other agents.

3.2 Breadth-first search and Uniform cost search

As the results show, `UCS` and `BFS` provide an optimal solution (best possible score) but with a large number of expanded nodes, and more slowly than the other algorithms, and this for all layouts.

They have the same results everywhere, and this comes from the way we implemented the priority. Indeed, we used the length of the list of actions leading to a state as priority, which means that `UCS` behaves exactly as `BFS`, exploring the shallowest nodes first.

We initially used the opposite of the cost of a state as priority, but we realized that it was not optimal. The score could change its sign and so the priority did not truly reflect the best path possible (the fact that eating a dot only gave 10 point and finishing the game gave several hundreds accounted for that too). However, the number of nodes expanded and the computation time were decreased.

3.3 A* (and his associated heuristic)

We chose as heuristic for `A*` the sum of the length of the sequence of actions leading to a state and the Manhattan distance to the farthest food dot. As we have seen in the theoretical course, the Manhattan distance is an admissible heuristics. That means that the score obtained via `A*` is optimal, and thus are the ones obtained via `UCS` (and `BFS` consequently), as can be seen on the graphs.

Regarding the computation time (and the number of nodes expanded), `A*` is better than `BFS` and `UCS`, but not as effective as `DFS`, and this for all layouts.

We can see that it is the algorithm that takes the less computation time amongst those that provide the best score.

An observation we made is that, if we take the Manhattan distance to the nearest node instead of the farthest, the computation time and number of expanded nodes is multiplied by 6 on the medium layout, but the score is still optimal. That can be explained by the fact that a lot of states will then have the same priority, as the distance to the nearest food is much more likely to be equal to that of another state than in the case of the farthest dot. The search agent will be less well-guided (the contour of the search will look more like a circle than an elongated ellipse), and so have poorer performance. However, as the heuristic is still admissible, the score will be optimal.

3.4 Conclusion

Based on the previous discussions, we can conclude that the algorithm `A*`, in general, provides the optimal solution (in terms of score, nodes expanded and total computation time).

The `DFS` algorithm is the fastest and provides an almost optimal solution for small layouts (in any case with a small number of food dots to eat). It could be used in these cases, because the solution would be acceptable and the computation time reduced, but would quickly become bad for other layouts.

The `UCS` and `BFS` algorithms provide, in our case, an optimal score, but with a long computation time and many expanded nodes. So, their use is limited : they quickly become unusable on very large layout. We will therefore prefer the algorithm `A*`.