INFO8006 : Project - Part 1

Roumache Grégoire - 20162533 and Stoquart Colin - 20161140

October 26, 2018

1 Formulation of the Pacman Game as a Search Problem

A search problem consists of a state space of the problem, a goal test and possibly a path cost. In all cases, we define the state space as follow :

- The state of an agent is defined by its position (x, y) on the map, if whether or not it remains dots and its score. If an agent uses a path cost (like UCS and A*), there is also the score that we have to take into account. The distance between Pacman and the remaining dots is important too in order to define our heuristic for A*.
- The set of actions Pacman is allowed to perform consists in going in the directions that are not obstructed by a wall. When Pacman does this action, he reaches a new state. This set is always a subset of $\{North, South, East, West, Stop\}$.
- For a given action (i.e. a direction), the transition model consists of updating the position of Pacman, if it remains dots or not, its score and possibly the distance between dots and our agent. This gives us all the characteristics of the resulting state.

We define our goal test as a function that returns true if pacman has eaten all dots, false if it hasn't. We discuss in section 3 about other function that we could have chosen.
For A* and UCS, we have to define a path cost ($g(n)$). We decide to define it as follow : for any action, if pacman increase its score by performing this action, the cost is null. In the other case, we increment the cost by one.
Finally, we have to define an heuristic function ($h(n)$) for A*. We have chosen the Manhattan distance between the pacman and the farthest dot. It verifies :

$$h(n) \leq g(n)$$

so the heuristic is admissible. We take the farthest dot in order to our heuristic to be as influential as possible

2 Results

In figure 1, 2 and 3, we present the results of our algorithms depending the maze.

3 Discussion

As we start looking at the graphs, we immediately notice that the BFS, UCS and A* algorithms *always* get the best possible score while DFS doesn't. But there's a good reason for that. As its name suggests, DFS privileges depth and doesn't always find the optimal path while the other three always do. DFS is not optimal. This also explain why it doesn't take as much computation time and why it doesn't expand as many nodes as the other path-finding algorithms.
Conversely, as the theory predicted, UCS and A* are optimal. BFS seems to be optimal too. Actually, it's because it finds the minimal distance between two positions on the map. Regarding the score function of the game, we see that every action has the same cost except if there's a dot. Since, in the end, all dots are eaten the only information that matters is that every action has the same cost. This is why BFS is optimal.

When it comes to the number of expanded nodes and the total computation time needed, the worst algorithm is BFS (although it always get a perfect score as explained earlier). Indeed, since BFS works as LIFO queue, pacman has to explore every path of the the same depth. UCS usually expands less nodes and takes less time than BFS. These two facts are of course linked. Note that if we had defined an other path cost, the result would have been different. For example, our first path cost was the number of action that pacman has to execute.

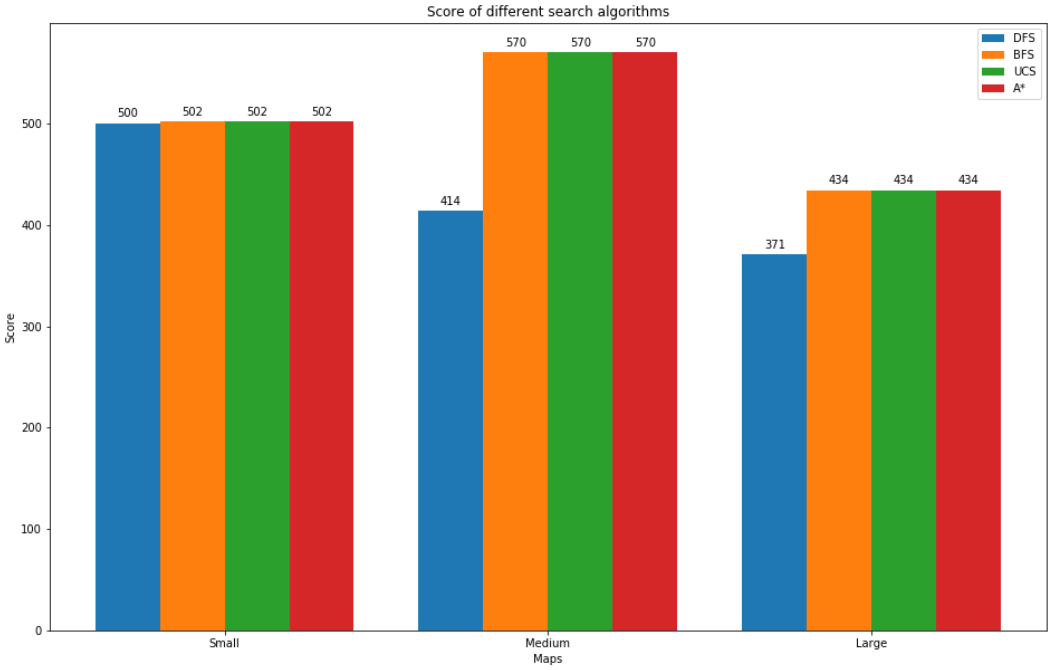

Figure 1: Score of different algorithm for different maze

This result in a LIFO queue and is very similar to the BFS algorithm. The way we have define our path cost in section 1 is more efficient (because it favours the path with dots to eat) as we can clearly see in figure 2 and 3. We can also notice that A* gets the best possible score while expanding a small number of nodes in a small amount of time. Indeed, A* is superior to UCS because it uses (admissible) heuristic which enhances its speed. Once again, the better the heuristic, the faster A* will be.

Finally, as we could have deduced it from the figures, the computation time and the number of expanded nodes depend on the number of food dots. A map may be large, if it doesn't contain as many food dots as a medium one, then the computation time and the number of expanded nodes could be higher in the medium maze, not in the large one. In fact, when there's a large number of dots it's not easy to attribute priority to each state. Indeed, it's difficult to make the difference between all the potential paths because they all have a high level of priority, seeing that they've all eaten some dots.
It seem that the major default of our algorithm is the computation time. To speed A*, we have to find better heuristic or a combination of several heuristic. However, it's not possible for BFS. In this case, we could imagine to change the goal test. If we define a goal test as "*true if we eat one dot*", this will significantly increase the speed. However, the algorithm is not optimal anymore. Let's consider figure 4. If Pacman goes westward first instead of immediately going to the east, the final score is two points higher. But if it we try to speed up BFS, it goes to the right because the food dot in (8,1) is closer. This is why we don't have used this goal test, we prefer optimally than speed.

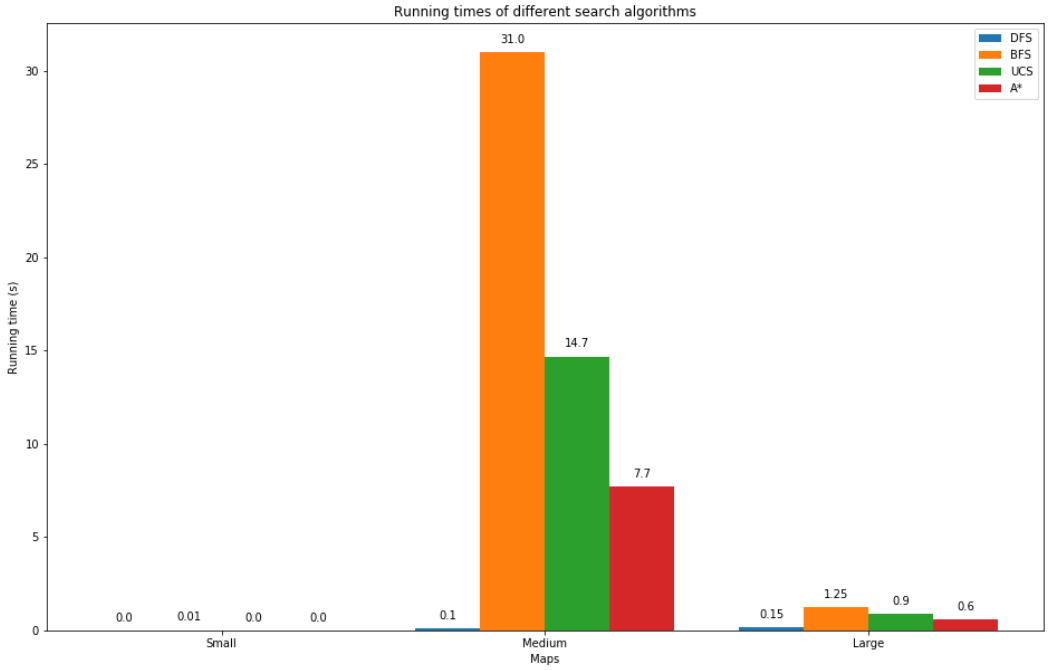

Figure 2: Computation time of different algorithm for different maze

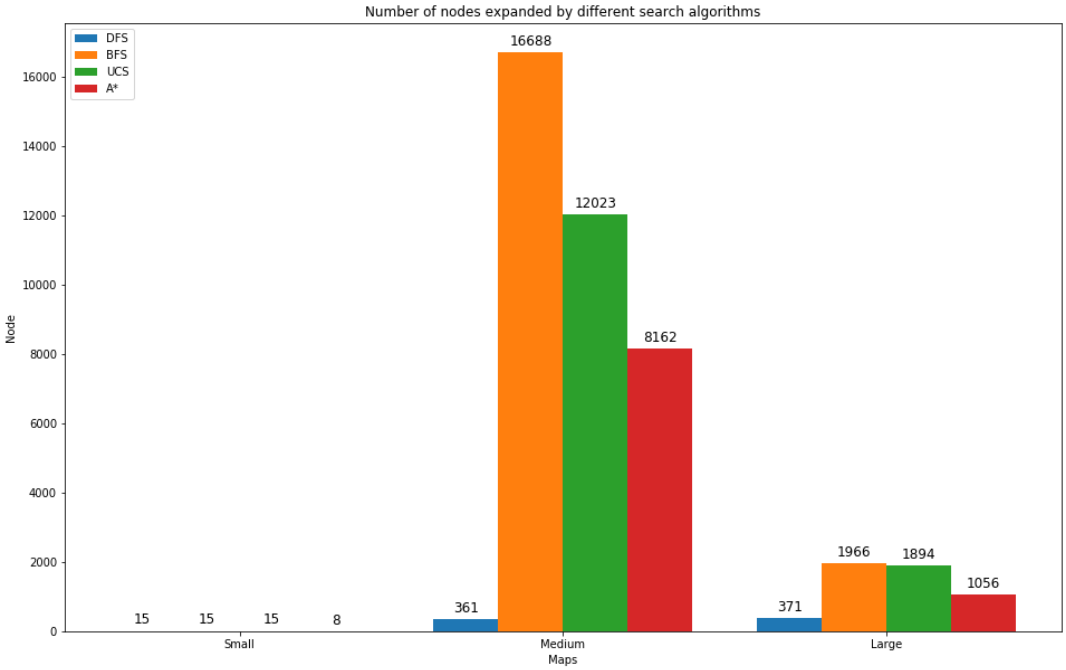

Figure 3: Number of node expanded for different algorithm for different maze

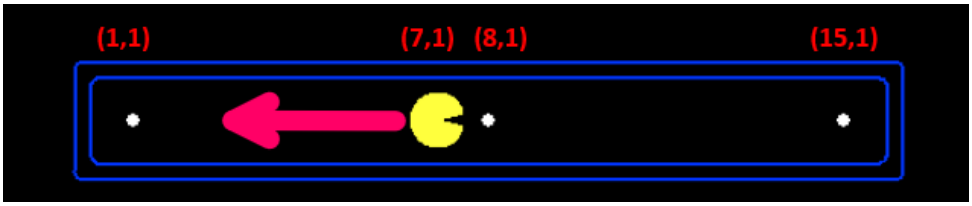

Figure 4: Optimal Pacman goes westward first while sub-optimal Pacman goes eastward.