INFO8006: Project 1 - Report

Mukolonga Jean-David - 20170679
Siboyabasore Cédric - 20175202

October 13, 2019

1 Problem statement

a. For each state we take account of the location *(x,y)* from a x-y axis placed at the lower left corner (x and y being respectively measured horizontally and vertically) and a boolean grid that indicates where there's food ($True$) and where there's not ($False$). For each layout, the agent has a different initial state :
-the agent starts at $(5, 5)$ for the *small layout*
-the agent starts at $(9, 3)$ for the *medium layout*
-the agent starts at $(5, 1)$ for the *large layout*
The agent can either go north, south, east or west.

The location must be updated when the Pacman agent moves and the boolean grid must be updated only when the agent eats.

The Pacman agent has achieved its goal if all the boolean variables of the boolean grid are false, meaning that it's eaten everything.

The step cost has a value of either 1 if the Pacman agent doesn't eat a dot or 2 if it does.

2 Implementation

a. As said in the previous question, a Pacman game state is uniquely identified by its coordinates (x,y) and by a boolean grid that indicates where there's food or not. The *key* function defined in ***dfs.py*** only returns the position associated to a Pacman state and not the corresponding boolean grid. When the Pacman agent eats food at a certain state $(x, y, True)$, the boolean at this position (x, y) in the boolean grid is updated to *False* and the key associated to this state (x, y) is added to the *closed* set without taking account of the boolean. It means that the Pacman agent won't go to that position again even though the state is different since the boolean has been set to *False*. This is a major problem if there are multiple dots and the Pacman agent needs to go to a position where a dot was previously eaten.

Therefore, the *key* function also needs to return the boolean associated to a Pacman state thanks to the *getFood* function defined in ***pacman.py***

b. We implemented the A* algorithm in a *astar* method using the heuristic function h in the **astar.py** file. The cost function is implemented inside the *astar* method.

c. We store the cost along with the state and the path in a fringe represented by a priority queue that uses $f = g + h$ as priority.
The cost function g represents the sum of the step costs that led from the initial state to the current state. The cost value for the initial state is therefore 0.
If the Pacman agent goes to a state where there's food, the step cost value is 1 and is added to the cost value for that state. Otherwise, the step cost value is 2.
We chose a heuristic h that computes the maximum Manhattan distance between the current Pacman position and the remaining dots. This way, the state with the lowest maximum Manhattan distance will be advantaged since f will be smaller.
The optimality of A* is guaranteed because our heuristic h is admissible since it always returns a lesser number than the actual cost of the traveling Pacman agent.

d. The cost function g preserves the completeness of A* because the step costs are strictly positive : its value is either 2 if the next state the Pacman agent goes to contains food or 1 if it doesn't. It also preserves the optimality of A* because it expands nodes in order of their optimal path cost.

e. We implemented the Breadth-First Search (BFS) algorithm in a bfs method in the **bfs.py** file.

f. If the g and h functions return the same number regardless of the considered state, every element of the fringe will have the same cost and same priority f. The state that the Pacman agent will therefore be the first element of the fringe which respects the FIFO aspect of the fringe.

3 Experiment 1

a. See $Figure1$ and $Figure2$

b. Both versions of A* have the same score (570) but when h(n) = 0, more nodes were expanded (14611 nodes) than when h(n) is not equal to 0 (10455 nodes).

c. These differences are due to the choice of h (n). Indeed, A* is optimized if h (n) is admissible i.e $h(n) < h^*(n)$ where h*(n) is the true forward cost associated with the problem. The closer h(n) is to h*(n) the more efficient the algorithm will be.

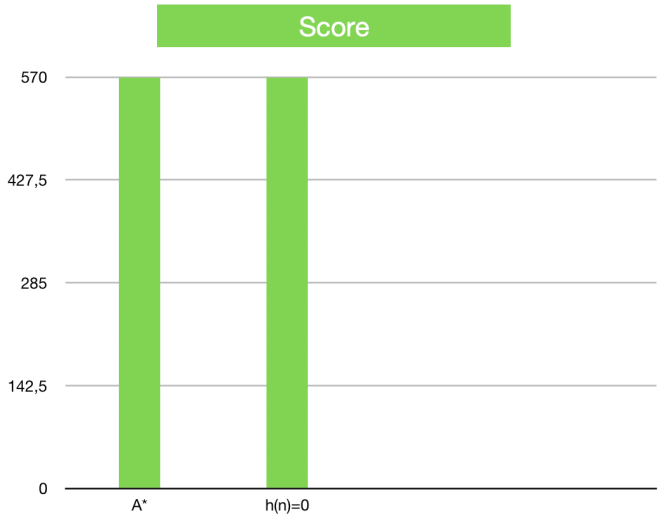

Figure 1:

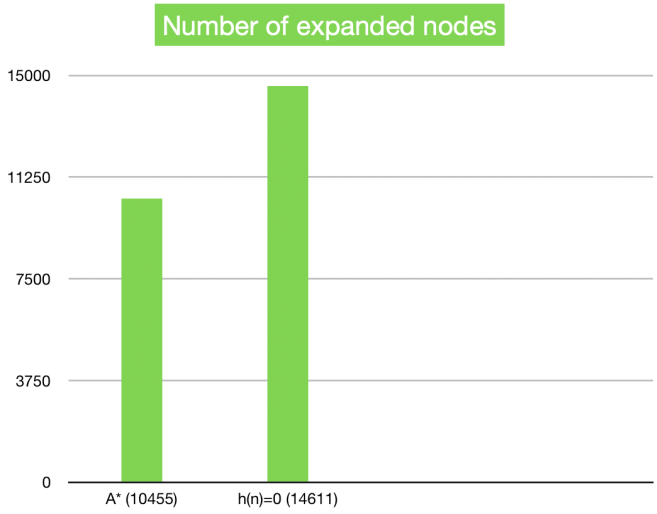

Figure 2:

d. The Uniform-cost search algorithm corresponds to A* when h(n) = 0 for all n.

4 Experiment 2

a. See *Figure*3 and *Figure*4

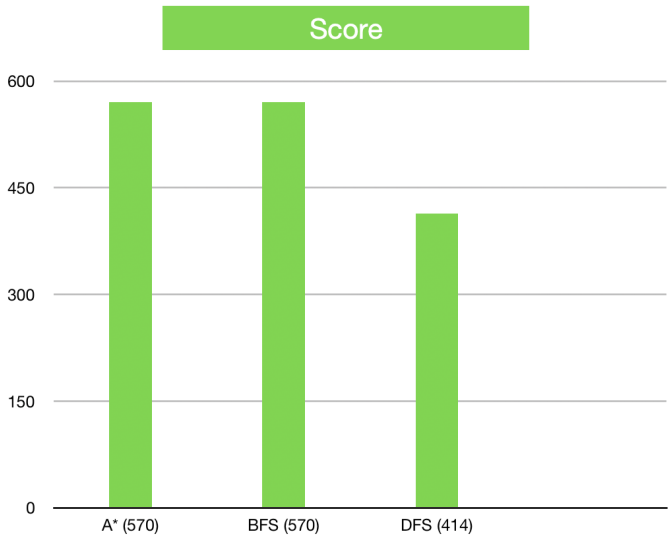

Figure 3:

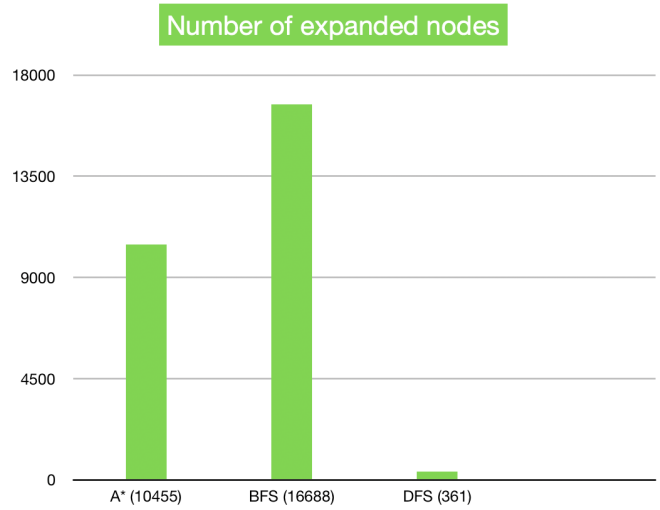

Figure 4:

b. The number of expanded nodes (10455) and the score (570) obtained with A* are respectively higher than the number of expanded nodes (361) and score (414) obtained with DFS.

c. DFS is not optimal because it finds the leftmost solution regardless of the cost that led to it. Since it stops as soon as it's found a solution,

the number of expanded nodes is smaller than the one with A* because A* finds the least-cost solution. The score of DFS is smaller because the solution is not optimal.

A* and BFS find the optimal solution (570) but BFS does so by generating all the shallower nodes. It expands much more nodes than A* because the goal node is not necessarily among the shallower nodes BFS has expanded.