INFO8006: Project 1 - Report

BARE Alexandre - s172388
RANSY Bastien - s174413

October 13, 2019

1 Problem statement

a. The Pacman game can be formalized as a search problem :

- The set of possible states contains all possible coordinates for the position of Pacman and whether there is (still) a food dot at this position or not : *s = ((x, y), food boolean)*
- The initial state is the position of Pacman at the beginning of the game, in terms of coordinates : *s0 = ((0, 0), false)*
- The legal actions that are available for Pacman are the four cardinal directions : *actions(s) = go(West), go(North), go(South), go(East) = go((-1, 0)), go((0, 1)), go((0, -1)), go((1, 0))*
- The transition model updates Pacman's position according to each new action and the corresponding food dot boolean if a food dot is eaten (that is when there is one on pacman's position).
- The goal test is to check if all food dot booleans are false.
- The step cost is the score we get from doing action *a* to go from sate *s* to state *s'* : *step cost c(s, a, s') = 1 + #time steps - #number of food dot eaten*

2 Implementation

a. The implementation error in *dfs.py* is located in the *key* function. Indeed, it must return an object key that uniquely identifies a Pacman state, as mentionned in the specifications of the function, but the key that is returned isn't unique : the position of Pacman in the maze doesn't take into account whether a food dot at this position has been eaten or not. Therefore, Pacman couldn't come back on a cell that he had already visited after eating a food dot because it was in the closed set of visited nodes.
To fix this, we added the food matrix as a second argument of the returned key (the first one being the position of Pacman) in order to uniquely identify a state.

b. cfr. *astar.py*

c. We chose a cost function $g(n) = 1 + \#steps - \#number\_of\_eaten\_food\_dots$ in accordance with the original score function.
About the heuristic *h(n)*, we decided to use the concept of manhattan distance to compute an admissible heuristic because the layout of the game is a maze, with walls that force Pacman to make detours. Indeed, $h(n) \leq h^*(n)\ \forall n$. This heuristic considers the goal to be the furthest food dot. If there is only one food dot, it returns the manhattan distance from Pacman to the food dot. If there are at least two food dots, we compute the highest manhattan distance from Pacman to one of the food dots and from this food dot to the furthest food dot. For each food dot (except the furthest one), if A is the position of Pacman, B the currently checked food dot and C the furthest dot, we compute this Chasles relation: $|AB|+|BC|-|AC|$. The result of the heuristic is the distance $|AB|+|BC|$ that maximizes the first relation. In short, $h(n) = |AB| + |BC|$ for the B position such that $|AB| + |BC| - |AC|$ is maximised. This approach preserves the admissibility of the heuristic. Indeed, Pacman must at least reach the furthest food dot to win. And if there are more than one food dot, it will also take a path that is at least passing through two food dot cells provided that the last of the two contains the furthest food dot. Thus, following these constraints, we can optimise the search by maximizing the estimated path length through 2 food dot cells.

   Regarding optimality, we tried to make the A* algorithm consistent in order to make it optimal. By definition, a consistent heuristic satisfies the relation $h(n) \leq c(n, a, n') + h(n')$ for every n and every successor n' generated by any action a. In our algorithm, the cheapest possible cost is $c = 1$ (cfr. point d), which represents a step towards a food dot. The value of the heuristic is then $h(n') = h(n) - 1$ if we are closer to the furthest dot in state n' than in state n, and $h(n') = h(n) + 1$ if not. Therefore, we have $h(n') \leq h(n) \pm 1 + 1 \Leftrightarrow h(n') \leq h(n) + 2$ or $h(n) + 0$, which is always true. In other cases, the step cost is greater because of the length of the path, and the relation is also satisfied. A consistent heuristic is always optimal, so A* is optimal. It also proves once again that $h(n)$ is admissible.

d. Our choice of $g(n)$ preserves the completeness of A* when $h(n) = 0\ \forall n$ because all step costs are greater than 0 no matter the state. Indeed, Pacman can't eat more food dots than the length of the current path. The smallest step cost occurs in the best case scenario in which Pacman makes 1 step, leading to eating 1 dot, and $g(n) = 1 > 0$. Therefore, $g(n) \geq 1\ \forall n$ and the completeness is assured. The cost function, inspired by the game score formula, penalises each new step by the current path length. Therefore, it pushes Pacman into minimising the path length. The cost function also rewards Pacman for eating food dots, pushing it to catch them all. These two driving forces make $g(n)$ optimal. As a result,

optimality of the algorithm when $h(n) = 0$ is preserved because the nodes are expanded in the order of their optimal path cost.

e. cfr. *bfs.py*

f. We chose $g(n) = 1 + \#steps$ and $h(n) = 0$ such that the fringe is a priority queue where the priority is only a non-decreasing function of the depth of the node. As a result, BFS is optimal. $g(n)$ still preserves the completeness of A* for the same reason as mentioned previously.

3 Experiment 1

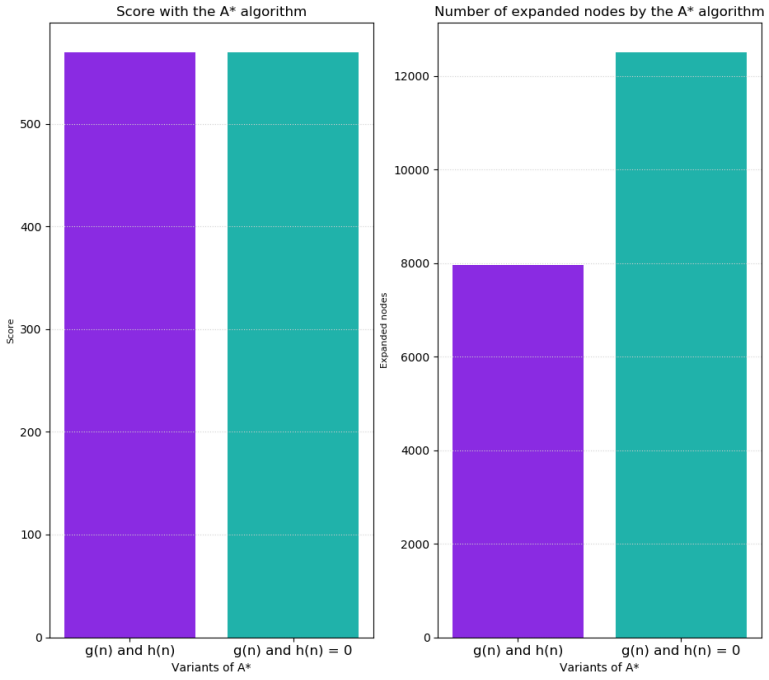

Figure 1: Performances comparison between A* and UCS

a. cfr. Figure 1

b. Comparison of the results of Figure 1 :
We see on figure 1 that the performances of both algorithms are similar

in terms of score, both having 570, while the number of expanded nodes is much higher with $h(n) = 0$ (12517 against 7960).

c. Explanation of the results of Figure 1 :
For $h(n) = 0$, the algorithm explores the maze in every direction without a clue on the nearest goal state. It only refers to the cost function to estimate the step cost of each new state. It is expensive in terms of space and time complexity because more nodes are explored. Conversely, the other version of A* has an heuristic function to estimate the plausible location of the nearest goal node, which is used to expand nodes from the fringe in decreasing order of desirability.

d. It corresponds to the uniform-cost search (UCS) algorithm.

4 Experiment 2

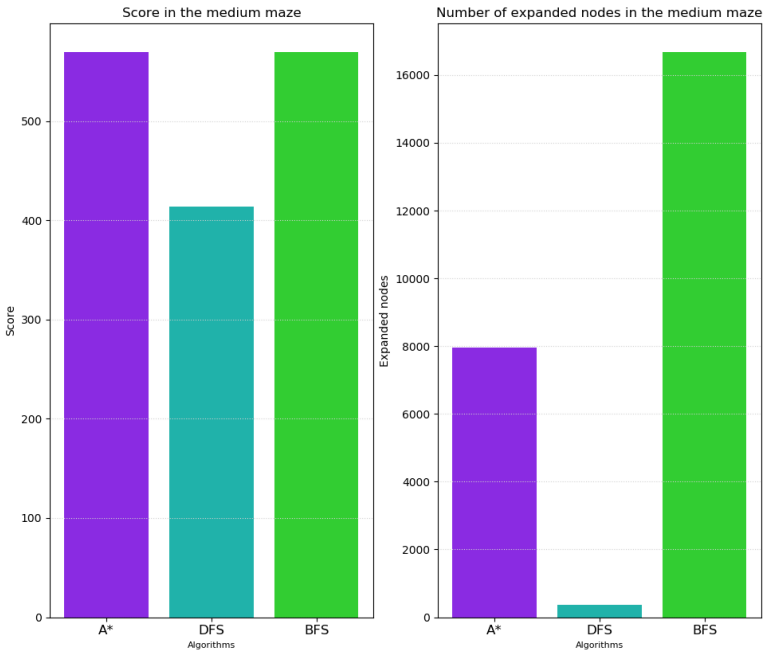

Figure 2: Performances comparison between A*, DFS and BFS

a. cfr. Figure 2

b. Comparison of the results of Figure 2 :

- Comparison between A* and DFS :
  The score generated by A* is far better than DFS' (570 against 414), while the number of expanded nodes follow the inverse tendency : there are much fewer nodes that are expanded with the DFS algorithm than with A* (361 against 7960).
- Comparison between A* and BFS :
  The score obtained is equivalent with both algorithms (570). Regarding the expanded nodes, Figure 2 shows that there are more of them in BFS (16688, which is twice as much as in A*)

c. Explanation of the results of Figure 2 :

- The DFS algorithm has fewer expanded nodes than A* because it doesn't search for the optimal path. Instead, it only explores the nodes of the maze in depth, i.e. by adding nodes to one path until it blocks and then trying another path the same way and so forth until reaching the goal. As there are tons of possible solutions in the case of the medium layout, DFS finds quickly one of them. The faster the goal state is encountered, the fewer the number of expanded nodes. Conversely, A* search for an optimal solution, thus exploring much more nodes. In terms of score, the difference is blatant because DFS finds a goal state without caring about optimization. It finds a food dot, then another and so forth until the goal state, while A* looks for food dots by exploiting information about a plausible location of the goal node in order to minimize the path length.
- Our BFS is the complete opposite of DFS. The algorithm always expands the shallowest node in the fringe, exploring thus many possible paths at the same time. Therefore, for each newly found food dot, BFS explores every possibility in increasing-depth order, and so on until the cheapest combination is considered. It represents a big amount of work, which is reflected into the number of expanded nodes. The score is as high as A* because BFS is optimal when the path cost is a non-decreasing function of the depth of the node.