INFO8006: Project 2 - Report

BARE Alexandre - s172388
RANSY Bastien - s174413

November 10, 2019

1 Problem statement

a. We define a state as a tuple

$$s = \Big((x_{pac}, y_{pac}), (x_{\text{ghost}}, y_{\text{ghost}}), ghost\_direction, food\_boolean\_matrix\Big)$$

where ghost_direction = {North, East, West, South}

Initial state : the initial state of the game is composed of the initial position of pacman and the ghosts, of the initial direction of the ghost as well as of a matrix of booleans giving the presence of a potential food dot at each position.

$$s_0 = \Big((x^0_{pac}, y^0_{pac}), (x^0_{\text{ghost}}, y^0_{\text{ghost}}), initial\_ghost\_direction, initial\_food\_boolean\_matrix\Big)$$

Player function : the function *player(s)* returns an id p for the player that has the next move in state s, ie: $p = 1$ for pacman and $p = 0$ for the ghost. In this project, pacman always starts, then the ghost moves.

Legal actions : pacman and the ghosts are allowed to move in the maze in the four cardinal coordinates, but their moves may be restricted by the walls. The ghosts can not make a half turn unless it is obliged to do so.

$$\begin{aligned} actions(s) &= \{go(West), go(North), go(South), go(East)\} \\ &= \{go((-1,0)), go((0,1)), go((0,-1)), go((1,0))\} \end{aligned}$$

Transition model :

$$s' = result(s, a)$$

is the new state with the updated position of pacman and the ghost, the ghost direction and the new food matrix which is updated if action a leads to eating a dot.

Terminal test :

$$terminal(s) = True$$

if pacman is eaten by a ghost and loses the game, or pacman has eaten all the food dots and wins the game.
Utility function :

$$utility(s, p) = \begin{cases} 1 \text{ if pacman wins} \\ 0 \text{ if the ghost wins} \end{cases}$$

b. In this game, pacman wants to maximize the utility function by eating the food dots as fast as possible, while the ghost wants to minimize it by catching pacman as soon as possible. We can define the game as a zero-sum game because the total payoff for all players (the ghost and pacman) is constant. Indeed, utility(s, p = 1) + utility(s, p = 0) = 1 at any terminal state s, i.e. for any game.
Both pacman and the ghost share the same score function. We can see that only two final states are taken into account by the score function: either the player loses (-500) or wins (+500) even though the actual score can differ from a game to another. Therefore, our utility function correctly preserves this property. And we do have a zero-sum game.

2 Implementation

a. Minimax is complete if the game tree is finite. In our case, we are sure that the tree is finite because every state is explored. To avoid an infinite tree, we store each newly-visited state in a dictionary. A state that has already been explored isn't considered unless the score is greater than the last time this particular state was encountered. Minimax is also optimal as it always provides the optimal move for pacman against an optimal adversary.

b. cfr. *minimax.py*

c. cfr. *hminimax0.py, hminimax1.py, hminimax2.py*

d. Note : (1) all the distances presented in this section are computed with the manhattan distance algorithm. (2) each cutoff function cuts the expansion of a state if it is a terminal state (either pacman loses or wins). (3) each cutoff function cuts the expansion of a state in 2 other useful cases when the ghost is far from pacman:
- if the distance between Pacman and the ghost is greater or equal to 2 and pacman has just moved. It ensures that at the end of the ghost's turn, the distance between pacman and the ghost will be at least 1.
- if the distance between Pacman and the ghost is strictly greater than 2 and the ghost has just moved. It ensures that at the end of pacman's turn, the distance between pacman and the ghost will be at least 2 so that the ghost can not reach pacman next turn.

Our first cutoff test, implemented in **hminimax2.py**, consists in assuming that pacman is in a good position in the maze if the distance between himself and the closest food dot is lower than the distance between the ghost and this dot. If it is pacman's turn, the equality between these distances is also considered as a favorable configuration. Indeed, in such a state, pacman is more likely to reach the dot before the ghost.
The evaluation function is

$$eval(s) = 5 \times pacGhostDist - minPacFoodDist - 10 \times numFood$$

where *pacGhostDist* is the distance between pacman and the ghost, *minPacFoodDist* is the distance between pacman and the closest dot, and *NumFood* is the number of food dots left in the maze. In this way, pacman tries to keep its distance from the ghost to maximise *pacGhostDist*, is attracted towards the closest food dot to minimise *minPacFoodDist* and wants to eat as fast as possible each food dot to minimise *NumFood*.

The second cutoff test, implemented in **hminimax1.py** stops the expansion of nodes when the mean distance between pacman and the food dots is lower or equal to the mean distance between the ghost and the food dots, if pacman has the next move. In this way, pacman will have on average less steps to make than the ghost to eat all the dots and therefore, he's probably in a good position.
The evaluation function is

$$eval(s) = score(s) - 4 \times minPacFoodDist - 10 \times numFood$$

where *score(s)* is the score in the state explored, and *minPacFoodDist* and *numFood* have the same meaning as before. Hence, pacman wants to quickly wins to maximise the score, is attracted towards the closest food dot to minimise *minPacFoodDist* and wants to eat as fast as possible each food dot to minimise *numFood*.

The final cutoff test, implemented in **hminimax0.py**, stops the expansion of nodes when pacman is located behind the ghost. pacman has the advantage because the ghost is unable to catch him in the next move, given the fact that the latter can't make a half-turn (unless he is forced to do so). The evaluation function is

$$\begin{aligned} eval(s) = {} & score(s) - currentScore - alreadyVisitedStatePenalty \\ & + \frac{pacGhostDist}{2} - minPacFoodDist - 2 \times NumFood \\ & + \frac{maxPacFoodDist}{2} + \frac{NbLegalActions}{2} \end{aligned}$$

where *score(s)*, *pacGhostDist*, *minPacFoodDist* and *NumFood* have the same meaning than before, *maxPacFoodDist* is the distance between pacman and the furthest dot, *NbLegalActions* is the number of available actions for pacman and *currentScore* is the score of the game after the last

played move. Finally, *alreadyVisitedStatePenalty* is a -400 penalty imposed to pacman if he visits a game state that has already been visited in the previous already-played moves. Therefore, in addition of the previous evaluation parameters, pacman is strongly penalised if he steps on a cell on which he has already been on. This allows him to come back on his steps but only if it is really necessary. The *NbLegalActions* term pushes pacman to avoid as much as possible to be blocked in a corner by the ghost.

3 Experiment

a. cfr. Figure 1

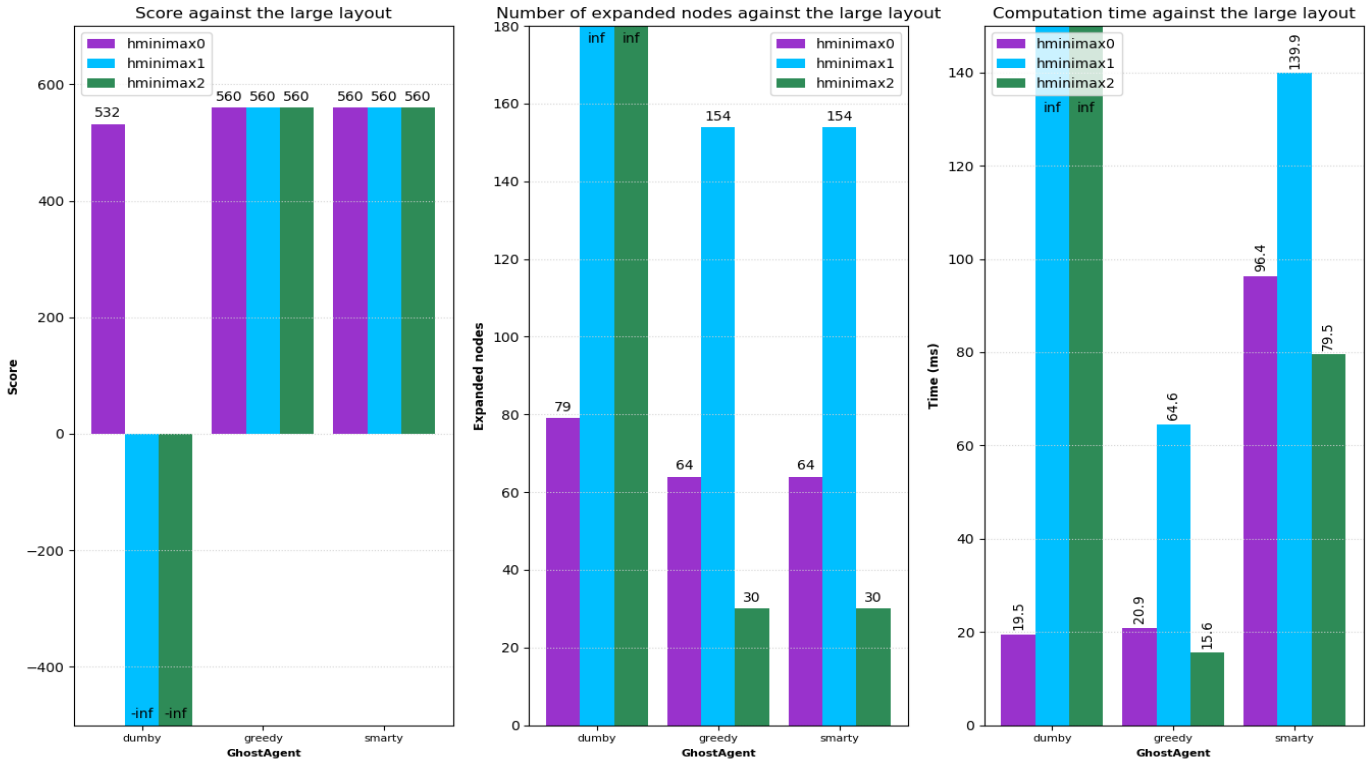

Figure 1: Comparison of the performances against the large layout

b.
- Against the **dumby** ghost, only the hminimax0 provides a satisfying result because pacman goes around in circles in other cases, a situation that we represented as an infinite computation time and number of expanded nodes, and a score of $-\infty$. It can be explained by the fact that pacman is allowed to come back to previous played states but only in exchange of a severe penalty in the eval function, thus discouraging in most cases pacman to do so. Moreover, a higher number of parameters are taken into account in hminimax0 which were carefully weighted to have a relevant evaluation of each state.
- Against the **greedy** ghost, we can observe that in terms of score, each version of hminimax produces the same score.
  The number of expanded nodes and computation time is much higher with hminimax1 while hminimax0 and hminimax2 have the same quantity of expanded nodes. hminimax2 is however the fastest of all. These observations can be explained by the fact that the least deep cutoff is in hminimax2, then in hminimax0 because the quiescent states are more easily reached. Pacman is respectively more often closer than the ghost from the closest food dot (relative to pacman's position) and can often be in the back of the ghost. The precision of the eval function is a bigger priority in this case as the cutoff tends to occur at shallow level in the game tree. It is also the fastest method as it is the least demanding in calculations. hminimax1, using average distances, is the deepest cutoff as this approach tends to attenuate the extreme results. Therefore, the precision of the eval function must not be as good as for hminimax0 and hminimax2 to obtain optimal results. It is also the most demanding approach in terms of calculations.
- Against the **smarty** ghost, we have the same observations as with the greedy ghost although computation time is on average higher than with the greedy ghost. This can be explained by the fact that a smart ghost will often be in a better position to confront pacman. The algorithm needs thus more calculations than with a sub-optimal adversary.

c. To handle several ghosts, we would add a *ghost_nb* argument in the minimaxAux function, so that minimaxAux recursively calls itself with *is_pacman* == *False* (meaning it is not pacman's turn but rather the ghost #*ghost_nb*) *nb_ghosts* times. Therefore, for each pacman's move, we consider each ghost's best move and so on until the terminal test is reached. The game can still be described as a zero-sum game. Indeed, the ghosts, all play with the same goal (catching pacman). Therefore, it can be considered that if one of the ghost catches pacman, they have all won (utility(s, p = 0) = 0 and utility(s, p = 1) = 1), while if pacman catches all food dots, he has won (utility(s, p = 0) = 1 and utility(s, p = 1) = 0) where p = 1 corresponds to the ghost team while p = 0 corresponds to pacman. The total payoff remains a constant for all games.