INFO8006: Project 2 - Report

Théo Stassen - s150804
Hubar Julien - s152485

November 10, 2019

1 Problem statement

a. **Adversarial search problem**

Initial state	The initial positions of the ghost are (3,3), (4,4) and (4,4) for the small, medium and large mazes respectively. The initial positions of the Pacman are (1,3), (2,3) and (2,3) for the small, medium and large mazes respectively. All the foods are initially present.
Player(s) function	player(s) = 1 if the ghost has the movement in state s, 0 if it is the Pacman (they alternate one after other, starting with pacman)
Legal action	The direction the pacman can takes, which depends on which are legal. If there are no walls next to pacman : $actions(s) = (left, right, up, down)$
Transition model	The transition model is "*the new position and new situation of food = result of the current position and the taken direction*"
terminal test	The game is over if pacman win (no food remaining) or if ghost win (same position as pacman)
Utility function	We use the terminal state score as a utility function.

b. Zero-sum games are a contradictory search that involves competition. In the zero-sum game, the total payoff is constant. Here Pacman tries to maximize his score, while the ghost tries to minimize it. The payoff of the ghost can be described as exactly the opposite of the payoff of the pacman, the score (max the time steps, min the eaten foods, want losing end). The total of the payoffs is so 0. So the game can be described as a zero-sum game.

2 Implementation

a. The Minimax algorithm is perfectly applicable to our implementation of pacman as a two-player zero-sum game. It leads us to explore the decision tree and assign to the leafs a value that takes into account the benefits for the Pacman. We can build a finished tree by prevents loops by keeping in memory a set of the already visited states. So the completeness can be guarantees by taking account the set in the Legal actions : *actions (s, closed)* which is now more strict.

b. The minimax algorithm is implemented in *minimax.py*

   An important note is that it is not a "pure" minimax algorithm. The minimax, given an initial state and a terminal-test and a utility function gives the optimal path assuming an optimal adversary. We obtain so one single path. The problem is that with this layout, it is strictly impossible to find a path that can beat the game with all types of ghosts (the first pacman step must be South which means that greedy ghost will go South and smarty/dumby will go North, and so there is no path which works). To resolve this problem, we do a first step which maximise the distance with the ghost. After this step, we call the minimax algorithm. The ghost will so be in different places and direction at initial state depending of it agent and so we can find two different correct paths.

c. The H-Minimax algorithm is implemented in *hminimax0.py* (Evaluation function n1 and Cutoff Tests 1), *hminimax1.py* (Evaluation function 1 and Cutoff tests 2), *hminimax2.py* (Evaluation function 2 and Cutoff tests 1).

d. Variables (all depending of s, the current state, and not on d, the current depth), with "distance" which means Manhattan distance :

   - d_p^f : min distance between pacman and the foods
   - $d_{p\_mean}^f$: mean distance between pacman and the foods
   - $d_{p\_init}^f$: min distance between the init position of pacman and foods
   - d_g^f : minimal distance between the ghost and the foods
   - d_p^g : distance between pacman and ghost
   - n_f : number of remaining foods
   - $n_{f\_s\_d}$: number of foods at the same minimal distance of pacman
   - n_{f_e} : number of foods eaten

   - Evaluation function 1:

$$Eval(s) = \begin{cases} 4 - d_p^f + d_p^g + (d_{p\_init}^f - d_p^f) * 5 & \text{if } n_f = 1 \text{ and } d_p^f <= d_g^f \\ 1 + (d_{p\_init}^f - d_p^f) * 5 & \text{if } n_f = 1 \text{ and } d_p^f > d_g^f \\ -d_{p\_mean}^f - d_p^f + n_{f\_s\_d} + n_{f\_e} * 10 & \text{if } n_f > 1 \end{cases} \tag{1}$$

   - Evaluation function 2:

$$Eval(s) = -d_{p\_mean}^f - d_p^f + n_{f\_s\_d} + n_{f\_e} * 10 \tag{2}$$

- Cutoff-test 1:

$$Cutoff\_test(s, d) = \text{if } (d >= 2) \tag{3}$$

- Cutoff-test 2:

$$Cutoff\_test(s, d) = \text{if } (d >= 4) \tag{4}$$

The evaluation functions are a weighting of a lot of simple characteristics of the game state.

In our journey to find an optimal solution we obtain chronologically five interesting function (The two first allow us to find the best but are not present in the code).

The first we find is composed of the two first cases of Evaluation 1 and the Cutoff-test 2, which works for all ghost type / layout cases except for the dumby / large.

The second has been found starting from this specific case and works for all cases except smarty and dumby / small. It is concretely the combination of Evaluation Function 2 and Cutoff-test 2.

It is easy to understand that this second evaluation function is not well adapted to the small layout because some components of the function are useful only if there are more than one ghost initially.

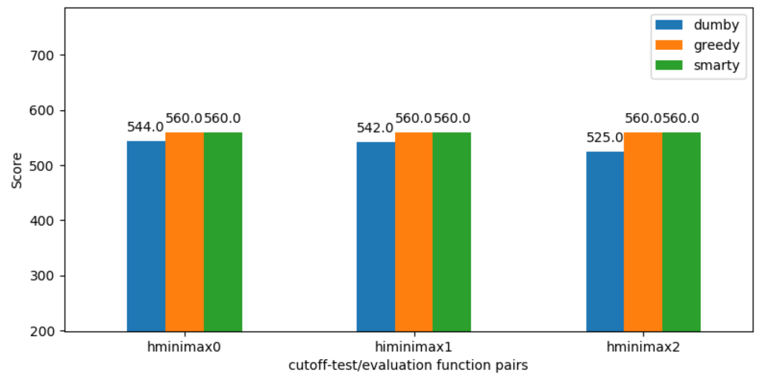
The third (*hminimax1.py*) is simply the combination of the second and third. We discuss about that in section 3.b. It works for all cases.

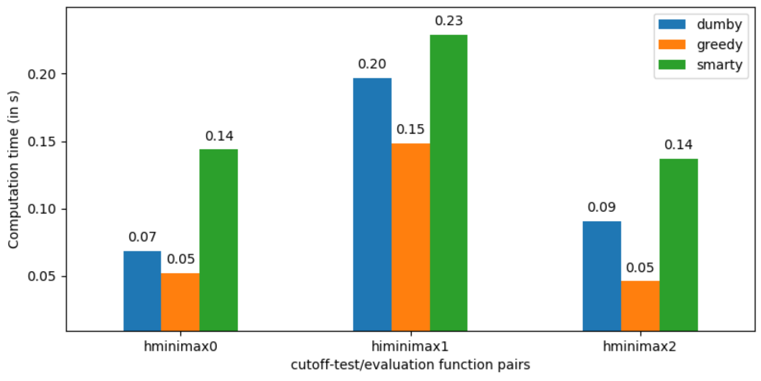
The fourth (*hminimax0.py*) has been found by simply decrease the limit depth : Evaluation function 1 ans Cutoff-test 1. We discuss about that in section 3.b.

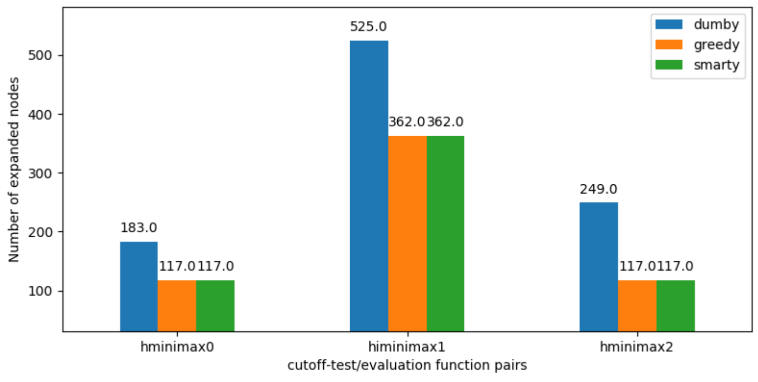
Knowing that a lower depth was better, we find the last (*hminimax2.py*) which is the same as the second but with Cutoff-test 1. We kept the three best in the code and report by comparing their score.

All the variable components of the evaluation functions are generalisable. The pacman must eat foods to improve it score : that is why we want to minimize d_p^f but also $d_{p\_mean}^f$ and that we want to maximize $d_{p\_init}^f$ - d_p^f (used to penalise hardly the branches where pacman does not approach the food) and n_{f_e} . We use $n_{f\_s\_d}$ to give an advantage to a case where two foods are at the same minimal distance compared to a case with only one. It is in general must convenient to have the ghost far away from pacman and the foods : maximize d_g^f and d_p^g. It is necessary to use the variables where there are significant due to the situation. That is why we have in our best evaluation function tree cases. The third is convenient to multiple foods cases. The first is convenient where the pacman is closer to the only food than the pacman. In this case it priority is to go to the food. The second is when the ghost is more closer to the food and so can, if smart, eat the pacman before pacman eat the food. In this case the goal is to avoid at any cost the ghost in order to reach the first case. Going directly to the food is not necessary, but going farther is not useful neither.

3 Experiments

a. 





b. *hminimax0* gives the better score for the dumby ghost (because it combines the different cases). *hminimax1* is slower because of the higher depth (two trees of depth 2 has less nodes than one tree of depth 4). *hmninimax2* is less aware of the different situations, so obtain a lower score in dumby/large.

The explication of why a low depth is good is that the more we go down in the tree, the more the difference between what we think the ghost will do (considering that it an optimal adversary that perfectly minimize the pacman score) and what the ghost real do increase significantly. If the ghost was perfect, it might be interesting to go through a high depth, but in our case, the hminimax works only (but really efficiently) with very low depth. We firstly choose 4 and it is in fact better with depth 2 (one move of pacman, one of the ghost). In this game, there are no problem of horizon effect (where a low depth is problematic because bad moves can appear as good moves), or it has a way lower effect than the imperfection of the ghost agent.

In addition of the problem of the depth, the fact that greedy / smarty / dumby are not equivalent to a perfect opponent has been the main problem to find a perfect evaluation function. We had relatively easily found some function that compute scores in the way that if the opponent was perfect the grid would be succeeded (we verify by hand with a 4 depth tree). The problem is that is not the case. We had so to find a function which gives results 'compatible' to a lot of different non-perfect (and possibly dumby) opponents. . We did succeed to find a solution only composed of 'general' blocks, not specific to the problem and the solution is general but we cannot be sure that it will works in all configuration. For example the first evaluation function we found was good and general, only composed of variable which are relatively obviously correlated to the idea of winning, but does not works on dumby/large case. The reason is because the dumby ghost do a first two steps which is counter intuitive and let the pacman in a position where, if it as the same behaviour than in the other cases, it will remains traped in one part of the grid. We needed to have a function, also general as the function used but that it is able to get out of these kind of situations.

c. We should see all ghosts as controlled by one agent whose goal is to minimize the score. To do this, it would be necessary to modify the feedback of the recursive function so that it takes into account all ghosts in the same time. Therefore, each ghost must takes into account the action of the other ghosts and the effect of this action on its performance. But the game is still a zero-sum game, as the ghost agent as the same payoff than before and the total payoff is always 0. If pacman win, the ghost agent loose and if the ghost agent win (if one ghost win), pacman loose.