Implementation of a Pacman game with Minimax

Antoine Louis
s152140

Tom Crasset
s154416

---

1 Formalising the game

Now that some ghosts appear in the grid, trying to defeat the Pacman agent by capturing it, we are now facing a multi-agent environment where agents may have either conflicting or common interests. That multi-agent environment, called a game, can easily be formalised as an adversarial search problem. In this case, it has at least two players : the Pacman agent, that wants to maximise the score by eating as many foods as possible as soon as possible without being captured by a ghost, and one or several ghosts that want to minimise the score by capturing the Pacman agent as soon as possible. This game consists in a game tree, a terminal test and an utility function.

1.1 Game tree

The game tree is a tree where the nodes are game states and the edges are the moves. It is defined by :

- The initial state of the game.
- A description of the legal actions available to a state.
- A transition model that returns the state that results from doing an action a in state s.

A game state is defined by whose turn it is, the position of Pacman, the ghosts' positions and the grid of remaining food.

The initial state of the game is defined by the turn being for the Pacman agent, the initial position of the Pacman agent, the initial positions of the different ghosts and the initial grid of foods.

The legal actions of a state are simply for the agent whose turn it is - Pacman or ghost - to move from its position to one position to the north, the south, the east or the west, without getting into a wall.

The transition model defines the result of a move, so basically the new state in which an agent has moved from its position to another, Pacman possibly having eaten a food dot.

1.2 Terminal test

Basically, the terminal test is a predicate telling if the game is over or not. A terminal test consists in checking if the current game state is a terminal state. In this case, a state is terminal if Pacman has eaten all the foods without having been captured by a ghost (it's a win), or checking if a ghost manages to capture the Pacman agent (it's a loose).

1.3 Utility function

The utility function defines the final numeric value for a game that ends in a terminal state s for a player p. The Pacman game is a zero-sum game because the total payoff to all players is the same for every instance of the game, it's either the Pacman agent that wins (1+0) or the ghosts (0+1) resulting in a constant payoff of 1 in either case.

2 Bar plots

2.1 Computation time

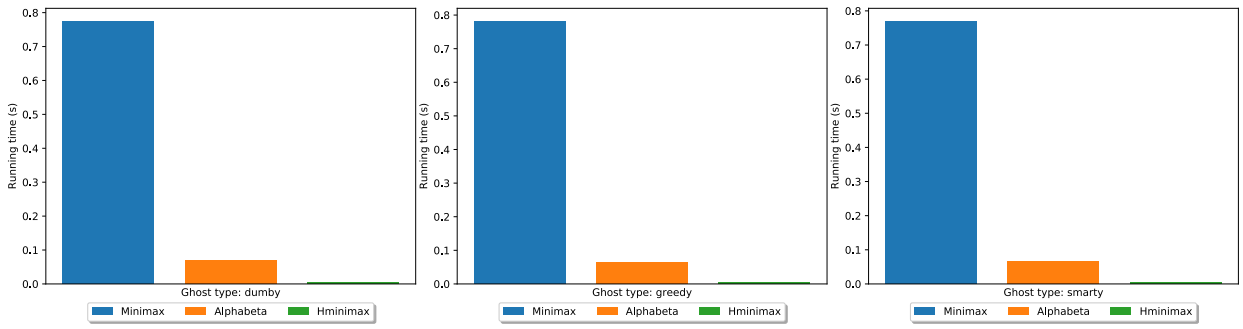

FIGURE 1 – Total computation times of the different agents for all ghost types

2.2 Expanded nodes

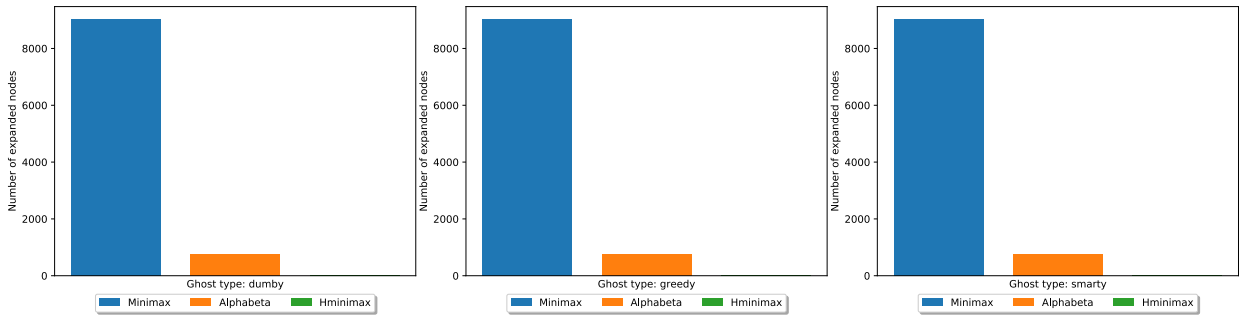

FIGURE 2 – Total number of expanded nodes of the different agents for all ghost types

2.3 Final score

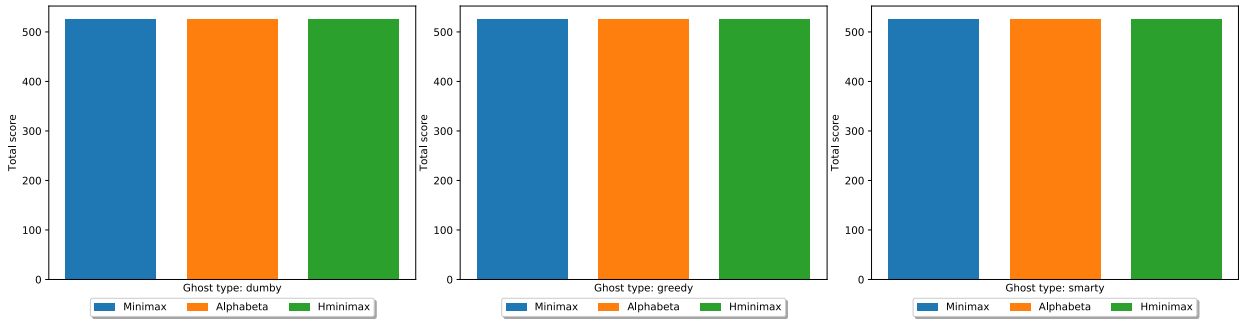

FIGURE 3 – Final score of the different agents for all ghost types

3 Performance analysis

As can be seen on the Figures 1 and 2, the Pacman agent implementing Hminimax seems to be the most performing for the depth we have fixed (a maximal depth of 4). It has the lowest computation time with the lowest number of expanded nodes for an optimal score in the small layout.

Behind Hminimax, the Pacman agent implementing Alphabeta outperforms, whitout surprise, the one implementing Minimax in both computation time and total expanded nodes, which completely makes sense because the aim of Alphabeta is to get rid off some parts of the Minimax tree during computation.

The agents implementing Minimax and Alphabeta work well on the small layout but have some limitations on the two other layouts, the level of recursion in the tree becoming too big making the computation of such a tree impossible for a reasonable amount of time.

The only Pacman agent capable of playing on the two biggest layouts (`medium_adv` and `large_adv`) is the one implementing Hminimax. The good working of the latter needs a good evaluation function and a cutoff function. Making a good evaluation function boils down to finding the features that count the most for Pacman to win, and then weighting each one of them according to their respective importance. The evaluation function will then be a combination of these weighted features. Our choice fell on the following features :

— The current state score.
— The closest distance from Pacman to a ghost.
— The closest distance from Pacman to a food.
— The number of foods left in the grid.

The current state score is computed with the given score evaluation function, left intact. It will be the only positive parameter of the evaluation function, all others being subtracted from this score in order to avoid that some non winning states get a better score than a winning one just because it finds itself in a advantageous position in relation to the others features.

Then, it is important that a ghost doesn't approach Pacman too closely as the risks of losing will increase. As this feature must be subtracted from the state score and as the the closest the ghost is, the lowest the score must be, the inverse of this feature is taken.

To conclude, the distance to the closest food and the number of remaining foods are added to the score as the former must be greater if they increase.

The last step consists in weighting each feature of the combination. Starting from some values found on the internet, we slightly modified them after repeated tests in order to find the ones giving the best scores. Finally, we came with the following equation :

$$\begin{aligned} score = \quad & 1 \ * state\_score + \\ & -1.5 * \frac{1}{distance\_to\_closest\_ghost} + \\ & -1.5 * distance\_to\_closest\_food + \\ & -6 \ * number\_of\_remaining\_foods \end{aligned} \tag{1}$$

A better way of finding the ideal weights for each feature would have been to train some machine learning algorithm on multiple parts.

Now that we have a good evaluation function, the ideal depth for the cutoff function must be found. After testing some depths, it turns out that a maximal depth of 4 seems to be a good compromise between computation time and optimal score.