Faculty of Applied Science
University of Liege

# Introduction to Artificial Intelligence

Part II : Minimax agent

November 2018

Derroitte Natan
Testouri Mehdi

*Prof. G.Louppe*

## 1 Formalization of the adversarial search problem,

An adversarial search problem is defined by several components : a state representation, the initial state, the player function, the legal actions, a transition model, a terminal test and an utility function. These components are described in this section.

1. **State representation**
   The state $s$ is represented by a tuple formed by the position of Pacman, the position of the ghost and the food matrix (matrix giving the positions of the remaining foods). More formally :

   $$s = ((i_p, j_p), (i_g, j_g), fm)$$

   where $i_p$ and $j_p$ describe the position of Pacman along $x$ and $y$ while $i_g$ and $j_g$ describe the position of the ghost. Finally, $fm$ is the food matrix.

2. **Initial state**
   The initial state $s_0 = ((i_{0p}, j_{0p}), (i_{0g}, j_{0g}), fm_0)$ is simply the initial position of Pacman and the ghost along with the initial food matrix.

3. **Player function**
   The player function $player(s)$ is described below :

   $$\begin{aligned} player(s_0) &= pacman \\ player(s_{i+1}) &= pacman \qquad \text{if } player(s_i) = ghost \\ player(s_{i+1}) &= ghost \qquad \text{if } player(s_i) = pacman \end{aligned}$$

4. **Actions**
   The set of actions $actions(s)$ are all the allowed moves for Pacman and the ghost for the given state $s$. The possible allowed actions are 'North', 'South', 'East' and 'West'.

5. **Transition model**
   The transition model $result(s, a)$ is the following, assuming that Pacman is the currently moving player.

   - $s' = result(s, North) = ((i_p, j_p + 1), (i_g, j_g), fm')$
   - $s' = result(s, South) = ((i_p, j_p - 1), (i_g, j_g), fm')$
   - $s' = result(s, East) = ((i_p + 1, j_p), (i_g, j_g), fm')$
   - $s' = result(s, West) = ((i_p - 1, j_p), (i_g, j_g), fm')$

   Where $fm'$ is the updated food matrix.

6. **Terminal test**
   The terminal test is defined by the two possible outcomes of the game.

   (a) Pacman has eaten all food without being eaten by the ghost.
   (b) Pacman has been eaten by the ghost.

   The terminal test thus checks for any of those two situation.

7. **Utility function**
   The utility function $utility(s, p)$ is defined below :

   $$\begin{aligned} utility(s, pacman) &= score \\ utility(s, ghost) &= score \end{aligned}$$

   where $score$ = -nbTimeSteps + 10nbEatenFood - 500 (if losing end) + 500 (if winning end)

   Pacman will thus try to maximizes the utility while the ghost will try to minimize it.

## 2 Algorithms Comparison

In order to obtain a complete comparison, it is necessary to take into account 3 main aspects: the **computation time**, the final **score** and the number of **explored nodes**.
In the following figures, the comparisons are made on the small map and for the different ghosts agent available : *dumby*, *greedy* and *smarty*.

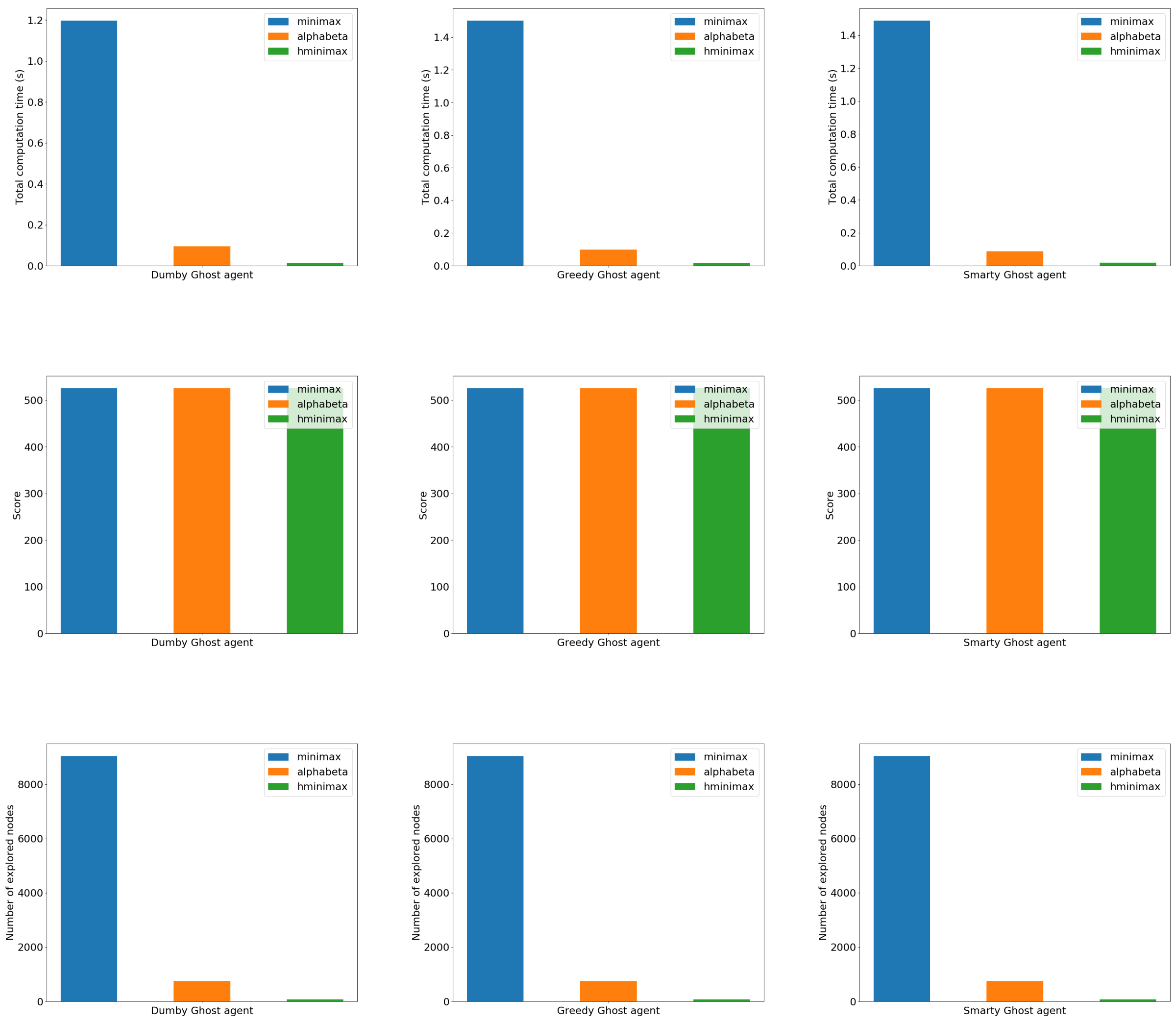

Figure 1: Comparison of the different ghosts agent. From up to down : Computation time, Scores and Number of explored nodes. From left to right : dumby, greedy and smarty ghost agent file

The first thing that appears from these graphs is that the influence of the ghost agent is not very marked: the results are almost similar independently of it.
It can be seen that `minimax` has a much larger number of explored nodes, which is also accompanied by a longer computation time.
Although the distance estimate used in `hminimax` also extends nodes, its total number of explored nodes remains very low. This was possible by applying alpha-beta pruning and adjusting the recursion depth according to the size of the maze. All this will be discussed in more detail in the next section.

## 3 Performance and limitations of the agents

### 3.1 Limitation

All three algorithms work fine on the small layout with all three ghost agents. However, the `minimax` and `alphabeta` implementations don't work on the medium and large layouts with any ghost agent because they perform too much recursions to compute the solution. Indeed, the `minimax` algorithm is a brute force search and the size of the state space increases sharply with the size of the layout. Even if the alpha-beta pruning is used, the algorithm takes too much recursion. Fortunately, the limited depth imposed in the `hminimax` implementation makes it possible to compute a solution on all layouts with all ghost agents.

### 3.2 `Hminimax` pruning

In order to ever increase the performance of the `hminimax`, the alpha-beta pruning was also used in this algorithm. The benefit is a reduced computation time by decreasing the number of explored nodes while preserving the optimality.

### 3.3 Heuristic : custom evaluation

In order to obtain a good evaluation function that accurately represented the quality of the movements taken by the Pacman according to his chance to win, it was necessary to recreate a score associated with the state of the game.

To do this, it was decided to start from the score already present in the GameState class but weighted it according to the different variables that influence the victory conditions. Let us start from the formula used to explain the choices made:

$$\texttt{score = state.getScore() - 4 distance\_closest\_food - 5 } \frac{1}{\texttt{distance\_closest\_ghost}+1} \texttt{ - 10 nb\_foods}$$

As announced, this formula is based on the score definition made in the GameState class. Since the Pacman must maximizes this score, the negative variables present in the numerator are the variables that should ideally take the smallest possible value. This is the case for the number of remaining foods and the distance to the nearest food.
On the other hand, the negative variable in the denominator should be as large as possible to maximize the score. The distance to the nearest ghost corresponds well to this definition. Note that this variable is accompanied by a $+1$. The distance between the Pacman and the ghost is between 0 and $+$ infinity, 0 corresponding to the case where both are on the same square. To avoid dividing by zero, we add a unit at this distance.

Let's now talk about the coefficients applied:

- `nb_foods` : The total number of food seems to be the most important variable between the 3. Indeed, the major goal of Pacman is to eat them all, and the variable should be weighted accordingly. Therefore, the largest coefficient, 10 is given.
- `distance_closest_ghost` : The distance from the nearest ghost is a crucial element that can detect how close the Pacman is to losing . A significant coefficient must be applied to it.
- `distance_closet_food` :The distance to the nearest food is the key element in the guidance of the Pacman. However, the latter must not endanger himself during these movements and a lower coefficient than the distance to the ghost is therefore assigned to it.

Finally, the choice of distance evaluation must be detailed for the two calculated distances.
A first idea simply corresponds to using the Manhattan distance. However, in cases where the Pacman is separated from food by a large wall, the Manhattan distance will be very small, which does not represent reality and this poses major problems. In order to avoid these, a more realistic but more expensive distance, calculated with a BFS algorithm, was chosen for the distance with the nearest food.
As for the nearest ghost, the situation is different: the ghost should hunt the Pacman and therefore, always be close to it. The Manhattan distance is then chosen for this variable. This may be less realistic if there

is a long distance between the ghost and the Pacman, but this is not very problematic since the situation allows the Pacman more freedom of movement.

### 3.4 Depth discussion

The size of the maze as well as the maximum recursion depth of the `hminimax` algorithm are the main influences on the number of nodes extended by our solution.
It seems quite logical that to obtain a reasonable number of nodes, and therefore a low computation time, the increase in one of these variables should have an impact on the other.

It was thus decided to adapt the recursion limit according to the size of the maze. When the latter increases, the number of recursions decreases.
Thus, our solution starts by computing the size of the layout and as a function of it, calibrates the number of recursive calls.

The bearings chosen are defined only according to the size of the puzzle. Thus, the code created will work regardless of the maze proposed. The values chosen should therefore fits perfectly any layout.
However, a user may want to optimize the score[1] for large layouts and modify the chosen values.

## 4 Improvements

### 4.1 `Minimax` and `alphabeta`

The `Minimax` and `alphabeta` algorithm don't work on the medium and large layout. The fact that `minimax` is a brute force algorithm make it difficult to optimize. There might be optimizations that could potentially make it possible for `minimax` and `alphabeta` to work on larger layouts. However, this is very likely that these modifications would lead to not perfectly implement those algorithms and in this case, the `hminimax` seems to be a good alternative.

### 4.2 Heuristic

Although the design of the heuristic makes sens, there is no guarantee that this is the optimal one and further research could be done to determine an heuristic that can give the best results as possible in any configuration of layout and ghost agent.

[1]It should be noted that there is not point to do so for the layout of this project. Indeed, even with a recursion depth of 1, the `hminimax` implemented finds the optimal solution for the large maze.