AI project 2 : Adversial Search

Adrien Minne s154340
Arnaud Delaunoy s153059

23 november 2018

1 Formalizing the game as a search problem

The initial state is a state with Pacman at its initial position, the ghost at its initial position, a set of food dispatched in the maze and where it is pacman's turn to play.

The player is defined as a function returning 0 if Pacman is playing, 1 if the ghost is playing.

The utility function is defined as a the score of the game.

The available actions for pacman are going north, south, east or west as long as there is no wall blocking the way. We will also consider the move where Pacman does nothing. In the game, Pacman can stop moving if he collide head on with a wall and is not given any direction after that. The ghost's available actions are also going north, south, east, west but has the constraint that it cannot turn around. Note that the ghost cannot stop.

The transition model is defined this way : given a state s and a legal action a performed in state s, the state s', result of the transition function is defined by :

— The position of the player given by the $player$ function equal to the player position in s, with a displacement of one unit in the direction given by a, or no displacement if a is $STOP$.

— A food matrix equal to the one in state s, minus the food on the position of Pacman, if there was a food in this position

Note that the transition model defined this way covers both states where it's pacman's turn to play and where it's ghost's turn to play.

The goal test consist in checking if all the food dots have been eaten, i.e. the state has no non-eaten food dots or if Pacman touched the ghost.

2 Algorithm Comparison

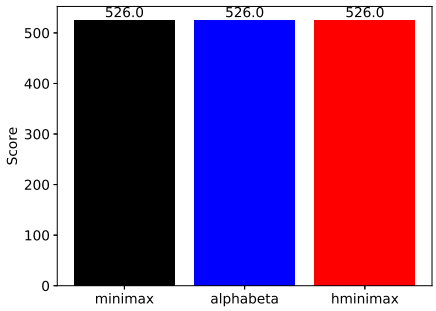

FIGURE 1 – Scores for the ghost Dumby

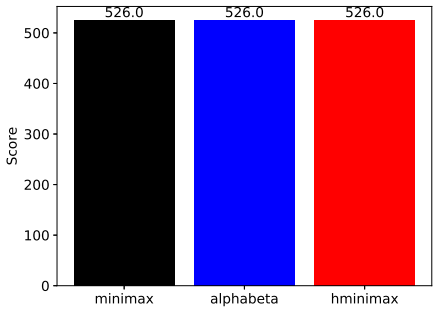

FIGURE 2 – Scores for the ghost Greedy

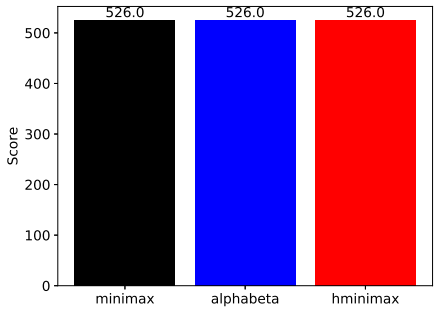

FIGURE 3 – Scores for the ghost Smarty

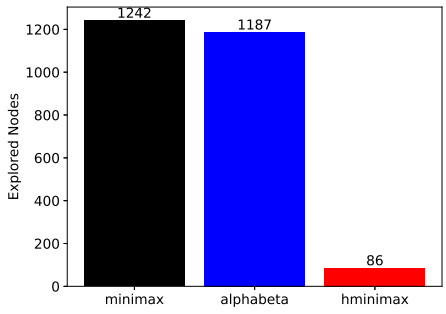

FIGURE 4 – Nodes explored for the ghost Dumby

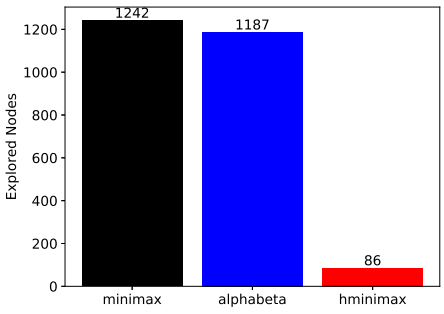

FIGURE 5 – Nodes explored for the ghost Greedy

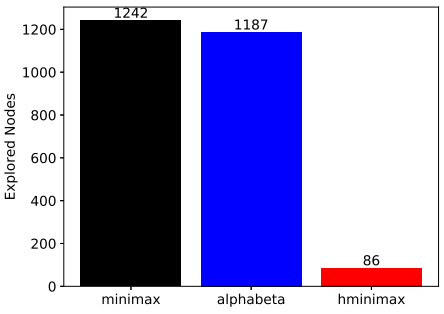

FIGURE 6 – Nodes explored for the ghost Smarty

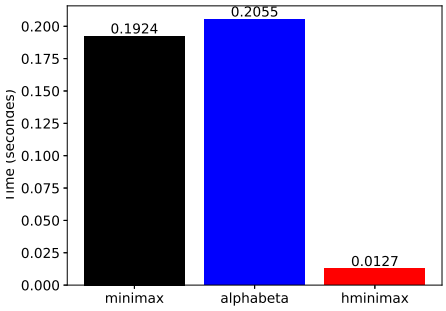

FIGURE 7 – Computation time for the ghost Dumby

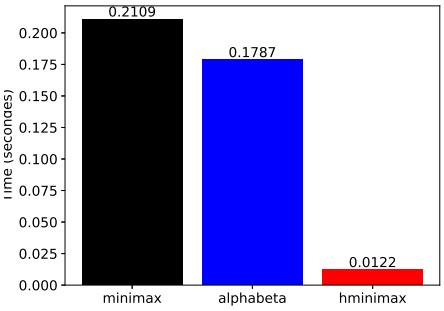

FIGURE 8 – Computation time for the ghost Greedy

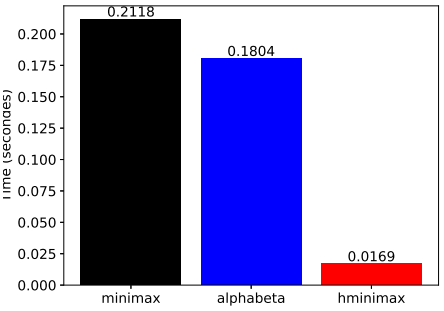

FIGURE 9 – Computation time for the ghost Smarty

2.1 Ghosts

We can notice on the figures 1-9 that the type of ghost has no impact on the performance of the algorithm on the small layout. That's because in the small layout, the game will end in 4 Pacman actions, and the 3 different ghost all do the same movements during this short time. On another layout, there could be differences between the score with different ghosts. On the medium and large medium, the performances change a bit from ghost to ghost. The performances for smarty and greedy are the same, because for these layouts they will actually make the same decision. On the other hand, dumby is so dumb that our algorithm that consider that its opponent will make the optimal move has an harder time predicting what dumby will do. Our algorithm will still win, but with slightly less points because dumby will block its way.

2.2 Scores

The scores are in the figures 1-3. It can be seen from those figures that Minimax and Alphabeta always give the same score. It is expected because they are in fact the same algorithms, only with one more optimized than the other.

For the H-Minimax, it really depends on the quality of our cutoff function and of our score estimator. In the case of the small layout with the cutoff function and estimator that we chose (see section 3.4), it give us the same solution as minimax, the best one. But if we use less precise estimators, or if the complexity of the problem goes up (eg. for the bigger layouts), H-Minimax will probably stop giving us an optimal solution. The effect of the ghost is the same as for the Minimax and Alphabeta algorithms.

2.3 Nodes Explored

The number of nodes explored are in figures 4-6. As expected, the Alphabeta is an improvement over the Minimax algorithm. The improvement is not massive because the algorithm Minimax remember the visited states, which already optimize it quite a bit (see section 3.1). The H-Minimax algorithm explore way less nodes because

we limit the size of the decision tree, instead of exploring it all searching for the optimal solution. The type of ghost has no importance for the same reason as before.

2.4 Computation time

The time of computations are in figures 7-9. These times were taken as a mean over 100 runs to try to get an accurate mean. But these time are so short (<0.5s) that event with a mean over 100 runs, these means are not really accurate. It is impossible to accurately compare those times, because if they are computed again, the results could be slightly different. The only thing that can be said, is that the time seems to be proportional to the number of nodes explored, which makes sense. That's why the H-Minimax algorithm takes less time during computation than the others. The type of ghost has no importance for the same reason as before.

3 General Discussion on Algorithms

3.1 Preventing Cycles

The main problem here is that there's cycles in the decision tree : Pacman can run back and forth or run in circles, and won't ever finish the game. That's not really a problem for the H-Minimax algorithm, since we have a cutoff function anyway, but it is for the Minimax and Alpha-Beta pruning algorithms. To solve this problem and optimize our algorithms, we decided to keep a dictionary of already visited states. We define a state by

— The position of Pacman

— The position of the ghost

— The food matrix

— The player currently playing

— The direction of the last ghost move

— Whether Pacman can stay still or not

It's important to remember the direction of the last ghost move because the ghost cannot turn around, so the possible moves of the ghost for the current state is limited by its last displacement. It's the same for if Pacman can stay still (see paragraph on Pacman available actions). If he can, the possibilities are different than if he cannot stay still.

But we cannot just indiscriminately throw away a node if its state is already visited. We can do that only if the visited state is a parent of the current node because it would create a cycle. In the other cases, we need a way to get the Minimax score of the current state from the Minimax score of the visited state. We implemented it this way : when we visit a new state in a node n, we put it in our dictionary, with a score of $None$. Then, if another node have this state and the score is $None$, we know this node is a child of the previous one. When we come back to the node n after our recursive calls, we update the value of the Minimax score in our dictionary. Note that it's possible that we get no result for the recursive calls (if all the children are already visited), in this case we remove the state from the dictionary.

When we get to a visited state and we have registered a Minimax score for this state, computing the Minimax score of the current node is easy. By computing the difference between the game score of the current node, and the game score of the state when we first visited it and adding this difference to the registered Minimax score, we get the Minimax score of the current node. For example if a state s was first visited with a current score of 0 and had a Minimax score of 500, when the algorithm is again in the state s but with a current score of 20, the Minimax score will be 520.

3.2 Minimax

The Minimax algorithm was implemented using a recursive function and a dictionary of visited state explained in section 3.1. It's worth noting that, in addition to the game state, and player currently playing, the recursive function also takes as argument the last ghost move (to get the visited state as defined in section 3.1), and the last Pacman move (to check whether Pacman can stay still).

3.3 Alphabeta Pruning

This algorithm is implemented roughly the same way as the Minimax algorithm, still with the visited state dictionary. They are however some key differences. First, obviously the presence of a score interval in the recursive

function call as defined in the Alphabeta pseudo-code of the theory course. Secondly, if a node is pruned, no assumption can be made on the Minimax score of the node with the current information. That's why instead of putting a Minimax score in the visited dictionary, we put the successors of the node in the dictionary. That way, when the algorithm is going through this state again, it don't need to compute the successors of the node anew.

3.4 H-Minimax

For this algorithm the first thing to do was to define cutoff function and an estimate function. In our case, the cutoff function is a maximal depth in the tree. The maximal depth cannot be too high, it would take too much time to compute, but not too low, or Pacman would make bad decision by lack of foresight. The estimate function is defined as :

(Current Score) + (Distance Between Pacman and Ghost) - 5*(Distance from Pacman to Closest Food Dot)
- 100*(Number of Food Dots Left in the Game)

This estimate will prioritize eating food dots, then getting closer to food dots, and finally escaping from the ghost.

Here, memorizing the visited states is useless because we're not computing the real, immutable Minimax score for this state, only an estimate. Since the estimate will not stay the same, we cannot deduce the estimate of the current node in state s from another node already visited also in state s. The visited dictionary cannot be used, but a simple alpha-beta pruning can still be used to optimize the algorithm.

The impact of the cutoff function is simple too see. The greater the maximal depth, the best moves Pacman makes, but the longer computation time it takes. The computation time is exponential in regard to the maximal depth, so we cannot increase the maximal depth too mush. On the other hand, it can be notice that for our estimate function, Pacman makes the same moves to solve the medium and large layouts for every max. depth equal or greater than 5. It shows that it's useless to take a too great maximal depth, at least for the proposed layouts. Our estimate seeks to get close to food dots. It works well for simple layouts, but the problem with that is that it doesn't take walls into account. It would pose probleme when the food is separated from Pacman by a wall. An ideal estimate would thus take wall into account. However, we didn't implement it for two reasons. The first one is that our estimate already works well in the layouts given in the context of the project. The second is that it would make the estimate more complex, and thus more computation intensive, where the estimate should be fast to compute.