INFO8006: Project 2 - Report

Siboyabasore Cédric - s175202
Mukolonga Jean-David - s170679

November 10, 2019

1 Problem statement

a. -**Initial state** : A game state s is identified by a set such as s ={Pacman position; Ghost position;Ghost direction; Boolean grid,player}.
Each layout has a different initial state:
-small : s_0 ={Pacman position; Ghost position; Ghost direction; Boolean grid,player} = {(1,3); (3,3);'Stop'; Boolean grid[x][y]= $True$ if (x,y)=(3,3), $False$ otherwise,Pacman}
-medium : s_0 ={(2,3);(4,4);'Stop'; Boolean grid[x][y]= $True$ if (x,y)=(2,4), (3,5),(5,5),(7,5),(6,6), $False$ otherwise,Pacman}
-large : s_0 = {(2,3); (3,4);'Stop'; Boolean grid[x][y]= $True$ if (x,y)=(1,2), (1,6),(5,6),(7,5),(9,1),(9,3),(15,1),(15,4),(15,6), $False$ otherwise,Pacman}
-**Player function** : $player(s)$ =1 if Pacman has the move in state s, 0 if it's the ghost that has the move in state .
-**Legal actions** : $action(s)$ = {'North','South','East','West','Stop'} if current player is Pacman not surrounded by walls and action(s) = {'North','South','East','West'} minus the previous action executed by the ghost (unless no other choice) if current player is the ghost.
-**Transition model** : $result(s, a) = s'$ where s' differs from s because of the updated Pacman and ghost positions, ghost direction and possibly the boolean grid after doing action a in state s.
-**Terminal state** : $terminal(s) = True$ when Pacman has eaten all the dots (when all the variables of the boolean grid are set to $False$) or when the ghost kills Pacman .
-**Utility function** : $utility(s, p)$ = the game score if the game ends in a terminal state s for the two players.
-**Game tree** : it is defined by the $action(s)$ and $result(s, a)$ functions. Nodes of the tree represent game states and edges represent actions.

b. Both players share the same utility function; the Pacman agent wants to maximize the the game score (utility) whereas the ghost agent wants to minimize it. If Pacman wins the game with a certain utility then the ghost loses the game with the opposite utility and vice-versa. This means

that the payoffs of the two players are always opposite and add up to a constant total payoff of 0 for all games. Therefore, the game of Pacman is a zero-sum game.

2 Implementation

a. The Minimax algorithm applied to a game is complete if the game tree is finite. With respect to the game of Pacman, the game tree (defined by the $action(s)$ and $result(s, a)$ functions previously mentioned) might not be finite because of transpositions: the agents might go to a game state they've already been in which can create infinite loops preventing the game tree from being finite.In order to guarantee the completeness of Minimax we create a list that stores the visited game states. If an agent considers going to a state it's already visited, a repulsive value for that state is returned ($-\infty$ for Pacman and ∞ for the ghost because Pacman wants to maximize the score and the ghost wants to minimize it) and the agent will just ignore that state.

b. We implemented the Minimax algorithm the way we do it in the course : we use two functions $minValue$ and $maxValue$ in order to build the game tree. In the *get_action* method we generate all the successor Pacman states of the initial state and we call the $minValue$ function on each of these states. The $minValue$ function represents the ghost's turn. It generates all the successor ghost states and call the $maxValue$ function on each of these states. It returns the smallest score returned by the $maxValue$ calls. $maxValue$ represents Pacman's turn. It generates all the successor Paccman states and call the $minValue$ functions on them. It returns the highest score returned by $minValue$
Both the $minValue$ and $maxValue$ functions maximize the utility function for its player. They call each other recursively on the successor state until one of the reached state is a terminal one. The score is then returned. The *get_action* method returns the moves that lead to that optimal state. The completeness is preserved with thanks to the list of visited states (cfr previous question).

c. The H-Minimax algorithm is mostly based on the Minimax algorithm except that H-Minimax doesn't explore the entire game tree. We stop expanding a state with a $cutOff(s, depth)$ function (with depth being the depth in the game tree) that cuts the search for the solution earlier if the condition(s) definied by this function is/are reached. We call that function at the beginning of the MaxValue and MinValue functions. When the $cutOff$ function cuts the search, an evaluation $Eval(s)$ of the utility for state s is returned(and not the actual utility which is the score). As Minimax and H-Minimax only differ because of the cutOff and evaluation functions, the terminal states in H-Minimax are ordered in the same way as in Minimax.

d. - Our H-Minimax with the best cut-off/evaluation functions combination (hminimax0) uses the evaluation function

$$\texttt{Eval}(s) = \begin{cases} -\infty & \texttt{if manhattanDistance(Pacman,Ghost) < 1} \\ -0.75 * \texttt{closestDotDistance} + 0.25 * \texttt{ manhattanDistance}(Pacman, Ghost) & \\ & \texttt{if manhattanDistance(Pacman,Ghost) < 3 and > 1} \\ \texttt{-closestDotDistance otherwise} & \end{cases}$$

If the Pacman agent executes an action that brings him to the ghost position (in which case Pacman loses), $-\infty$ is evaluated so that the Pacman agent (which wants to maximize the evaluation) ignores that state. If the ghost is near Pacman (manhattan distance of 1 or 2), the evaluation is the weight between the manhattan distance between Pacman and the ghost and the distance between Pacman and the nearests dot so that Pacman can go to the nearest dot position while remembering to avoid the ghost. If Pacman is far enough from the ghost, it goes straight to the nearest dot without thinking about the ghost.
The **cutOff(s,depth)** function stops the the expansion of a state if $depth = 1$ or if the state is a terminal state.

-Our H-Minimax with the second best cut-off/evaluation functions (hminimax1) uses the same evaluation as our best H-Minimax.

The **cutOff(s,depth)** function stops the expansion of a state when $depth = 1$ or when the state s is a terminal one. When this condition is reached, the evaluation function is returned. It also uses an algorithm that treats transpositions. The algorithm creates a dictionary that stores the already reached states in the past.

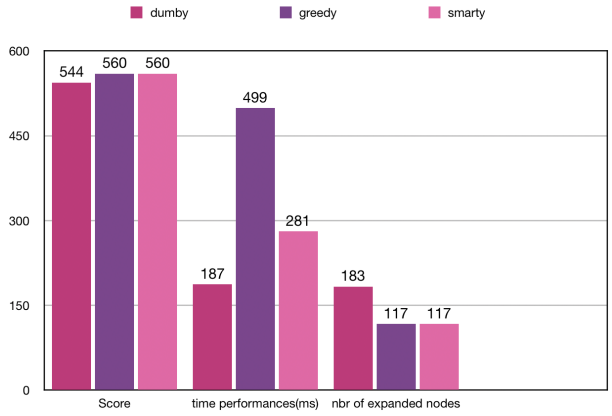
-Our H-Minimax with the third best cut-off/evaluation combination (hminimax2) uses a **cutOff(s, depth)** function that stops the expansion of a state when $depth = 3$ or when a terminal state is reached. This H-Minimax uses the evaluation function

$$\texttt{Eval}(s) = \begin{cases} -\infty & \texttt{if manhattanDistance(Pacman,Ghost) < 2} \\ -0.5 * \texttt{numFood} + 0.5 * \texttt{ manhattanDistance}(Pacman, Ghost) & \texttt{otherwise} \end{cases}$$

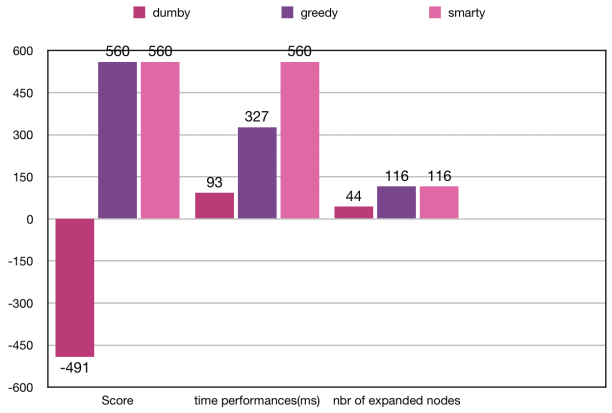
If the distance between Pacman and the ghost is less than 2 then we estimate that Pacman might lose : it might die because of its closeness to the ghost. Otherwise, the evaluation weights the opposite of the number of dots in the state and the manhattan distance between Pacman and the ghost because that's what Pacman wants to maximize (less dots and a greater distance between Pacman and the ghost).

3 Experiment

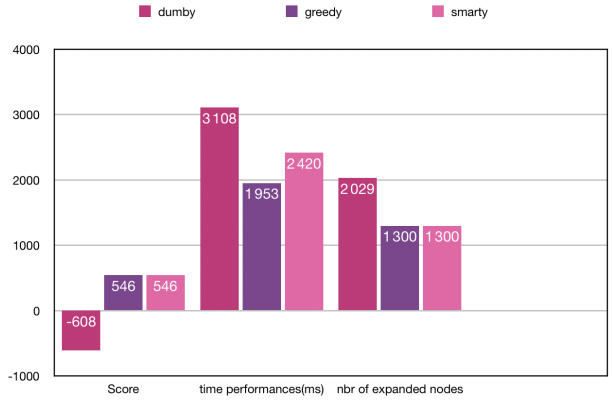
a. hminimax0 :



hminimax1:



hminimax2:



b. For the **dumby** ghost :
Pacman wins for hminimax0 but loses with the two other H-Minimax . This is due to multiple factors such as the behaviour of the ghost, the evaluation and cut-off functions choice and the way we go through the tree.
Pacman doesn't make assumptions about the strength of the ghost: Pacman assumes that the ghost plays optimally, that the ghost wants to maximize the worst-case outcome for Pacman. This is however not the case here because the dumby ghost constantly rotates on itself. Pacman might thus expand nodes in vain.
The depth choice in the cut-off functions has a huge influence on the behaviour of Pacman. If we don't look deep enough in the tree, efficient moves might not be spotted by Pacman because the solution search is cut too early. Therefore, the actions of Pacman might not be good actions and lead to losing states that we didn't know were losing states because we cut the search too early :this is the horizon effect. hminimax1 cuts the search too early ($depth = 1$) and therefore the number of expanded nodes is small (44).
The evaluation function choice is tricky because this function is always imperfect but important because Pacman wants to maximize it. The evaluation function in hminimax2 considers the number of dots and not the distance between Pacman and these dots. This is why hminimax2 gives worse score than the other heuristics.

For the **greedy** and **smarty** ghosts :
Pacman wins for all three H-Minimax. The score is however lesser for hminimax2 because we look deeper in the tree ($depth = 3$) which means that the quality of the evaluation function matters less (horizon effect). No matter the H-Minimax, the score is the same for both the greedy and smarty ghosts because Pacman assumes that the ghosts play in an optimal way. This is the case for both greedy and ghosts.

c. Pacman would have to assume that all the ghosts play in an optimal manner. Therefore, in the $MinValue$ function of minimax.py of our Minimax and H-Minimax implementations, we'd have to create a loop that takes account of all the ghosts and in that loop we'd loop on the successor states of the considered ghost and we'd call $MaxValue$ on each one of thoses states. If we still consider the game score as being the payoff of Pacman, each ghost must have a payoff of $\frac{-score}{numberofghosts}$ so that the total payoff = score + number of ghosts* $\frac{-score}{numberofghosts}$ is still constant (= 0) and the game can still be considered as a zero-sum game.