Introduction to Artificial Intelligence , Project 2

Olivier Moitroux (s152122) - Pierre Hockers (s152764)

November 22, 2018

1 Adversarial search problem

As seen in the lecture 4 of the class, a search problem is formalized by a few elements :

- **Initial state (s_0)**: A state is an association of the position of Pacman, the foodMap, and the position of the ghost. The initial state is therefore the initial position of both agents associated with the untouched foodmap corresponding to the layout.
- **Player function** : Both the ghost and pacman try to achieve their goal that are respectively: minimizing the score and maximizing it. In this way, pacman tries to maximize the score through max_value while the ghost does the opposite in min_value. Depending on which of the 2 functions is being executed we know what player is acting. The api self determines who plays: a call to get_action() means it is pacman turn, and thus, a call to max_value() is done to launch the recursion[1].
- **Legal actions from a state** : the legal movements of Pacman allowed by the game (where there are no walls, one step at a time, food automatically eaten, ...): North, West, East, South. Stop is removed. The same goes for the ghost.
- **Transition model** : updates the position of the ghost, of pacman and potentially updates the foodmap.
- **terminal test**: the function terminal_test will check wether or not the game is over, i.e. if all the food has been eaten or if the ghost catched Pacman. Note that for hminimax, terminal_test will also stop us when we reach a certain depth in the possibility tree
- **Utility function**: this function will in our case simply return the score of the situation reached. It will be replaced by a heuristic function in hminimax.

2 Performances on the small layout

All the graphs are reported in figure 1 and 2.

3 Performances and limitations of the different algorithms in diverses situations

All the graphs have been generated and averaged on 8 runs.

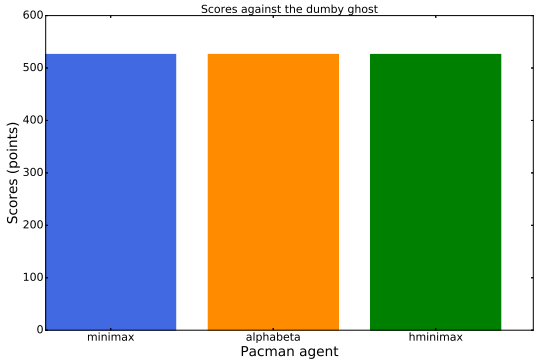
3.1 minimax

On the small layout, minimax is fast enough and optimal. The different ghosts require slightly different amount of nodes to expand : around 700 for dumby, around 650 for greedy and around 800 for smarty. The difference in time is negligible, all algorithms run in about 0.09s

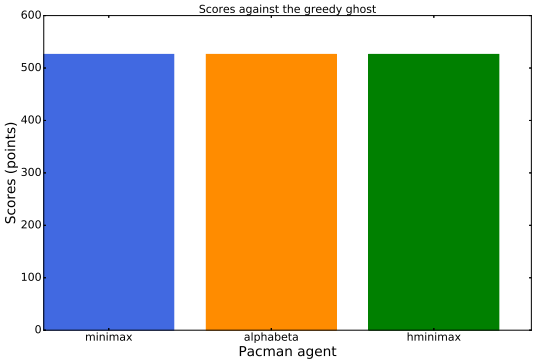
Problems arise with bigger layouts. On medium and large, we can observe that Pacman get killed for greedy and smarty, and gets stuck in a loop for dumby. This comes from the fact that the map is too

[1]Note that another version of the algorithm could call a single recursive function which, depending on a boolean, could determine what player is currently playing and therefore fit more to the definition. We chose the first version.

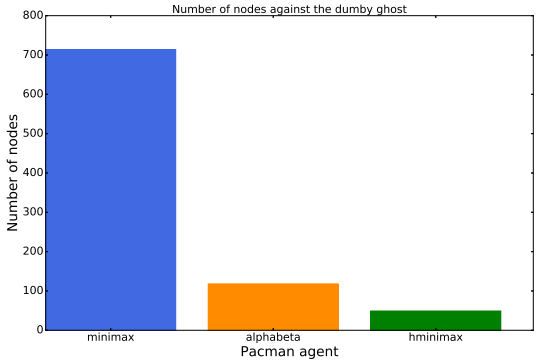
big and the recursion reaches a limit that stops it. This leads to illegal moves, and therefore Pacman stays motionless and gets killed. For dumby, Pacman doesn't get killed because he isn't being chased but he gets stuck nonetheless for the same reason. Since the problem comes from the depth of the tree to explore, we can simply limit the max depth that the code can reach. We tried this approach and the results were actually quite good on bigger maps, even if it would still fail sometimes.



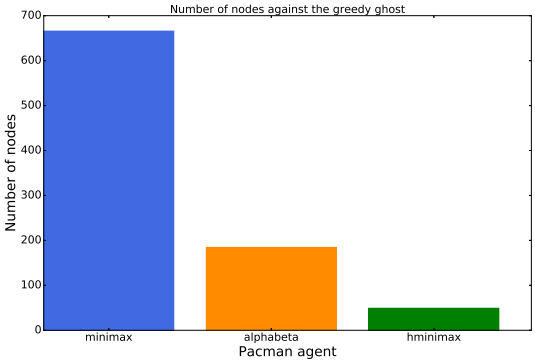
(a) **Dumby** score



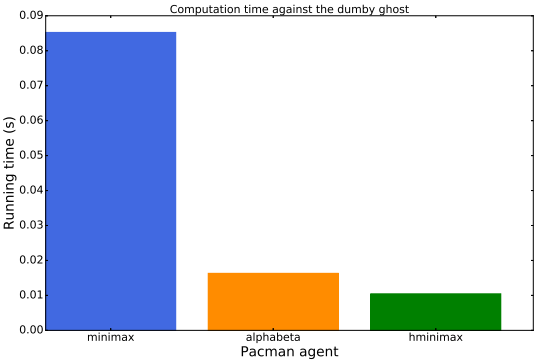
(b) **Greedy** score



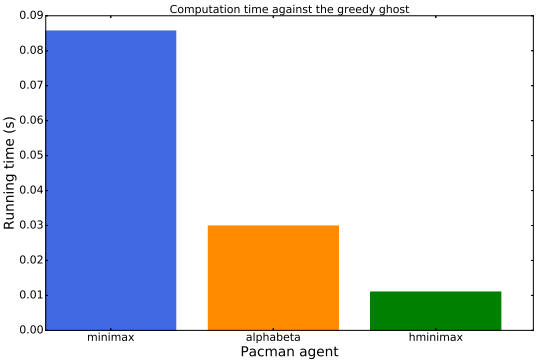
(c) **Dumby** node



(d) **Greedy** node

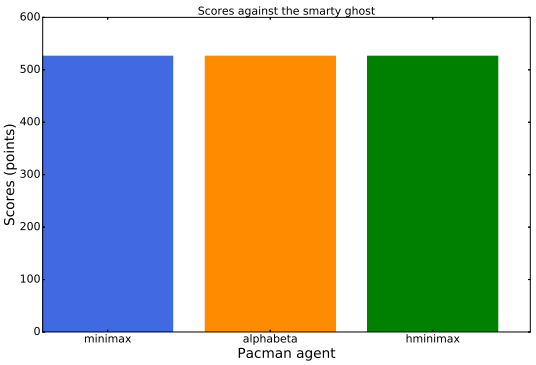


(e) **Dumby** time

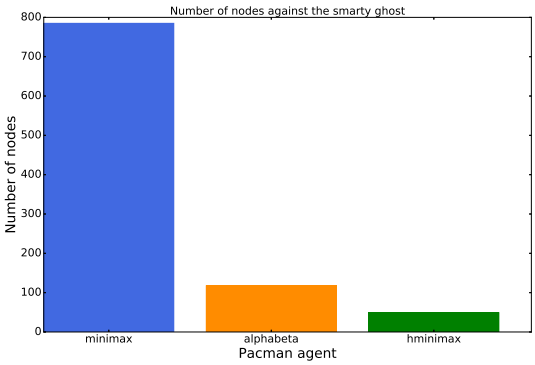


(f) **Greedy** time

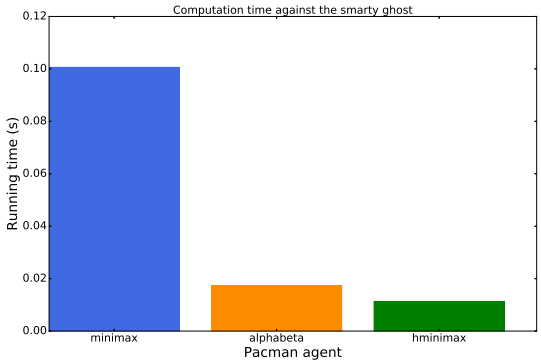
Figure 1: Dumby and greedy results



(a) **Smarty** score



(b) **Smarty** node



(c) **Smarty** time

Figure 2: Smarty results

3.2 alphabeta

On the small layout, alpha beta is doing very well : it is optimal, expands way less nodes than minimax (around 64), and therefore is computed way faster (around 0.02s for all the ghosts, greedy being a touch slower). On the two other layouts, it has the same problems as minimax as the number of recursion is still too high.

3.3 hminimax

3.3.1 Results

We have experimented a lot to find a good heuristic but couldn't find a linear combination of the "game state" information that would allow pacman to win in an optimal way in every situation. However, we designed 2 heuristics that succeeded to do so when coupled with the right depth. While there was never an heuristic/depth pair that worked for all ghosts/layout combination, we could always find one heuristics/depth pair that would work well on it. We have retranscripted the combinations that gave the best results in the following table 1:

Layout	**Dumby**	**Greedy**	**Smarty**
small_adv	`evalFct`(depth=3)	`evalFct`(depth=3)	`evalFct`(depth=3)
medium_adv	`evalFct`(depth=1)	`evalFct`(depth=1 or 3)	`evalFct`(depth=1 or 3)
large_adv	`evalFct2`(depth=3) ♣	`evalFct`(depth=1)	`evalFct`(depth=1)

Table 1: Best evaluation function and associated max depth for each configuration

Let's note that, in the situation ♣, Peter's computer can also produce an optimal result with `evalFct`(depth=1) while Oliver's computer requires `evalFct`(depth=4). Both runs ok with `evalFct2` and a maximum depth of 3. Actually, it turns out that Oliver's computer is acting differently than all

the other computers we have tried the algorithms on (the difference being that the order of the legal directions given by the pacman api is completely random on Oliver's computer and not on the other ones[2]). All in all, `evalFct` with depth 1 is the way to go but the depth needs to be changed for the small layout to 3.

3.3.2 Design and implementation of the evaluation function

To design `evalFct` and `evalFct2` we followed the following guidelines : we tried to use data that was a good indicator of an advantageous situation potentially leading to a win, and we avoided any costly operation to keep a fast runtime. With that in mind, we used data such as the real minimum distance from pacman to a food. We did this using Lee algorithm and not the Manhattan distance to avoid dumby to loop blindly forever on the large layout. This is more costly to compute than the Manhattan one but it is a more precise information. Improvements could be done to compute the real distance more effectively (a basic cut-off has been added).

We always take into account the score (notion of time passing by), of course, and give a bonus for winning. In `evalFct2` we divide a factor by the number of food left so that there is a real interest in eating food, and to push pacman to actually finish the game. We indeed had a lot of issues with pacmans being very good but somehow not eating the very last food cap. In order to have a bigger impact on the score when there was not much food left, we used the division.

In previous tries we also used malus in case of loosing, or in case of a ghost that would be too close but without good results. We also tried other data like the number of food left, the Manhattan distance to foods, etc.. But in the end, `evalFct2` and `evalFct` both work better without using these features.

3.3.3 Discussion on the results

On the small layout, hminimax is optimal and needs less nodes to compute (49). This is exactly what we expected. This obviously leads to a faster computation time even though the gain is less pronounced as the evaluation function takes more time to compute.

On medium, pacman manages to win everytime in an optimal way against all ghosts but we had to change the maximum allowed depth to 1 for dumby. Because the depth is small, the cut-off arise soon and the computation time is very low (from 0.055s for dumby to 0.085 for smarty). The number of nodes stays around 40. Depth 3 does not work for dumby as we assume the ghost is optimal while it is not.

On the large layout, pacman wins against dumby, but very slowly (14.65s of computation with 6526 nodes) in a non optimal way (540). The real maze distance is not enough to determine the optimal path (he forgets the upper left food in the beginning f.e.). Moreover, our implementation of the real_minimum_food_distance() is heavy (Lee algorithm with cut-off) and it has a bigger impact with a larger depth. However, like the remark we made (♣), on Peter's computer, pacman always win with a score of 556 with `evalFct` and depth = 1 in ≈0.257 seconds.

4 About the code

In order to memorize the already explored state, we decided to use a set called `visited`. It is a class variable which is cleared at the root before exploring another branch. For this system to work, we also use a dictionary that store the score associated to a given state (=key). In this way, when pacman is "playing", a new state is considered as visited only if it is in `visited` **AND** if the old score is less interesting than the new one. Otherwise, it is not worth exploring because we already reached this state before with a better score. Another solution would have been to pass a `visited` set as an argument to `min_value` and `max_value` and to build it recursively via a copy each time. We expect this solution to be quite slow and therefore preferred to use our method.

[2]Only difference apart hardware: python 3.5.2 and not python 3.7 installed via anaconda.