PRACTICE 2:
Reinforcement learning

Made by:
María Ángeles Magro Garrote (100472867)
Marina Gómez Rey (100472836)

Index

INTRODUCTION

In this assignment we are asked to apply a Q-learning algorithm to the Pac-man game in order to create an automatic agent that automatically works in a variety of mazes with different characteristics. On top of that, the aim is to maximize the reward taking into account the structure of the map and the elements present on it.

In order to do this there are several tasks that have to be done, related to programming some functions as well as designing rewards, building an agent and training it, everything distributed in two different phases.

PHASE 1

State tuple

In this phase we are asked to create the state tuple that includes attributes we consider relevant for the Pac-man to make decisions.
Firstly, based on the idea we followed in the previous practice and in order to have the most simple tuple, we thought about creating only one attribute based on the distances to the closest ghost taking Distancer distance, that takes into account the walls. The idea behind this was computing the distance from the next tick to the closest ghost in all possible future positions, so the result is the minimum one. That would lead to four possible results that are the four directions and the one selected should be the one taken into account.

However, after having done this, we realized that we needed another variable in order to be able to have a track about the distance because if not, our Pac-man could not learn that going further from the ghost is not good. For that reason we added a discretization of the distance in four intervals: close, mid-close, mid-far and far.

We wanted these variables to be useful for any map so the first approach of the discretization was to divide the map in four having done the mean between the width and height but we saw that this was not a good approach as it was always close with these distances, so we took absolute values instead. These absolute values had a difference of three units each, it was close being between 0 and 3, mid-close between 3 and 6, mid-far between 6 and 9 and far larger than 9.

Reward function

The reward function was a main issue in this practice. Our first idea was to keep giving rewards all the time (positive and negative ones) based on the following different cases:

- On one hand, if the distance to the ghost increased, a negative reward (-10) was returned.
- On the other hand, if the distance was reduced, the reward was positive (+10).
- If a ghost was eaten a huge reward was returned (+50)
- If a pac-dot was eaten, the reward was +25.

After trying to make the pacman learn (in the first map) with this reward function we saw that it was not working as it did not learn and did not approach the ghost because too many rewards were given. For that reason, we decided to start again from the beginning with a different approach.

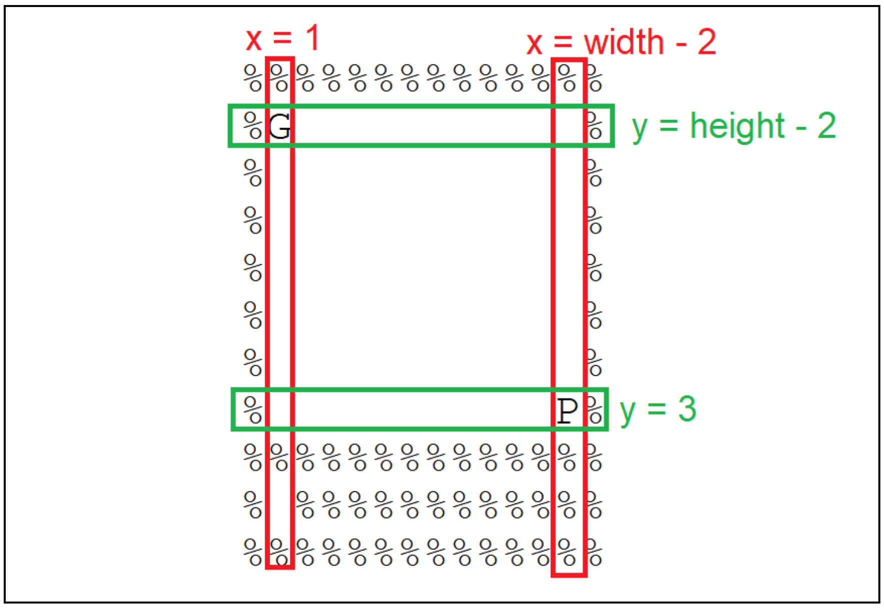
First of all, as our goal was that the Pacman should learn that eating both a ghost or a dot was a positive thing, we checked when this happened: if the score increased 99 units in the next state, it meant that a pac dot has been eaten, if the same happened but with a score 199 points higher, a ghost was eliminated. Furthermore, eating a dot was more rewarding than eating a ghost as all dots should be eaten before the game ends (the game ends when all ghosts are dead).

After this change we trained Pac-man again and this time, it learned the way to eat the ghost in both the first and the second map in a very efficient way. However, when trying the third map, our agent kept hitting the wall that separated it from the final ghost. With this in mind, we decided to include something that would prevent Pac-man from getting stuck in the walls.

Our idea was that if Pac-man was facing the direction of a wall, the reward was reduced in -10 units. In order to do that, the following should be checked: if the action taken is not in the legal actions of the next state, it means that we faced a wall.

However, we also had to introduce some changes to this first idea, as trying this approach led to the Pac-man avoiding the limits of the map and , consequently, the ghosts that were placed very close to them. For that reason, we implemented a condition where if the Pac-man was facing the limits of the map this negative reward was not given. So, only facing a wall that was inside the map was penalized. It was done this way:

We noticed that all maps followed the same structure so we needed to know the coordinates of the limit that was transitable for Pacman. The following images displays the displacement of the grid and the location of the borders:



Borders of the standard map

With this information, to the previous condition of the legal actions, others were added so that if the Pacman was colliding with a wall that was part of the border, it was not penalized.

With this reward function we finally arrived at a solution where the Pac-man was able to learn how to get to the ghosts positions and eat them without getting stuck in a very efficient way. Nevertheless, the score was still not optimum as the pac dots were not taken into account so we thought of a way to include them.

As one of the attributes studied was the direction that should be taken to arrive at the nearest ghost, the coordinates of the pac dots were included. A method was created that was called closestElement() so that it was easier to compute the direction which returned us the closest element including ghosts and dots.

PHASE 2

In this phase the agent has to be implemented using the methods, q-table and finally training it using them.

General methods

In this part we are going to explain the methods that are needed in every reinforcement learning project.

- update

The implemented update method was very similar to the one computed in Tutorial 4. In order to change the value of the correct cell of the q-table the correct values of the row and column were taken with the computePosition function and the action taken in that tick. Also, we took the q-value with the getQValue function and the best q-value for the next state using computeValueFromQValues. Finally, the corresponding cell is updated using the following formula: *self.q_table[position][action_column] = (1-self.alpha) * q + self.alpha*(reward + self.discount * max_q_nextstate)*.

- getReward

The getReward method aims to implement the reward function explained before in phase 1. Firstly, we introduce the reward variable with an initial value of 0, which would be the value returned if none of the following conditions are met:

1. If the Pacman faces a wall, -10 will be subtracted to the reward. As mentioned before, in order to do that, the action should not be in the legal actions of the next state and the pacman should not be in a border.
2. If the score of the next state has increased 199 points, a ghost has been eaten and the reward will be increased 50 points.
3. If the score of the next state has increased 99 points, a pac dot has been eaten and the score will be increased twice more than if a ghost has been eaten (100 points). This is done to prioritize that all dots are eaten before the game finishes.

- computePosition

This method aims to get the number of the row of the q-table we are at on the current tick.

In order to create this method we took into account the variable that represent our state that, as explained before, are the relative positions of the closest ghost with respect to the agent (represented as the four coordinates: North, South, East, West) and the discretize distance to it (in four intervals), which leads to 16 rows in the q-table with the combinations of all the values.

To get the number of the row we first create a list called attributes where the values of the two attributes of the state tuple are kept. Firstly, the direction has to be analyzed. Our approach was to compute the distances from the Pac-man to the closest element (whether pac-dot or ghost) using the closestElement method that we will discuss in detail later. Then, we compute the distance between it and our agent four times, one for each possible direction, so it is the four possible distances in the future. After that, we check which is the smallest one and append a different value according to this result to the tuple “attributes”. North is 0, South is 1, East is 2 and West is 3. Now, we have the first value of our tuple.

After that, we discretize the actual distance to the closest element. If it is close (between 0 and 3), we append a 0; if it is mid-close (between 3 and 6), we append a 1; if it is mid-far (between 6 and 9), we append a 2 and finally, if it is far (more than 9), a 3 is appended.

Finally, to get the number of the row, the following operation is computed: *attributes[0] * 4 + attributes[1]*.

Extra implemented methods

Apart from the previous implemented methods, we created some more that were necessary in order to make our agent work correctly.

- countFood

This method, as its name explains, is used to count the number of pac dots that are left in the grid. It is used to know if there is any food left in order to add the coordinates to the remaining elements Pac-man must eat. This method was already present in other agents so we copied it here in order to be able to use it.

- closestElement

This method is crucial to be able to compute the distances between Pac-man and the ghosts and pac dots and choose which is the closest one so that Pac-man can know where to go and to compute the correct value of the row of the q-table.

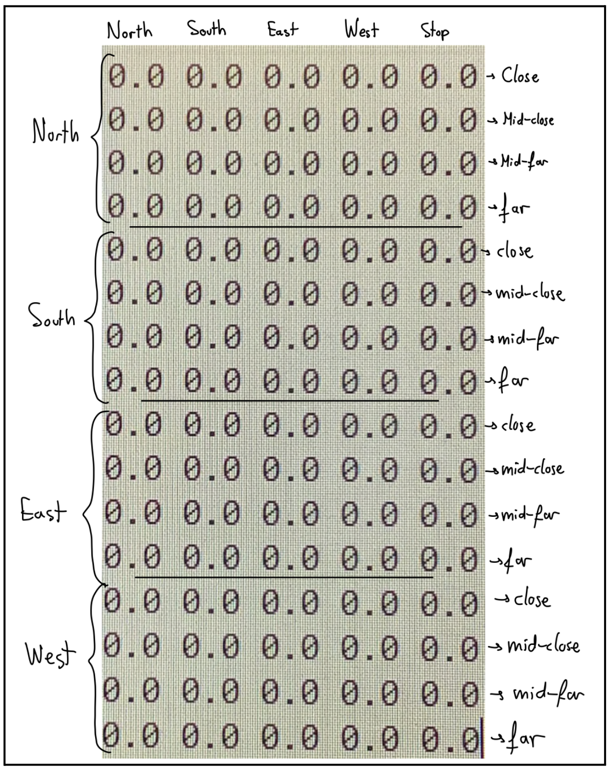
First, we check using countFood to know if there are pac dots left in the map- This is done because if there are, then we should compute the positions of the pac dots. If there are not pac dots, we should go straight to the ghost positions.

If the number of the food is bigger than 0, then, an algorithm is done to compute their positions. We create a list where the coordinates are going to be saved and we call the getFood method, which returns as a list the map with a T in the coordinates where there are dots, and a F elsewhere. Then, we check this matrix cell by cell with a loop and with the hasFood method, we check if there is a pac dot there. If this is the case, we append these coordinates to the list.

Now that we have the pac dots, the coordinates of the ghosts are taken easily with the getGhostPositions method that are included in a list with the pac dots ones.

Finally, what we have to do is computing the distances between all these coordinates and the actual Pac man position (getPacmanPosition) and the smallest one is finally returned as the closest element.

Q-table



	North	South	East	West	Stop	
North	0.0	0.0	0.0	0.0	0.0	→ Close
	0.0	0.0	0.0	0.0	0.0	→ Mid-close
	0.0	0.0	0.0	0.0	0.0	→ Mid-far
	0.0	0.0	0.0	0.0	0.0	→ far
South	0.0	0.0	0.0	0.0	0.0	→ close
	0.0	0.0	0.0	0.0	0.0	→ mid-close
	0.0	0.0	0.0	0.0	0.0	→ mid-far
	0.0	0.0	0.0	0.0	0.0	→ far
East	0.0	0.0	0.0	0.0	0.0	→ close
	0.0	0.0	0.0	0.0	0.0	→ mid-close
	0.0	0.0	0.0	0.0	0.0	→ mid-far
	0.0	0.0	0.0	0.0	0.0	→ far
West	0.0	0.0	0.0	0.0	0.0	→ close
	0.0	0.0	0.0	0.0	0.0	→ mid-close
	0.0	0.0	0.0	0.0	0.0	→ mid-far
	0.0	0.0	0.0	0.0	0.0	→ far

Representation of the used q-table

As we have been explaining throughout the document, our q-table has 16 rows and 5 columns (that correspond to the 5 possible movements Pac-man can compute: North, South, East, West and Stop). However, Stop is never taken so its value stays as 0.0 forever.

Then the rows have the attributes of the coordinates, that represent in which direction is the closest element; and the discretized distance, again to the closest element.

As a result, we obtained the q-table that is visible in the left image, in this case, empty.

Training

The training was done in the order of the maps provided, as they go from the easiest to the most complicated one.

Firstly, the values of the parameters had to be selected so that the learnt information is maximum. The parameters that have to be defined are: epsilon, which is the probability that the agent, instead of doing the optimal movement makes a random one so that new possibilities and paces are taken into account; alpha, which is the learning rate, it controls the degree to which newly acquired information overrides old information and finally; the discount, which is the weight given to future rewards in the agent's decision-making process. To this last parameter we gave a value 0,95 so the new information is relevant.

The other two parameters had to be changed depending on the moment we were training. When we wanted the q-table to be updated, alpha was set to 0,15 so that the values were changed but the previous information was still kept. The value of epsilon had to keep changing depending on the moment of the training. When the map had not ever been tried, this value was 1 so that the movements were completely random and it could learn the maximum possible information traversing through the grid. After that, we kept reducing it so that it also learnt new information more related to the optimal policy.

Finally, we checked that the agent worked on every map provided for this lab until all were successfully completed. Then, we also tested new maps which have not been used for the training of the q-table and again, the game is completed again in the most optimal way.

FINAL RESULTS

After the final result was obtained, the scores of the games in the different maps (with alpha and epsilon as 0.0) can be seen in the following table as well as the number of ticks to finish it:

Map	1	2	3	4	5
Score	183	383	569	763	1238
# Ticks	17	17	31	37	62

Apart from this quantitative results, the behavior is very good as the paths taken are optimal and it does not take any unnecessary step, as it goes directly to its objective (whether closest dot or ghost)

CONCLUSION

This practice involved a lot of processes related to reinforcement learning in order to create an automatic Pac-man that maximized the obtained score. In order to fulfill this task our final model has different characteristics.

First, the state tuple which is used to identify the situation of the game. This tuple has two attributes: the direction in which the closest element (ghost or pac dot) is at and the discretized distance based on if it is close, mid-close, mid-far or far.

Furthermore, our q-table is generated initially with 16 rows as our two attributes have 4 different possible values. So the total number of possible combinations is 16 states. Also, there are five columns that correspond to the different actions the agent can take (Stop remains at 0 always).

What is more, the definition of alpha, epsilon and discount were studied so that the best value could be setted when testing. Specifically, the value of epsilon was changed constantly depending on the phase of learning the Pacman was (the first time it was set in 1 and then it kept decreasing).

On top of that, thanks to this project, we gained acknowledgement about how the reward function works and that not every tick should be rewarded (this was one of our main issues). In fact, our reward function is much simpler than the initial one, which worked less efficiently. Our actual function only rewards when something is eaten (it gives more reward to the pac dots so it prioritizes it as the game finishes when the ghosts are eaten) and only penalizes when a wall inside the grid is faced (with the exception of the borders) so that it does not get stuck in any place.

Finally, to get these previous ideas done, we had to code some methods inside the QLearningAgent class such as: update, where the q-table is updated in every tick depending on the reward and the previous value; getReward, where the reward value was returned based on the previous idea; and computePosition, where the row of the q-table we are at right now is computed based on the attributes present on the q-table (closest element direction and distance). Regardless of them, we included two extra methods that were necessary to compute the previous ones: countFood, that returns the number of pac dots that are left in the grid, and closestElement, which computes the coordinates of the pac dots and the ghosts and calculates the Distances distance to them to return the smallest one.

Having taken everything into consideration, we have learnt q-learning efficiently throughout this game. In addition to this, this model is much better than the classification Pacman model, which did not succeed in eating all ghosts. This model not only does that, but it also eats all the pac dots in the most optimal way and gets very huge final scores.