Assignment 2

Source: https://en.wikipedia.org/wiki/Q-learning

$$Q^{new}(s_t,a_t) \leftarrow (1-\alpha)\cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} \bigg)}^{\text{learned value}}$$

Iker Rosales Saiz - 100475397
Enrique Colmenarejo Ortega - 100472767

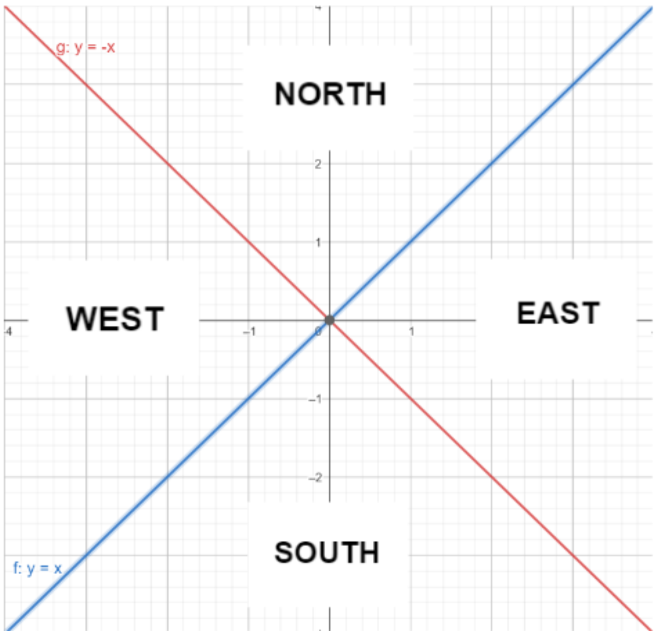
Contenido

INTRODUCTION

Q-learning is a reinforcement learning algorithm used to learn the optimal action-selection policy for an agent in a *Markov Decision Process* . In simple terms, it is a technique for finding the best possible action to take in a given situation to maximize the long-term rewards.
In Q-learning, the agent learns a function called the Q-function or the Q-value. The Q-function takes a state-action pair as input and outputs the expected cumulative reward that the agent can obtain by taking that action in that state and following the optimal policy thereafter. The optimal policy is the one that maximizes the expected cumulative reward.
The Q-learning algorithm works by maintaining a table, called the Q-table, that stores the expected discounted reward for each possible action in each possible state. The expected discounted reward is often referred to as the Q-value. The Q-table is initialized to all zeros, and the agent interacts with the environment by taking actions and receiving rewards. During the learning process, the agent updates the Q-table based on the observed rewards and the new state that it transitions to after taking an action. The goal is to learn the optimal policy, which is the sequence of actions that maximizes the expected discounted reward over time.
One of the key advantages of Q-learning is that it can learn optimal policies in environments with large or continuous state spaces. However, it can also suffer from the "curse of dimensionality," as the size of the Q-table grows exponentially with the number of states and actions. On the whole, Q-learning is a powerful and widely used algorithm in reinforcement learning that allows agents to learn optimal policies in complex environments without explicit knowledge of the environment we are dealing with.

PHASE 1

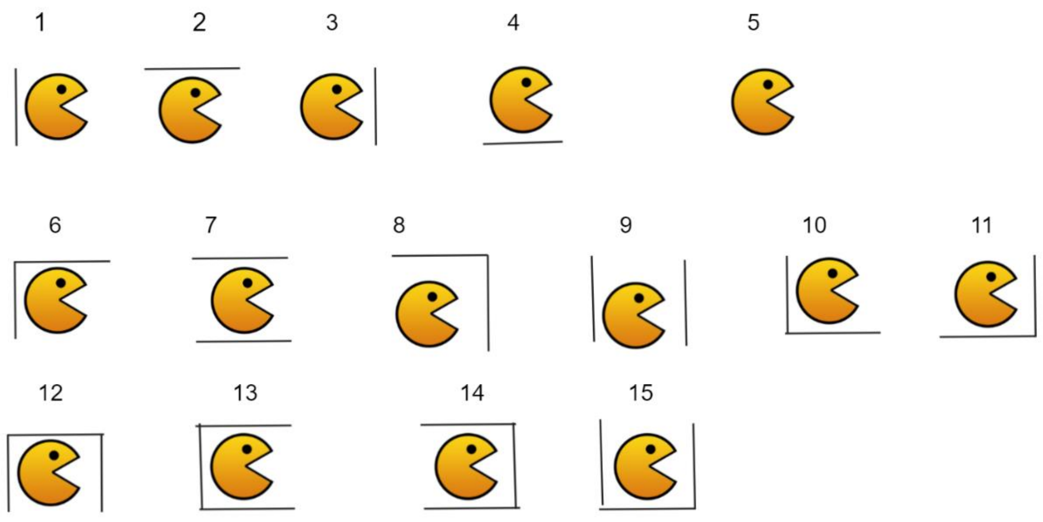
One of the main tasks we had to perform was to decide the selection of states that would satisfy our needs in order to achieve our goal. This, obviously, wasn't easy and therefore we had to make different approaches until finally reaching the final selection.
Our first approach was selecting a few states in order to find a simple but effective selection. This first thought was to make use of some axis built with the functions *f: y =x* and *g: y = -x* building four regions which were *north, south, west* and *east.*



Once we computed this, we realized that it wasn't enough and therefore we decide to add to the previous idea, the states not only from north to south and west to east bust also the straights and also, we realized that it was important to focus on the legal actions as the walls could be something that stopped our Pac-Man from learning properly. The approach we gave was considering all the possible options regarding walls. The possibilities were:

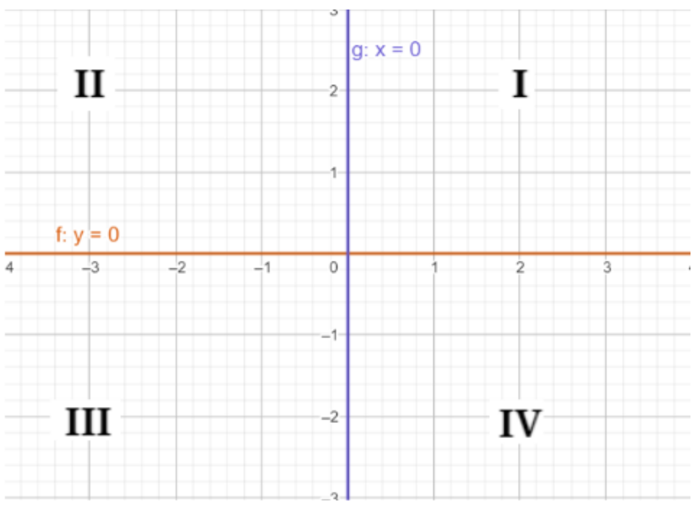
- One wall in each direction,
- Two walls which gave six new states.
- Three walls.
- (We didn't consider the four walls option as it didn't make sense for us as it is not a possible case and if it did happen nothing could be done).

In total, **15 possibilities**.



With this approach we felt like we had found the key to success but after testing it, we realized that it wasn't as good as we thought it will be, although it did learnt quite well in the different maps, what it learnt from map was a complete mess in other one and therefore it wasn't what we were looking for.

As this wasn't working for us, we decided to change our main focus and instead of the previous approach, we decided to use the known quadrants created when dividing the map, as shown in the picture.
Additionally, we also used as states the axis of the map, creating four new states with *x = 0* with *y, positive* and *negative* and also with *y = 0* and *x, positive* and *negative.*
With this new approach, the results were much better and the agent was capable of learning in a very efficient way in most of the maps.



Although this approach was much more effective, we still had an issue with the map number 5 when the ghosts weren't random but the improvement was enormous and therefore we decided that this approach added to the use of a function that calculates the distance to the ghosts taking into account the walls which was very useful for us. The development of this, ended up by creating 120 states because every time we added one, all the **possible combinations** of the rest of the states with this new state were also created.

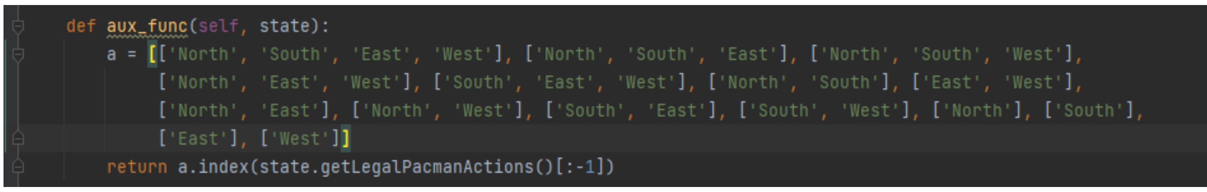
We really know that this state's approach can be improved in a way that is more efficient. Since it is easy to see how most of the states are even unreached with these 5 maps given. Anyway, we thought that it was an approach that was full of common sense since not all the maps given in the Pacman game are going to be like this, and we think that it is better that this Q-Table should be prepared for other maps.

PHASE 2

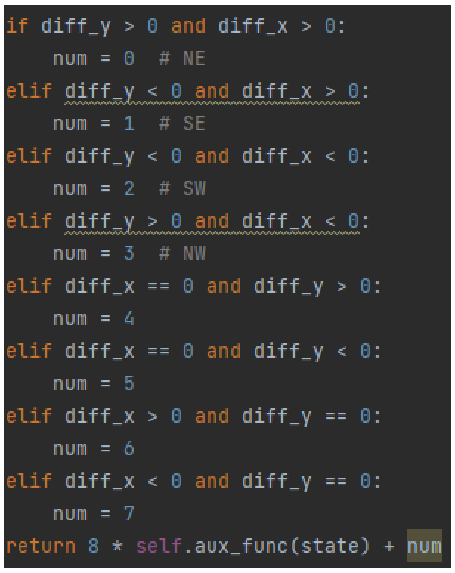
How did we move these ideas to Pycharm?

All these states of conception were needed to code it in the Q-Learning class. Precisely in the ***computePosition*** method,which is the one that given the state of the game should give the state(row in the Q-table) we are.

By extracting the pacman position and the ghost nearest position and subtracting them we were able to split the map in the 8 regions explained before by the relative position within the nearest ghost. The nearest ghost could be chosen by the design of a function that extracted the index position.



```
def aux_func(self, state):
    a = [['North', 'South', 'East', 'West'], ['North', 'South', 'East'], ['North', 'South', 'West'],
         ['North', 'East', 'West'], ['South', 'East', 'West'], ['North', 'South'], ['East', 'West'],
         ['North', 'East'], ['North', 'West'], ['South', 'East'], ['South', 'West'], ['North'], ['South'],
         ['East'], ['West']]
    return a.index(state.getLegalPacmanActions()[:-1])
```



```
if diff_y > 0 and diff_x > 0:
    num = 0  # NE
elif diff_y < 0 and diff_x > 0:
    num = 1  # SE
elif diff_y < 0 and diff_x < 0:
    num = 2  # SW
elif diff_y > 0 and diff_x < 0:
    num = 3  # NW
elif diff_x == 0 and diff_y > 0:
    num = 4
elif diff_x == 0 and diff_y < 0:
    num = 5
elif diff_x > 0 and diff_y == 0:
    num = 6
elif diff_x < 0 and diff_y == 0:
    num = 7
return 8 * self.aux_func(state) + num
```

Furthermore, it was required to know the position in respect to the wall so we had to extract a list of all the possible combinations of walls and given a determined state of the game, to return which was the index in that list.

Once the index was returned, we set that index multiplied by 8 since it has to cover all the possible combinations within the relative position pacman-ghost, and we added the index of the map regions.

Finally we covered the 120 lines of the Q-table.

Reward function

Not only the State ideas were important for the behavior of the Pacman over the game, but also the reward it was being given over every action performed in each state. This reward is in fact, one of the main keys of the Q-learning approach, since it will be the guide to the optimal policy that our Pacman will take.

By giving a high-reward to pac-dots eating, we would encourage our agent to move toward these eating points, however, if we penalize taking illegal actions and we encourage eating ghosts, we would converge our optimal policy to the persecution of ghosts.

The first approach we thought was by giving no reward until a ghost was eaten, but this procedure lasted for a long time to converge even with a high discount factor. Although this took a long time playing by itself, it didn't converge that well as expected.

So this led us to think that we should give a reward by performing good, that is to say, getting closer to the ghost and not performing illegal actions(although this was going to be corrected by our own Busters algorithm). So we add these conditions, with a high reward by getting near to it and penalizing the agent for getting far away and taking illegal actions. This reward helped to converge in a better way than before, but we were still really far from what we expected.

We noticed that the pacman was getting stuck in several determined positions along the different maps so we decided to create a list (inspired in our Tutorial1) that takes all the positions steeped by the pacman until a ghost was eaten, so in this way we could penalize when the pacman takes a position that was already visited over the ghost persecution.

When we were almost finished we noticed the method getDistance() of the self.distancer, that calculated the distances taking into account the walls. So this was finally the element that made it successfully converge in every map of the game.

The final reward was:

$$\text{Reward} \begin{cases} +5 & \text{if} \quad \text{distance} < \text{prev\_distance} \\ -3 & \text{if} \quad \text{cell visited} \\ -5 & \text{if} \quad \text{action not LEGAL} \\ -5 & \underline{\text{otherwise}}\ (\text{distance} > \text{prev\_distance}) \end{cases}$$

Updating the Q-table

Once all the previous steps are done, we could analyze in detail how the Q-table is updated and moreover, what are the tuning parameters that we should fit and take into account regarding the formula used to update the Q-Values.

The formula used, is the following:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a')]$$

This equation is the Stochastic Update Equation, and it shows how the q value associated with an action is updated in each iteration in which that action is taken.

It is important to remark how it does have in mind the previous Q-value associated with a weight of (1-α) and how the new value is updated with the other part of the equation with weight α. This α is indeed known as the "Learning rate", and it does make sense since the higher we select our alpha, the more we would take into account the reward in comparison with the previous Q-value associated.

It is important to emphasize the immediate reward, which is referred to with the letter "r" that is accompanied by the long term reward.

This long term reward selects which is the maximum Q-value of the following state so it considers the following state rewards for noticing if we are going the right way or we are not taking the good path. This long-term reward is penalized with a discount factor γ. It is simple to see that the higher the γ the more we would take into account the future rewards.

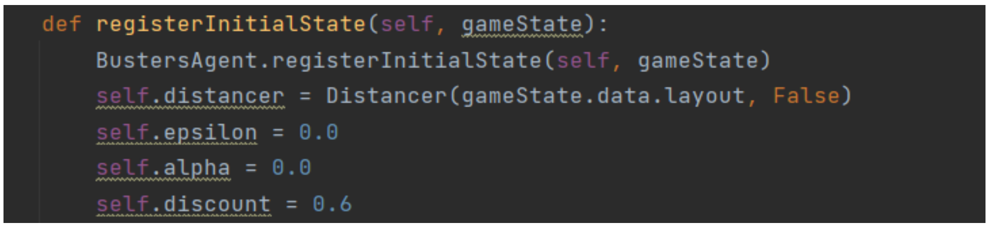
Last but not least, it must be clear that this formula can only be used when we are NOT in a final state, since in a final state we do not have the long term reward so the formula will be changed to:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \cancel{\max_{a'} Q(s',a')}]$$

By this interpretation we should then select the parameters that are going to be used in the update function.

Epsilon-Greedy

Before explaining all the combination of parameters, it should be introduced the effect of the Epsilon-greedy algorithm.
This algorithm is made to try to explore as many states as the agent can, by taking random actions with a



```
def registerInitialState(self, gameState):
    BustersAgent.registerInitialState(self, gameState)
    self.distancer = Distancer(gameState.data.layout, False)
    self.epsilon = 0.0
    self.alpha = 0.0
    self.discount = 0.6
```

probability ε so the table can converge and reach the ghosts in the least time possible.

To do this we made use of the code that already existed in *bustersAgents.py* in `def registerInitialState(self, gameState):`. The way this works is simple, we modify the value of epsilon depending on how much emphasis we want to put in the q-table. Basically if we increase epsilon, we increase the percentage of decisions taken randomly which can be useful in a task where we don't want to focus as much on the value of the q-table and we want to explore more. When trying to learn this can be very interesting as we can find new values relevant for the design of our q-table, but with a high value for the epsilon, we wouldn't be able to build a real policy as it would be taking a percentage more or less high of random actions. Therefore, if we want to show how good our q-table performs, we would implement an epsilon = *0.0* which makes no random decisions, and all of them are taken by the values in the q-table.

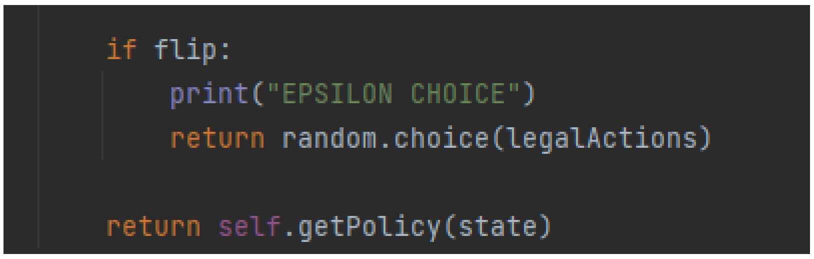
To start the algorithm, we considered high values of α and ε and a fixed γ = 0.6 so we could learn in a quick way the Q-values of Pacman. By many games in different maps we began watching the Q-table converge.

But taking a look on the Q-table.txt, some of the rows were even unvisited, but as explained in the Phase 1, this was not that bad since we have states in which only one action can be taken or even it is really difficult to reach them.

Once we have performed many initial games in several maps and with different conditions (such as RandomGhost, # of ghosts = 3,...) we shifted the epsilon to low values so randomness is not that frequent and α to low values so it does not learn that much now, and to see how it performs but always with some knowledge acquisition.

From there, it was where we could really see how the Pacman Agent started following the agents in a more direct way(taking into account some random movements).

Regardless of the good performance, we were noticed by some strange momentaneous movements so we thought they were performed by the ε-greedy. But to make sure of our assumptions we decided to print on the terminal whenever a random choice was made, so we include a printing command into the ***getAction()*** method.



```
if flip:
    print("EPSILON CHOICE")
    return random.choice(legalActions)

return self.getPolicy(state)
```

Once the Qtable converged into the expected optimal policies(the Q-values remain similar and actions taken by the pacman did not change considerably) we could finally say that our pacman Q-learning algorithm was ready to be tested without taking any random movement.

PERFORMANCE and ANALYSIS

In the performance of the Pacman Agent, we set epsilon and alpha to 0, and then we could see how it worked.
We decided to test each map by its own and we obtained the following results:

GHOST TYPE	MAPA	NUM_GHOST	EPSILON	SCORE
RANDOM GHOST	1	1	0.0	179,8
RANDOM GHOST	1	1	0.1	181
RANDOM GHOST	1	1	0.3	177,8
RANDOM GHOST	2	2	0.0	379,8
RANDOM GHOST	2	2	0.1	369,4
RANDOM GHOST	2	2	0.3	372,2
RANDOM GHOST	3	2	0.0	337
RANDOM GHOST	3	3	0.0	542
RANDOM GHOST	3	2	0.1	349,6
RANDOM GHOST	3	3	0.1	550
RANDOM GHOST	3	2	0.3	339,8
RANDOM GHOST	3	3	0.3	487,6
RANDOM GHOST	4	2	0.0	333
RANDOM GHOST	4	3	0.0	565
RANDOM GHOST	4	2	0.1	369,8
RANDOM GHOST	4	3	0.1	541,3
RANDOM GHOST	4	2	0.3	362,8
RANDOM GHOST	4	3	0.3	552,6
RANDOM GHOST	5	3	0.0	551,3
RANDOM GHOST	5	3	0.1	531,3
RANDOM GHOST	5	3	0.3	520

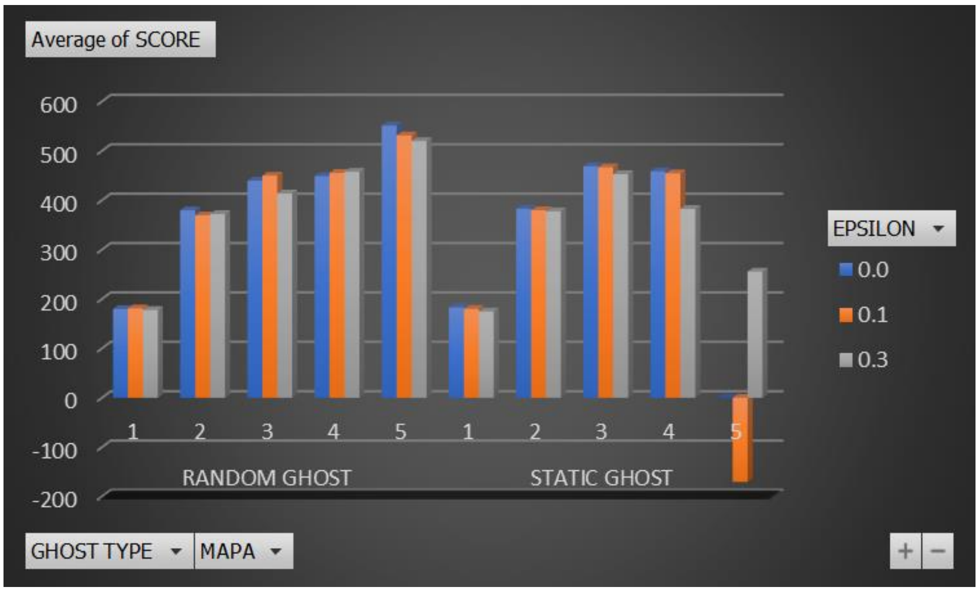
GHOST TYPE	MAPA	NUM_GHOST	EPSILON	SCORE
STATIC GHOST	1	1	0.0	183
STATIC GHOST	1	1	0.1	180,2
STATIC GHOST	1	1	0.3	175
STATIC GHOST	2	2	0.0	383
STATIC GHOST	2	2	0.1	380
STATIC GHOST	2	2	0.3	377,8
STATIC GHOST	3	2	0.0	369
STATIC GHOST	3	3	0.0	569
STATIC GHOST	3	2	0.1	368,2
STATIC GHOST	3	3	0.1	565,8
STATIC GHOST	3	2	0.3	352,2
STATIC GHOST	3	3	0.3	553,4
STATIC GHOST	4	2	0.0	371
STATIC GHOST	4	3	0.0	546,6
STATIC GHOST	4	2	0.1	365,8
STATIC GHOST	4	3	0.1	543
STATIC GHOST	4	2	0.3	351,8
STATIC GHOST	4	3	0.3	413,4
STATIC GHOST	5	3	0.0	ERROR
STATIC GHOST	5	3	0.1	-170
STATIC GHOST	5	3	0.3	255,6

Note: This part of the project was already achieved more than one time but when we saw some errors, we have to restart all the performance analysis and restarting the Q-table too.

To obtain these results, an average of the score of **5** consecutive games was done.

After analysing all this relationship with **Pivot Charts**(Microsoft Excel tool), it is essential to point out the error when performing map 5 with static ghosts and epsilon 0.0. It was impossible for us to make the table converge with that concise map without affecting the other optimal policies for the other maps, so after long time trying, we considered that it was better to have that small issue rather than affecting the efficiency of the others.

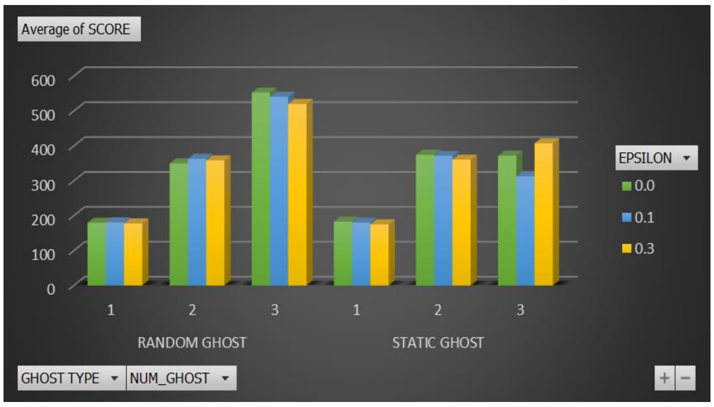
As we can see in this plot, epsilon 0.0 performs the best in static ghost in all the maps but number five which was kind of a nightmare for us as it didn't work as expected. The fact that epsilon 0.0 is the one which is best performing for static ghosts is quite satisfactory as it means that Pac-Man is performing as we want it to do it. And as we can see in static ghosts as we increase the epsilon, the score is reduced which follows the logic explained before.



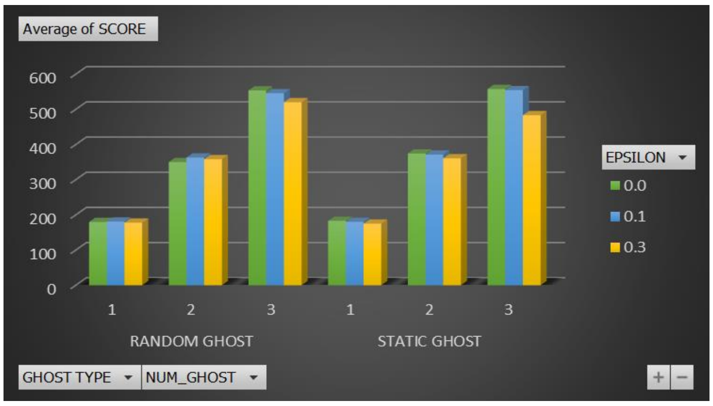
On the other hand, random ghosts work in a completely different way as in some maps it may reduce the time taken to eat them as they get closer to Pac-Man but it doesn't let us see the relevance of the use of different epsilon. It is important to mention that the scores between different maps are not comparable as some maps can only be played with 1, 2 or 3 ghosts and some others can be played with more than one option and what it is important here is to check the differences between same maps, with different epsilon and different type of ghost.

We can conclude that our Agent performs better in static ghost with a low epsilon excluding the bad performance shown in map number 5, and that differences between static and random ghost are infimum because although the algorithm works better in static ghost, the luck also appears when playing with random ghost and they can be eaten in less ticks.

WITH MAP 5



WITHOUT MAP 5



These two graphs were plotted to show how map 5 caused so much trouble for us when played in static mode. Map number 5 was only played with three ghosts, therefore we can see how the barplots remain the same for all the possibilities, included for 3 ghosts in random cases. As I've explained before, we didn't have this issue with random ghosts. But if we focus on the rightmost bars from the plot, we can see a huge difference between taking map number 5 and not taking it into account. The scores grow for all three epsilon and show again how our algorithm works great with a value for epsilon of 0.0.
For the rest of the maps we can say that 1 and 2 perform excellent no matter what type of ghost, map 3 performs well with static ghost but it does perform better when the type of ghost is random as there exist a possibility of entering in a loop with one of the ghosts and although it ends up finding the exit to that problem it obviously takes a little more time. In map 4 the epsilon 0.3 for static ghosts performs worse but it also gives very satisfactory results for the rest of the epsilons and it doesn't give any issue when it is played with random ghosts. Therefore, the results obtained overall satisfy the goal we had to achieve.

CONCLUSION

To sum up, this project on Reinforcement Learning with Pacman(specified in Q-learning approach) leds to a overall comprehension of how the algorithm works, from the creation of all the different states in which the agent can live and its reflection over the Q-table rows, to the creation of the rewards that it can be obtaining in each of them by the talking of any action.

Remark too, how the epsilon-greedy algorithm directly affects the effectiveness when learning the optimal policy or any of the parameters can affect the Q-table values.

In our personal experience we would have preferred to be advised on how the distances problem could be solved by the only attribute of self.distance.getDistance(). Nonetheless, it was quite entertaining as we had a really enjoyable experience when designing the perfect states and reward approach to converge into the optimal policy.
To finish, we know that 120 states could be lots of states but as we have commented few times over this project, we selected all that to cover all the possible states in every Pacman so it could learn and work correctly in every map given.