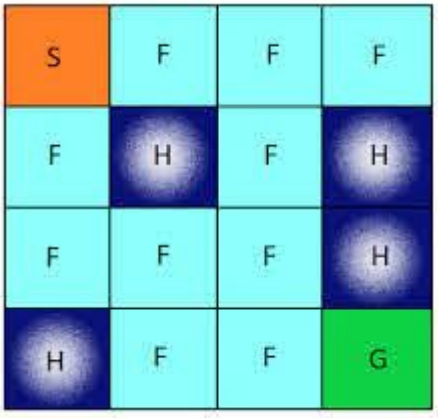
Number of page (excluding cover) : 10

Introduction and Problems choice

1- Purpose of the assignment:

The purpose of this assignment is to apply reinforcement learning techniques to help an agent make decisions in two different environments.
The task is to explore two problems in Markovian settings (Markov states with transition probabilities) and apply value iteration, policy iteration and Q-Learning to solve for the optimal decisions that will lead our agent to the best reward possible.
We will consider various numbers of states and we will make one of our problems a grid problem (discrete) whereas the other will be continuous i.e (continuous values for the states).
We will compare the convergence time all of the three methods and see which methods are better in which contexts.

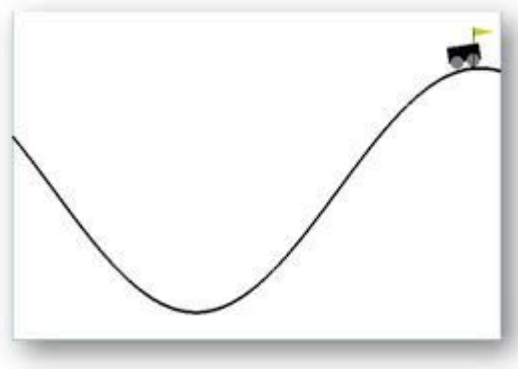
2- First Problem - Frozen Lake:



This is a grid problem from openAI gym where the state space is a 2D grid, the agent starts at the top left and can choose to either go UP, DOWN, LEFT or RIGHT. He has to reach a goal G on the bottom right of the map to obtain a positive reward and end the game, but if he lands on a hole (there could be as many holes as we want in the map) the episode ends.

This problem is interesting for us because it is a classic grid problem, with a relatively easy to understand environment, we do have knowledge about the model, so it is easy to implement both policy and value iteration.
Another advantage of this problem is that we can easily scale the state number as well to study the effect of this parameter on each of our decision making algorithms.
We will using a transition probability of ⅓ of going in the actual action's direction and ⅓ chance for each of the perpendicularly neighboring states.

3- Second Problem - Mountain car:



This is a two-dimensional state problem, every state is defined by one dimensional location (x-axis) and the velocity of a car. To reach to the goal (top of the mountain) the car must build momentum by moving either right or left (a third action corresponds to taking no decision)
What's special about this problem is that the state space is continuous, the position can take a range of values between two values and the same goes for speed.

The number state can be chosen by sampling our space in a way that makes every state relevant without losing too much information.
The transition probability is 1 of going in the commanded direction.

Implementation and evaluation

1. Frozen Lake

We will run each of the two planning algorithms and the learning algorithms on two grids of size (4x4) and (50x50) i.e 16 states and 2500 states.

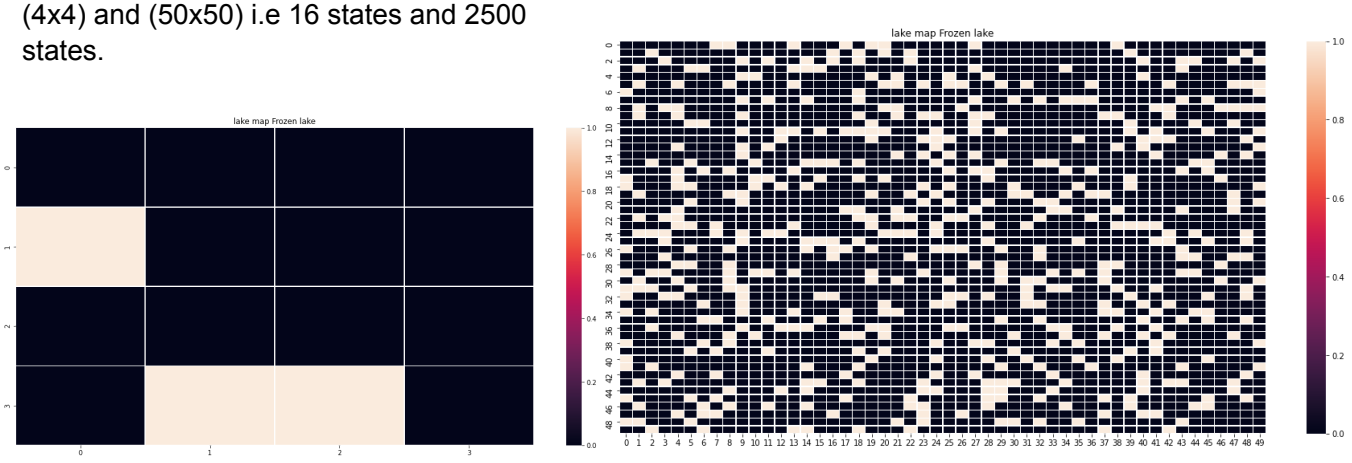


Figure 1 - 2-D visualization of lake maps for two different state numbers (holes are clear cells)

Value Iteration:

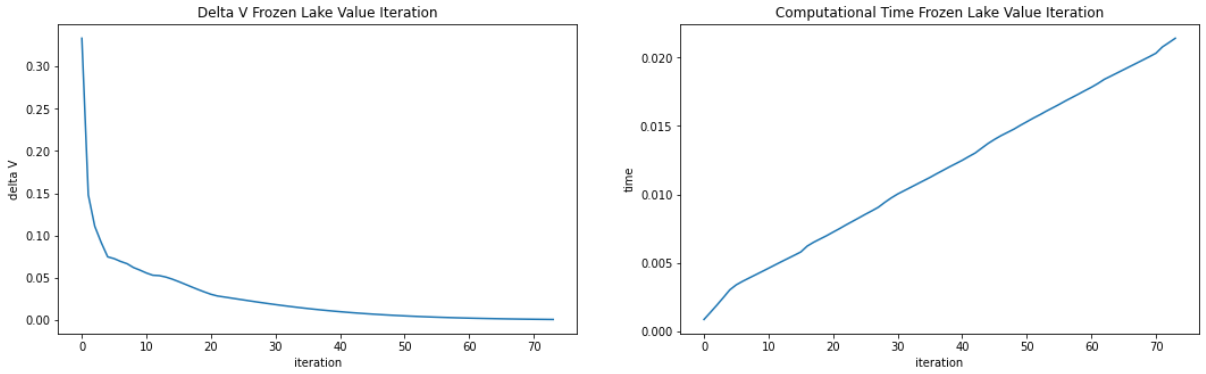


Figure 2 - delta V (left) and computational time (right) evolution over iterations (4x4 grid)

We set the delta threshold for convergence to 0.0001, and achieved convergence in 0.02 seconds after 73 iterations. This is a relatively simple scenario since the number of states is low, and the rewards propagate fairly quickly to update the value function.

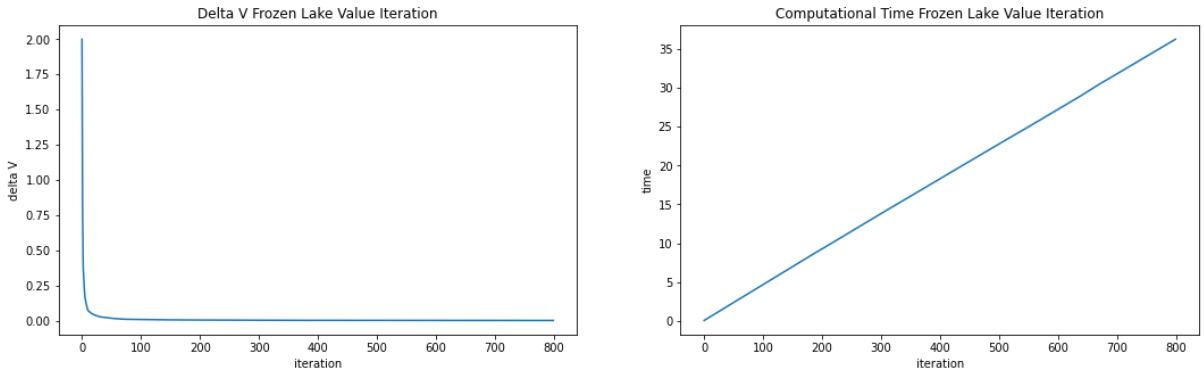
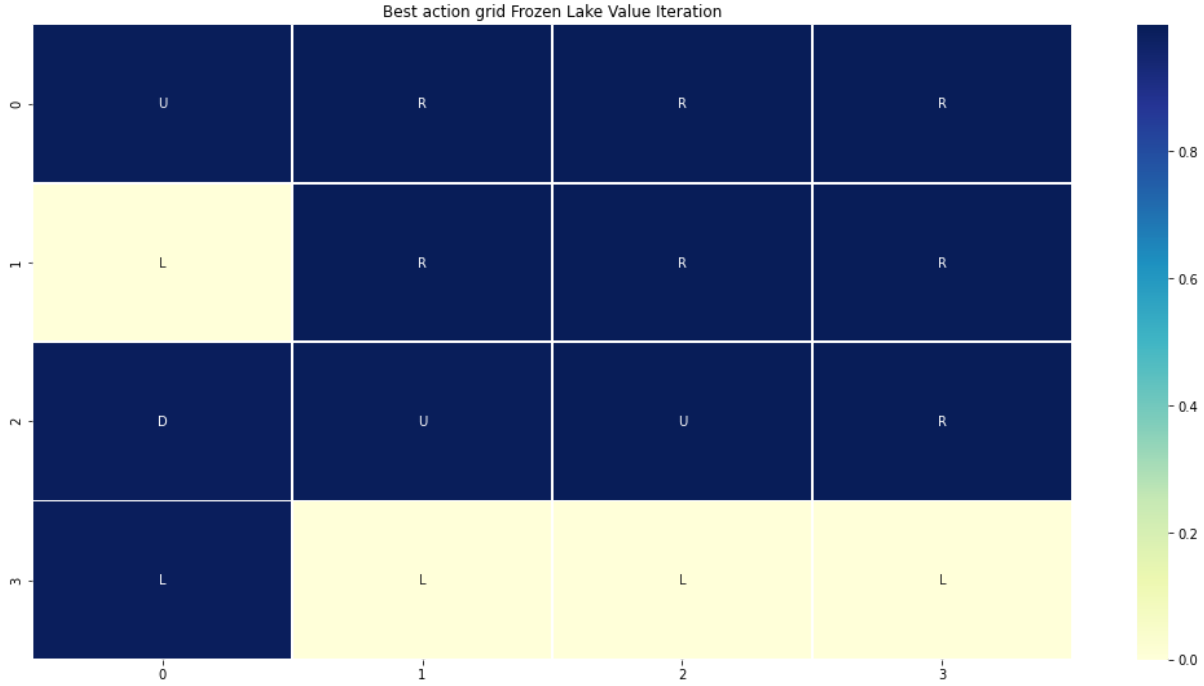


Figure 3 - delta V (left) and computational time (right) evolution over iterations (50x50 grid)

We still reached convergence with 2500 states, but one important tweak was to add a negative reward for falling into holes, it helps achieve a much faster convergence. As having one reward at the final state did not propagate very well to the early states, due to a strong decay of the discount factor. The runtime of the algorithm is 25 seconds which is not very high with respect to the number of states but we will do a deeper analysis of this later one.

Figure 4 - optimal action map (L - left, R - right, U - up, D -down) on a V base heatmap (4x4 map)

We easily observe that the value function is low exactly in the holes' position. And this even more relevant to see in the heatmap for the 50x5



0 lake grid, by comparing it to the actual lake's visualization (figure 1) we can see that the value is lowest around holes and takes the strongest value in the neighborhood of the goal at the bottom right.

Note : the best actions for each is not necessarily the direction that leads directly to the goal because taking an action only has ⅓ probability of actually following its direction.

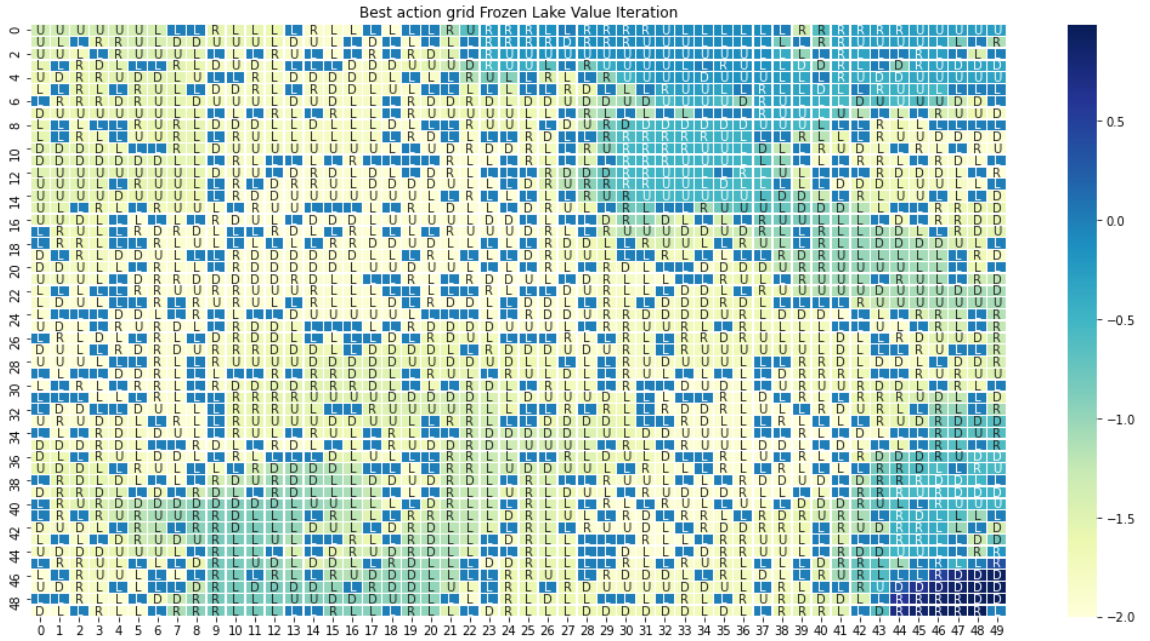


Figure 5 - optimal action map (L - left, R - right, U - up, D -down) on a V base heatmap (50x50 map)

Policy Iteration:

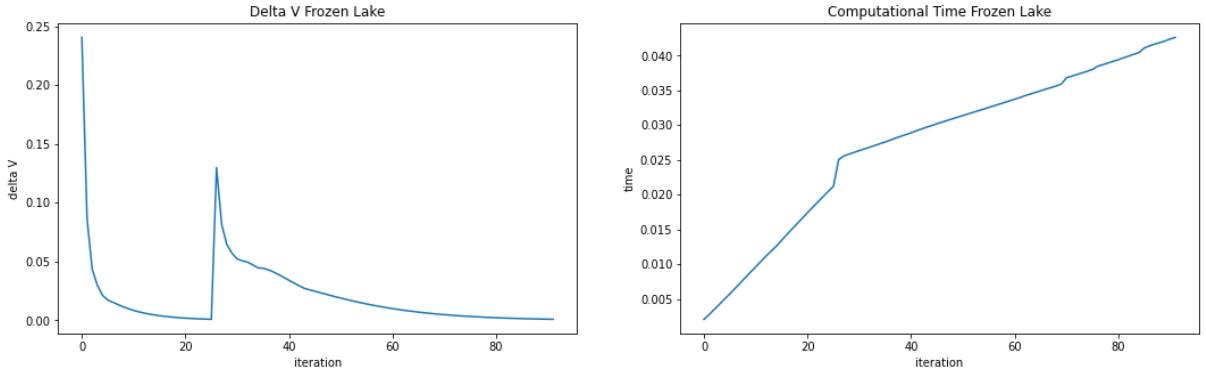


Figure 6 - delta V (left) and computational time (right) evolution over iterations (4x4 grid)

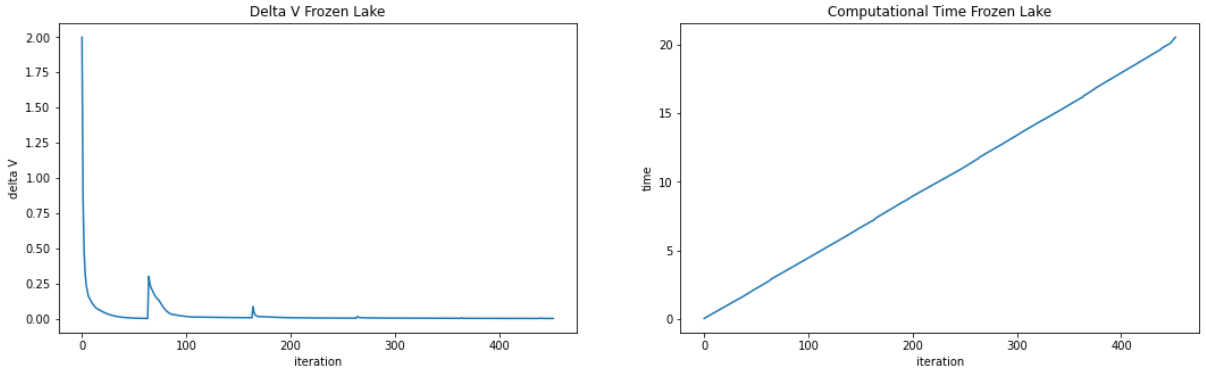


Figure 7 - delta V (left) and computational time (right) evolution over iterations (50x50 grid)

We can see that for a 50x50 grid the policy function takes less iterations (but more time complexity per iteration) to converge. Since for each step we compute the value function for the current policy.

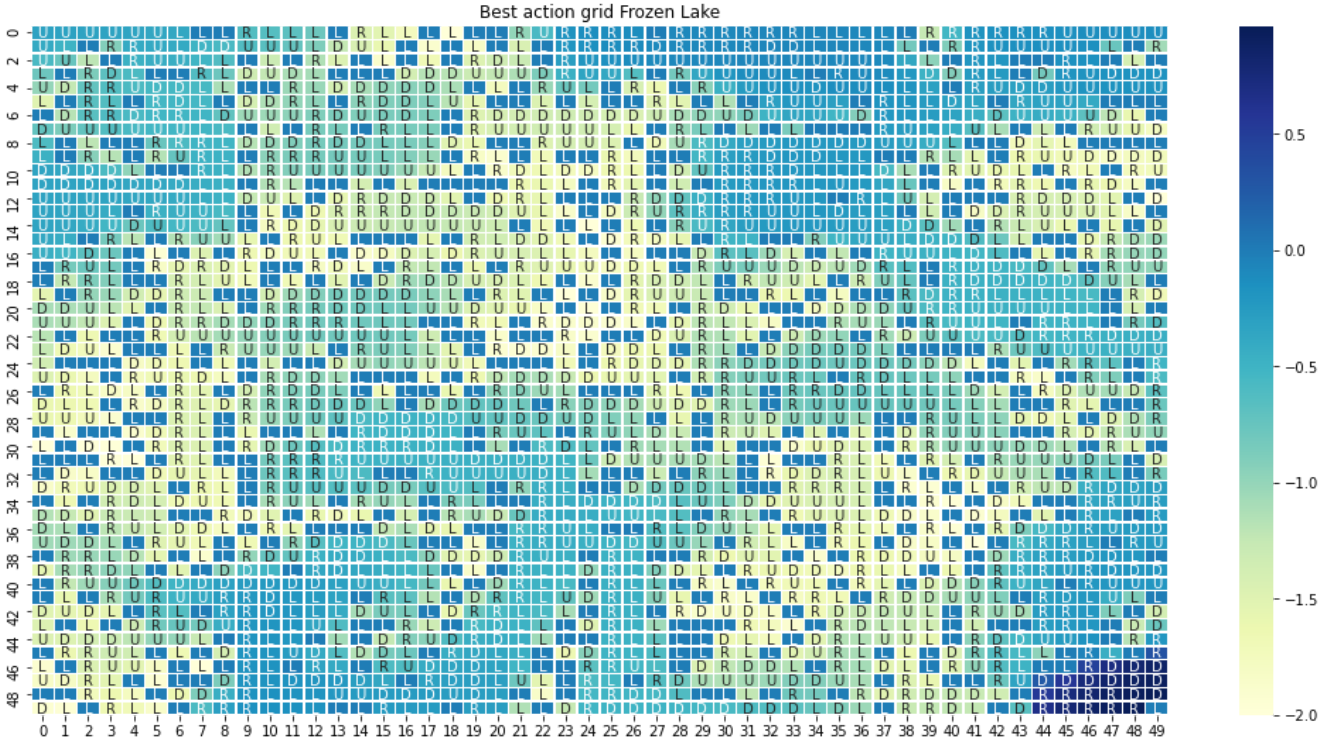


Figure 8 - optimal action map (L - left, R - right, U - up, D -down) on a V base heatmap (50x50 map)

The optimal value function seems to be slightly more coherent with the lake visualization, but this is counter-intuitive since this method computes the value function from policy updates as opposed to value iteration that finds that best value function. The reason behind that is that the heatmap seems more coherent only because the value function follows the same patterns as the holes distribution but that does not necessarily mean it is indeed the optimal value function (when we take into account that the movements are stochastic and we have more chances of ending up sideways of the target cell than on the actual target cell)

The optimal policy map for the 4x4 map using policy iteration is exactly the same as for value iteration (figure available in the code).

Q-Learning:

We used a Q learning algorithm with values initialized at 0, and exponentially decaying exploration factor (epsilon) and learning rate.
We tried tuning all of these parameters as well as the decay rates, but it seems to be insufficient for the 2500 states case.

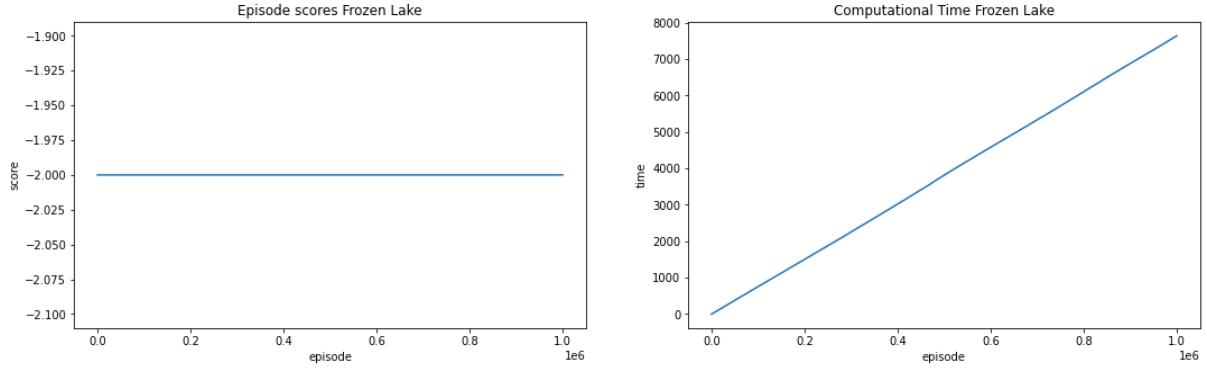


Figure 9 - Score per episode (left) and computational time (right) evolution over iterations (50x50 grid)

Although the runtime scales linearly with episode numbers, the problem is that we probably need a lot more episodes to converge, in a million episodes , with different exploration settings, it seemed impossible for the model to reach the final winning state. This is logical because unlike planning models, the reward will not propagate from the goal unless the model gets there, and it has to do it by chance and the insufficient information of simply not going into holes, which is extremely unlikely. The agent's behaviour will simply consist of using what it learned to avoid the holes it fell into before, by staying in the initial states until it falls into a new hole, or even a known hole if it decided to explore (take a random action).

On the other hand it works very well for the small number of states:

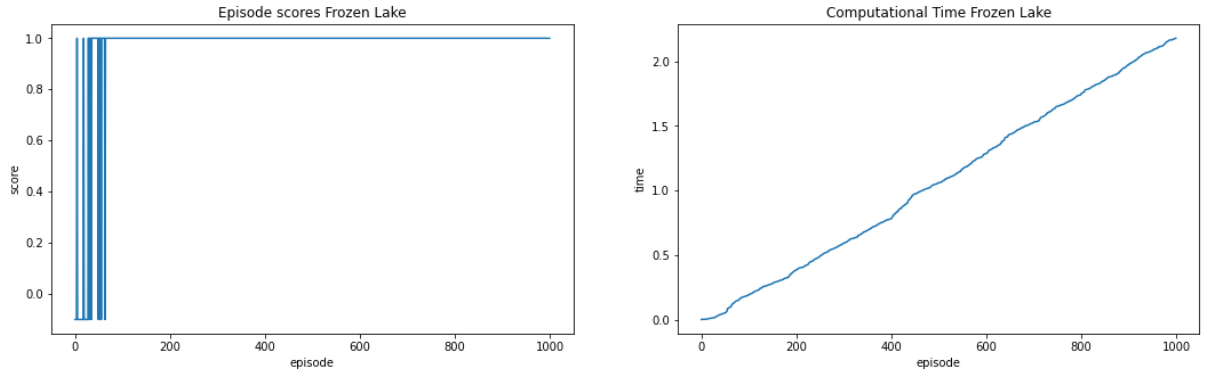
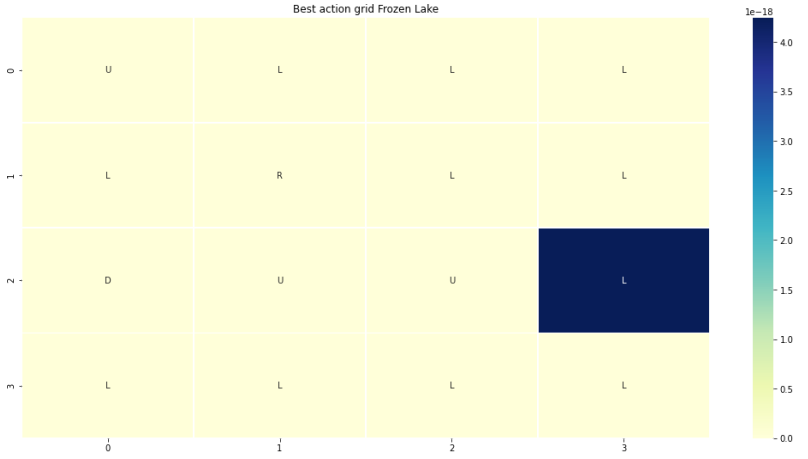


Figure 10 - Score per episode (left) and computational time (right) evolution over iterations (4x4 grid)

It finds a winning path in the first few episodes, and starts using it the time that the exploration factor decays.



But although we get a winning policy, our value function is far from being optimal, the agent has not explored all the possibilities, but just enough for it to know a way to win, we can see how this can be problematic if there were local optimas.

Figure 11 - optimal action map (L - left, R - right, U - up, D -down) on a V base heatmap (4x4 map)

2. Impact of the number of states (frozen lake)

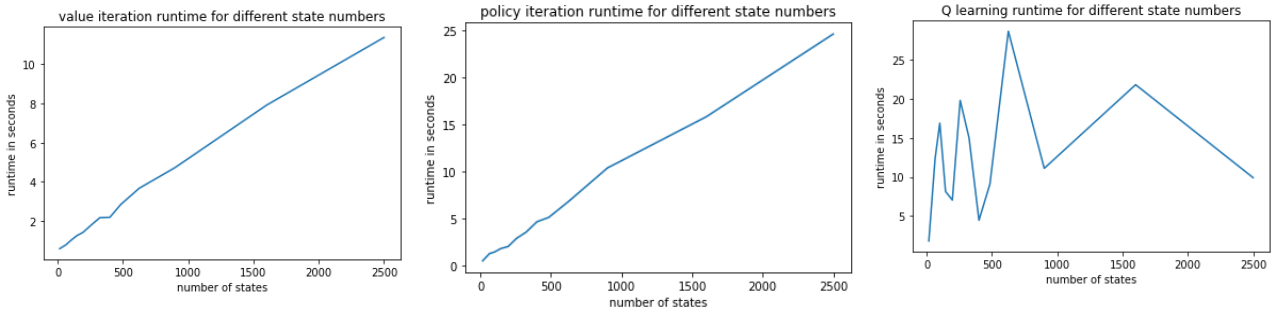


Figure 12 - Algorithm scaling time with number of states (value iteration left, policy iteration middle and q learning right)

The planning algorithms scale very well (linearly) with the number of states, and as we saw earlier, even in the 2500 states case, they reached convergence and were able to generate a winning policy.
The Q-learning algorithm on the other hand seems to be less efficient and more random, the first reason is that since the algorithm that we used does not have a convergence criteria but rather a number of episode where we trained our model (updating the Q function), which means that it's more than the number of required episodes that should increase rather the runtime per episode. And we also saw before that even with a million episodes and various tuning we were not able to find a winning policy for the 2500 states case.

Another interesting point is that the duration of every episode is very variable, since the episode only ends in certain states and the actions can be quite random due to exploration and due to the fact that the model can't learn much in the early episodes.

3. Mountain car

In the settings of these problems every step induces a reward of -1, and termination occurs upon reaching destination, so maximizing reward means reaching the goal in the least steps. We consider a win to be an episode with a reward greater than -200.

Discretization:

We achieve discretization simply by sampling the position and velocity, and we then create a transition matrix so that we can reuse the functions for frozen lakes.
The position values go from -1.2 to 0.6 (to win we have to reach a position higher than 0.45)
Velocity values range from -0.07 to 0.07.
We discretize by sampling each of these two ranges into 10 values for a total of 100 states.

Value Iteration:

After finding the optimal policy we would still not be able to win 100% of the game instances, so we tried setting low convergence thresholds.

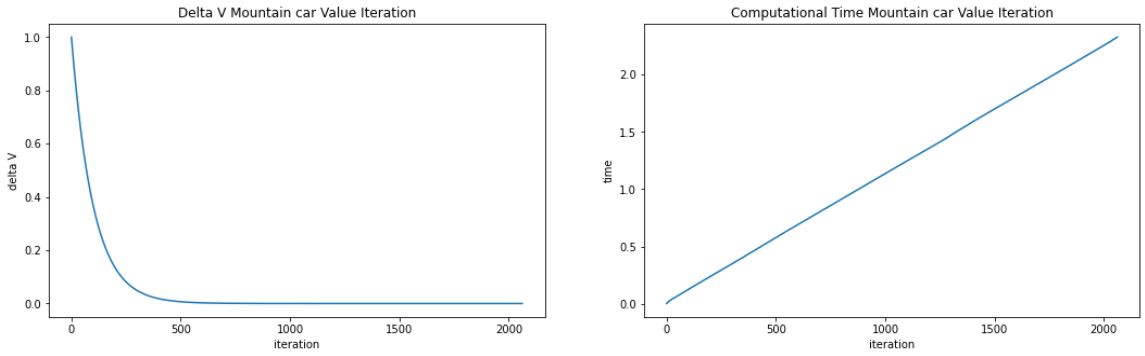


Figure 13 - delta V (left) and computational time (right) evolution over iterations (value iteration)

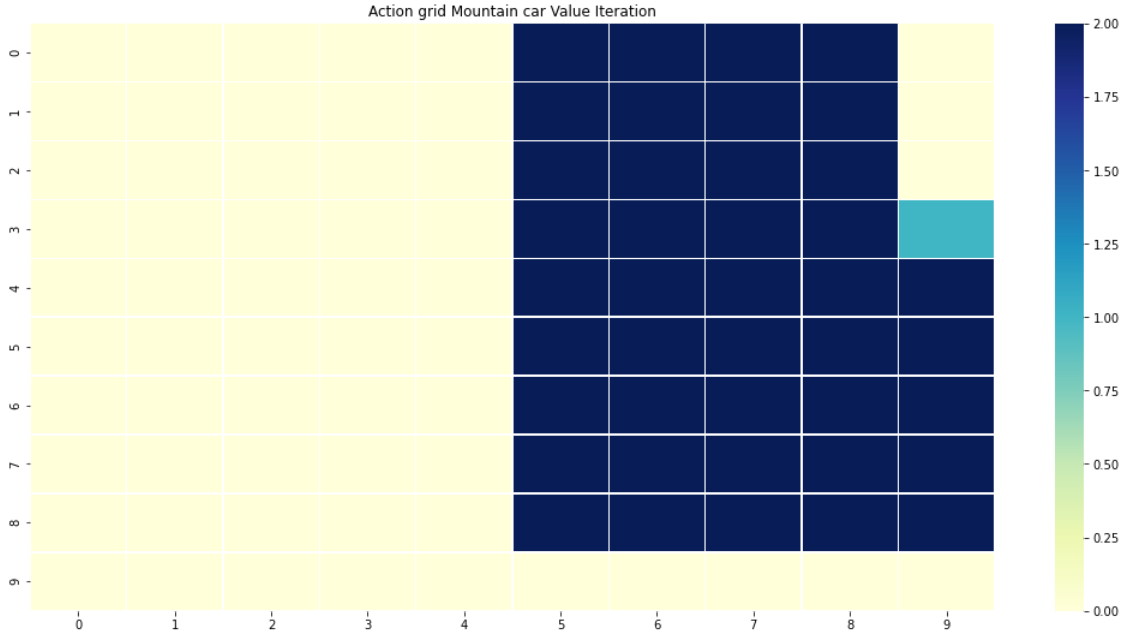


Figure 14 - best policy map (axis x is position and axis y is velocity), yellow for RIGHT, blue for LEFT and turquoise for NONE (value iteration)

Policy Iteration:

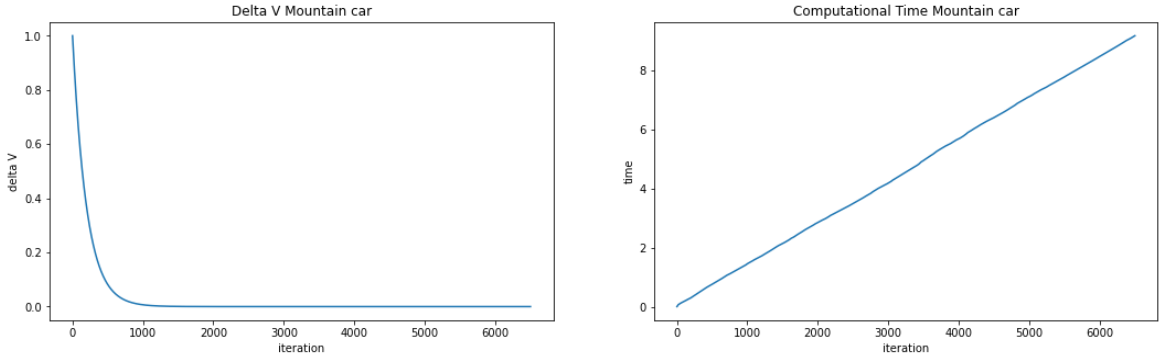


Figure 15 - delta V (left) and computational time (right) evolution over iterations (policy iteration)

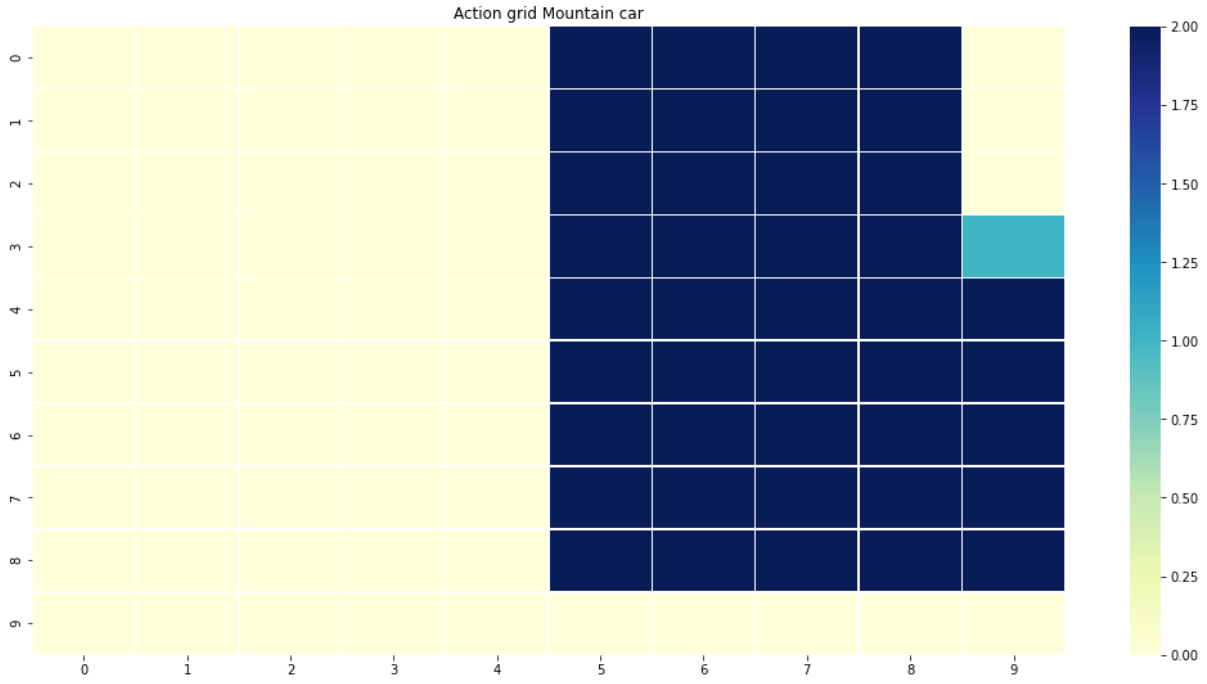


Figure 16 - best policy map (axis x is position and axis y is velocity), yellow for RIGHT, blue for LEFT and turquoise for NONE (policy iteration)

Value and policy iteration converge to the same solution and take more or less the same amount of time to converge (20 seconds for value vs 8 seconds for policy iteration)
The optimal policy consists of going down the hill (right when we're on the left half and left when we're on the right side) to build momentum, except when we're close to the goal and we have enough speed in which case the decision : moving right allows us to reach the goal.
Both those policy result in a winrate of around 50%

Q-learning:

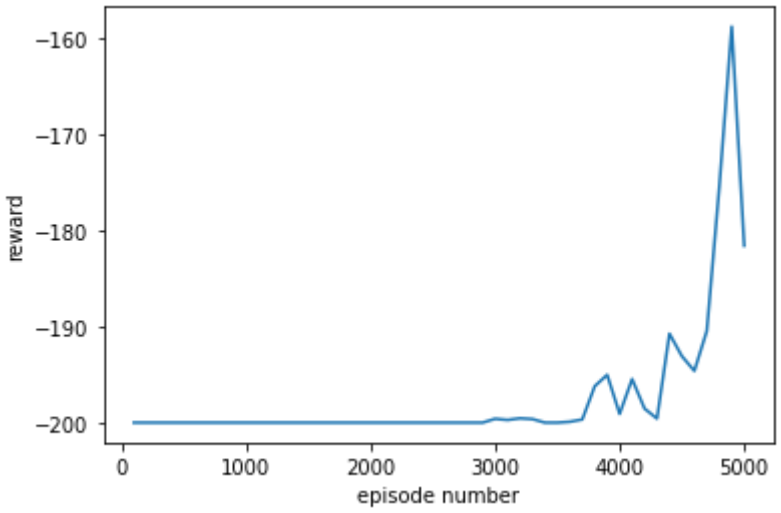


Figure 17 - reward evolution with number of episodes Q-Learning

This Q-learning algorithm works well with this environment, since there is only one optimum and not too many states.
We use a discount factor of 0.9 and a linear decay for our exploration factor (experimentally worked better)

Total runtime was 60 seconds (3 times slower that planning algorithms)

Conclusion

In conclusion it is interesting to compare the MDP planning solvers (value and policy iteration) with a Q-learning algorithm, the 2 problems at hand allowed us to see the limitation of Q-Learning, but it is important to keep in mind that the constraint behind planning algorithms, is that we must know the model, whereas in many real life problems, we do not know the model exactly, and we can only rely on observations of the environment after the agent takes a certain action.
In such scenarios we are obliged to use a learning algorithm.

And although Q-Learning performed fairly poorly in large state environments, there are many methods that allow to overcome this, such a using a neural network to predict the Q-function instead of updating the Q function at every step, this method can be powerful and is widely used for example in video games, where the number of states can easily reach the order of millions.