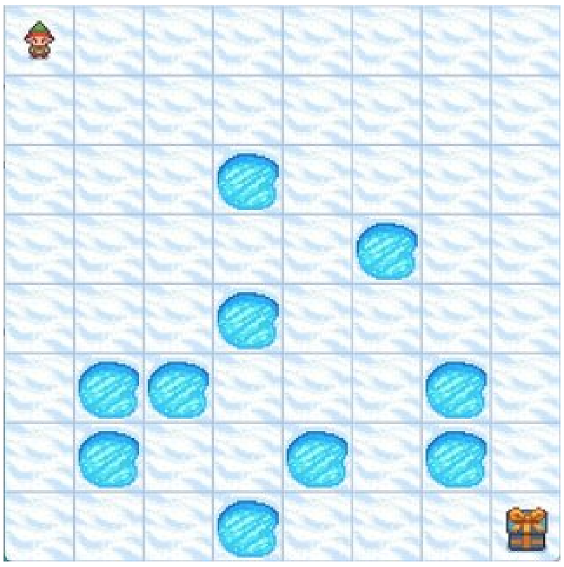
Reinforcement Learning - FrozenLake

Introduction

We have chosen the premade gym environment "Frozen Lake". This is a stochastic environment, where the agent wants to move from the starting state to the goal state without falling into any of the holes in the ice. The ice is slippery, so actions taken are stochastic, not deterministic. The agent will perform the chosen action with a probability of ⅓, so with ⅔ the agent "slips" and moves either left or right with probability ⅓ respectively. Whether the ice is slippery or not is managed by the boolean variable "is_slippery".



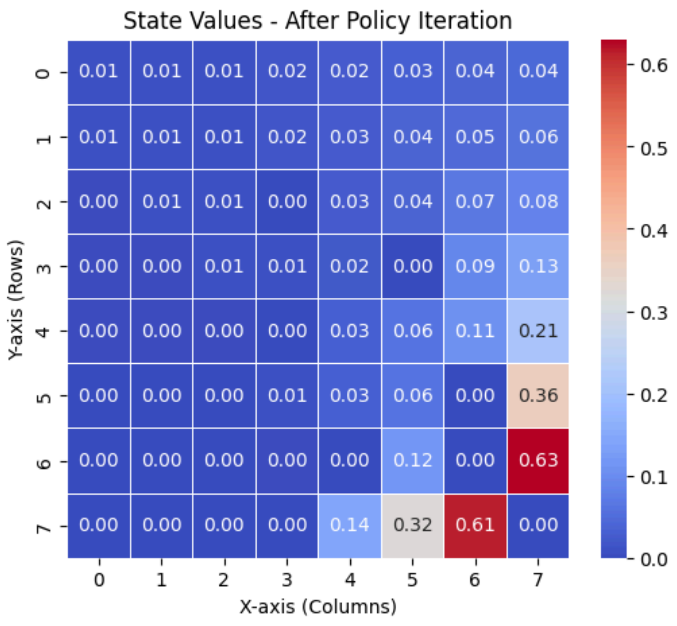
The purpose of this report is to compare the complexity and results of different RL algorithms when finding the best policy to reach the goal state. In this report, we will answer the following: Which algorithm is the most effective at finding a good policy for the agent in the Frozen Lake environment? We will answer this question by answering the following 2 sub-questions: How many computations does the algorithm need to converge on a policy? And what is the rate of success for the found policy?

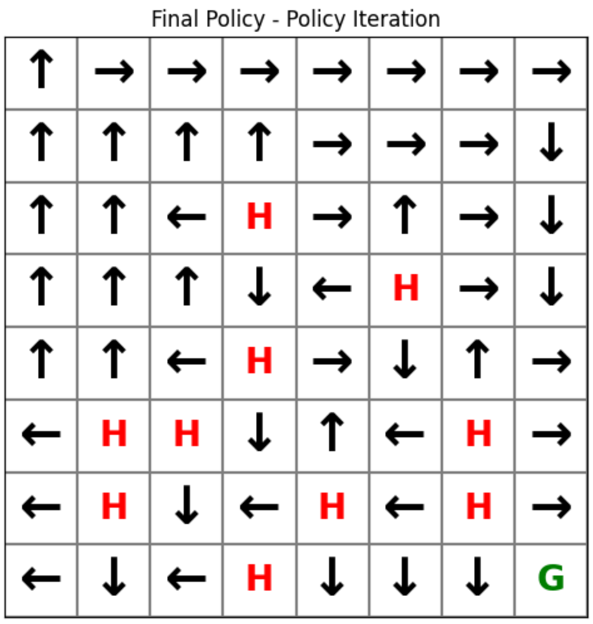
Dynamic Programming algorithms

Dynamic programming algorithms, as opposed to other algorithms that will be discussed, require full knowledge of the environment beforehand. The idea behind how dynamic programming can come to a solution is through dividing the main problem into subproblems, and through looking ahead in the steps and solving them. Here we'll be looking at policy iteration and value iteration.

With policy iteration we first start with a uniform random policy and start with 0 for all the state values. Then we go into the main loop, which we have set to run 1000 times at most. In each loop we do policy evaluation, as well as policy improvement. From that we get a new potential policy each time, but if the new policy is the same as the old policy it can't be improved anymore, and we exit the loop early.

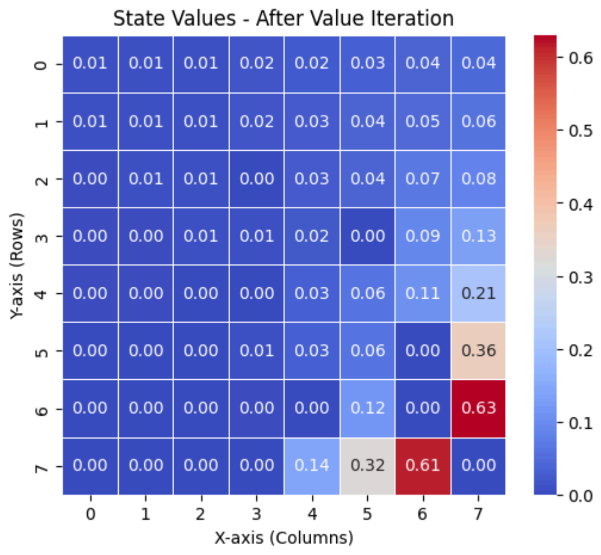
In the policy evaluation we're updating the state values using the bellman expectation equation. We do this by going over every action for every state, and after the calculations are done we can get the best action, and update the value of the state for the new policy. After every state has been updated they will be passed to the policy improvement part of the algorithm. Here it chooses the best action according to a greedy approach, and that way creates an updated policy.

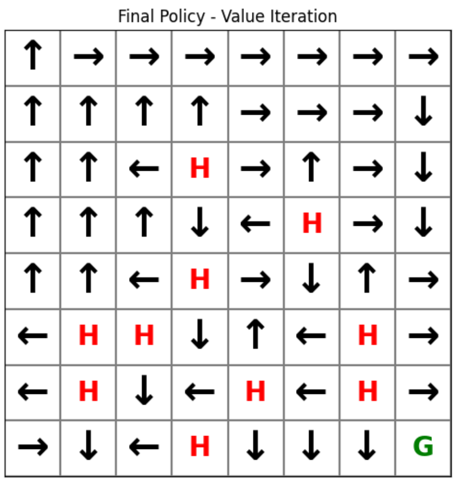




The plots were made from a run with gamma = 0.9.
Here we have an episode count of 8 for gamma=0.7, 4 with gamma = 0.8, 6 with gamma = 0.9, and 4 for gamma = 0.99. For 1000 runs it has 742 successful runs with this policy.

With value iteration we iteratively apply the Bellman equation until we converge on a certain value. We choose a small threshold, θ, and we iterate until we have a value Δ that is smaller than it. After the threshold has been reached we can define a policy based on the final state values.





What we do for each episode here is the following: we get all the possible actions for each state, and then for each action we get the info from the state action pair on the possible transitions. From that info we determine the action values for that action, and update the

state values. Then after having done that for every action we can get the best possible action for that state based on the values. At some point this will lead to reaching the threshold, at which point using the state values we can derive a policy.
The plots were made from a run with gamma = 0.9.
For value iteration we had an episode count of 21 with gamma=0.7, 24 with gamma=0.8, an episode count of 46 with gamma=0.9, and an episode count of 221 with gamma=0.99.
For values 0.5 and below it took too long to compute. For 1000 runs it has 762 successful runs with this policy, meaning that in that regard it is very similar to the result policy iteration got, which isn't too surprising as they ended up landing on the same policy in the end.

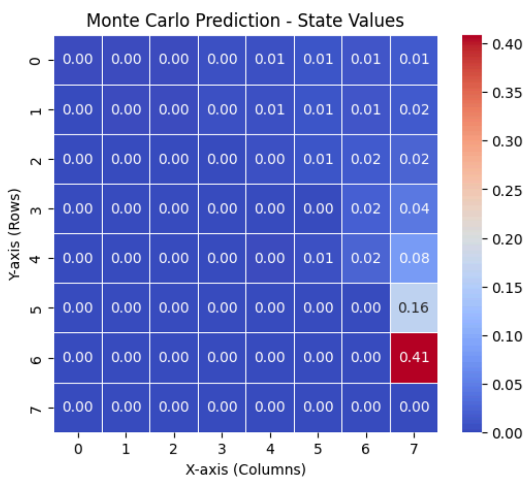
Monte Carlo algorithms

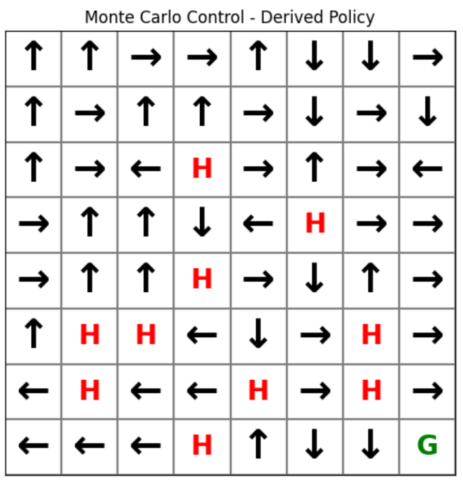
While DP requires a complete model, the Monte Carlo algorithms use sampling to estimate values. It runs complete episodes for a predetermined amount of times, averaging the values of either every time a state is visited or taking only the value of the first time that particular state is visited during an episode. It's divided into a prediction (to evaluate the policy) and a control algorithm (to find the optimal policy with a greedy approach) and it can use different exploration strategies, including exploration starts, on-policy methods and off-policy methods. For the off-policy methods it's necessary to use importance sampling in order to correct the mismatch between the two policies used. Since these algorithms require to run complete episodes it's not suited for continuous tasks and it can be quite intensive for tasks that have long episodes.

The code we wrote uses a first-visit only on-policy method with an ε-greedy policy. With on-policy we mean that this improves the policy it's using, and doesn't have an additional target policy to work with, so no relevance sampling is necessary. The ε-greedy policy refers to the fact that we ensure every action has some degree of probability of getting chosen, giving the best action, so the greedy choice, a probability of 1-ε and any other action a probability of ε spread on them.

Monte Carlo results
Training variables used:
episodes = 20.000
gamma = 0.999
epsilon = 0.5

The policy that the Monte Carlo method derived with the values above was decent at the stochastic variant of the 8x8 FrozenLake map. Using this policy it got a success rate of 65 percent. Using 20.000 episodes to train the agent is low considering the Monte Carlo method and the success rate could probably be higher if you ran many more episodes and let the agent learn more about the environment.





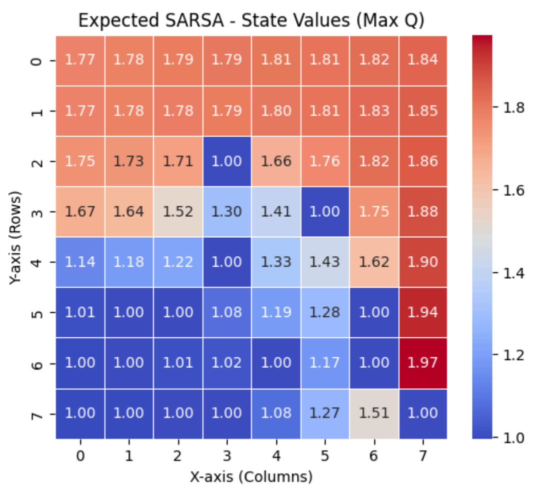
Temporal Difference algorithms
Temporal difference learning methods combine ideas from both dynamic programming and Monte Carlo methods. We don't need a full model like DP but also don't need to wait until the end of an episode to update their values. Instead, we update estimates after every step using the reward and the estimate of the next state.

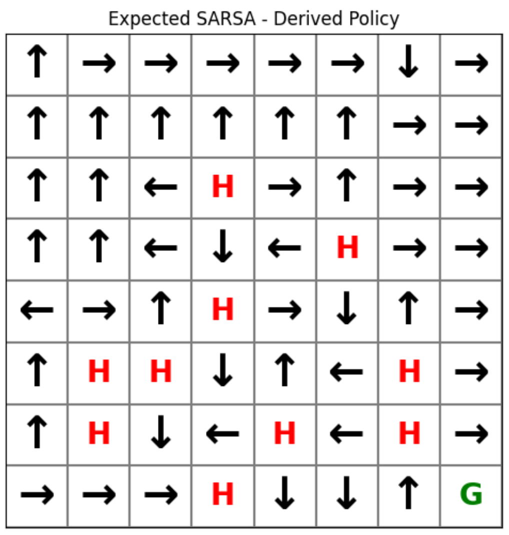
Expected SARSA is an on-policy method, meaning it improves the policy that it is currently following. It uses an epsilon-greedy strategy to balance exploration and exploitation. That way most of the time the best action is chosen to exploit the game in order to get the most time, but sometimes we take a different action just to see if there are possible ways to traverse the lake. This is all based on how high you set the value of epsilon. The higher it is, the more you want the agent to explore, meaning the agent will try more random actions.

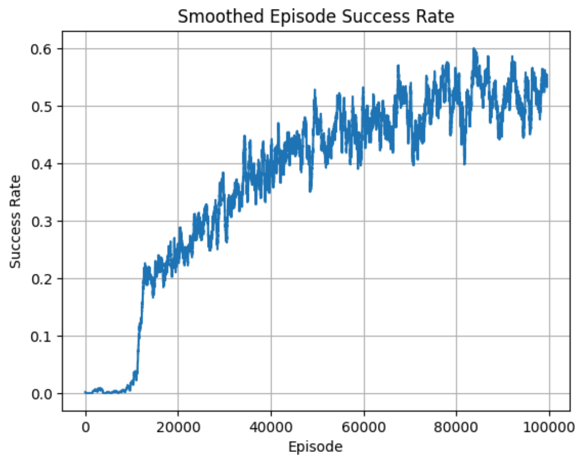
Q-learning is an off-policy method, which means it learns the value of the optimal policy regardless of the policy being followed. It also uses an epsilon-greedy to explore the environment. However unlike SARSA it always updates its Q-values as though it had taken the best possible action, even when it didn't actually take that action. This allows Q-learning to directly learn an optimal policy

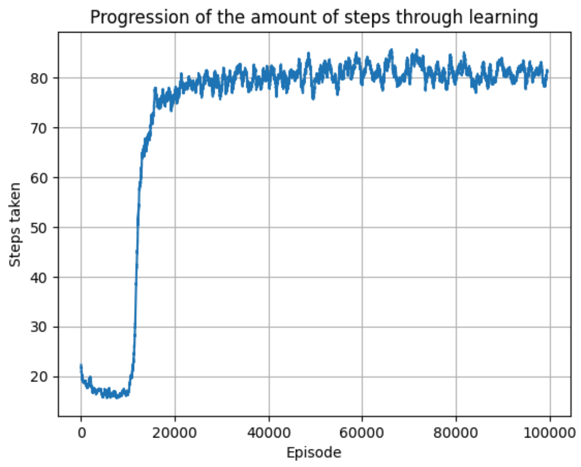
SARSA results
Training variables used:
gamma = 0.999
episodes = 100000
epsilon = 0.45
alpha = 0.05

SARSA had a great success rate of 87% when running the found policy on a 1000 episodes. We noticed that SARSA does not work well when the epsilon is too high so we set it at the halfway point while letting it diminish, and this resulted in a great policy! The graph at the bottom left shows how the policy is improving through the episodes and is receiving more and more rewards for each run.



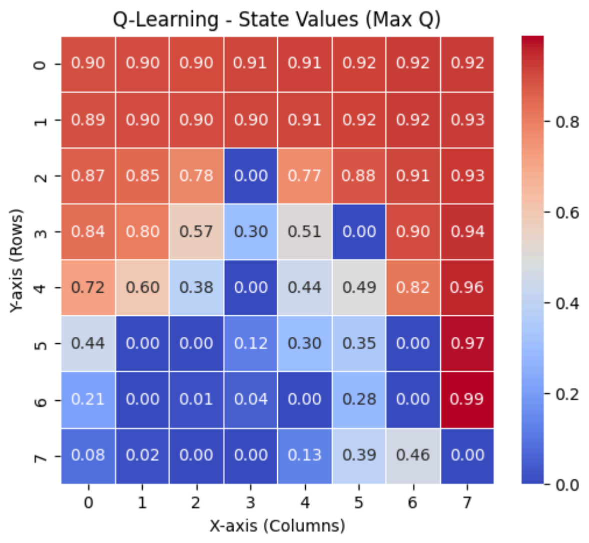


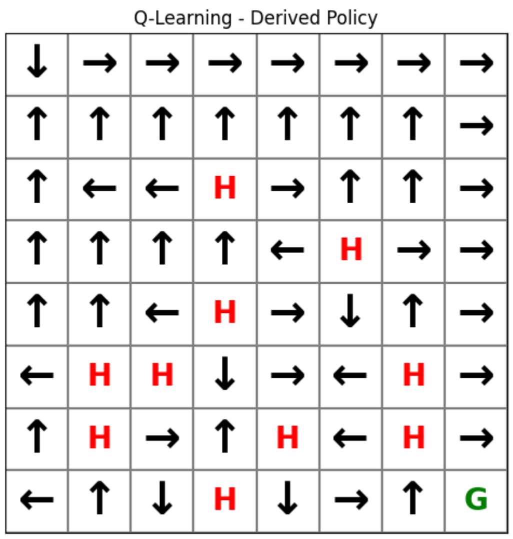


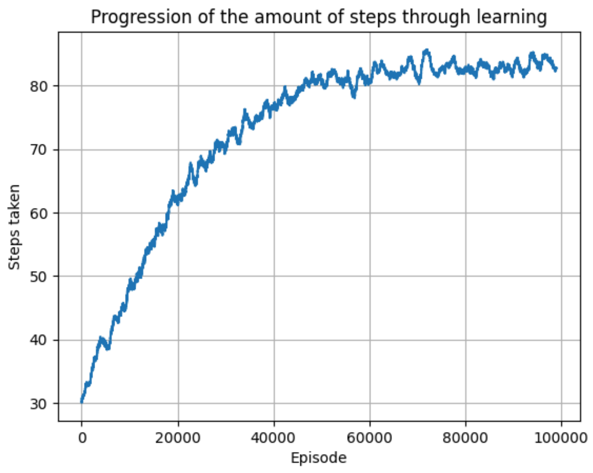


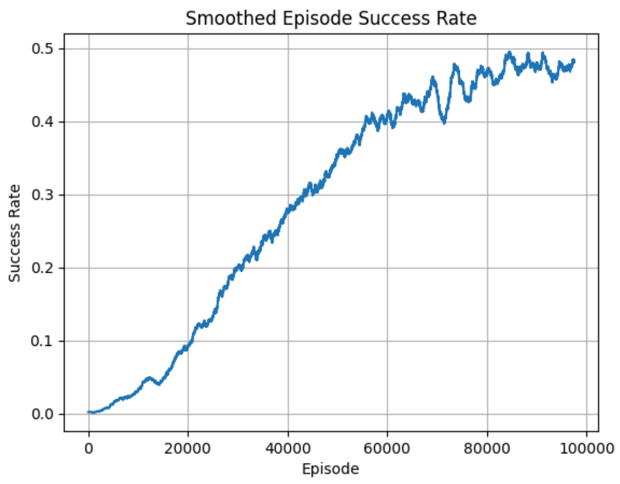
Q-Learning results
Training variables used:
gamma = 0.999
episodes = 100000
epsilon = 1.0
alpha = 0.05

The epsilon value could be much higher with Q-learning since it did not mind so much exploring at the start while it diminishes and slowly changes exploration into exploitation. With Q-learning the agent again found a great policy that has a success rate of around 88 percent when ran on 1000 episodes. Which is a great result.









Discussion

Final policies comparison

For each algorithm we also measured the number of episodes it took to find the final policy they came to, as well as the success rate of that policy.

Algorithm	Number of episodes until final policy	Success rate of final policy
Policy iteration	6	74,2%
Value iteration	46	76,2%
Monte Carlo	20.000	65%
SARSA	100.000	87.7%
Q-Learning	100.000	88.5%

From these results it may seem like policy iteration and value iteration are very efficient algorithms, but that is definitely not the case. This is because those two are very computationally heavy per episode. This means that the other algorithms can be allowed to do many more episodes with the same computational resources.
Still, it does show that those two algorithms can come up with a relatively strong policy in a low number of episodes.
Our application of Monte Carlo here fared worse when finding a good policy than the other algorithms. It has the lowest success rate than the final policies that the other algorithms ended up finding. This may be due to not having run enough episodes.
SARSA and Q-learning could learn the environment after a good chunk of episodes and did not mind the stochastic nature of the game.

Conclusion

What we have seen throughout the project is that DP methods converge quickly but require full knowledge of the environment which is not always a given when you want to use the methods in the real world. Monte Carlo methods are simpler but require a lot of episodes to find a decent policy, since its only option is learning through randomness.. For us SARSA and Q-learning were very effective methods to use for this game and when you get the training variables right the algorithms converge to a well-founded policy where it exploits the right parts of the map in a great way.