MARKOV DECISION PROCESSES

In this assignment we will explore the notion of making decisions, which is different from the previous assignments where several ML techniques were used to perform function approximation. Particularly, we will focus on how an agent should act in the world. The basic idea is to model to world such that we can find a way to act in it, and Markov Decision Processes (MDPs) help us represent this world. In this report, we will explore the notion of finding an optimal policy that tells us how to act in a well-defined MDP using the *Value Iteration* and *Policy Iteration* algorithms. Also, the same problems will be solved using *Q-Learning*, a model-free Reinforcement Learning algorithm that learns a policy exploring the world and has no information about it.

TWO INTERESTING MDPS

Markov Decision Processes help us model the universe. An MDP is defined by:

• $\boldsymbol{S}$: States	→ Set of states s_i that describe the state of the world
• $\boldsymbol{A}$: Actions $A(s)$	→ Set of actions a_i we can take to act on the environment. Allows to change states
• $\boldsymbol{T}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')$: Model	→ Transition model $\Pr(s'\|s,a)$, Physics of the world
• $\boldsymbol{R}(\boldsymbol{s})$: Reward signal	→ The value of being in a state

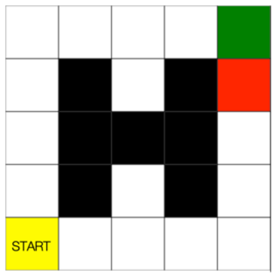
Given the MDP defined by the four elements above, and an Initial state $\boldsymbol{s_0}$, we want to find policies that tell us what action to take in each state, and in particular among all the policies, we are looking for the optimal policy:

• $\boldsymbol{\pi}(\boldsymbol{s}) \to \boldsymbol{a}$: Policy	→ tells what action to take in each state, a solution to an MDP
• $\boldsymbol{\pi}^*$: optimal policy	→ the policy that maximizes expected long term reward

What makes this Markovian is that the only thing that matters is where you are, and not where you have been. Another reason is that it is stationary, the model (T) does not change over time. This formalism is different from supervised and unsupervised learning. Our goal is to learn the policy. We are doing sequential decision making. We are never told what the right action is. In this problem because what matters is the reward we get over time, we can afford to take a decision that is locally bad but overall good. We want to maximize our long term expected reward. Given this we present the following interesting MDPs.

1. Slippery World Treasure Hunt

In a 5 x 5 Grid World (figure at the right) there is a robot located at the START state, and this robot wants to go treasure hunting. The possible actions (↑ ← ↓ →) the robot can take is to move to any neighboring state that is not a **black** wall, or a wall along the edge of the world. If the robot moves against the wall, it will stay in the same spot. It has been raining all day so the world is very slippery, so the robot's actions are non-deterministic: with 80% change it will move in the direction it intended, and with 20% chance it will end up moving perpendicularly (10% in each direction). The robot wants to get to the green spot, where a +100 treasure awaits! Unfortunately, there are thieves hidden at the red spot, and if the robot goes there, they will steal -100 from it. While the robot is in the world, its battery charge diminishes, therefore for each state except the green spot and red spot, the robot gets a negative -4 reward.

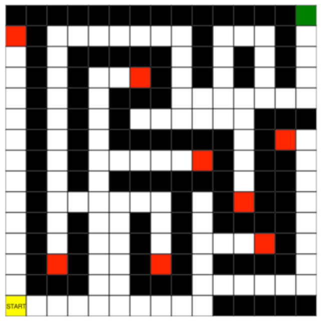


5 x 5 Slippery World

This problem is interesting because it has a few numbers of states (5x5 grid world with 18 states), so it is faster for performing Value Iteration and Policy Iteration analysis. Also, because the grid world is small, in Q-Learning where the robot has to explore the world to learn about it, the explorations are not going to be too time consuming. We can also easily interpret the tradeoffs of Value Iteration vs Policy Iteration when there are few states compared to a large number of states (from our next problem).

2. Battery Recharge Maze

In a 15 x 15 Grid World (figure at the right) there is a robot located at the START state, and this robot's battery is about to die! Therefore, it wants to get as fast as possible to the green charging dock, where it gets +100 battery charge. Unfortunately, there are defective charging docks at the red spots, and if the robot goes there, it will suck -100 charge from it. The possible actions (↑ ← ↓ →) the robot can take is to move to any neighboring state that is not a **black** wall, or a wall along the edge of the world. If the robot moves against the wall, it will stay in the same spot. Because of the robot's motor design, the robot's actions are non-deterministic: with 80% change it will move in the direction it intended, and with 20% chance it will end up moving perpendicularly (10% in each direction). While the robot is in the world, its battery charge diminishes, therefore for each state except the green spot and red spot, the robot gets a -4 battery discharge.



15 x 15 Battery Recharge Maze

This problem is interesting because it has a large number of states (15x15 = 225 possible spots in the grid world) compared to the previous problem which has a small number of states. Because the transition model is known, Value Iteration and Policy Iteration analysis is straightforward. However, because the grid world is large, in Q-Learning where the robot has to explore the world to learn about it, the explorations take much longer to converge to an optimal policy, because the robot has to explore more. This problem will help us see how the process is of learning an optimal policy different when there are a large number of states vs a small number of states as well.

The structure of this report is as follows. We will first solve the MDPs using Value Iteration and Policy Iteration. It is worthwhile to note that both algorithms try to find an optimal policy given that we already know the Transition Model and the Reward structure of the world. We will then use Q-Learning, a model-free Reinforcement Learning approach to learn the optimal policy.

Value Iteration

This is an algorithm to find an optimal policy by calculating the utility of each state. These utilities help us choose an optimal action to perform in each state. The utility of each state is the expected sum of discounted rewards, in other words the reward for being at that state + the discounted utility of being in the following state by taking an optimal action. (Norvig & Russell, 2010) This gives us the Bellman Equation of utility for each state:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s,a,s')U(s')$$

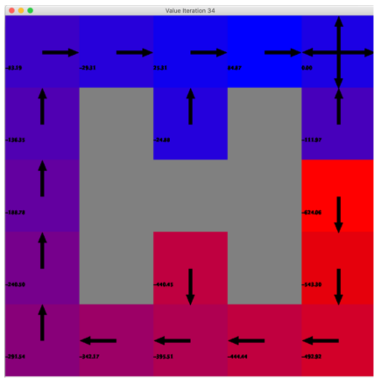
Because a reward in the future is not the same as a reward now, an optimal utility calculation considers the gamma γ value between 0 and 1 as a discount factor. In other words, if we have a discount factor of 0.01, it would make future reward value insignificant compared to an immediate reward. On the other hand, a large value such as 0.99 will make immediate rewards as much as future rewards. This gamma value often tends to change the optimal policy. For our problem, we set gamma to 0.90.

There is one equation per state, and we want to know the utility of each state. n equations and n unknowns. However, they are nonlinear because of the max, so we cannot solve it using system of equations. Therefore, we follow the follow an iterative approach algorithm to get the utilities (Isbell & Littman, s.f.):

- Initialize random utilities for each state
- Update the utilities based on neighbors
- Repeat the update step until convergence.

Basically, instead we are improving the utilities for each state at every iteration until the value function converges. After applying this algorithm, we get unique solutions for the utilities of each state. Then we get the policy at each state by performing $\pi^*(s) = argmax_a \sum_{s'} T(s,a,s')U(s')$.

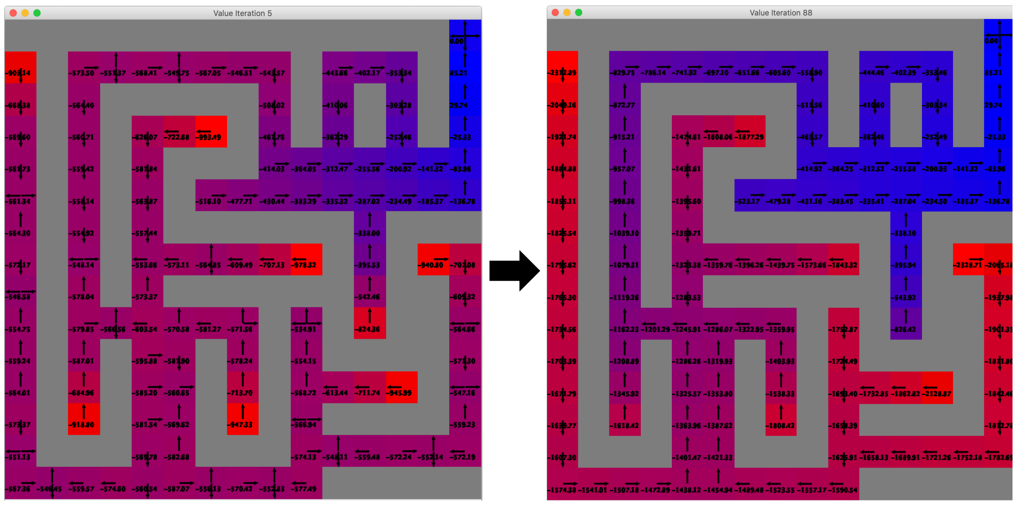
SLIPPERY WORLD TREASURE HUNT



Let's try Value Iteration on our Slippery World. We get the policy shown in the image at the right. It took **34 iterations** to converge to the optimal policy. As we can observe, the policy recommends moving away from the negative -100 spot (red areas) and move closer towards the +100 spot (blue areas). It is also interesting to note that if the robot reaches the -100 spot, it recommends move up straight to the +100 spot it is neighboring than to take the long route to get there. The results make sense as the robot has to take a small negative reward at each step, so to minimize the total rewards, it should just move up to the +100 spot.

BATTERY RECHARGE MAZE

Once again, we use this algorithm to find an optimal policy by calculating the utility of each state until it converges. Let's try Value Iteration on our Battery Recharge Maze. We get the policy shown in the image at the below. On the left we see one of the initial policies obtained at iteration 5, and at the right we see the final policy obtained after it converges at **iteration 88**.



As we can observe, it took 88 iterations to converge to the optimal policy. The policy recommends moving away from all the multiple negative -100 charging docks (red-pink areas) and move closer towards the +100 spot (blue areas). The results make sense as the robot has to take a small negative reward at each step, so to minimize the total rewards, it should just move as quickly as possible to +100 spot and far away from all the -100 spots.

POLICY ITERATION

This algorithm is based on the idea that we are looking for a policy we shouldn't bother to learn the true utilities of each state. We just care that the magnitude is correct. So, Policy Iteration algorithm is as follows (Isbell & Littman, s.f.):

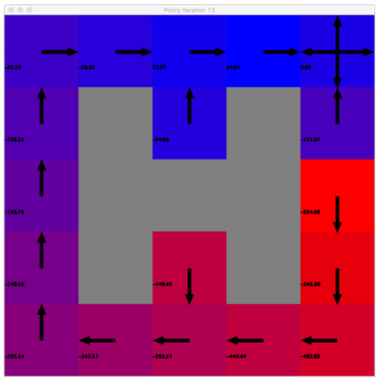
- Start with some initial policy $\pi_0 \leftarrow$ guess
- <u>Evaluate</u>: given a policy π_t calculate the utility of each state $U_t = U^{\pi_t}$
- <u>Improve</u>: calculate the new policy $\pi_{t+1} = argmax_a \sum T(s,a,s')U_t(s')$
- Repeat the Evaluate and Improve steps until convergence.

And the way to obtain the utilities for each state is straightforward compared to the Bellman equation update in Value Iteration:

$U_t = R(s) + \gamma \sum_{s'} T(s, \pi_t(s), s')U_t(s')$. Since we have got rid of the max we now have n equations and n unknowns, so it is linear and can be solved by linear algebra.

Basically, instead of improving the utilities at every iteration (as in Value Iteration algorithm), we are redefining the policy at each step and computing the utilities for this new policy until the policy converges.

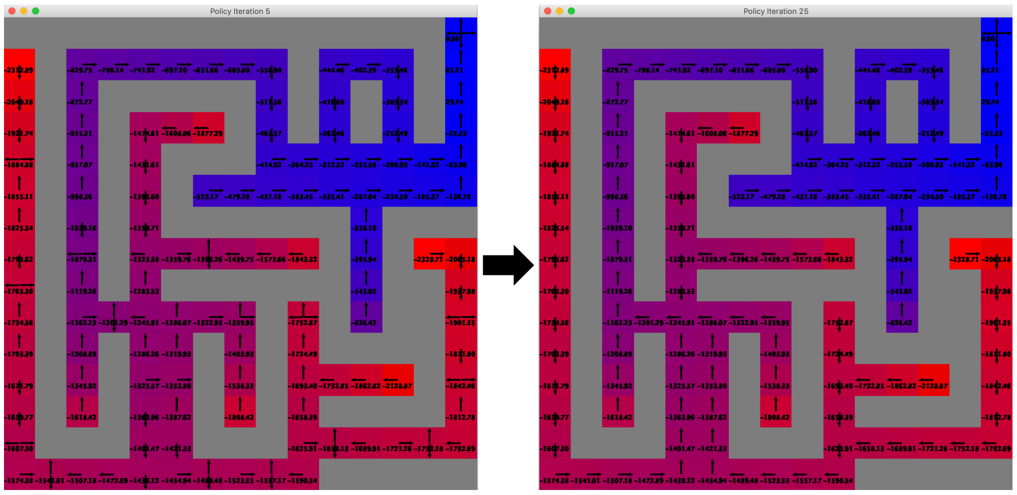
Slippery World Treasure Hunt



Let's try Policy Iteration on our Slippery World. We get the policy shown in the image at the right. It took **13 iterations** to converge to the optimal policy. As we can observe, the policy also recommends moving away from the negative -100 spot (red areas) and move closer towards the +100 spot (blue areas). Policy iteration converges much faster. In fact, it has converged to the same policy obtained by Value Iteration within fewer iterations.

Battery Recharge Maze

Once again, instead of improving the utilities at every iteration, we are redefining the policy at each step and computing the utilities for this new policy until the policy converges. Let's try Policy Iteration on our Battery Recharge Maze. We get the policy shown in the image at the below. On the left we see one of the initial policies obtained at iteration 5, and at the right we see the final policy obtained after it converges at **iteration 25**.



As we can observe, it took 25 iterations to converge to the optimal policy. This policy also recommends moving away from all the multiple negative -100 charging docks (red-pink areas) and move closer towards the +100 spot (blue areas). We can see that Policy iteration converges faster than Value Iteration. In fact, it has converged to the same policy obtained by Value Iteration within fewer iterations.

Q-Learning

This is the Reinforcement Learning algorithm chosen for this assignment. This one is different from the previous two algorithms; it is a model-free approach where the agent doesn't spend time learning the transition and reward models, but rather discovers what actions are good or bad by interacting with the world. MDP's are easy to solve, because we know everything about them (states, actions, rewards, and transition model), hence we could use Value Iteration and Policy Iteration to find the optimal policy. What would make it hard is if we didn't know some things about these. In this section, we will put a robot in the world and not give it any information about the transition model T and reward model R. Given these new conditions, we want it to explore the world and learn the optimal policy. The fundamental tradeoff of Reinforcement Learning is exploration vs exploitation. You can either exploit what you know and run the risk for not doing any better, and just explore and not take advantage of what you knew. Rather than learn the true utility of a state $U(s)$ (that represents the long term expected value of being in that state), we should learn a new function $Q(s, a)$ (long term expected value). Each time we interact with the world we gain an experience tuple. The components of an experience tuple we get are $< s, a, s', r >$. From those experience tuples, we can learn a policy. $Q(s, a)$ is the value of being in state s if you take action a, and then the optimal thing afterwards.

- Experience tuple $< s, a, s', r >$
- $Q(s, a)$ → expected immediate reward for taking action a in state s + sum of discounted reward for acting optimally afterward (in the future according to Q)

The formula to calculate $Q(s, a)$ given an experience tuple $< s, a, s', r >$, is (Balch, s.f.)

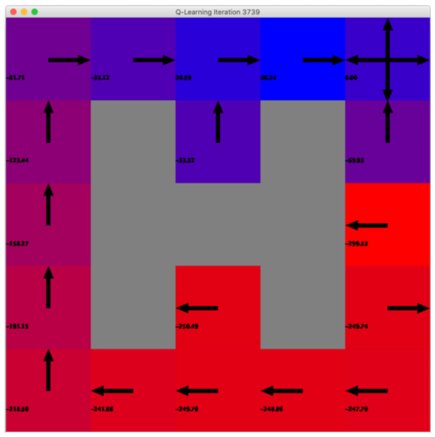
$$Q'(s,a) = (1-\alpha) \cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot Q\left(s', argmax_{a'}(Q(s',a'))\right), \text{where:}$$

- $r \ = R(s, a)$ → immediate reward for taking action a in state s
- $\gamma \ \in \ [0, 1]$ (gamma) → discount factor used to reduce the value of future rewards
- s' → the next state
- $argmax_{a'}(Q(s', a'))$ → action that maximizes the Q-value from all valid actions a' from state s'
- $\alpha \ \in \ [0, 1]$ (alpha) → learning rate used weigh new experiences versus past Q-values

These are the different things we experiment with so that our robot can learn the optimal policy. We follow the following algorithm:

- Initialize Q (zeros, or small random numbers near zero)
- Act and observe (s, a, s', r)
- Update Q
- Repeat

Slippery World Treasure Hunt



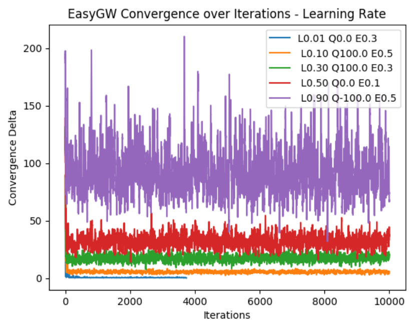
Let's try Q-Learning on our Slippery World. We get the policy shown in the image at the right. It took **3739 iterations** to converge to the optimal policy. As we can observe, the policy also recommends moving away from the negative -100 spot (red areas) and move closer towards the +100 spot (blue areas). The resulting policy is mostly similar to the one obtained by Policy iteration and Value Iteration, except for 3 states. However more broadly it yields the similar expected behavior, move away from the negative rewards and go closer to the positive rewards. As we can see, it has taken many more iterations for Q-Learning to obtain this policy because it has no information beforehand about the world, so it has to take steps and learn from each experience during the exploration.

For Q-Learning I experimented with the following that yielded different exploration techniques:

- **Discount factor γ** (gamma) `[0.90, 0.99]` where `0.90` did a better job weighing more current rewards but still giving some weight to future rewards. This one helped determine the importance of future rewards. Because in the small grid world there was only a big positive reward at the goal and many negative rewards for each step, it was necessary to adjust the discount factor such that the value of the +100 at the goal is significant enough, however not disregarding how many negative -4 it has to get before reaching the goal. Setting the discount factor at `0.90` allowed the learner to a obtain a policy that suggests taking a longer path to the goal, avoiding the shorter path which contained a big -100 reward on the way to get the +100 reward.
- **Learning rate α** (alpha) `[0.01, 0.1, 0.3, 0.5, 0.9]` where `0.01` did a better job weighing more newer experiences but still giving some weight to past experiences. This value basically allows us to weight how much importance to give to the result obtained via exploration, where a value closer to 0 forces the learner only to only exploit past knowledge and not learn much from the newly encountered experience. On the other hand, a value closer to 1 makes the learner pay more attention to the newly explored results and ignore prior experience. Ideally small values of alpha, or a slowly decreasing alpha make the Q-values converge faster. In the paragraph and plots at the end of this section we can observe that the lower the value of alpha, the faster the Q value converges.

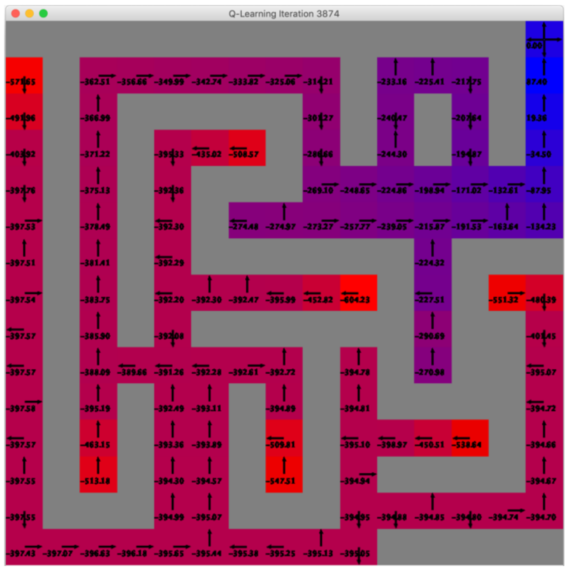
- **Initial Q values** `[-100, 0, 100]` where `0` did a better job to converge to a better policy.
- **Epsilon** `[0.1, 0.3, 0.5]` which corresponds to the exploration rate, saying how many steps will be done randomly. We generate a random number at each iteration, and if it is higher than epsilon we perform "exploitation" following the learned policy, and if not, we perform "exploration" taking a random step. Epsilon value of `0.3` gave a better policy. This value is used for the epsilon-greedy approach in Q-Learning which helps us make sure that we sufficiently explore our world to learn about it and converge to an optimal policy. For this small grid world exploring 30% of the time helped us get a policy comparable to Value Iteration and Policy Iteration.
- **Convergence Criterion** to check for convergence, I check the average performance over the last 10 episodes and check if it is smaller than < `0.25`

One of the values that gave the most changes in terms of early convergence and better policies was the alpha value. In the plot to the right for the Slippery World Treasure Hunt, I plotted the Convergence Delta vs Number of Iterations with best combinations of learning rate alpha, initial Q value, and epsilon. We can see the smaller the learning rate, the quicker the Q table converges. In this case the learning rate of `0.01` with initial Q values of `0` and an epsilon of `0.3` effectively converged at iteration 3739 (blue curve), yielding the policy in the previous figure, which is quite similar to the policy obtained by Value Iteration and Policy Iteration.



Battery Recharge Maze

In this section we apply the model-free approach Q-Learning to our Battery Recharge Maze problem. Once again, we will put a robot in the world and not give it any information about the transition model T and reward model R. Given these new conditions, we want it to explore the world and learn the optimal policy.



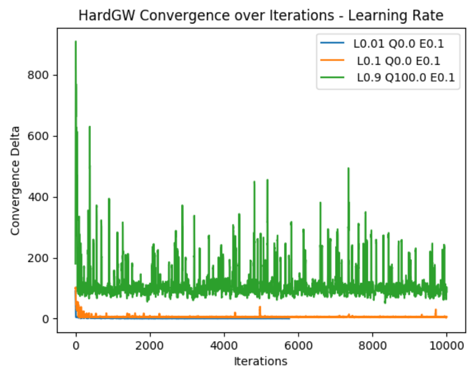
Let's try Q-Learning on our Battery Recharge Maze. We get the policy shown in the image at the right. It took **3874 iterations** to converge to the optimal policy. As we can observe, the policy also recommends moving away from the negative -100 spot (red areas) and move closer towards the +100 spot (blue areas). The resulting policy is mostly similar to the one obtained by Policy iteration and Value Iteration, except for certain states. However more broadly it yields the similar expected behavior, move away from the negative rewards and go closer to the positive rewards. As we can see, it has taken many more iterations for Q-Learning to obtain this policy because it has no information beforehand about the world, so it has to take steps and learn from each experience during the exploration.

For Q-Learning I experimented with the following that yielded different exploration techniques:

- **Discount factor γ** (gamma) `[0.90, 0.99]` where `0.90` did a better job weighing more current rewards but still giving some weight to future rewards. Because in the big grid world there was only a big positive reward at the goal and many -100 reward traps and negative rewards for each step, it was necessary to adjust the discount factor such that the value of the +100 at the goal is significant enough due to the presence of a lot of negative rewards present in the world. Like for the small grid-world, for this one setting the discount factor at `0.90` allowed the learner to a obtain a policy that suggests taking a path that heads directly to the goal, avoiding (moving away) from all the trap paths containing big -100 rewards.
- **Learning rate α** (alpha) `[0.01, 0.1, 0.9]` where `0.01` did a better job weighing more newer experiences but still giving some weight to past experiences. Once again, we could see that small values of alpha, made the Q-values converge faster. The alpha value of `0.01` could allow the learner to give more weight to the past experiences it has encountered vs the current experience it is exploring. In doing so, for this large grid world, the learner could converge to the optimal policy in fewer iterations.

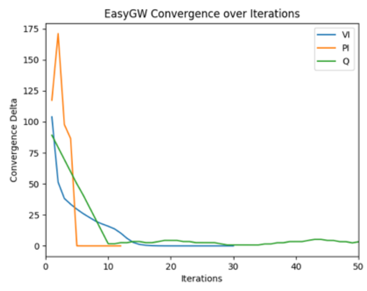
- **Initial Q values** `[-100, 0, 100]` where `0` did a better job to converge to a better policy
- **Epsilon** `[0.1, 0.3, 0.5]` which corresponds to the exploration rate, saying how many steps will be done randomly. We generate a random number at each iteration, and if it is higher than epsilon we perform "exploitation" following the learned policy, and if not, we perform "exploration" taking a random step. Epsilon value of `0.1` gave a better policy saying that we should explore randomly 10% of the time. This is fewer number of explorations required compared to the smaller grid world.
- **Convergence Criterion** to check for convergence, I check the average performance over the last 10 episodes and check if it is smaller than < `0.25`

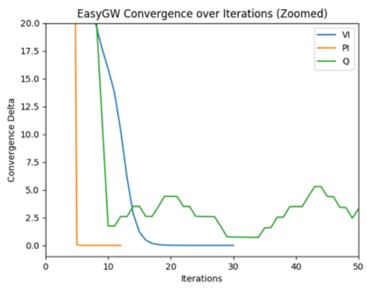
Once again, we can see that the value that gave the most changes in terms of early convergence and better policies was the Learning Rate alpha value. In the plot to the right for the Battery Recharge Maze, I plotted the Convergence Delta vs Number of Iterations with best combinations of learning rate alpha, initial Q value, and epsilon. We can once again see the smaller the learning rate, the quicker the Q table converges. In this case the learning rate of `0.01` with initial Q values of `0` and an epsilon of `0.1` effectively converged at iteration 3874 (blue curve), yielding the policy in the previous figure, which is quite similar to the policy obtained by Value Iteration and Policy Iteration. It is also worth to observe that for this larger maze it preferred to take random steps only 10% of the time and stay with the optimal policy 90% of the time. In other words, more exploitation and less exploration worked for this maze to converge to the optimal policy sooner.

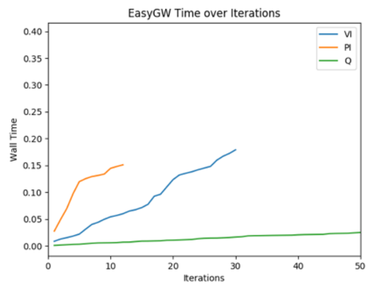


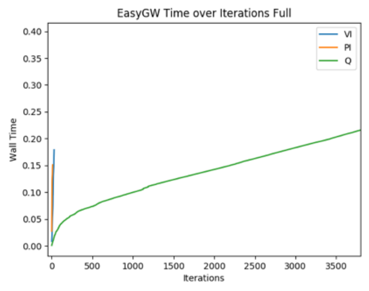
Analysis of Results

As we can observe from the previous sections, Policy Iteration converges faster than Value Iteration for both the problems. Policy Iteration may be converging faster because in Value iteration each iteration updates the utilities for each state, and implicitly the policy as well (via the max operation). However, because we are waiting for the utility values to converge, this process may take longer than the actual policy to converge. In practice, getting the magnitude of the utilities to be precise is unnecessary to get the optimal policy, which is a reason why policy iteration works, and therefore converges faster in the term of number iterations. Hence, when the policy stops changing, Policy Iteration converges, which is an early convergence criterion compared to Value Iteration where the utilities have to converge to then get the optimal policy.

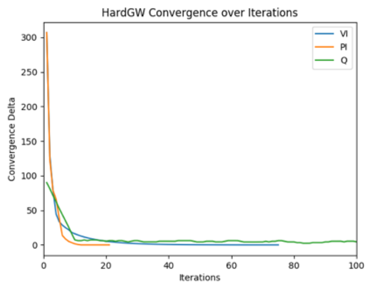


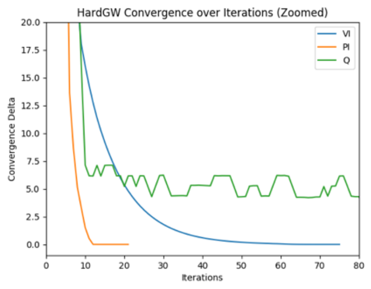


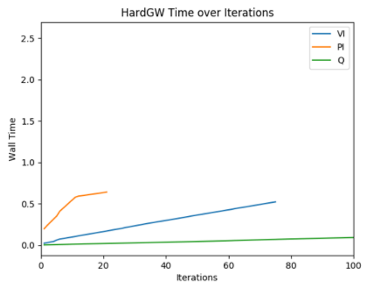


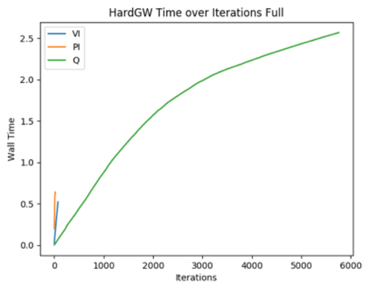


However, it is also worth to note that each iteration in policy iteration is more computationally expensive, because during each iteration we have to solve the system of n equations and n unknowns, which would take $O(n^3)$ time per iteration using standard matrices calculations in linear algebra. For the Slippery World Treasure Hunt (plots above), because the grid-world is very small, it seems that Policy Iteration is a more efficient approach than Value Iteration taking fewer iterations to converge to the optimal policy. In the above first two plots at the right we can effectively see that Value Iteration takes 46 iterations to converge, Policy Iteration takes 12 iterations to converge, and Q-Learning takes a lot more to converge (3739 iterations). It makes sense that Q-Learning takes longer as it has no information about its environment, and it must perform a balance between exploration and exploitation to learn to optimal policy while interacting with the world. With respect to wall time in the two plots on the right side above, we can see Value Iteration takes 0.18s wall time which is more compared to Policy Iteration with 0.15s wall time. This seems to reassure that if we know the model of the world, and we have a very small number of states, and we want to learn an optimal policy, Policy Iteration will perform better than Value Iteration in terms of number of iterations and runtime due to early convergence criterion. On the other hand, it is interesting to note that Q-Learning converged to a policy within a little more time than Value Iteration, however it takes many more iterations, and the policy is not as good as the one obtained when the models of the world are known. However, it is easy to believe that in a problem with a larger state space, $O(n^3)$ may be more prohibitive. It may still take fewer iterations than Value Iteration, but each iteration may take more wall clock time. This is something we can experimented with second grid world in the Battery Recharge Maze problem.









In a problem with a larger state space, $O(n^3)$ may be more prohibitive for each iteration update. It may take fewer iterations than Value Iteration but more wall clock time. In fact, our results do suggest this as well in the plots above for the Battery Recharge Maze Problem. Due to what was explained in the previous section, similarly Policy Iteration converges faster in the number of iterations. However, because this grid-world is much larger, it seems that Value Iteration is a more efficient approach than Policy Iteration despite taking more iterations to converge to the optimal policy. We can effectively see that Value Iteration takes 75 iterations and 0.52s wall time which is less time compared to Policy Iteration with 21 iterations and 0.64s wall time. We can effectively see that the $O(n^3)$ increased the runtime of each iteration in the Policy Iteration algorithm, so despite taking fewer number of iterations to converge to the optimal policy, it takes more wall clock time. In general, it seems that Policy Iteration is a better choice when the number of states is small, and Value Iteration when the number of states is large. On the other hand, we can see that in a bigger grid world, it takes even more wall time to obtain a good policy and many more iterations (3874 iterations) to converge to a good policy.

Conclusion Remarks

In general from these experiments we can conclude that if we know the model of the world, the transition model and the rewards model, then using Value Iteration or Policy Iteration are better options to get to the optimal policy, as these algorithms tend to converge pretty quickly and make use of all of the information about the model of the MPD to propagate the utilities and get the policy. Value Iteration solves the MDP by iteratively calculating the utilities for each state to then obtain the optimal policy. Policy Iteration, on the other hand, moves between obtaining a policy, calculating the utilities of each state using this policy, and then reupdating the policy. Value Iteration performs better when the world has a lot of states, and Policy Iteration performs better when there are a few numbers of states. In practice, Policy Iteration is overall computationally efficient as it takes a lot less iterations to converge to the optimal policy compared to Value Iteration, however its downside is that each iteration is more computationally expensive.

If we do not have a well-defined model of the world (transition model and reward function), such as with a robot navigating in an unknown environment, then Q-Learning is useful. Q-Learning does not worry to learn about the models of the transition and reward functions, but rather learns the policy by taking a step in the world and learning something. Tuning different hyper-parameters, we can change the way the exploration is performed. For example, altering the epsilon value we can decide how much how often to trust our current policy and do exploitation, and how often to take random actions to perform exploration. Altering our alpha learning parameter, we can decide how much to weigh newer experiences compared to past experiences. I learnt that using small alpha values help our Q-table to converge faster. Finally, I also experimented the discount factor gamma, which helped decide how much to weigh current rewards compared to future rewards to change how the exploration is performed.

Bibliography

Balch, T. (n.d.). *Machine Learning for Trading - Q-Learning*. Retrieved from Udacity: https://classroom.udacity.com/courses/ud501/lessons/5247432317/concepts/53538285920923

Isbell, C., & Littman, M. (n.d.). *RL 1- Markov Decicion Processes*. Retrieved from Udacity: https://classroom.udacity.com/courses/ud262/lessons/684808907/concepts/6512308840923

Norvig, P., & Russell, S. (2010). *Artificial Intelligence, A modern approach 3rd Edition.* Pearson.