Markov Decision Processes

Tevon Walker

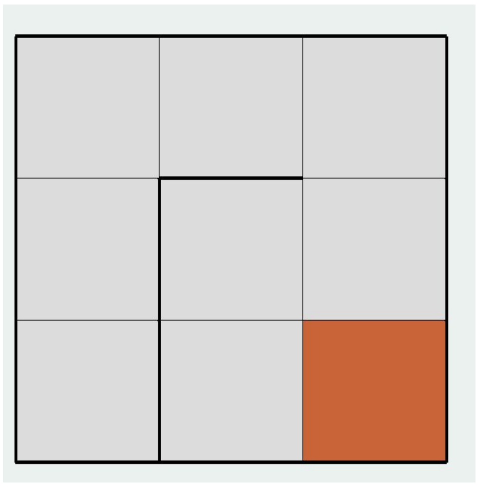
Introduction

This report will detail the analysis of a few algorithms solving some Markov Decision Processes (MDP). The MDPs shown here will be simple grid world problems. On paper, these MDPs may seem boring; but, they could be abstracted to more complex concepts such as path planning, sales optimization, and healthcare. I found a program written in Java that simulates reinforcement learning algorithms. This tool includes the value/policy iteration algorithms and the q-learning algorithm. Additionally, this tool includes several gridworld mazes; it gives the user the option to create their own maze, which includes options for grid sizes, wall placement, and goal state placement. I chose to use this tool because it is simple to use and understand. It also includes an implementation of *modified policy iteration*. This is important for analysis because it shows that, with some engineering, vanilla policy iteration can become faster. The simulator doesn't seem to include any implementation of discount factor, however. Because I have not written any code for these experiments, you will need to run the simulator yourself with my MDP parameters for experiment recreation.

Experimental Design

The experiments will be performed as follows: two different MDPs will be crafted. These two MDPs will differ in both their state space size and transition probabilities. After I have these two problems, I will run value/policy iteration and Q-learning on both MDPs. The simulator reports the time taken for the off-line planning algorithms to run. Multiple runs on the same MDP can yield different runs times, so experiments will be run more than once and times will be averaged. I will run experiments showing the effects of modified policy iteration, epsilon greedy exploration, and comparing offline vs online planning. After these algorithms have run, the policies will be visualized. The simulator outputs a visual representation of the resulting policy. The visualization is simply a line from the center of the grid cell toward the cell that it should move toward.

MDP One

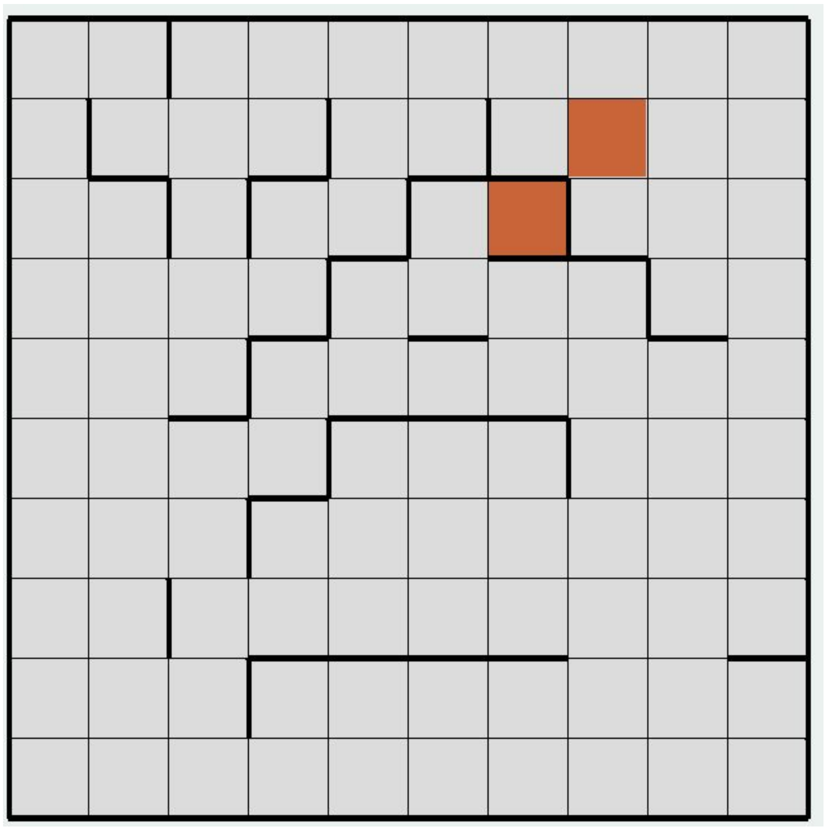


The first MDP is a simple one with only 9 states and 4 actions. I have made the environment transitions deterministic; that is, the transition probabilities between states are 1 for a single action and 0 for all other actions. The darker lines indicate walls in the environment. Taking an action that runs the agent into a wall results in a penalty of 50, and the agent remains in the state it was in before it hit the wall.

Results

Value Iteration	Policy Iteration	Modified Policy Iteration (5 policy eval steps)
0.167 ms / 7 steps	0.8333 ms / 4 steps	0 ms / 3 steps

MDP Two



The second MDP gives a much larger state space than the first. This time, there are two terminal goal states. I have also changed the transition probabilities from deterministic transitions to stochastic transitions. There is a 70% chance of successful moving and a 30% chance of moving in a direction that the agent did not choose.

Results

Value Iteration	Policy Iteration	Modified Policy Iteration (5 policy evaluation steps)
31.833 ms / 84 steps	41.5 ms / 7 steps	12.17 ms / 13 steps

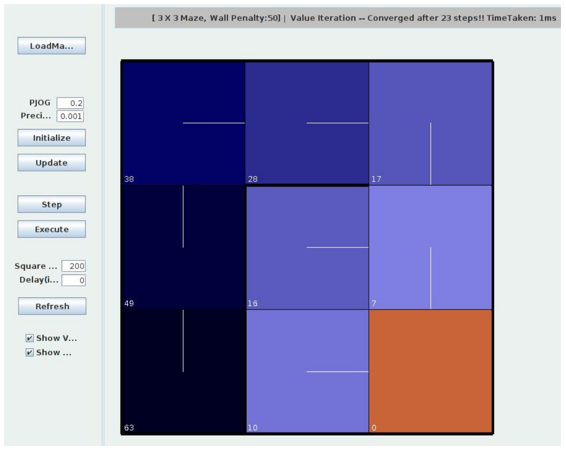
Analysis

Value Iteration

The value iteration solved the first MDP in only 0.167 ms on average using 7 steps to do so. I attribute the fact that the transitions are deterministic to the quickness of the algorithm. Because there is no stochasticity in this environment's transitions, the cost of each grid cell collapses to minimum number of steps needed to reach the goal state from the grid cell. Let us examine the bellman equation for value iteration:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

The equation must compute the expectation over possible transitions states in the sigma term. When the environment is stochastic, the algorithm must consider the values contained at all other states. This makes the value fluctuate more and causes slower convergence. Whereas in the deterministic case, that sigma term disappears. Because only one s' will have a 1 and all others will be zero, the value at that cell is not influenced by all neighboring cells. The cell that is being updated gets its information from only one cell, and that makes the convergence much faster. If I simply increase the noise of transitions from 0 to 0.2, the algorithm takes an additional 16 steps for a total of 23 steps to solve the MDP. **The determinism in the MDP makes the algorithm use less steps**. Look at the picture below showing what happens when the MDP variable, PJOG, is changed from 0 to 0.2.

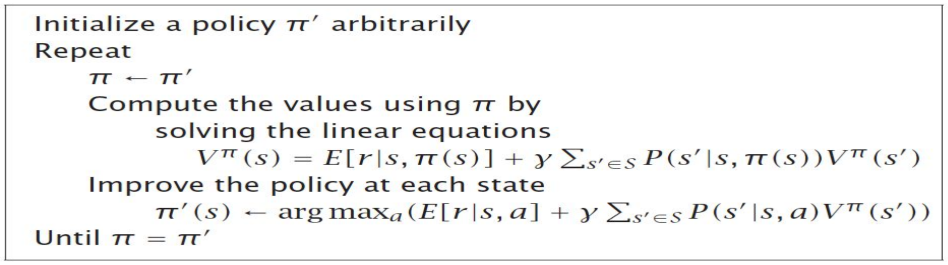

[3 X 3 Maze, Wall Penalty:50] | Value Iteration -- Converged after 23 steps!! TimeTaken: 1ms

PJOG 0.2
Preci... 0.001
Square ... 200
Delay(i... 0

The second MDP had stochasticity, and much more states. This caused value iteration to take much more iterations and time. The summation term computes over all possible other states, and this happens for every state. This creates a time complexity of O(S^2*A), where S is the number of states and A is the number actions. This property of the algorithm makes it scale very poorly to larger state spaces. Even though this property is true, value and policy iteration have properties that make it suitable for dynamic programming. Using memoization, these algorithms can be optimized to O(S*A) time complexity. That is what is implemented in the simulation, and why the ~10-times increase from 9 states in MDP one to 100 states in MDP two results in ~10-times more steps as opposed to 100-times more steps. Value iteration took 7 steps in MDP one and 84 steps in MDP two. The dynamic programming optimization prevented value iteration from using ~700 steps that a naive implementation likely would have.

Policy Iteration

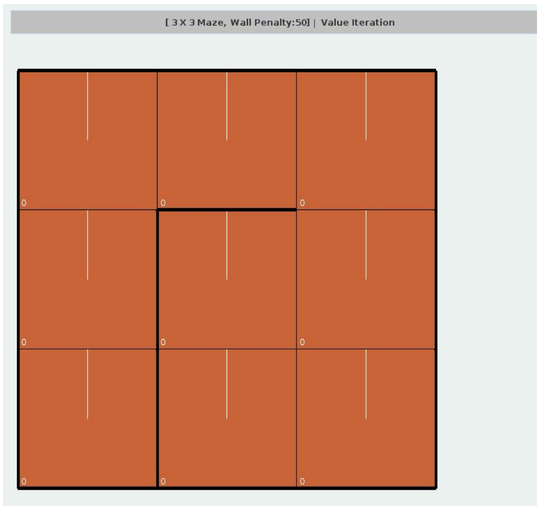
Policy iteration used way less steps, but it took more time to converge. How could this be? The reason that policy iteration converges in less steps than value iteration is that policy iteration is biased towards finding the optimal policy, whereas value iteration is biased toward finding the optimal value function. Let us examine the policy iteration algorithm.

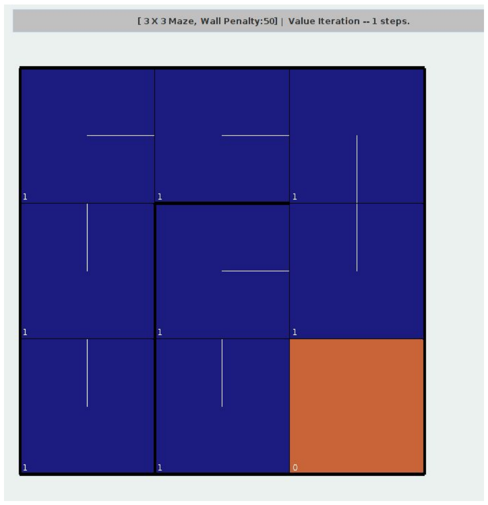


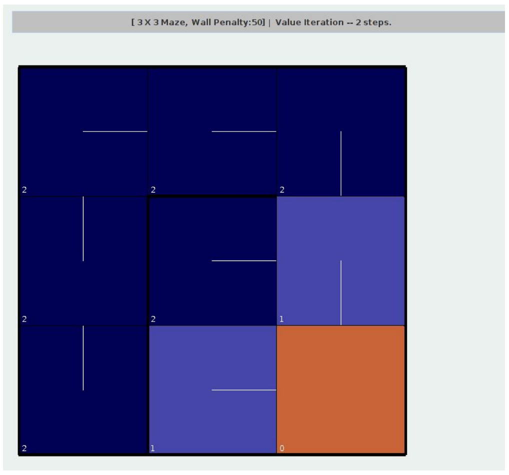
```
Initialize a policy $\pi'$ arbitrarily
Repeat
    $\pi \leftarrow \pi'$
    Compute the values using $\pi$ by
        solving the linear equations
        $V^{\pi}(s) = E[r|s, \pi(s)] + \gamma \sum_{s' \in S} P(s'|s, \pi(s)) V^{\pi}(s')$
    Improve the policy at each state
        $\pi'(s) \leftarrow \arg\max_a (E[r|s, a] + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi}(s'))$
Until $\pi = \pi'$
```

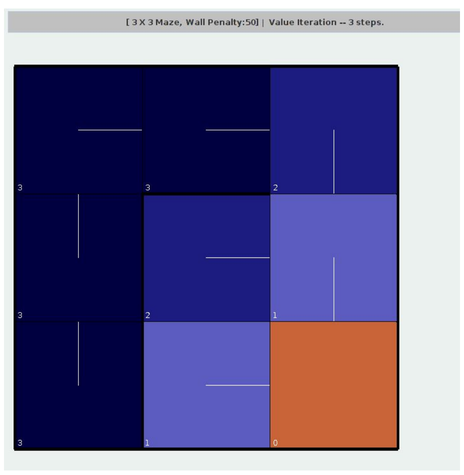
Photo taken from https://medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-978f9e89ddaa

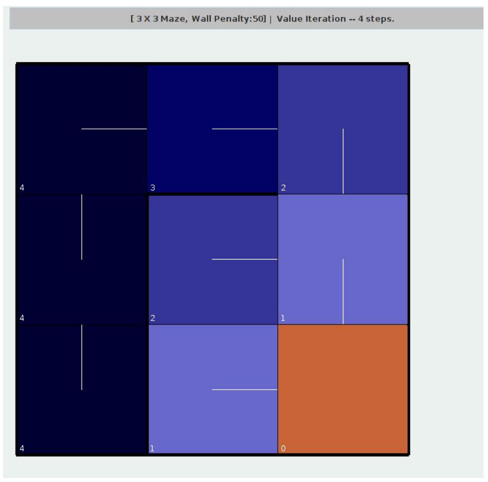
Policy iteration works by evaluating a policy and then improving that policy. To evaluate a policy, value iteration is used, but instead of computing expectations over optimal actions, the algorithm computes expectations over actions selected by the policy, and this is what determines the value of the policy. After policy evaluation has been performed, policy extraction is used to make a new policy using the value function from the policy evaluation step. This policy extraction step is guaranteed to improve the policy monotonically. What this means is that after each iteration of policy iteration, the policy can either improve or stay as it was. The policy will never get worse. What makes policy iteration use less iterations than value iteration is the fact that a policy often, if not always, converges before the value function converges. Let us look at the evolution of the policy across iterations of value iterations for MDP one.

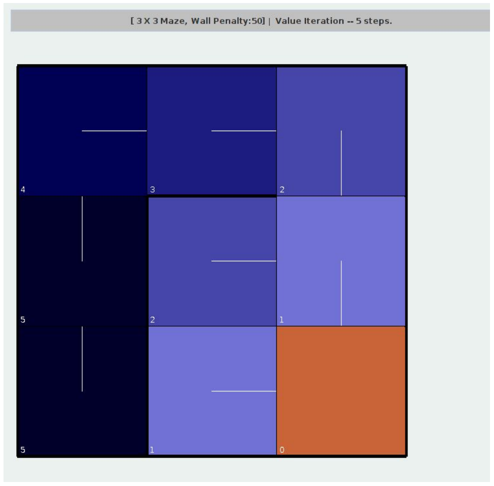


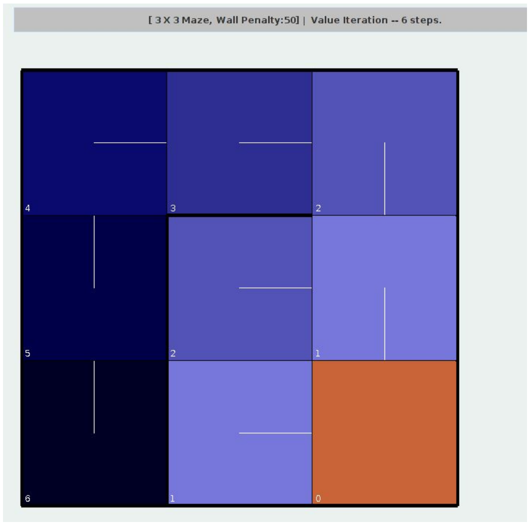

[3 X 3 Maze, Wall Penalty:50] | Value Iteration -- 1 steps.
1 1 1
1 1 1
1 1 0

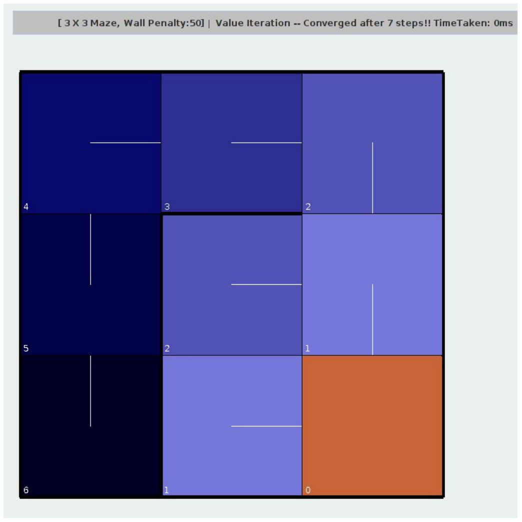

[3 X 3 Maze, Wall Penalty:50] | Value Iteration -- 2 steps.
2 2 2
2 2 1
2 1 0


[3 X 3 Maze, Wall Penalty:50] | Value Iteration -- 3 steps.
3 3 2
3 2 1
3 1 0


[3 X 3 Maze, Wall Penalty:50] | Value Iteration -- 4 steps.
4
3
2
4
2
1
4
1
0

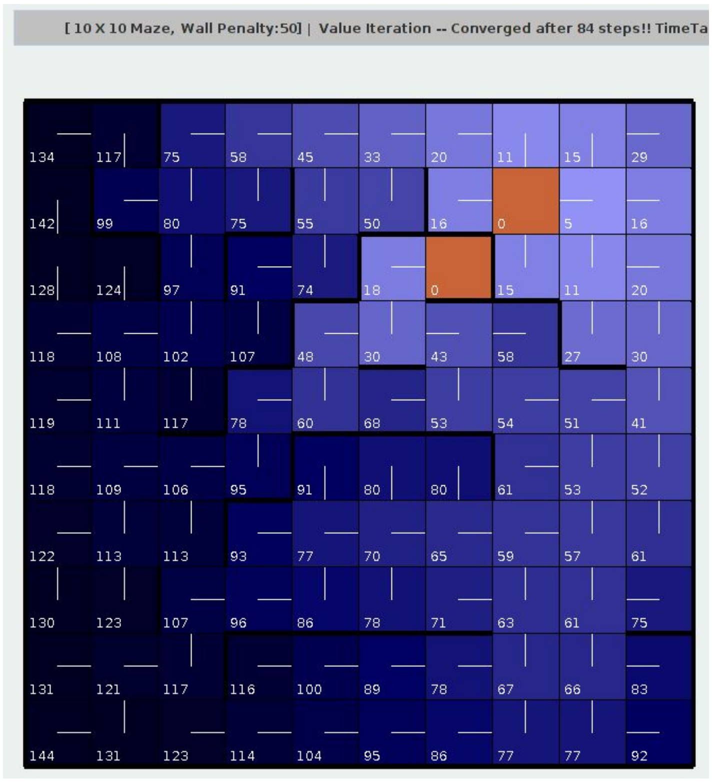

[3 X 3 Maze, Wall Penalty:50] | Value Iteration -- 5 steps.
4
3
2
5
2
1
5
1
0


[3 X 3 Maze, Wall Penalty:50] | Value Iteration -- 6 steps.
4
3
2
5
2
1
6
1
0


[3 X 3 Maze, Wall Penalty:50] | Value Iteration -- Converged after 7 steps!! TimeTaken: 0ms
4
3
2
5
2
1
6
1
0

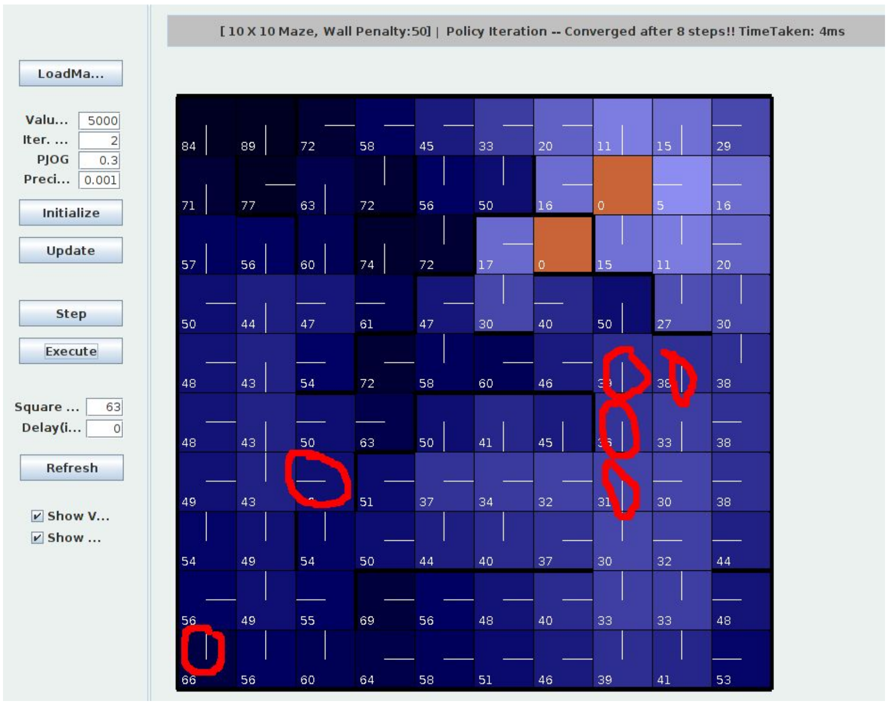
Notice that between steps 2 and 7, the policy does not change at all. Value iteration does not return a policy until the value at each state has stopped changing. Policy iteration improves upon this shortcoming by monitoring the **change in policy** as opposed to **change in value function**. As demonstrated above, policy convergence can occur in much fewer steps than convergence in value function, and **this is why policy iteration takes fewer iterations**. Even though the policy iteration algorithm uses less steps, it is often slower than value iteration in terms of time taken to converge. The reasoning behind this is that in each iteration of policy iteration, value iteration is performed in the policy evaluation step. This becomes quite expensive as the state space gets larger. Notice that in MDP one, the time and number of steps of value iteration and policy iteration are comparable; however, in MDP two, the discrepancy is much more felt. Luckily, researchers have come up with a way to counteract this undesirable property of the policy iteration algorithm. *Modified* policy iteration is a version of policy iteration that uses shortcuts to speedup the bottlenecking step of policy evaluation. One such version is a version that uses the value function from the previous policy evaluation iteration to 'warm start' the computation and save algorithm iterations. Another modified policy iteration algorithm is one that simply limits how many times the policy evaluation can run. This is what is implemented in the CMU simulator that I am using. Instead of running policy evaluation until the value function converges, I can limit the number of iterations that the policy evaluation step does. This feature enables my policy iteration experiment to run just as quick or quicker than value iteration while using less steps. This is a neat piece of engineering that gives some performance for solving the MDP. Look at the performance of modified policy iteration when I constrain the policy evaluation step. The algorithm uses less time to complete than both value iteration and vanilla policy iteration. One thing to keep in mind when using this version of modified policy iteration is that one should be very careful with how they limit the number of iterations of policy evaluation. Watch what happens when I allow a maximum of two iterations of policy evaluation on MDP two. The optimal policy found by value iteration is shown below. The policy found by modified policy iteration with max 2 policy eval steps is also shown. Notice how the two policy are not the same; I have circled some differences in red. Even though the algorithm has converged, it has found a non-optimal policy. Because only 2 iterations were allowed, the value likely wasn't able to propagate throughout the MDP, and policy wasn't able to properly update. **This shows that modified policy**

iteration must be used with care, it may be fast, but it could also return a sub-optimal policy since policy evaluation is not allowed to fully run.



[10 X 10 Maze, Wall Penalty:50] | Value Iteration -- Converged after 84 steps!! TimeTa

134	117	75	58	45	33	20	11	15	29
142	99	80	75	55	50	16	0	5	16
128	124	97	91	74	18	0	15	11	20
118	108	102	107	48	30	43	58	27	30
119	111	117	78	60	68	53	54	51	41
118	109	106	95	91	80	80	61	53	52
122	113	113	93	77	70	65	59	57	61
130	123	107	96	86	78	71	63	61	75
131	121	117	116	100	89	78	67	66	83
144	131	123	114	104	95	86	77	77	92



Valu... 5000
Iter. ... 2
PJOG 0.3
Preci... 0.001
Square ... 63
Delay(i... 0
☑ Show V...
☑ Show ...

[10 X 10 Maze, Wall Penalty:50] | Policy Iteration -- Converged after 8 steps!! TimeTaken: 4ms

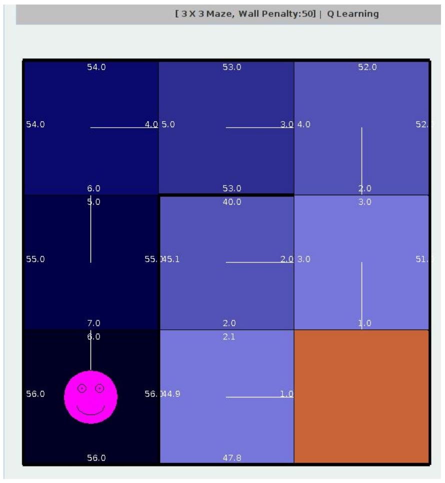
84	89	72	58	45	33	20	11	15	29
71	77	63	72	56	50	16	0	5	16
57	56	60	74	72	17	0	15	11	20
50	44	47	61	47	30	40	50	27	30
48	43	54	72	58	60	46	39	38	38
48	43	50	63	50	41	45	36	33	38
49	43	[illegible]	51	37	34	32	31	30	38
54	49	54	50	44	40	37	30	32	44
56	49	55	69	56	48	40	33	33	48
66	56	60	64	58	51	46	39	41	53

Reinforcement Learning

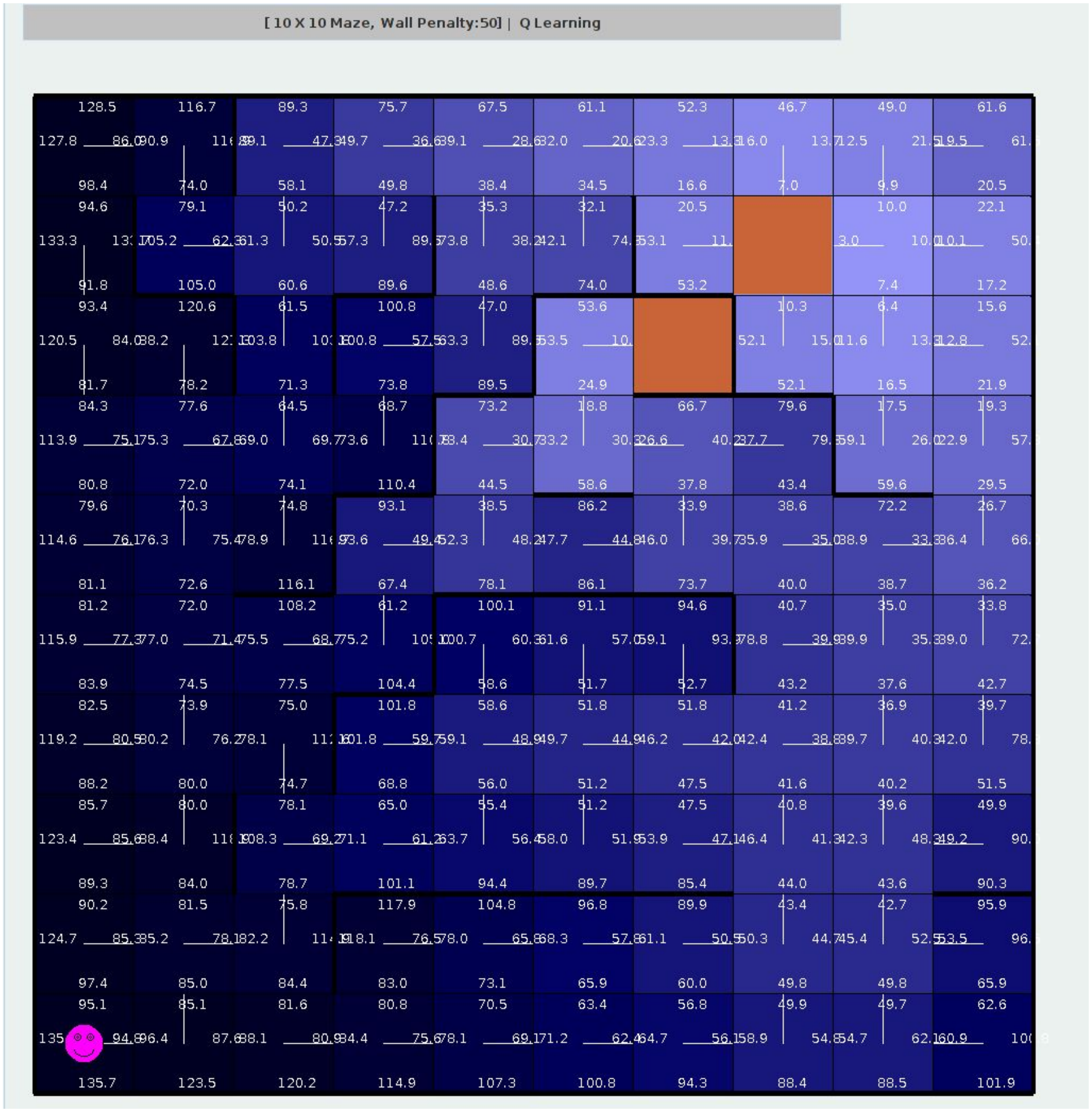
Now, I will use Q-learning to solve the two MDPs. Q-learning is an online planning method to solve MDPs. Q-learning is considered online because it **cannot** plan out actions prior to actually interacting with the environment. Q-learning is a **model-free** reinforcement learning algorithm. This means that the algorithm does <u>not</u> have access to the dynamics nor the reward probabilities of an environment. Because of this, Q-learning must interact directly with an environment to learn from it. For these experiments, the simulator doesn't report training times for the algorithm, so I will simply show the resulting state-action values.

Results

MDP One

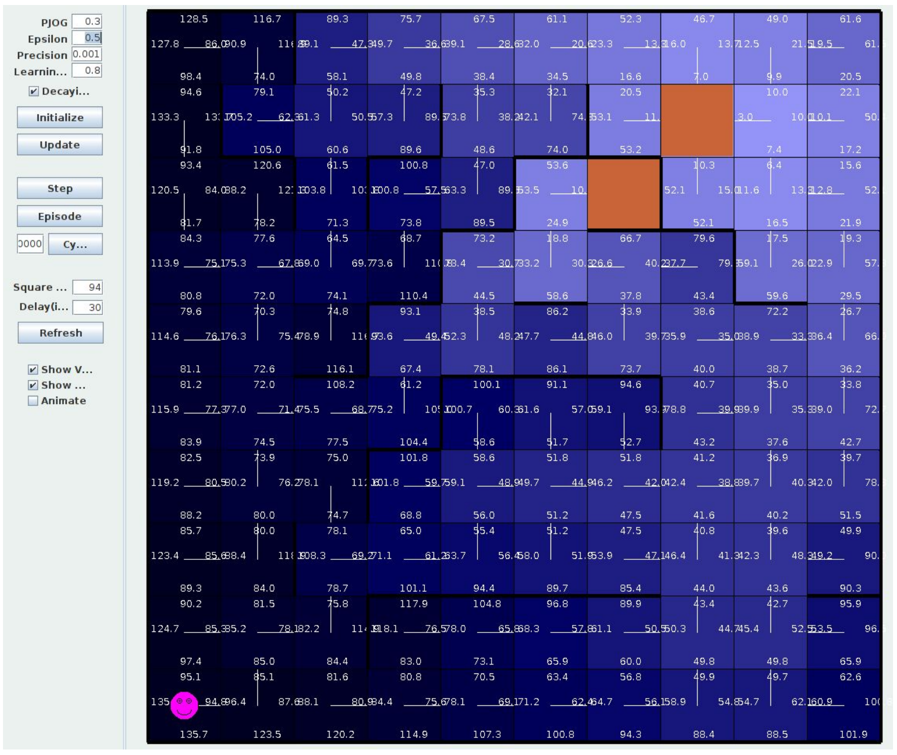


MDP Two

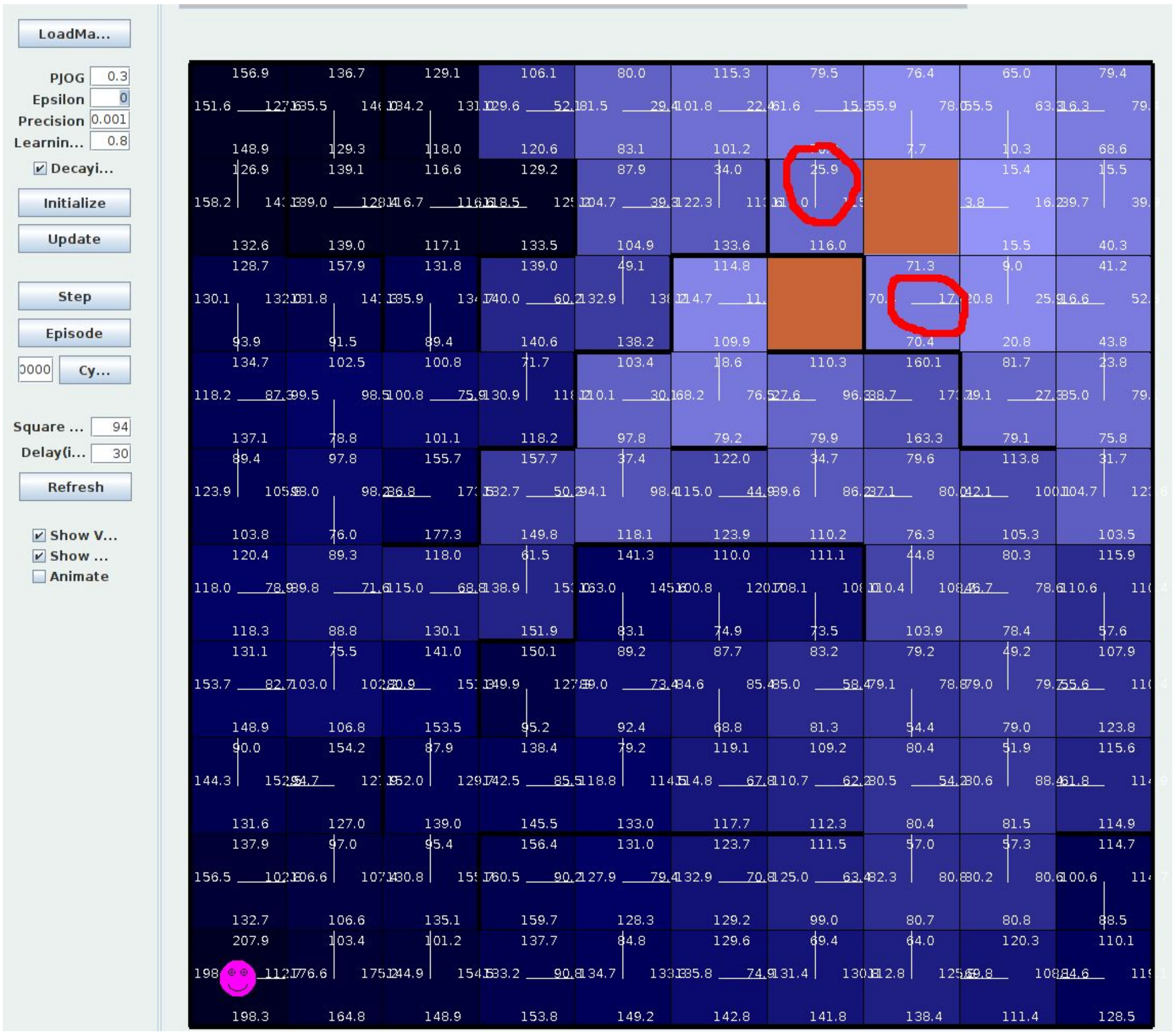


Analysis

Like I've stated above, the simulation tool did not record run times for Q-learning. So, to analyze this algorithm, I will look at the effect of epsilon-greedy exploration and state size on this algorithm. In the first MDP, the Q-learner easily finds the optimal Q-values. Because this is such a small state space, the Q-learner has high probability of visiting all states many times. Look at the minimum value at each state; these values are equal to the state values found by value iteration. It is the case that $V^*(S) = \min_A Q^*(S,A)$, and that is shown here. The reason that the value function optimum and the Q-value optimum are equivalent is due to the small state space of MDP one. If the state space of the problem is much larger, as in MDP two, then this may not be the case. In fact, Q-learning is only guaranteed to converge with infinite training time. This becomes apparent in the second MDP. Notice that the values are not equivalent to the optimal value function found by value iteration. This may not matter, however; looking at the resulting policy from Q-learning, it seems to be acting identically to how the policy from value iteration would. My thoughts on this are that **Q-Learning doesn't have finite convergence conditions for all environments and environments with larger state spaces must have a Q-learner run for much more training iterations.** Additionally, Q-learning benefits from **dense reward functions that give feedback often**. Imagine for a second that the MDPs used here didn't give a +1 cost each time an action was taken, and the only feedback was when the agent entered the goal state (sparse rewards). It would take *significantly* more time for the q-values to propagate throughout the state-action pairs of the environment. The fact that the Q-learner constantly receives cost when it moves helps to drive it to the goal state. Another important aspect of Q-learning is exploration. Exploration helps the agent learn about the environment when it first starts its interacts. When the agent initially starts its interactions, it does not know anything about the environment: typically the q-table is initialized to zero or randomly. So, if the algorithm tried to take maximal actions according to values, it may end up stuck in local optimas, or the agent may find itself repeatedly visiting the same states. To counteract this, I introduce some randomness in the form of **ε-greedy exploration**. This method takes a random action with probability ε and takes the maximal action value with probability 1-ε. This enables the agent to learn about the environment without getting stuck in the same routine of visiting the same states. In most implementations of this method, ε is annealed with more iterations. **This is similar to how the learning rate is decayed in the training of neural networks**. The higher learning rate allows the optimizer to **explore** and escape potential local minimas that the parameters may be in. When the learning rate is decayed, the parameters are starting to converge upon a single space in the optimization landscape, and **exploitation** comes into play. For the first MDP, exploration wasn't as crucial since the state space size was so small. I ran experiments with epsilon set to 0.5 and set to 0, and both runs resulted in the same policy with similar Q-values. Exploration was important, however, in the second MDP. When I prevented the agent from exploring (set ε to 0), I found that the resulting policy was suboptimal, I found neighboring states with arrows pointed at each other (causing cycles) and I also found that sometimes the optimal action for a state next to one of the goal states was <u>not</u> for it to enter the goal state. This is due to the fact that the agent wasn't able to gain enough knowledge about the environment. The agent was constantly exploiting the knowledge it gain from the beginning, and as a consequence, it found Q-values that yielded a *sub*optimal policy. When I set ε to 0.5 and allowed the agent to explore more, the policy improved. The Q-values may not have been the same as those found by value iteration, but the policy found was near the same. I ran Q-learning for 500,000 iterations to get the agent to learn; but again, **no optimal Q function is guaranteed to emerge unless the agent explores the environment *<u>indefinitely,</u>* especially with such a complex state space**. Direct your attention to the two pictures below. The top picture shows the resulting policy when ε was set to 0.5. The policy makes sense, the agent has learned desirable behavior. Now, look at the picture below it. I have circled in red two actions that are suboptimal. Obviously, the agent should move into the goal state; however, it has learned to miss the goal state because it did not explore enough.



PJOG 0.3
Epsilon 0.5
Precision 0.001
Learnin... 0.8
[✓ Decayi...]
Square ... 94
Delay(i... 30
[✓ Show V...]
[✓ Show ...]
Animate



PJOG 0.3
Epsilon 0
Precision 0.001
Learnin... 0.8
☑ Decayi...
0000
Square ... 94
Delay(i... 30
☑ Show V...
☑ Show ...
☐ Animate

One problem with ε-greedy exploration is that it treats all actions equally when randomly selecting an action. Another method of exploration, called *Boltzmann Distribution*, uses the Q-value of each action to weigh its probability of being chosen during explorations. Perhaps this could have worked better in the second MDP since the state space was so large, and reaching some states could be very unlikely. I *would* have experimented with this technique, but the tool that I selected only had options for the ε-greedy technique. If I had to compare Q-learning to the off-line planning methods, I would say that the offline planning algorithms worked better. They gave optimal policies in finite time. Q-learning gave an optimal policy on the first MDP, but I suspect that was only the case because of the small state space size of the problem. When the problem became much larger, as in the second MDP, the Q-learner needed far more iterations (500,000) and exploration to achieve a similar policy; but this was not guaranteed to be the optimal policy. The key takeaway is the policy/value iteration perform better when a model of the environment is known; however, **it is often the case that bigger, more interesting problems have unknown dynamics and no environment model is available**. This is when value/policy iteration do not work and you must learn a Q-function via Q-learning. Also, sometimes a state space is too large or the state space is continuous, and the Q-function cannot be learned within a table. This is when *deep* Q-learning comes in. Instead of using a fixed table in memory to store Q-values, a neural network approximates the Q-function. This has shown great performance on many atari games with discrete action spaces. In the case of continuous action spaces, policy gradient methods directly learn mappings from state to action without worrying about associating a value to a state or state-action pair.