Reinforcement Learning

Swati Prasad
sprasad33@gatech.edu

CS7641: Machine Learning

Abstract:
In many cases, agents must deal with environment that contains nondeterminism. Perhaps the agent's action can fail, or its sensors can be inaccurate or outside forces might change the environment. In this assignment we will focus on a particular sort of nondeterminism: nondeterministic actions. Example of nondeterministic environment is a block might slip out of robot's gripper or web page might fail to load, or a doorway might be locked. In these environments, taking same action from the same state could lead to two different outcomes. In this case, our transition functions takes an input state s, action a, and successors state s' and return the probability of reaching s' from s when taking an action a.
This project will explore various reinforcement learning techniques an agent can use to make decisions. An analysis of two interesting Markov Decision Process (MDPs) is conducted using two planning algorithms, Value iteration and policy iteration and one algorithm of choice Q learning. The two MDPs are selected to demonstrate the different behaviors of reinforcement learning for MDPs with small and large number of states. We will explore MDPs on the Tiny Grid Sample and Big Grid Sample.

Markovian environments
The role of history in determining a successor state. Probability of reaching state s' from s given action a might also depends on states that were previous to s. If the number of previous states can be bounded, we call this environment Markovian or says the environment obeys the Markovian assumption. To solve this environment, we construct a policy. A policy is mapping from states to actions that tells our agent the optimal action to take in any given state with respect to a particular goal. One thing to note that a policy is only useful with respect to a particular goal that our agent wants to achieve. If we change goals, it will need a new policy.

Markov Decision Processes
MDP are a way to model sequential decision making under uncertainty. To formalize this, we must introduce a few concepts:
Our problem has an initial state s0
States and actions are discrete.
Each state is our world will have reward r associated with it.
Our problem has a transition function T(s, a, s') -> [0,1] that indicates the probability of transitioning from state s to s' when action a is taken.

A discount factor $\gamma \in [0, 1]$ applied to future rewards. This represents the notion that a current reward is more valuable than one in the future. If γ is near 0, future rewards is almost ignored; a γ near 1 places a great value on future reward.
$T R = EU(P\infty t=0 \gamma tR(st)|\pi)$
This says that the reward from a policy is the sum of the discounted expected utility of each state visited by that policy. The optimal policy is the policy that maximizes this

equation. In this assignment, we will look into at three algorithms for discovering that policy.

Value Iteration:
The idea behind the value iteration: if we knew the true value of each state, our decision would be simple: always choose the action that maximizes expected utility. But we don't initially know a state's true value; we only know its immediate reward. Example, a state might have low initial reward but be on the path to a high-reward state.

The true value (or utility) of a state is the immediate reward for that state and the expected discounted reward if the agent acted optimally from that point on:
$U(s) = R(s) + \gamma \max_a P\ s'\ T(s, a, s')U(s')$
Note that the value for each state can potentially depend on all of its neighbor's values. If our state is acyclic, we can use dynamic programming to solve this. Otherwise, we use value iteration.
1) Assign each state a random value
2) For each state, calculate its new U based on its neighbor's utilities.
3) Update each state's U based on its calculated above:
$U_{i+1}(s) = R(s)+\gamma \max_a \text{sigmas}'\ T(s, a, s')U_i(s')$
4) If no value changes by more than δ, halt
This algorithm is guaranteed to converge to the optimal (within δ) solutions.

Policy Iteration
Value iteration has two weaknesses: first, it can take a long time to converge in some situations, even when the underlying policy is not changing, and second, it's not actually doing what we really need. We actually don't care what the value of each state is, that's just a tool to help us find the optimal policy. So why not just find the policy directly?
We can do so by modifying value iteration to iterate over policies. We start with random policy, compute each state's utility given that policy and then select a new optimal policy.

1) Create a random policy by selecting a random action for each state.
2) While not done:
    a. Compute the utility for each state given the current policy.
    b. Update state utilities.
    c. Given the new utilities, select optimal action for each state.
3) If no action changes, halt

Q Learning:
Value iteration and policy Iteration work wonderfully for determining an optimal policy, but they assume that our agent has a great deal of domain knowledge. Specifically, they assume that the agent accurately knows the transition function and the reward for all states in the environment. This is actually quite a bit of information, in many cases our agent may not have access to this.

Fortunately, there is a way to learn this information. In essence, we can trade learning time for a priori knowledge. One way to do this is Q learning. Q learning is a form of model-free learning, meaning that an agent doesn't need to have any model of the environment. It only needs to know what states exist and what actions are possible in each state.
The way this works is as follows: we assign each state an estimated value, called Q-value. When we visit a state and receive a reward, we use this to update our estimated of the value of that state. Q-value can be written as:
$Q(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a')$

The value of taking action a in state s is the immediate reward for the (s,a) pair, plus the value of the best possible state-action pair for the successor state.
We can use this to update the Q-value of a state-action pair as a reward r is observed:

$$Q_{t+1}(s, a) = r + \gamma \max_{a'} Q_t(s', a')$$

However, how should an agent choose which action to test? We are assuming that the agent is learning in an online fashion. This means that it is interleaving learning and execution. This brings up a trade off: learning is costly; time that our agent spends making mistakes is time that can't be spent performing correct actions. On that other hand, time is needed to learn, and some time spent making errors early on can lead to improve overall performance.

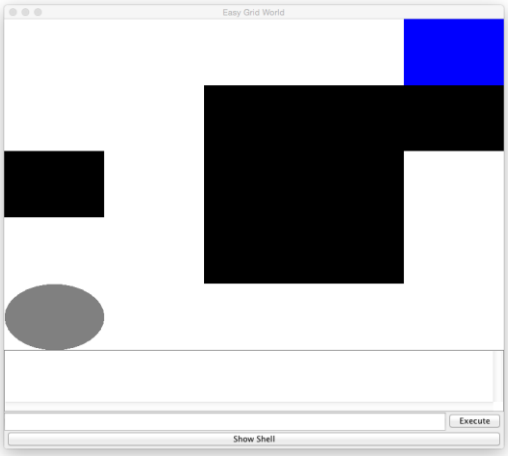
In the early stages of execution, it is important to explore and try unknown actions. There is a great deal to be gained from learning, and agent doesn't have enough information to act well in any case. Later in its life, the agent may want to almost always choose the action that looks best; there is little information to be gained, and so the value of learning is negligible.

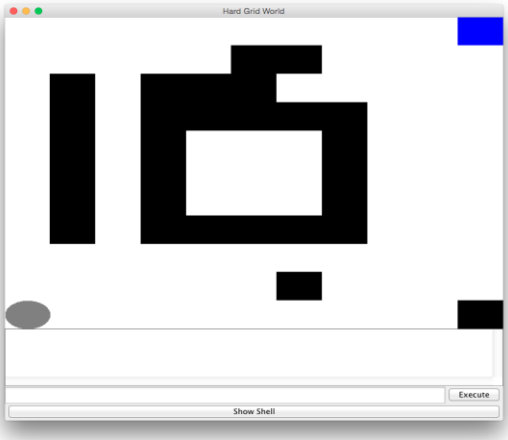
Algorithm Implementation:

The implementation of MDPs utilized existing Java code, which was adapted from original source BURLAP Reinforcement Learning Package. The behaviors of the three reinforcement algorithms explored, using various parameters to observe the impact on the convergence, computation time and policy. The results were exported to CSV files and analyzed using Excel to explore trends and generate plots.

The original BURLAP code contains BURLAP Grid Worlds. The examples maintain the ideas f cost per action and a terminal reward. The original BURLAP package includes a Grid World example in which the agent exists in a 2-dimensional plane. The datum is in the bottom-left corner (0,0) and has the choice of 4 actions to move in any cardinal direction to navigate the plane and avoiding walls (in black) to find the terminal state (blue). The agent is encouraged to reach the terminal state by forcing it to make an action, where all actions have a small negative cost function (for example, -1 per action) except when the terminal state is reached, in which a large reward is awarded (for example, 100). The objective of the agent is to maximize the net reward by reaching the terminal state in the fewest number of actions.
As a way of modeling the "real world", BURLAP can make the actions probabilistic (instead of deterministic) where the agent's individual actions do not always have the intended consequence. For example, when the agent wants to go Up, there is an 80% chance the agent will actually go Up, but a chance to go Down, Left or Right (the remaining 20% is split equally 3 ways). To model the agent's priorities of the present and future, a discount factor is applied to the expected value of each action. MaxDelta allows the algorithms to end when the delta in utility between consecutive iterations is below a threshold.
In Q-Learning, there are additional parameters to consider. The Q_Initial value is the value assigned to all states before updating the values with "truth". When Q_Initial is high, this initially makes all states viable for the agent to pursue, whereas when Q_Initial is low, all states are less viable and thus the agent is encouraged to exploit more than explore. The exploration strategy determines how the agent decides whether to explore or exploit. The decision process is a function of the approach (such as epsilon greed, random, or a form of simulated annealing) and the value of epsilon ε _(a threshold for random generation below which an agent chooses to explore). Lastly, the learning rate is a relative measure of the importance of recent and older information when an agent makes a decision.

Grid Worlds (Tiny and Big):





fig(1) fig(2)

Interesting Problem:

Markov decision processes exemplify sequential problems, which are defined by a transition model and a reward function, situated in an uncertain environments. The environment we will be observing is the grid world, which is analogous to the problem of solving for the best moves in a board game. We can view a sequential decision problem as a situation where an agent's utility is dependent on a sequence of decisions that the agent can make. Our focus is on sequential decision problems because sequential decision problems lay the groundwork for reinforcement learning.

Sequential problems are intriguing due to the components that contribute to the makeup of the problem and how each of the components affects the solution. The infrastructure that we will focus on is the Markovian transition model and additive rewards which is called a Markov decision process (MDP). Given a grid world with obstacles, terminals, a start state, rewards, and moves, we will find the optimal policy of the grid world, the policy that yields the highest expected utility. The result is a world where each state has a direction and each direction points to the best direction to take when the agent is at that state; the best direction being the direction that will yield the highest expected utility. To solve for the optimal policy, we use two different algorithms: value iteration and policy iteration. Each of the algorithms will produce an optimal policy. We will be observing the results of these two algorithms and comparing them with each other.

Implementation:

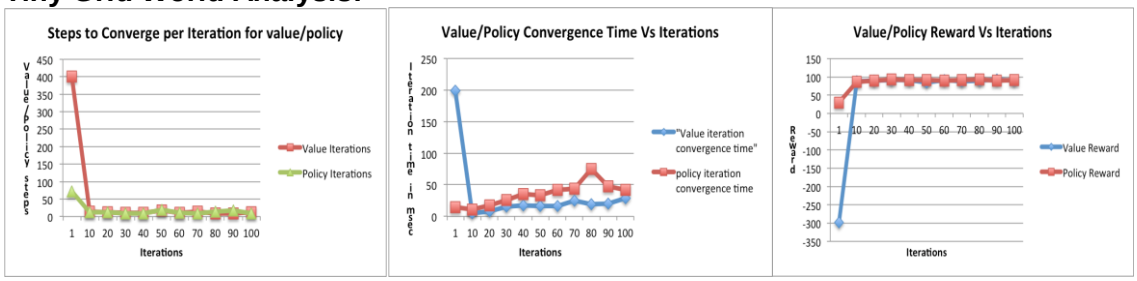
The first problem is Tiny Grid problem of 5X5 state with possible 17 possible states shown in fig(1) and second problem is Big Grid Problem of 11X11 states with possible 104 possible states fig(2). As these examples are implemented on BURLAP, the starting position is always at bottom left of grid and terminal position is at the top right. Reward is function of state, In order to motivate the agent move towards the goal, the terminal position was setup with reward of 100 while each step is set up with negative reward of value -1. We're going to define our domain so that our four (north, south, east, west) actions are stochastic: with 0.8 probability they will go in the intended direction, and with 0.2 probability, it will randomly go in one of the other directions. To encode this

stochasticity, first dimension indexes by the selected action, and the next dimension indexes by the actual direction the agent will move, with the values specifying the movement in that direction, given the action selected. With this problem, we will let the value iteration, policy iteration and Q learning decide if it is worth penalty to achieve greater reward and how long does it take algorithm to make their decisions.

Value And Policy Iterations:

The following section presents MDP behavior using a discount factor of 0.99. Value Iteration and Policy Iterations shares similar behavior in the number of iterations to reach convergence. While VI and PI behave similarly for convergence, reward value and number of actions, the duration per iteration for PI is more than that of VI.

Tiny Grid World Analysis:

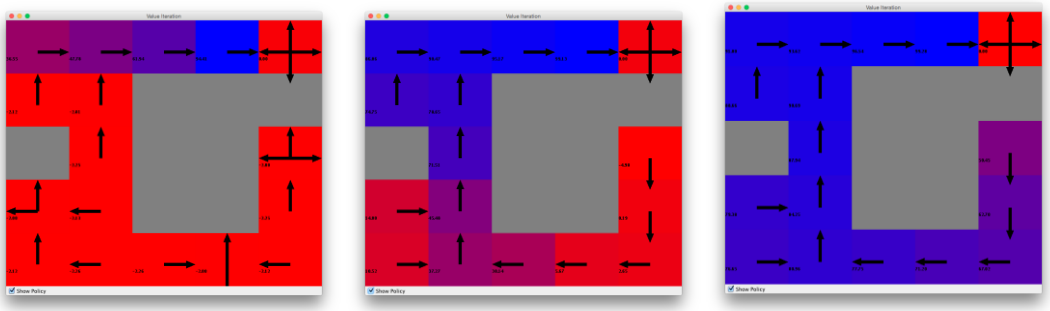


Value Iteration development of Tiny Grid World:

To get started with value iteration, an initial discount factor γ of 0.99 were used. Using the tiny grid world, some tests were ran on convergence and output the optimal policies and the number of iterations it took to get there. Below figures shows the policies at 3 such different time steps (1, 5, and 10 respectively). As the number of required steps to get to the goal drops considerably within the first few iterations. From the below, it is clear that with number of iterations reward has got positive value (color blue) compared to lower iteration with negative value shown in red.

The Reward graph above shows that Value Iteration had a large negative utility for the first iteration, which is expected, as it has not updated the policy based on expected values. By the 10th iteration, VI achieves a total utility of 89. Therefore, convergence is considered at I=10, when the total utility is within 5% for two consecutive iterations. This is as expected, since the VI must make calculations for each action in the policy.

I = 1 I = 5 I = 10

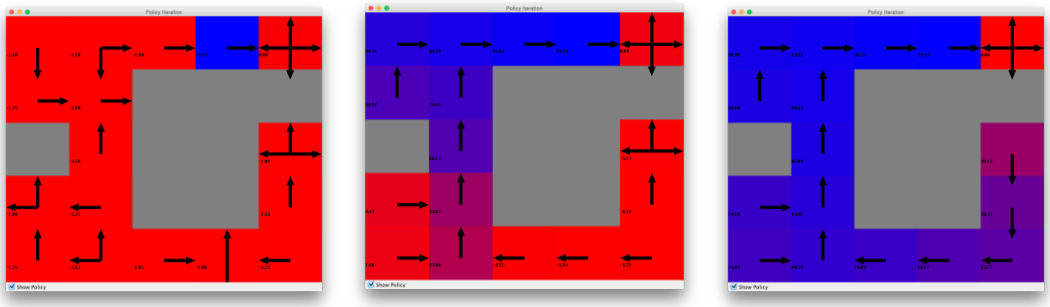


Policy Iteration development of Tiny Grid World:

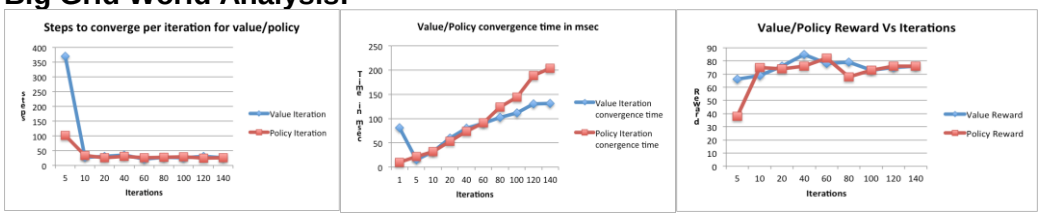
Policy iteration is similar to Value Iteration. It takes almost same number of iteration to converge as Value iteration. The Reward graph above shows that that Policy Iteration also converged to the same reward as VI. PI also takes longer to perform iterations –

almost exactly double of VI. While this agrees with the expectation that PI takes longer per iteration, it is also expected that PI takes fewer iterations because PI evaluates the utility of each state/action pair for a given policy, which takes more time to compute but provides more information to learn from.

I = 1 I = 5 I = 10



Big Grid World Analysis:

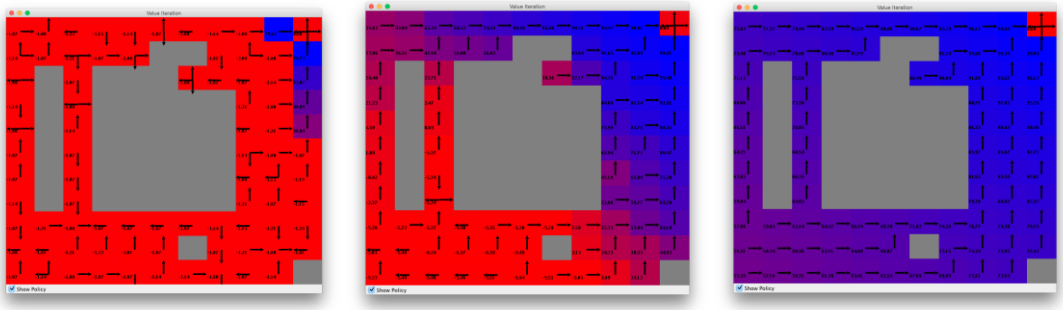


<u>Value Iteration development of Big Grid world:</u>

The Reward graph above shows that that Value Iteration had a large negative utility for the first 10 iterations. In the initial process before converging at I=40. As expected, big grid requires more number of iteration to find the optimal path.

The duration grows linearly with the number of iterations, which is expected since the time will increase linearly with the increased number of actions. The small (and sometimes large) spikes in time are again, due to the probabilistic nature of the agent's actions.

I = 1 I = 5 I = 40

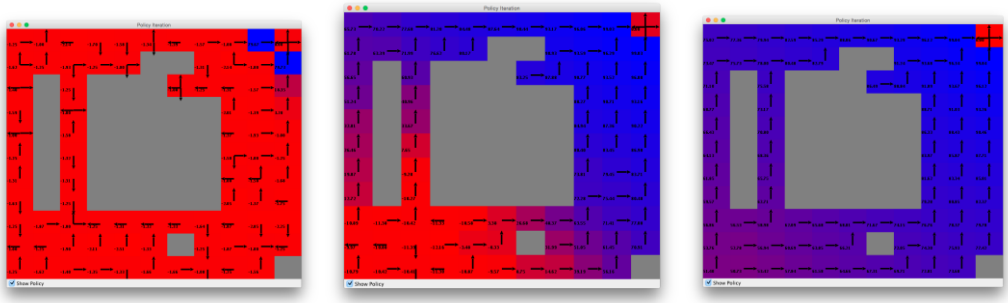


<u>Policy Iteration development of Big Grid World</u>:

Policy iteration takes way less iterations to find an optimal policy. In the case of the tiny grid world example, it only took 10 iterations, while the maze grid world took 19 iterations to converge.

From the policy output at each iteration we can also see the more granular step size of policy iteration in respect to updating the policy. In figure below we can see the policies at iterations 1, 9 and 19 (converged).

I = 1 I = 9 I = 19



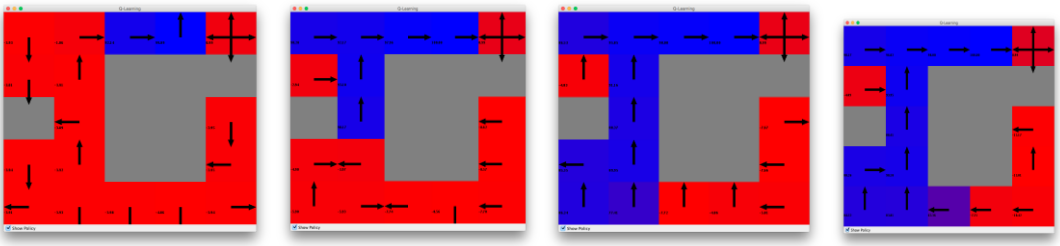
Part-2

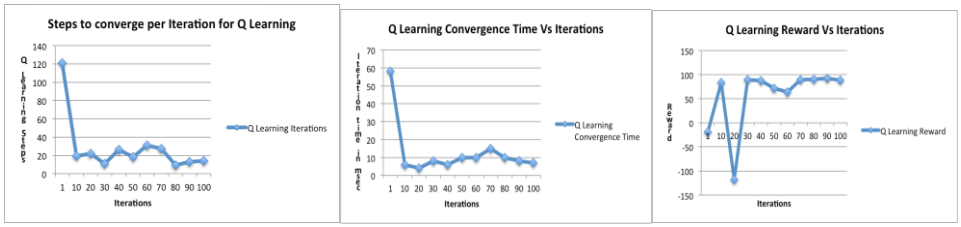
Q-Learning Analysis:

Q Learning takes much different approach than PI and VI. PI and VI must know the model, before begin the search. QL has no concept of model when it begins learning. This causes much more iterations, but can result in creative solution. For this simple Grid example, it also puts QL at bit of disadvantage.

Q-Learning development of Tiny Grid World:

I = 1 I = 5 I = 10 I = 20



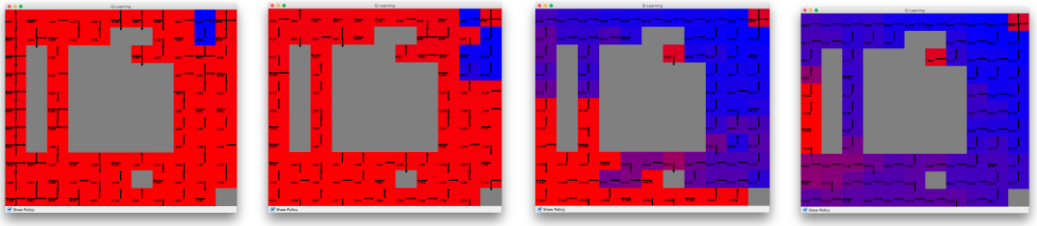


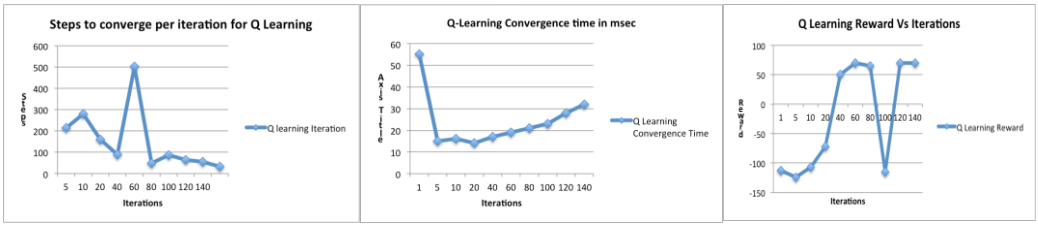
There are a number of differences in QL that are not observed in VI or PI. Observing the Reward graph in the upper middle, VI and PI converge at Reward of 10, while QL has an average around 30, with significant drops in Reward during exploration. It is evident from Reward picture, reward is low during first few iterations (0-30) which explains the exploratory nature of QL, and also that it converges more slowly. The criteria for QL convergence was to approximate the limit of the converged utility using the average of utility bet. This produces a converged utility of 20, and the iteration for convergence is I=20 (defined as when the utility for two consecutive iterations is within 5% of the converged utility). While this is double the number of iterations than for VI or PI, the time per iteration is considerably less. In fact, considering overall clock time to reach convergence, QL is faster than PI. This is explained by the model-free characteristic of

QL, which frees the method from calculating the action-utility pairs, like in VI and PI.

Q-Learning development of Big Grid World:

I= 1 I = 10 I = 40 I = 140





Duration for each iteration increases linearly, although the lower middle graph shows that QL computes each iteration significantly faster than VI or PI. This computation advantage of Q-Learning is much more pronounced than in the Small Grid Problem, which suggests that the number of total reachable states has a larger impact on the computing Analysis.

The Rewards in QL are clearly different from VI and PI. While QL is capable of generating policies that perform just as well as VI and PI, the exploration component of the algorithm lowers the average performance of QL. The converged total utility of VI and PI were nearly 70. QL meets the convergence criterion at I=120, at least 3x longer than VI or PI.

As opposed to VI and PI, QL does not simply change the policy incrementally to improve performance; it tends to be more “creative” and integrates very different approaches to explore the set of policies. This has an advantage of overcoming local optima, which can be particularly beneficial when the global optima is not intuitive.

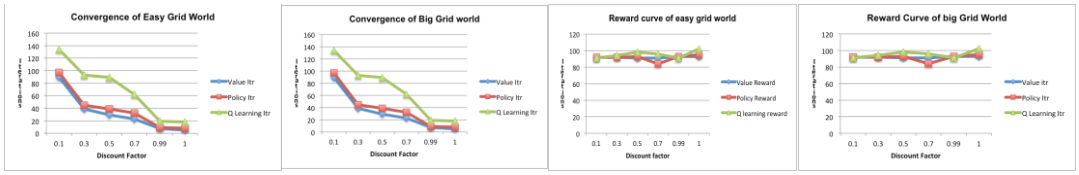
The policies for iterations between 1 and 140 are shown above. At I=1, it can be seen which states QL has begun to evaluate. QL explores radially from the initial state, and already in the first iteration, the Q-values suggest an optimal path along the top right corner of the grid. At I=10, QL has begun to exploit this local optima and by I=20 QL has a strong understanding of the shortcut. At I>100, it can be observed that QL continues to explore beyond the local optima in search for another local optima.

OBSERVATIONS & COMPARATIVE ANALYSIS

The graphs demonstrate the exploratory nature of QL, seen by the many & frequent spikes in the reward and the number of actions. While exploring is useful in discovering new policies, the simplicity of both MDPs favor exploitation, which VI and PI are very effective at.

In both MDPs, VI and PI converge in much fewer iterations than QL (by 3-5 times), and at a higher value of net reward (by 10-20%). Therefore, for these examples, the main advantage for QL is the computing time, but not the performance. Value Iteration appears to converge in slightly fewer iterations than Policy Iteration, with a duration of half as well. Overall, Value Iteration performs the best out of the 3 algorithms, for both MDPs.

Discount Factor



The results above were conducted using a discount factor of 0.99, which makes the agent value future rewards almost the same as present rewards. The following section presents the findings of how the discount factor affects net reward, convergence, the number of actions and the time. The graphs below evaluate the algorithms for various discount factors, and the convergence behavior.

Overall, this study shows that convergence occurs slightly faster for high discount factors, which aligns with the intuition that the algorithms can explore different policies a little more if the final reward is not worth as much. However, it is observed that QL performs better than VI and PI overall across all discount factors. This characteristic may compensate for the fact that at convergence, QL does not reach the same net reward as the other algorithms. Also, it is observed that PI varies the most for all tests. It is suspected that this variation is largely due to differences between runs in general. This may suggest that the initial policy assigned by PI is much more random than VI or QL. This may explain why VI converges slightly faster than PI.

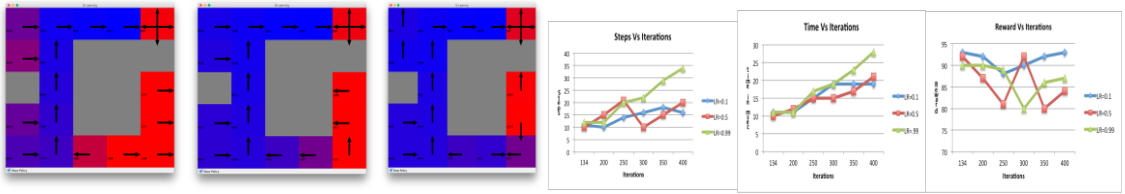
Q-Learning Parameters: Learning Rate & Exploration (Epsilon) and Q_Initial

For the majority of this project, the default values for QL were selected to observe the general behavior of the 3 algorithms. However, for further study, the default values were changed to assess how they affect the results.

The learning rate, L, controls the level at which new information overrides existing information. When L=0, the agent does not learn from new information whereas when L=1, the agent only considers new information. A constant learning rate was used for this MDP of the value of L=0.99 was initially used for experiments, however L=0.50 and L=0.10 were explored on the Tiny Grid World MDPs. The results demonstrate that the behavior is similar, shown by the policy and the per-iteration and cumulative graphs below. While the performance is similar, it appears L=0.5 performs best.

It was observed that QL continued to explore after it had converged on the optimal policy (as compared with VI and PI). This is controlled by the GreedyEpsilon strategy and default value of ε=0.1. For further study, it will be interesting to evaluate RandomPolicy or Boltzmann Policy, using values of epsilon between 0.1 and 0.99. It is expected that Boltzmann will perform best, as the high temperature dictates a high level of exploration, which cools down over time to prioritize exploitation.

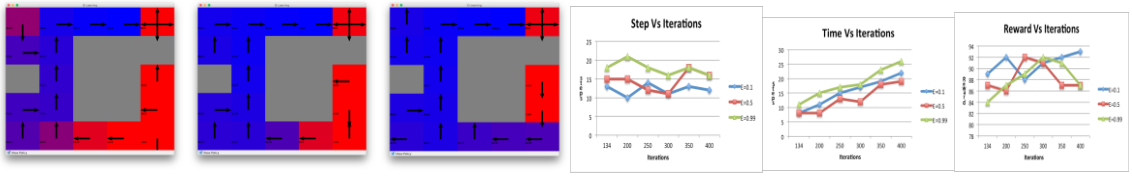
L = 0.1 L = 0.5 L = 0.99



To initialize the algorithm, QL applies an initial Q-value to all states prior to iteratively updating the values. There are generally two approaches to Q_Initial: use high initial

values (0.99) to create optimistic conditions and encourage exploration; or to use low initial values (0.10) to create conservative/realistic conditions to encourage exploitation.

Q_initial=0.1 Q_initial=0.5 Q_initial=0.99



CONCLUSION

To conclude, both MDPs demonstrate that Policy Iteration performs better than Value Iteration because they converge to the same policy but Value Iteration converges faster and computes faster per iteration. Q-Learning computes the fastest of the 3 algorithms, but with the epsilon greedy strategy the ability to explore was not advantageous for the QL converged reward. However, in the first several iterations, Q-Learning performs significantly better than Value Iteration and Policy Iteration by several magnitudes. The analysis on discount factor demonstrates a quantitative measure of exploration, however use of other MDPs may better highlight the impact of discount factor. As an extension to this paper, it is recommended to explore different grid designs and reward/cost structures. In addition, MDPs may be applied outside of the 2D GridWorld such as MountainCar (tutorial in BURLAP that models momentum and gravity to climb a mountain) or Real-Time Bidding in online advertising as used in Facebook ads or Google Adwords (an interesting study combining the Knapsack problem in Random Optimization with MDPs).

References:

CS7641 Assignment 4 BURLAP Extension https://github.com/stormont/cs7641-assignment-4
[2] Brown-UMBC Reinforcement Learning and Planning (BURLAP) java code library http://burlap.cs.brown.edu/
[3] Udacity Lecture on Machine Learning