Markov Decision Processes
(ML Assignment 4)

Silviu Pitis
GTID: spitis3
silviu.pitis@gmail.com

1 The Skip Ladder Environment

The first part of the assignment asks us to come up with two "interesting" MDPs. Because we are required to apply value and policy iteration to both, we are (practically speaking) restricted to choosing discrete MDPs. But the universe of discrete MDPs can be described with relatively few choices:

- a set of $n \geq 1$ states: s_i for $0 \leq i < n$
- for each state, s_i, a set of $m_i \geq 1$ actions: a_{ij} for $0 \leq j < m_i$
- for each (s_i, a_{ij}) pair, an n-by-m_i transition matrix (of probabilities)
- for each (s_i, a_{ij}) pair, a reward, r_{ij}
- a discount factor gamma[1]

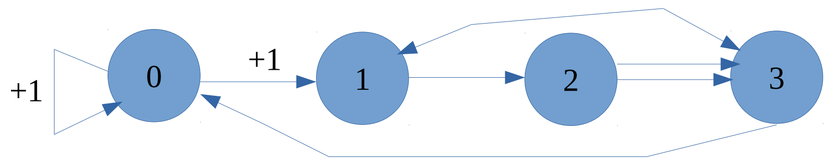
Terminal states can be modeled by restricting the action set for a terminal state to a single action that loops back into the terminal state and has a reward of 0.

Given this setup, we could scale an abstract MDP model along each dimension and obtain the desired comparisons (policy iteration vs. value iteration vs. online learning algorithms) for entire universe of discrete MDPs. This would be an interesting exercise, but would certainly take more than 10 pages, and so I place the following three simplifying restrictions on the universe:

1. The environment is deterministic
2. All states have an equal number of actions; there are no terminal states
3. Gamma = 0.9 and there is a fixed integer number of reward, each unit of which is assigned uniformly to a state (with replacement). Rewards for each action, given the state, are equal.

With these restrictions, we could create an arbitrary number (many more than "two") of random MDPs using just three variables: n_states, n_actions, and n_rewards. There is only one more thing I dislike about this, which is that randomly generating the actions can result in an MDP that is just a collection of several smaller disconnected state-action graphs. To get around this, I fix the first action in every state to return the next state (or, in the case of the final state, loop back to the start). This guarantees that every state of the random MDP will be connected in one dimension. I call this a "**Skip Ladder**" environment, because we can view it as an infinitely repeated ladder (or loop) that has skip connections (one skip connection leaving each additional action).

An example Skip Ladder environment, with n_states = 4, n_actions = 2, and n_rewards = 1:



1 Gamma is often considered a part of the learning algorithm, but in my view this is incorrect, at least to the extent that rewards are part of the environment (and not the agent): gamma is on equal footing as the rewards when determining utility, and should share whatever status rewards do. Since the formal treatment of rewards is as a part of the environment, so should the formal treatment of gamma (even though they are both agent-specific (cf. real humans experience different rewards and have different time values of rewards). Of course, we may find instances where learning is made easier by choosing a surrogate gamma that is different from the true gamma defined by the environment (e.g., in Open AI's Lunar Lander environment, the true gamma = 1, but the most effective performance with a DQN agent is gamma = ~0.995).

A Interestingness and real world Skip Ladders

The experiments in this paper are primarily of interest to me in the abstract, because they reveal the impact of the key MDP characteristics on MDP learning algorithms. By capturing a range of MDPs on multiple dimensions, and introducing random generation, I am able to obtain results that are for more representative than those that experiments on any two specific MDPs would provide.

As noted above, the Skip Ladder family of MDPs covers a large swath of all possible MDPs. It is no surprise then that it can be related to numerous real world scenarios. Basically, any scenario where there is a forward linear progression (e.g., driven by time, or 1-dimensional space), along with certain actions that can skip forward or backward, can be represented by a Skip Ladder MDP. For example:

- Mario games progress linearly, but have shortcuts that can be used to skip entire worlds
- One's career typically progresses linearly, but certain key decisions can cause one to advance faster or perhaps reset
- Our daily lives can be seen as a linear loop, where certain actions throughout the day can be completed out-of-order --- we may occasionally pull an all-nighter: instead of going from the final state back to the beginning, we would skip to somewhere in the middle of the MDP

B Note re: similar states and hierarchical abstraction

A very important part of MDPs, as they apply to real life, and as reflected in the RL algorithms, is the idea of groups of similar states. E.g.:

- hierarchical RL groups states/actions together into the higher order concept of "options"
- value iteration works by propagating the value of nearby (~similar) states
- optimal actions in the solutions to Grid World MDPs (or other MDPs with a spatial structure) are typically point in the same direction as nearby states

The Skip Ladder environment captures this idea of similarity in the unidirectional linear progression of states. In general, if state n has value V, then state (n-1) mod n is guaranteed to have value at least equal to gamma*V, because there is an action with reward at least 0 going from state (n-1) mod n to state n. This relation does not apply in the opposite direction.

C Skip Tree MDPs

Another interesting family of MDPs that I was unable to pursue for this assignment are Skip Tree MDPs, which are terminal MDPs with a tree-like progression, where the leaf nodes are terminal, and there are no backwards skips (and therefore no loops). Skip Trees would be a better representation of human life than Skip Ladders, since life is terminal, and has no do-overs. It would be interesting to see if the general results found below apply to Skip Trees as well.

2 Experimental Methodology

A Description of Experiments

I compared three reinforcement learning algorithms (value iteration, policy iteration, and Q-learning) on a variety of Skip Ladders of different sizes. For each "play", a Skip Ladder environment was randomly generated based on the three variables noted above: n_states, n_actions, and n_rewards. A play was defined as a single application of each of the three algorithms, each to convergence (as described for each algorithm below): thus, even though the environments were random, within each play, the algorithms were applied to the same environment for comparability.

The purpose of the experiments was to determine how each of the three variables impacts the three algorithms. To test this, I tested each variable independently, by holding the other two fixed. As there may be interactions between the variables, I tested each variable in two different settings, one “small” and one “large”. The settings are summarized below:

Variable tested	**Tested range**	**Small setting**	**Large setting**

n_states	[10, 100, 1000, 10000*, 100000*, 1000000*]	n_actions = 2
		n_rewards = 3
			n_actions = 5
			n_rewards = 50

n_actions	[2 .. 6, 8* .. 10*] by 1	n_states = 50
		n_rewards = 3
			n_states = 500
			n_rewards = 50

n_rewards	[1 .. 101] by 10	n_states = 50
		n_actions = 3
			n_states = 500
			n_actions = 5

*Q learning was not tested for settings marked with asterisk due to slow convergence speed

I believe the above ranges to cover a variety of “small” and “large” MDPs, as required by the assignment, along with everything in between.

Since MDPs were randomly generated, the results of a single play do not necessarily make for good comparisons, even if all algorithms are tested in the same MDP. Therefore, for each setting of the variables I ran 30 different plays (30 different randomly generated Skip Ladders) and averaged the results. This should be more than sufficient to establish statistically significant comparisons (from experience), although I did not actually perform the pairwise t-tests (I am lazy, and that would be beyond the scope of the assignment).

In addition, I also tested different exploration schedules for Q-learning, and different policy valuation schedules for policy iteration.

B Description of Algorithms

Each algorithm works by using a version of the **Bellman Q Operator**:[2]

$$BQ((s, a) \mid \alpha, \gamma, Q, R, PV) = (1-\alpha)*Q(s, a) + \alpha*(R(s, a) + \gamma * PV(s))$$

Where:

- s is a state, a is an action, α is the “learning rate”, γ is the discount factor, Q is a Value function of a state and action pair, R is a reward function of a state and action pair
- PV is a Policy-Value function of the state that implies a value based on some policy; the most commonly seen PV function is MaxQ, which takes the maximum Q value over all the actions for state s; MaxQ represents the greedy policy

Both value and policy iteration utilize **policy valuation**: given a known MDP, policy valuation, with respect to a policy **pi**, converges on the value function, V, that represents the expected discounted future rewards for policy pi. It does so by applying the following update to the value function, V (which can be initialized arbitrarily), until convergence:

```
for all states:
    for all actions:
        $Q(s, a) \leftarrow BQ((s, a) \mid \alpha=1, \gamma=0.9, Q=Q, R=ER[pi], PV=EV[pi])$
    $V(s) \leftarrow Q(s, pi(s))$
```

where ER is the expected reward (we know it because we know the MDP model) under policy pi, and EV[pi](s) is the expectation of V(s’) under policy pi

Convergence within some epsilon of the true value is guaranteed by theory (as a result of the Bellman Q Operator being a contraction mapping).[3] Based on a few preliminary experiments, I set

2 This notation is based on Professor Littman’s notation in the RL course (8803-003), but I *think* some elements (i.e., PV) are my own to make it generalize to all algorithms.

“epsilon” equal to 1e-4 to provide a margin of safety over 1e-3, for which policy iteration and value iteration agreed on the optimal policy for all tested plays.

Value iteration (VI) interchanges single iterations of policy valuation (i.e., a single value function update for each state) with **greedy policy selection** in a 1-to-1 ratio. Greedy policy selection accepts V and R functions, and returns the policy that maximizes expected value under the value functions. Converge for VI has the same criteria as for policy valuation, though the policy is changing at every iteration. I use epsilon = 1e-4.

Policy iteration (PI) interchanges policy valuation iterations with greedy policy selection in an z-to-1 ratio. A typical value of z might be 20 (this is what AIMA’s python repo uses), which I adopt. Since convergence within a small epsilon (1e-4) (of the inner policy valuation loop) is often found early (i.e., within w << z iterations), my implementation stops early if such convergence is occurs. Below, I run an experiment to show the effect of z. I consider that PI has converged when the policy has remained unchanged after z iterations of policy valuations.

Both VI and PI begin with an arbitrary initial value function / policy (usually the zero value function).

Both VI and PI use policy valuation, which requires a known model in order to compute ER and EV[pi]. In many cases, however, we will not have a known model.

Q-learning overcomes this limitation by approximating ER and EV using a Monte Carlo approximation based on observations from interacting with the environment:

```
Until convergence (or satisfactory performance):
    a ← action taken according to ExploreQ(Q, s) in current state, s
    r, s’ ← observed result
    Q(s, a) ← BQ((s, a) | α=α γ=0.9, Q=Q, R=I[r], PV=EQ[GreedyQ(Q, s’)])
    s ← s’
    α ← anneal(α)
```

Import differences to policy valuation based methods are as follows:

- We use the observed result, consisting of r and s’ to approximate ER and EV.
- Whereas policy valuation uses the same policy to obtain a probability distribution over s’ and to compute V(s’) when computing the expectation EV, Q-learning uses ExploreQ(Q, s) to obtain an observation of s’, and GreedyQ(Q, s’) to create an expectation over the value of s’. This is known as **off-policy** learning: observations/actions are taken with respect to one policy, but expectations/learning are taken with respect to another policy. Off-policy learning allows us to learn the optimal policy while not actually following it. This is important because exploration is necessary in order to prove that Q-learning converges in the limit.
- GreedyQ(Q, s) is simply greedy policy selection (as described above), but computed using Q instead of V. ExploreQ(Q, s) returns a policy that allows for exploration. The typical choice is an epsilon-greedy policy that returns a random action epsilon percent of the time (and otherwise chooses actions based on the greedy policy). We could also choose other exploration strategies such as softmax action selection.
- Due to the Monte Carlo approximation of ER and EV, we cannot (in general) simply use $\alpha = 1$ as in policy valuation. Instead α must be satisfy certain conditions to guarantee convergence; namely, the sum of the infinite series of alphas must diverge, and the sum of the infinite series of squared alphas must converge. However, because Skip Ladder environments are deterministic, and the Monte Carlo approximation is exact, I was able to (and did) use $\alpha = 1$. Note that this means the results for Q-learning may not generalize well to stochastic environments, as those will require alpha to be appropriately annealed.
- Testing for convergence in Q-learning is a bit harder than testing for convergence in policy valuation methods. Technically, we could apply the same result as described above for policy valuation methods, but (1) we would never be 100% certain of convergence,

3 Convergence is typically determined with respect to the largest state-wise difference in value: |V’(s) – V(s)| < epsilon for all s.

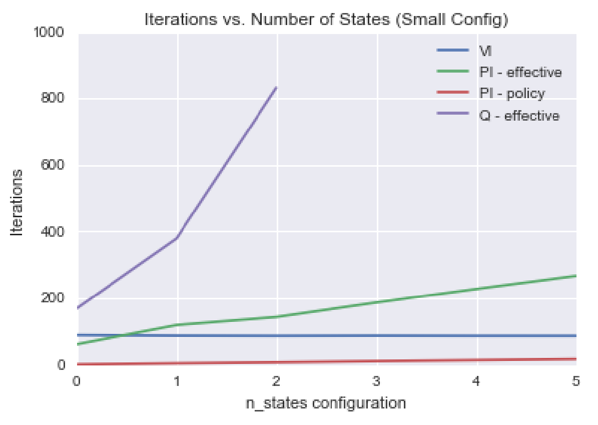
because of the use of Monte Carlo approximation; instead we would need to use something like the Hoeffding Inequality to create a confidence interval, (2) states in the MDP are not visited uniformly (as in policy valuation), and so we may require many cycles before the MDP finally convergences (e.g., if an important state is visited very rarely by the exploration schedule, and our estimate for it is poor, Q-learning iterations are basically wasted until that state is adequately explored; this does not happen with policy valuation methods). Skip Ladders, being deterministic, make convergence in Q-learning much simpler, since our Monte Carlo approximations are 100% accurate in a single observation. Thus, I was able to use an epsilon-based method effectively (stop after update has been < epsilon in the last n plays). It is important to note, however, that the results for Q-learning may not generalize well to stochastic environments, due to convergence issues with Monte Carlo approximations.

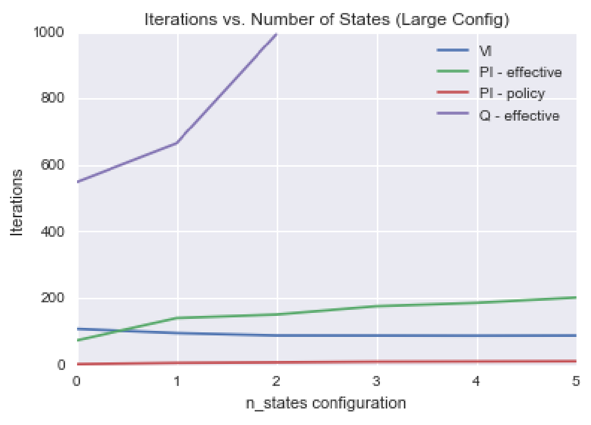
3 Experimental Results and Analysis

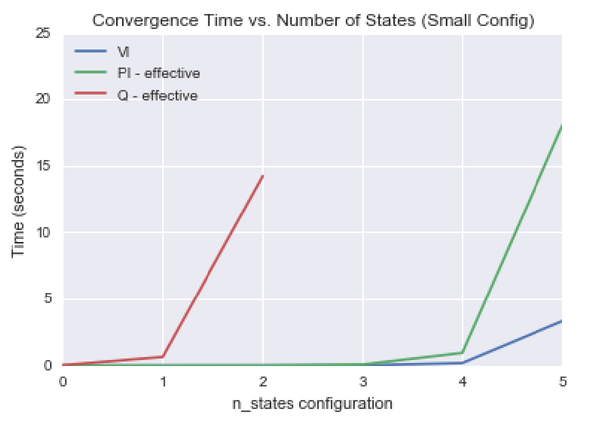
A Number of States

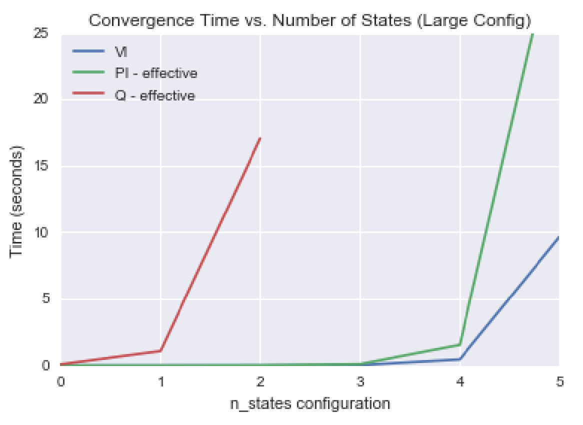
Variable tested	Tested range	Small setting	Large setting

n_states	[10, 100, 1000, 10000*, 100000*, 1000000*]	n_actions = 2
		n_rewards = 3
			n_actions = 5
			n_rewards = 50









Trends were consistent across both small and large configurations. The number of VI iterations decreased slightly when the number of states increased, but were more or less constant. However, the time started to become large as the state space entered the millions. As expected, the number of policy updates required for PI to converge was much smaller than the number of value iterations for VI to converge, and was also relatively stable as the state space increased. However, the number of effective iterations (i.e., value iterations in the inner loop) of PI grew approximately logarithmically with the state space. This growth is reflected in the convergence time graphs, where the convergence time for PI increases more rapidly than the convergence time for VI. Extrapolating, we can conclude that for large state spaces (100,000+ states), we should prefer VI to PI.

Q-learning was tested with epsilon = 1 (i.e., 100% pure exploration). Lower values of epsilon (discussed in more detail below) sharply increase convergence time and number of iterations, and caused the experiments to run too slow.

Note that the number of iterations for Q-learning is “effective”, which means that I divided the number of Bellman updates by the number of states, in order to make the number more comparable to VI and PI. It is still higher than VI and PI due to the stochastic exploration schedule.

Q-learning performs poorly for two reasons. First, my implementations of VI and PI are 100% vectorized in numpy and highly efficient. Q-learning is a sequential algorithm and cannot similarly be parallelized (unless we were to run multiple agents and allow them to share the Q-function). Second, Q-learning suffers from stochastic exploration: whereas VI and PI treat each state as equally, Q-learning does not (even when epsilon = 1, it will tend to stick to areas of the state space that have lots of state-action loops); since convergence was determined based on the maximum delta for all state values, this causes the converge of Q-learning to take longer. Note that, as discussed above, this latter point is beneficial in more practical scenarios, since Q-learning will allow us to focus our attention on more relevant parts of the state space when all we seek is an approximately correct policy.

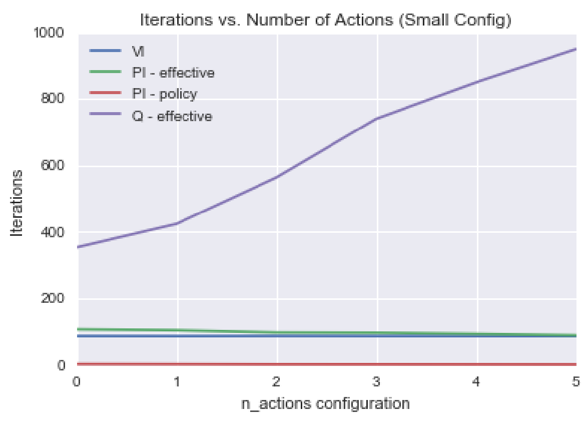
In all “large” configurations, VI, PI and Q-learning converged to the same value function (and therefore policy) (within an error bound of 1e-3). The same was true for all “small” configurations, except that the final value function found by q_learning diverged from the true value function by a slightly larger amount in about 20% runs where n_states was 10. This was more or less the case for all experiments (i.e., my implementations are correct and my convergence criteria tight enough), and so I will omit this comment going forward.

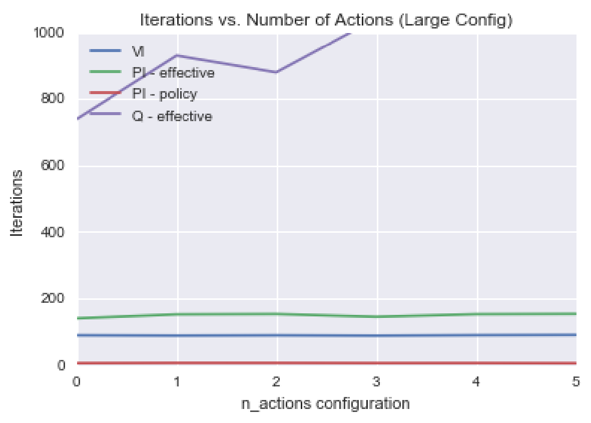
B Number of Actions (branching factor)

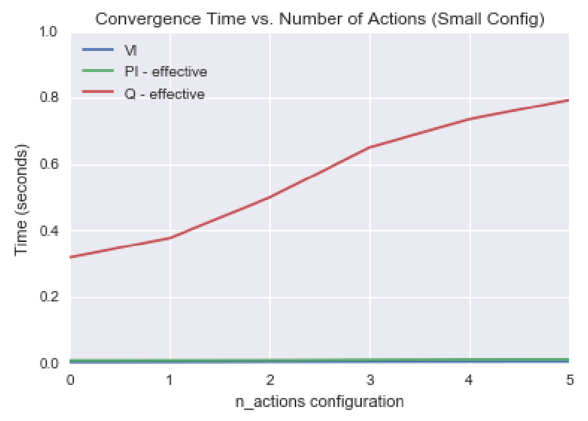
Increasing the number of actions per state had an approximately linear effect on the performance of all three algorithms for both small and large configurations. Once again, for the reasons discussed above, Q-learning performed the worst. VI performed slightly better than PI at all tested settings, with the gap between them growing larger as the number of actions increased. This suggests that the more actions an MDP has, the more we should prefer value iteration to policy iteration. Note that although the number of actions is very much the branching factor of the state tree, the repeated states that result from the Skip Ladder environment make this quite different from certain typical state trees, such as the game tree of a chess game --- thus it may not be appropriate to generalize this conclusion to such scenarios.

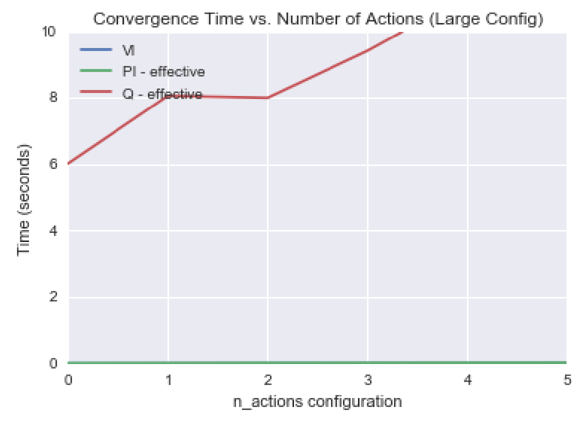
Variable tested	**Tested range**	**Small setting**	**Large setting**

n_actions	[2 .. 12] by 2	n_states = 50
		n_rewards = 3
			n_states = 500
			n_rewards = 50

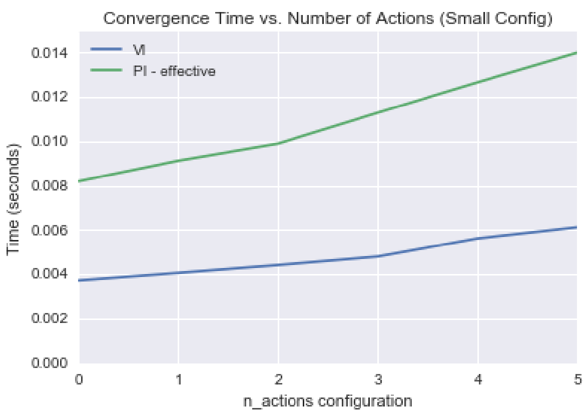


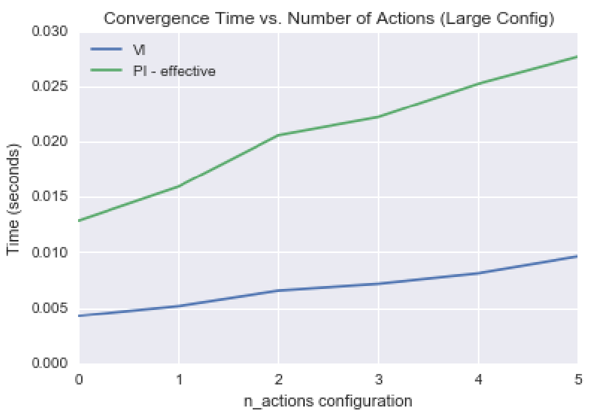






(note that y-axis is on different scale from small config)



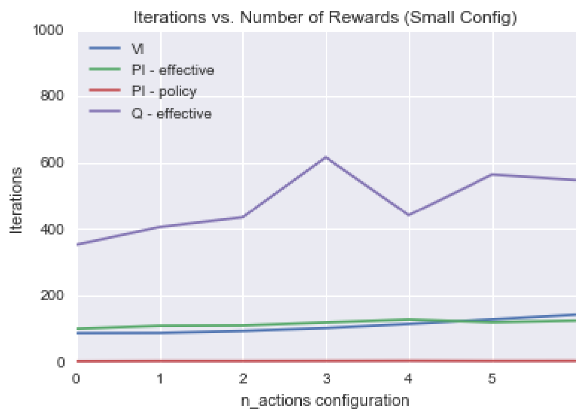


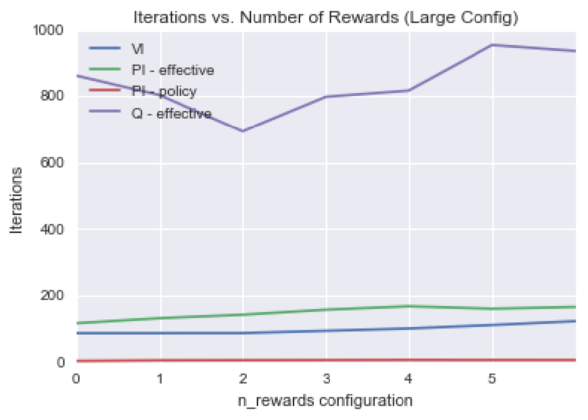
C Number of Rewards

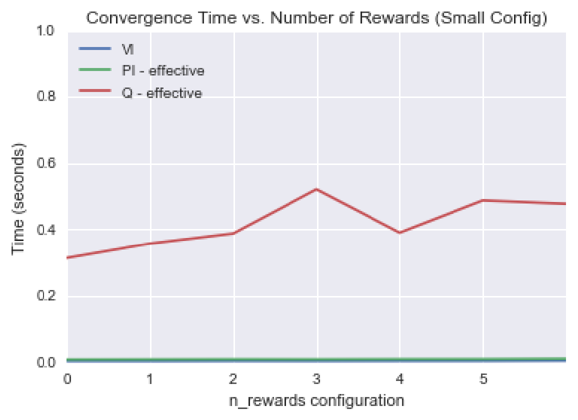
As described above, each unit of reward was assigned to a random state based on a uniform distribution. Thus, increasing the rewards increases the expected reward per state and changes the distribution of rewards. Smaller number of rewards result in sparser rewards and more spiky distributions of rewards across the states. As the number of rewards increases, the distribution of rewards at a given state approaches a normal distribution with a mean equal to n_rewards / n_states (by the law of large numbers).

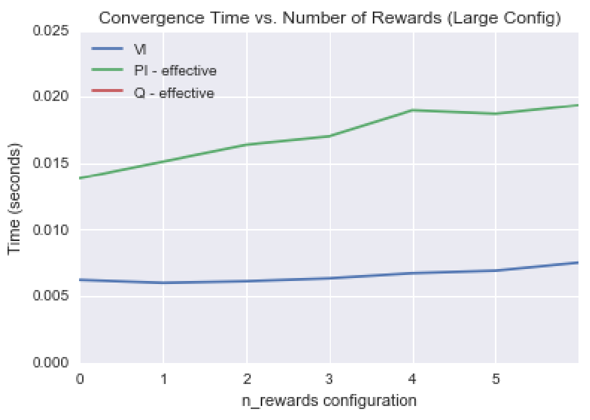
Variable tested	**Tested range**	**Small setting**	**Large setting**

n_rewards	[1, 5, 25, 125, 625, 3125, 15625]	n_states = 50
		n_actions = 3
			n_states = 500
			n_actions = 5









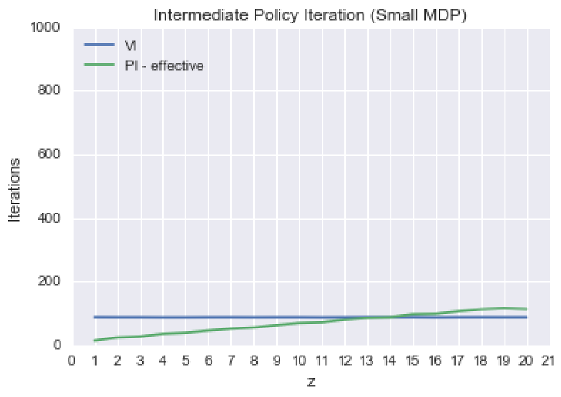
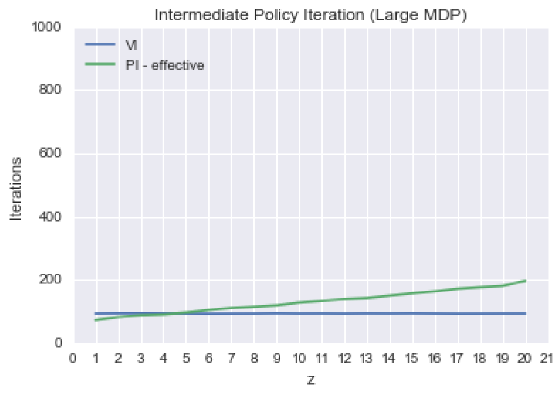
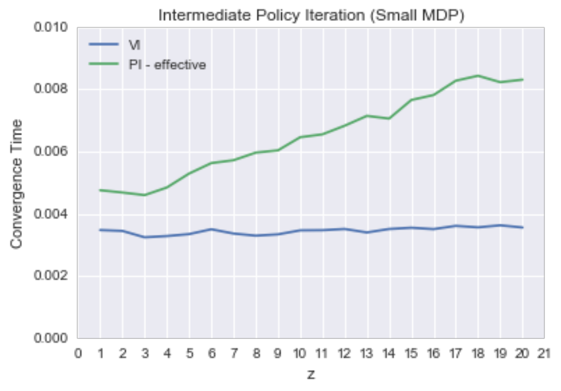
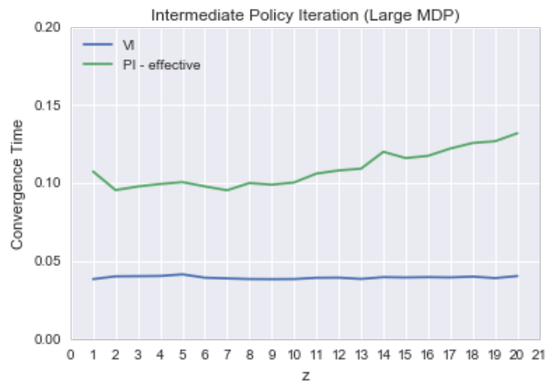
(Q converges in approximately ~5 seconds)

The effect of number of rewards is similar to the effect of number of actions (but note that reward configurations were scaled exponentially --- so the results are substantively different). This result surprised me, as I was expected iterations and run time to remain consistent or perhaps even drop as the distribution of rewards became more even. Having seen the result, my explanation is that this occurs because I am using a tight convergence criterion rather than seeking an approximately correct policy. When rewards are closer together (more normal), it should be easier to find an approximately correct policy. But, given the surprise here, I am not confident making that generalization with an experiment (not performed).

D Intermediate Policy Iteration

As described above, the policy valuation phase of PI ran until convergence. But this is not necessary: convergence is guaranteed for any incomplete policy valuation. Therefore, one might ask whether an intermediate version of policy iteration would perform better.

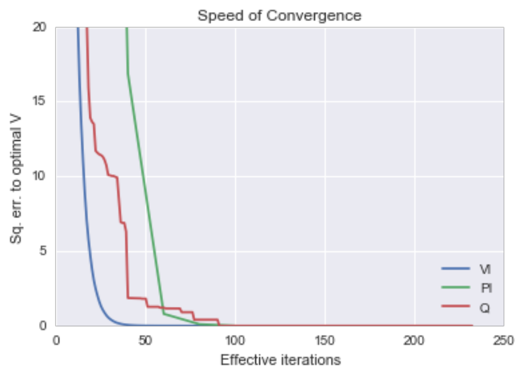
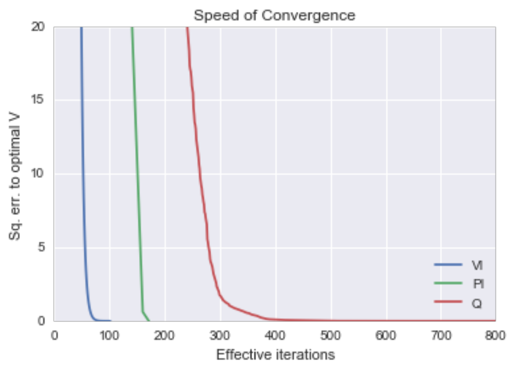
I compared VI and PI for values of z in between 1 and 20 (inclusive) (recall that above, z was set to 20), on two different MDPs (results shown below). The trend is clear for both large and small MDPs: as we move from VI to PI by increasing z, our algorithm takes progressively more (inner loop / effective) iterations to converge, and progressively more time. Note that the convergence criterion for PI is a stable policy, which explains why for small values of z, PI converges in fewer iterations than VI (which uses an epsilon-based convergence criterion). Nevertheless, because tested for policy convergence requires actually comparing policies, this introduces enough overhead so that VI is more efficient than PI for all values of z: even when PI converges in fewer effective iterations. Further, it appears that the “policy convergence” criterion of PI is imperfect. Whereas PI generally converged to the same policy as as VI (VI representing the *correct* policy), for very small values of z (i.e., 1-5) on the small MDP only, PI occasionally (~20% of the time) did not converge to the correct policy. This suggests that PI may benefit from a dual criterion: not only must the policy remain stable, but deltas during policy valuation must be epsilon-small.

Small (n_states=100, n_actions=2, n_rewards=3)	Large (n_states=10000, n_actions=5, n_rewards=1000)
	
	

E Speed of Convergence

Often, and particular when MDPs get very large, we are more interested in an approximately correct answer than a correct answer. How fast do VI, PI and Q-learning approximate the answer?

To test this, I once again use a small and large configuration, for which I compute the optimal value function using VI to convergence. Then, I run each of the three algorithms and periodically evaluate the mean squared error between that algorithms current value function at every state and the optimal value function. Unlike the other experiments, I was unable to average the results over 30 trials, due to the sporadic nature of PI's effective iterations. I thus ran the experiment several times to confirm that different random MDPs obtain the same result (they do):

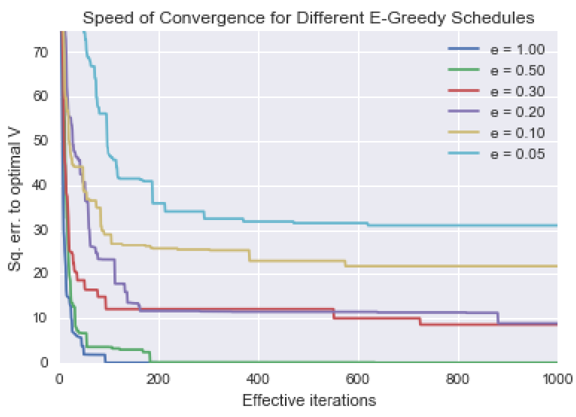
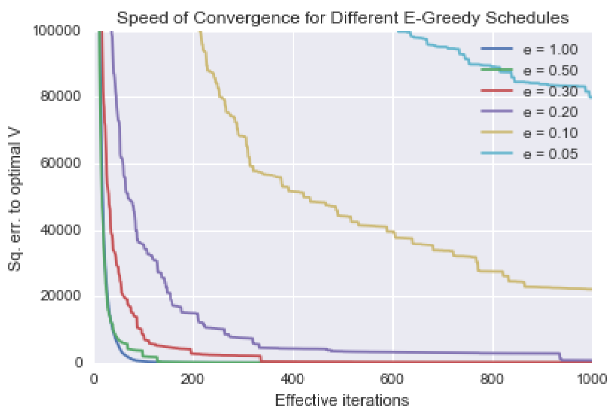
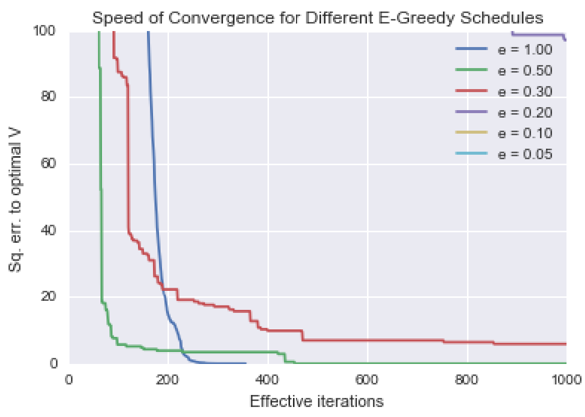
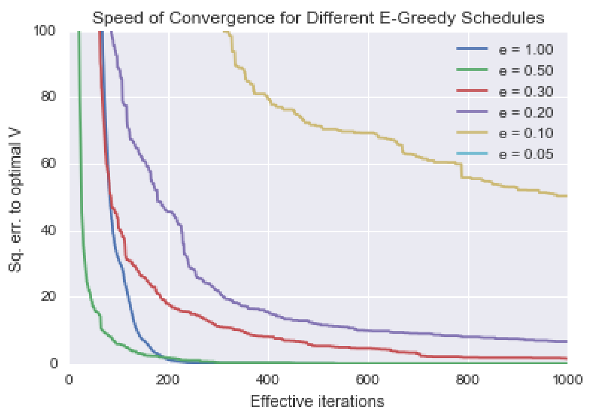
Small (n_states=50, n_actions=2, n_rewards=3)	Large (n_states=1000, n_actions=5, n_rewards=1000)
	

What is interesting about these results is that the difference between Q and PI on the large MDP is about the same as the difference between PI and VI --- so even though PI seems to perform far better than Q-learning when testing for convergence, it is does not perform that much better for obtaining approximate policies. In fact, on the small MDP, Q-learning tends to outperform PI in the short run.

F Exploration schedule

There was no requirement for real-time learning performance in the description (e.g., exploration vs exploitation trade-offs for purposes of obtaining rewards early on in learning), and so I took the liberty to focus this assignment on convergence. Exploration vs exploitation for purposes of early reaping of rewards is sufficiently explored in RL class, which I am taking concurrently, and it does not really make sense to compare to PI and VI, or in isolation for Q.

Exploration vs exploitation may still be relevant, however, insofar as it affects convergence. This can be the case if exploitation allows us to focus on more “interesting” parts of the state space, which is characteristic of an MDP that I observed in RL class.

Small (n_states=50, n_actions=2, n_rewards=3)	Large_1 (n_states=1000, n_actions=5, n_rewards=1000)
	
Medium_1 (n_states=100, n_actions=5, n_rewards=1)	Medium_2 (n_states=300, n_actions=10, n_rewards=1)
	

These results make it clear: for certain MDPs, exploitation can help convergence. Although most MDPs I tested were ordered by epsilon, in the bottom row you can find two MDPs were sub-1 epsilons greatly increase convergence speed. Based on my experiments, and contrary to my hypothesis (which was that it would occur when rewards are dense), this occurs when rewards are sparse. Incidentally, the MDP on which I experienced this phenomenon in RL class was Lunar Lander, which, in retrospect, had an extremely sparse reward profile (so don’t ask me why my hypothesis was that dense rewards would cause this….).

Quite surprisingly, these results suggest that for certain MDPs, if our objective is fast convergence to an optimal policy (rather than early exploitation for rewards---e.g., this might be the case when we are training the next Alpha-GO offline), we should experiment with an inversely decaying epsilon schedule!