The Study of Q-Learning Iterations in Comparison to Value and Policy Iterations in Solving Simple MDPs

Nicholas C. Park, November 16, 2020

Introduction

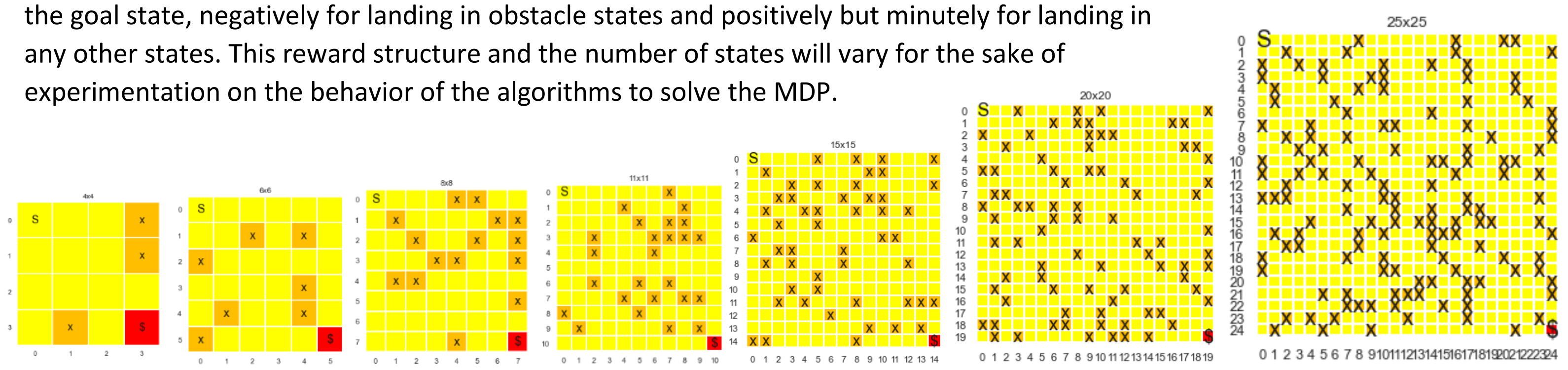
An expertise can be learned computationally by an agent in a decision-making process given that the environment is mathematically specified in the framework of a Markov Decision Process (MDP), where "the future is independent of the past given the present," for all state values in the process. The learning is complete when the optimal policy is determined such that maximizes the expected sum of rewards discounted over a certain time horizon. This optimal policy can be solved for by exact, model-based methods such as the value iteration and policy iteration or otherwise be planned by model-free learning algorithms such as those in the Q-learning family. These algorithms turn out to be sensitive both to how the MDP is set up (e.g. discount factor, reward structures) and to the used method's native parameters (e.g. learning rate – alpha, its decay rate). To analyze these behaviors, two MDP scenarios are implemented and experimented with using said algorithms – with various parametric settings -- with attention to changing behaviors in the utility value convergence, algorithmic runtimes, number of total iterations, and the output policy.

Markov Decision Processes and their Implementation

An MDP is defined via 4 or 5 parameters -- S, A, T, R (& Gamma). S (state value function): A set of all possible states, A (action value function): A set of all possible actions, T (transition probabilities): The probability function of transitioning from one state to another, R (reward function): The immediate reward for landing in a state and taking in an action and Gamma (discount factor): The degree of valuing more immediate rewards higher relative to later rewards.

2D Grid World

The toy grid-world model that is reformed from the code base of "Forest Lake" from OpenAI Gym is ran with a privately edited version of PyMDPToolBox-hiive, a fork of a library made to resolve discrete-time MDPs. This is a 2-dimensional (n x n) setting where the states are any location in this grid world, indexed column-wise first. The environment is used to discover what the optimal path is to take to go from the upper-left corner to the bottom-right corner in the presence of obstacles in the other areas of the map. The action to take defines how the agents move in this world: up (3), down (1), left (0) and right (2). This is generally known as a deterministic, finite MDP but this is altered; the probability of moving to another state is disturbed by a random chance of slipping that causes the agent to move in unintended ways. The rewards are defined positively for reaching the goal state, negatively for landing in obstacle states and positively but minutely for landing in any other states. This reward structure and the number of states will vary for the sake of experimentation on the behavior of the algorithms to solve the MDP.



Simple Forest Management with Two Actions

The Forest Management problem is a non-grid-world MDP to enact the goal of managing a forest either to preserve wildlife/scenery or to make profit by cutting wood down for sale. The state is defined as the age in years of the forest, rounded down, and ranges as an integer between 0 to n-1, i.e. state of (n-1) is the oldest. There is a probability of the age of the forest to become 0 with a probability p where the forest becomes burnt down and renewed by a fire. This probability p is subject to variation as part of experiment. The action is 1 or 0 depending on whether to cut down the forest for sale or not, respectively. There are certain rewards that are greater than 1 for choosing to cut the trees in the forest's oldest state and for choosing not to. The reward for choosing to cut the trees in any age other than 0 is 1. In all other cases, the reward is zero. This reward structure is subject to variation to explore the behavior of the MDP.

Methodologies

The so-called "exact methods" used to solve for the MDP are value iteration and policy iteration. These methods deterministically solve the MDP in a finite number of iterations by solving the given linear set of Bellman equations and assume the domain knowledge of the problem in the form of the state transition probabilities and rewards structures. The value iteration and policy iteration therefore obviate the need for an agent to learn the environment, because it simply solves the MDP. However, some stochasticity is introduced to the problems to examine in the scope of the analysis.

On the other hand, a model-free learning method hereon implemented as a type of Q-Learning learns without the use of such a domain knowledge. The agent must end up in the various states and find out for itself that the states provide certain rewards as well as that the states allow certain, subsequent state transitions. If the transitions are stochastic, the transition probability matrix is learned through trial-and-error based on how frequently the various transitions are observed. The Q-learning therefore has the advantage that it can solve any MDP even those with stochastic transitions without prior knowledge of transition probability matrix or rewards. This suggests that even when the agent's ability to learn on its own is not important e.g. due to determinism, it may still be necessary to use Q-learning to approximate the best policy if the state space is too large to perform linear programming to solve the MDP.

Algorithms & Implementations

PyMDPToolBox-hiive is a forked version of the library PyMDPToolbox, originally developed by Steven Cordwell. This forked version is edited to include additional iterating factors for the three main algorithms in order to observe the convergence of the utilities of all states. Some modifications include the following:

- The "error" metric corresponding to each of the algorithms – *PolicyIterationModified()*, *ValueIteration()* and *QLearning()* – are iteratively collected to produce an error vector that contains the error at each iteration. The "runtime" for each iteration is also collected and appended per the three algorithms above to see behaviors on the runtime over the iteration horizon beyond the cumulative CPU time.
- A function *pymdp_PR()* is written to transform the OpenAI gym's probability transition and reward matrices into versions compatible to the above PyMDPToolbox initializations.
- A function *return_term_goal_name()* is written to read randomly generated map and visualize the environment, policies and value functions of the gridworld MDP using privately modified functions from helper.py script from Denny Britz's RL resource library to fit *return_term_goal_name().*
- Hyperparameters "epsilon_decay_method", "epsilon_decay", "alpha_decay_method" and "alpha_decay" added to the QLearning init.

Value Iteration

In the value iteration algorithm, the value function as the expected sum of discounted rewards assuming optimal actions and starting from the state is calculated -- for every state in the state space -- per iteration, i.e. uses the optimality Bellman operator: $\mathrm{v(s)} \leftarrow \max_{\mathrm{a}} \sum_{\mathrm{s'}} \mathrm{p(s'|s,a)[r(s,a,s')} + \gamma \mathrm{v(s')]}$. Initially, a random value function is initialized as near-zero for all states and the value function is improved each iteration until the optimal value function is reached. Once this optimal value function is reached, the deterministic policy (with argmax) is extracted. For each state in the state space, the optimal value. Unless otherwise stated, the threshold value for errors, theta, is set as 1e-5 and gamma as 0.9.

Policy Iteration

In the policy iteration algorithm, each iteration involves a policy iteration step and a policy improvement step. Initially, a random policy is chosen. During the policy iteration step, based on the most recent output policy, the value function is calculated for each state in the state space. Afterwards, during the policy improvement step, a new policy is chosen based on the previously calculated value functions. The Bellman operator is used to obtain the value function: $\mathrm{v(s)} \leftarrow \sum_{\mathrm{s'}} \mathrm{p(s'|s,}\ \pi\mathrm{(s),s')[r(s,}\pi\mathrm{(s),s')} + \gamma \mathrm{v(s')]}$. Unless otherwise stated, similar to value iteration, theta is set as 1e-5 and gamma as 0.9.

Convergence Criteria using "Error"

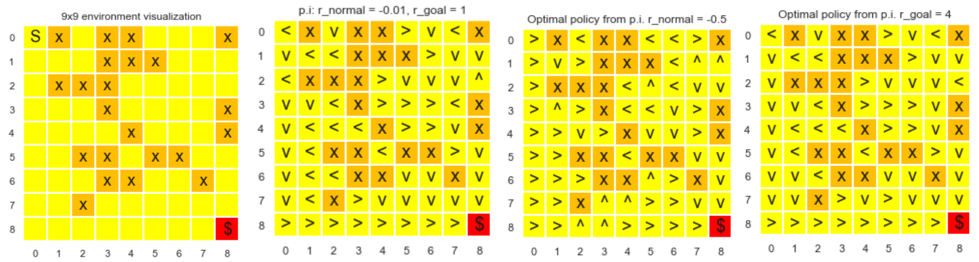
The total number of iterations to take to converge are dependent on both the settings of the MDP environment such as the gridworld size or the maximum age of the forest. However, the convergence criteria need to be primarily consistent across the method of the reinforcement algorithm. This will allow a fair comparison of the role that other settings of the reinforcement algorithm plays such as the epsilon decay rate, alpha decay rate, etc within the same method. To evaluate convergence, an "error" value or a function is defined per the methodology implemented. This definition of "error" will vary as follows:

- For policy Iteration, "error" = _np.absolute(next_v - policy_V).max() for policy iteration
  - The method converges when the error defined by the maximum value of the difference in the value functions given by two consecutive policy evaluations across the various states is less than a certain threshold theta.
- For Value Iteration, "error" = _util.getSpan(self.V - Vprev) for value iteration.
  - The span of the linear vector listing the difference of the two consecutive value function's values for each state is used as the error, and the method converges when this value becomes smaller than a certain threshold theta.
- For Q-Learning, "error" = np.absolute(self.alpha * (r + self.gamma * self.Q[s_new, :].**max()** - self.Q[s, a]))
  - The current iteration's learned utility, as opposed to the learned utility from the past iteration, or the amount to update to the Q-function values across the various state-action pairs in the Q-function is defined as the error. IF there is less than a certain threshold theta to learn, the learning algorithm is considered converged.
  - Q[s, a] ← Q[s, a] + *alpha*(R + gamma*Max[Q(s', A)] - Q[s, a])*

Examining the MDPs and their Unique Parameters

2-D GridWorld MDP Exploration

Let's observe how changing the reward structure affects the convergence of the policy iteration or the value iteration algorithm. Let the immediate default reward of landing in a non-hole and non-goal state be $r_{normal} = -0.04$, i.e. for all non-terminating states other than the start, the immediate reward of landing in a hole be $r_{hole} = -1$ and the immediate reward for landing in the goal state be $r_{goal} = 1$. Comparing $r_{normal} = -0.04$ to $r_{normal} = -0.4$ and $r_{goal} = 1$ to $r_{goal} = 4$, the policy outputs for policy iterations.

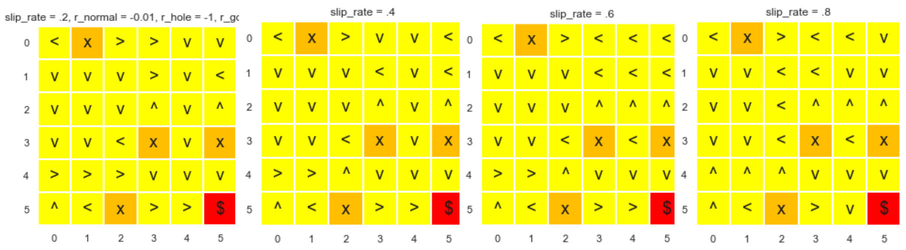

9x9 environment visualization
p.i: r_normal = -0.01, r_goal = 1
Optimal policy from p.i. r_normal = -0.5
Optimal policy from p.i. r_goal = 4

Here it can be shown that when the immediate reward r_{normal} is minutely negative, the agent prefers actions that avoids falling into a hole due to its negative reward of -1.

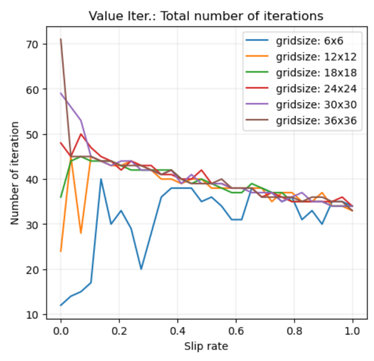
However, when r_{normal} is increased tenfolds to be greatly negative, the agent always prefers entering a hole any chance it gets despite the negative reward -1 of entering the hole; it exhibits a self-destructive behavior due to the disincentivizing reward structure to continue the iteration despite the penalty. This shows that the expected reward is higher when the agent is self-destroyed by entering a hole compared to when the agent continues to roam around -- unless it is very near the goal state. On the other hand, it turns out changing the r_{goal} from 1 to 4 does not result in a great change of the output policy because the positive reward of $r_{goal} = 1$ relative to other states' rewards is motivating enough for the agent to seek the goal state. The same phenomenon has been observed for decreasing the r_{hole} into a slightly more negative value, beyond which the above self-destructive behavior ceases to be observed – noting the output policy is highly sensitive to the relative reward structures of r_{normal} to r_{hole}, of r_{hole} to r_{goal} and of r_{goal} to r_{normal}.

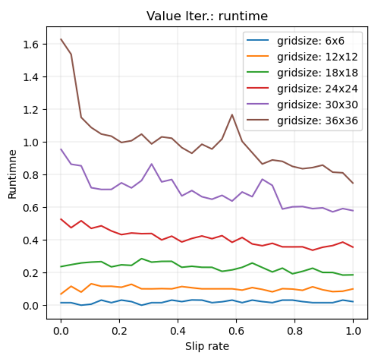
Varying the Slip Rate Values

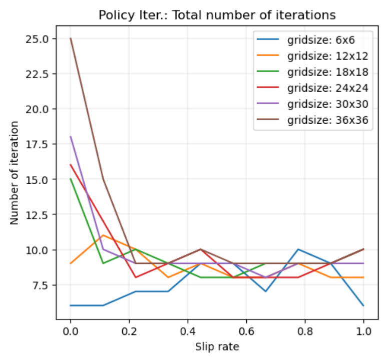
The slip rate introduces stochasticity to the deterministic MDP and is hypothesized to influence the convergence behavior of the policy iteration and the value iteration as well as the Q learning. Let's see how the behavior affects the unit in terms of the algorithmic runtime, number of iterations and the output policy using a new 10x10 gridworld but with the reward structure as the default of the previous gridworld. The output policy is as follows using value iteration.

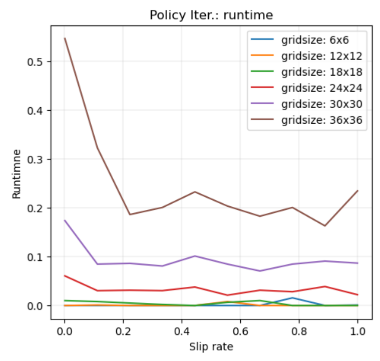


In terms of the output policy it can be shown that, as the slip rate becomes higher, more randomness is introduced thus converges incorrectly. There is a limitation to solving the MDP because it is no longer a linear set of equations to solve. On slip rate = 0.6 and 0.8 it is seen that the agent is not able to enter the 2x3 region on the bottom right corner. Now let's view this in Q-learning iterations.



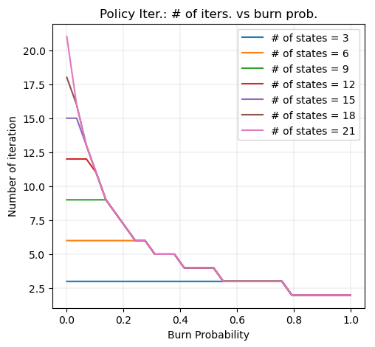


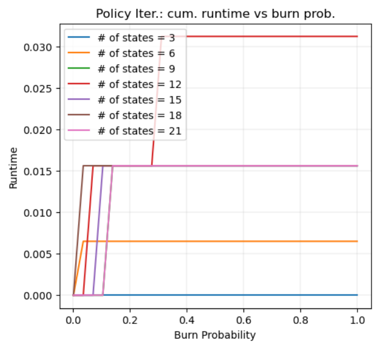




For the value iteration and policy iteration it can be shown that the slip rate does not have a clear influence on the total number of iterations it takes for the algorithms to converge. However, it is shown from the figure for the value iteration that when the slip rate is equal to zero the number of the iteration to converge is almost directly proportional to the width of the gridworld. It reveals that for a gridworld of size n, the number of iterations to converge is approximately 2n when the slip rate = 0. However, regardless of the size, the numbers of iterations to converge for every grid size interestingly tend to all converge near 33~34. On the other hand, the runtime also is unequivocally dependent on the size of the gridworld when the slip rate is zero. However, for larger gridworld sizes, the runtime generally decreases as the slip rate increases -- likely due to having less obligation to heavy computation instead of outputting random actions.

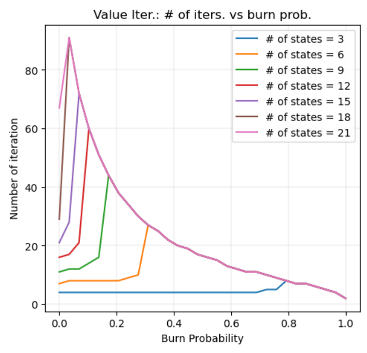
Forest Management MDP Exploration

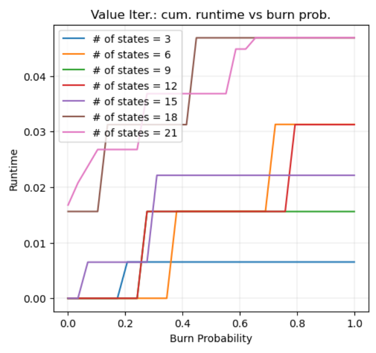




This MDP is more stochastic than the previous MDP whose randomness is determined by the sliprate at most, since the probability of burning down can be flexibly determined based on the age of the forest. For the sake of having control over the probability of the trees being burnt down and see its effects on the behaviors of various-sized MDP's, only one value of p is used hereon. Additionally, the reward structure between the action of waiting vs cutting down the trees are varied to see the effects on the iteration and runtime behaviors on this problem.

The value iteration and the policy iteration are different in iteration and runtime behavior depending on the burn probability. Yet in both cases, it turns out that the number of iterations it takes to either converge or terminate is decreased by the burn probability. This is likely because the optimal policy regarding the forest management becomes more obvious when the risk of forest fire is given as high – to cut them down when they are not burnt. This hold most clearly for larger state spaces in both value iteration and policy iteration. However, it is also shown for both policy and value iterations that increasing the chance of fire always never results in the decrease of the cumulative runtime for any state-space forest MDP despite the reduction in iteration.





For policy iteration, it appears that for smaller state space sizes, the burn probability needs to be higher than a certain level to reduce the number of iterations. Below this probability level, the number of iterations the algorithm takes is the same. This can be interpreted as that if the MDP state space size is already small, it terminates or converges to find the optimal policy early due to the simplicity in solving the linear equation, without introducing the ease in solving the MDP by the increasing the risk of forest fire. For value iteration, however, there appears to be a peaking of the number of iterations it takes to converge for a certain probability p, for each level of state space size observed, after which the higher burn probability results in a less amount of iteration to do. Also, value iteration is in general shown to be faster in runtime on this MDP as well.

	R0 = 4 and R1 = 2	R0 = 2 and R1 = 4	R0 = 3 and R1 = 3	R0 = 5 and R1 = 1
Policy, when burnprob = 0.2 with state size = 50.	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
Policy, when burnprob = 0.8 with state size = 50.	0, 1, 1, 1, 1, 1, 1, 1, 1,1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0

Let's observe if this pattern holds clearly for very high state size with attention to the individual runtime. For the consistency in comparison, a state size S = 10,000 years with rewards r0 = 4 and r1 =2 is used for the comparison. The results on the table below confirm, the lower the burn probability p, the longer it take s the policy iteration to converge. On the other hand, for the value iteration, it can be shown that having a p < 0.5 while p > 0.1 requires the most computation and wall clock time for the method to converge.

	Burnprob = 0.1	Burnprob = 0.3	Burnprob = 0.5	Burnprob = 0.7	Burnprob = 0.9
# of policy Iteration	11	6	4	3	2
Policy Iter. runtime	13.030817985534668	6.5503094	4.04089236	2.712913	1.34257245
# of value Iteration	8	28	17	10	6
Value Iter. runtime	0.5453324	1.8167505	1.0890676975250244	0.71879339	0.38639879

Altering the Reward Structure of Forest MDP

Above experiment assumed the default PyMDPToolbox value where waiting (action = 0) receives 4 points and cutting (action = 1) receives 2 points. This structure prefers waiting to preserve the forest unless the model-based approaches learn the risk of fire to reap the profit by cutting down the trees. Let "wait" reward be r0 and "cut" reward be r1. Let's observe how changing this structure affects the behavior of the policy while keeping state size as 28 and burn probability as 0.1. It turns out that, with the exception of the very first year, for the initial few years, the optimal plan by the value iteration is to cut the trees down. However, there is some level of clear structure that is agrees with intuitive explanations. For instance, if the reward to wait instead of cut is lower, it shows that more values of zeroes are populated towards the last states. Also, when the reward to wait is 100, most of the states are populated as the 0 for the optimal policy. Otherwise, there are some uninterpretable patterns present in the output policies.

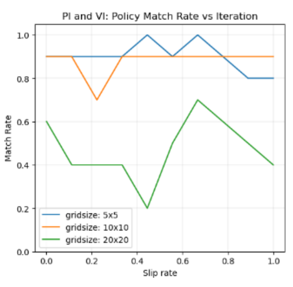
$[r_0, r_1]$	[4, 2]	[2, 4]	[100, 1]	[1, 100]	[3, 3]	[50, 50]
Output Policy	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0	0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0	0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

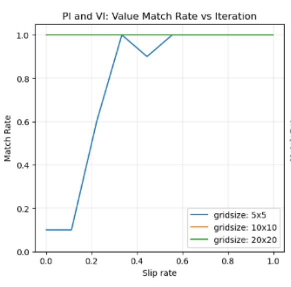
<u>Does Value Iteration Match Policy Iteration?</u>

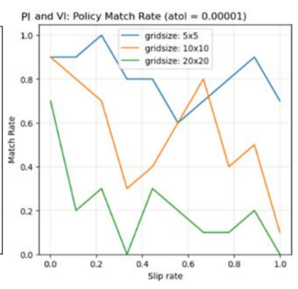
The two algorithms are similar. The key differences are that the value extracts the policy only once at the end and that it involves a max operation in the Bellman equation to find the optimal value function -- i.e., it uses the "optimality" Bellman operator. Another difference is that the policy stability is checked before it is extracted in the policy iteration. For both MDPs, the match rate is calculated by calculating the frequency of np.isclose() == True given multiple seeds over various values of burn probability. This curve is considered for various state space size to observe any significance of state space as well. The match rates have been tested over various state space sizes for both problems, using an absolute tolerance values of 0.01 and 0.0001 for the gridworld MDP (& values of 0.1 and 0.0001 for forest management MDP), while the relative tolerance remains the default as 1e-5 -- where these tolerances are defined in the comparison of two floats, a and b, as follows: absolute(a - b) <= (atol + rtol * absolute(b)). Then the process repeated with the atol reduced significantly for a stricter match criterion.

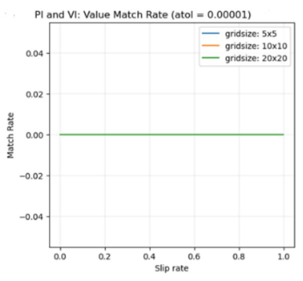
Gridworld: PI vs VI output policy

As the slip rate increases, due to more random actions the two algorithms are hypothesized to diverge. Using an absolute tolerance value of 0.01 in determining closeness using the numpy.isclose(), the two algorithms VI and PI are compared whether they are close in the output policy as well as the value function. This comparison is done across varying 10 linearly spaced slip rates between 0 and 1, and this span of comparison is repeated over



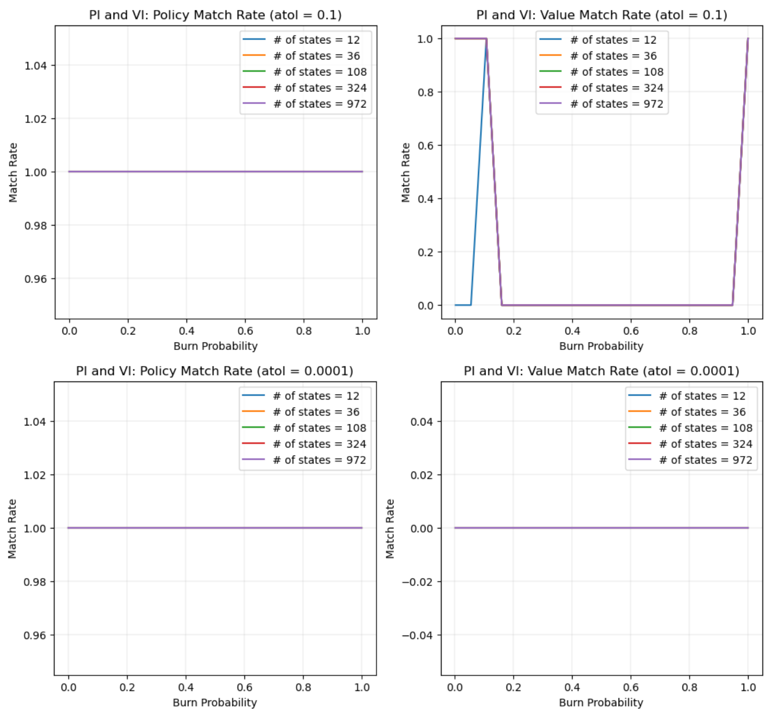






varying 10 seeds to see the fraction of matches per seed for the corresponding slip rate. This is compared for varying seed sizes to observe the effect of the size of the problem also. When the absolute tolerance, or atol, is reduced to 0.00001 there is zero match rate across all slip rates between the policy iteration and the value iteration. The match rate generally decreases as the comparison tolerance value is lower showing high variation and no clear pattern. It is clearly shown however that within this atol setting, the policy iteration and value iteration do not match at all for any slip rates, no matter the grid size.

Forest Management: PI vs VI output policy



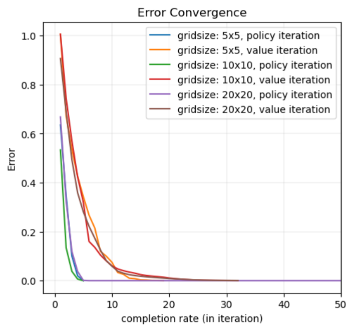
The curves are all plotted correctly for all state sizes. The last plot of # of states = 972 tends to overshadow the other states curves, meaning the various state sizes had identical policy match rates. While keeping in mind the output policies are integers, the atol parameter is used in the np.isclose() to judge how closely the value functions match rather than the output policies. For all state sizes, it shows that the exact value function values do not match given the atol of 0.1 but the policy matches exactly given this threshold. When the burn probability however reaches near 1.0 the two methods converge in the value functions as well, for all state sizes – likely due to that a guaranteed fire outputs an obvious policy to cut the tree as early as possible.

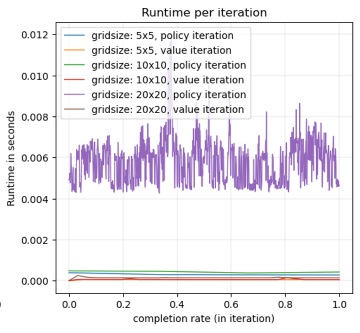
The match rate continues to hold 100% for atol = 0.0001 for the output policy while outputting 0% for all the function values for all state sizes even near burn probability = 1. Even when the burn probability is 1, the match rate of the value functions do not budge because the threshold is too strict. It appears, within the forest MDP setting, the value iteration and the policy iteration rarely show discrepancies.

VI & PI Convergence and Runtime Behaviors

Gridworld: VI & PI Convergence and Runtime Behaviors

Using the reward structure of r_normal = -0.01, r_hole = -1 and r_goal = 1, sliprate of 0.2, gamma = 0.9 and epsilon decay from 0.9 to minimum 0.1 using 0.99 geometric decay rate, the two model-based methods are compared in error convergence over the continued iterations. Additionally, the two algorithms in various state sizes are examined of their runtime per iteration for any behaviors.



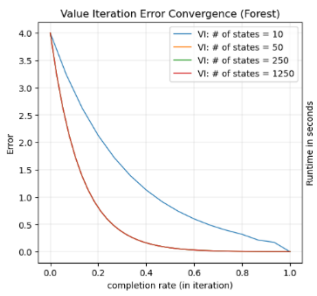


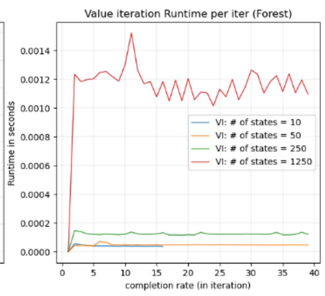
The # of iteration: gridsize: 5x5, policy iteration: 3
Mean runtime per iteration: gridsize: 5x5, policy iteration: 0.000301133
The # of iteration: gridsize: 5x5, value iteration: 21
Mean runtime per iteration: gridsize: 5x5, value iteration: 4.5171427e-05
The # of iteration: gridsize: 10x10, policy iteration: 999
Mean runtime per iteration: gridsize: 10x10, policy iteration: 0.0003873072
The # of iteration: gridsize: 10x10, value iteration: 35
Mean runtime per iteration: gridsize: 10x10, value iteration: 5.594286e-05
The # of iteration: gridsize: 20x20, policy iteration: 9
Mean runtime per iteration: gridsize: 20x20, policy iteration: 0.00543558889
The number of iteration: gridsize: 20x20, value iteration: 35
Mean runtime per iteration: gridsize: 20x20, value iteration: 0.00018189143

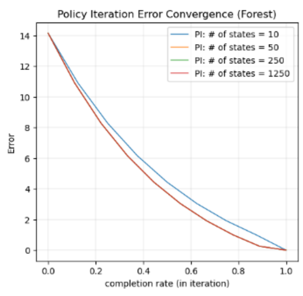
Remarking that the "error" definitions for the value iteration and the policy iteration are unequal, the figure shows that the downward curves are clustered based on the algorithm used. Viewing the value iteration curves separately, it shows that the value iteration starts off higher for the MDP with a lower state space size. The policy iteration starts off higher for the MDP with a higher state space size. The policy iteration tends to converge more rapidly in terms of iteration, though not in terms of runtime. The policy iteration is shown to show higher runtime per iteration. Policy iterations on the other hand takes more numbers of iteration for the error to reach near-zero and consistently requires less time per iteration. The policy iteration shows a higher variation in the runtime per iteration and this variance is more pronounced as the size of the problem increases. Let's observe with the forest management problem if the patterns hold for the two algorithms.

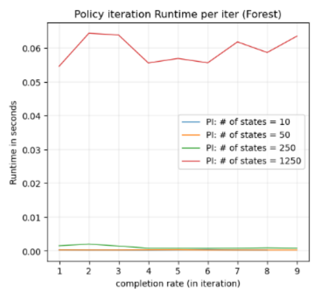
Forest Management: VI & PI Convergence and Runtime Behaviors

The burn probability of 0.1, reward structure of [r_waiting, r_cutting] = [4, 2] and gamma = 0.9 is used to compare within the forest MDP setting.









The number of iterations: state size: 10, vi: 16
Average runtime per iteration: state size: 10, vi: 3.88875e-05
The number of iterations: state size: 50, vi: 39
Average runtime per iteration: state size: 50, vi: 4.68128e-05
The number of iterations: state size: 250, vi: 39
Average runtime per iteration: state size: 250, vi: 0.00012
The number of iterations: state size: 1250, vi: 39
Average runtime per iteration: state size: 1250, vi: 0.00114
The number of iterations: state size: 10, pi: 8
Average runtime per iteration: state size: 10, pi: 0.000276
The number of iterations: state size: 50, pi: 9
Average runtime per iteration: state size: 50, pi: 0.0002879
The number of iterations: state size: 250, pi: 9
Average runtime per iteration: state size: 250, pi: 0.0011066
The number of iterations: state size: 1250, pi: 9
Average runtime per iteration: state size: 1250, pi: 0.0594454

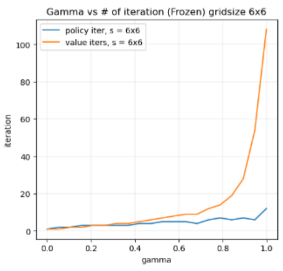
It is shown unexpectedly that changing the state space size does not result in a very different variation of the error in the policy iteration as well as the value iteration. In fact, for state size other than 10, the error of the algorithm did not show any difference over the entire iteration progression. Even the number of iterations remained the same. This is a behavior unique to this MDP implying that there is an occurrence where the maximum of the difference of the two consecutive value function vectors is constant for each iteration regardless of the state size. The runtime is the highest in average and standard deviation for the largest state space size however, despite the exact same error output and number of iterations. Similarly, the value iteration had the same output for all state size greater than one. This suggests than the span of the linear vector as the difference of two consecutive value functions is equal for all state sizes other than 10. And strangely enough, like the case of the policy iteration, the runtimes differed, with the largest state size showing the highest mean and variance.

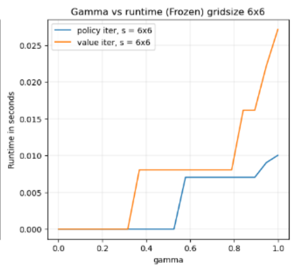
Gamma, the discount factor – "Far-Sightedness"

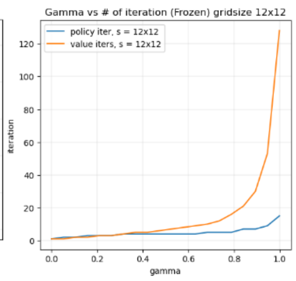
The discount factor accounts for how far-sighted the algorithm is in terms of not valuing immediate rewards over future utilities. Gamma = 0 causes the algorithm to focus on the immediate rewards only, whereas gamma equal to or greater than 1 can cause the algorithm to diverge and run infinitely due to that long-term rewards are valued more than or equal to immediate ones. Therefore, it is hypothesized for both problems to show increasing number of iteration and runtime as the gamma increases. Intuitively, if one can see more into the future rewards, the candidate space for the optimal plan is effectively greater, and the planning is constantly adjusted to fit beyond the current scope. The runtimes and the numbers of iterations that has been taken for the policy iteration and value iteration are plotted against various 25 values of gamma linearly spaced between 0.001 and 0.999.

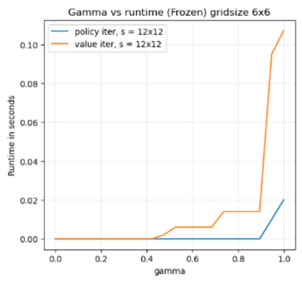
Gridworld MDP: Varying Gamma Values

The different discount factors are used in the value iteration algorithm in a 6x6 randomly generated grid and the 12x12 grid. The slip rate is given by 0.2. The reward structure used is r_normal = -0.01, r_hole = -1 and r_goal = 1. There is a clear increase of runtime and the number of iterations for convergence when the gamma is increased.





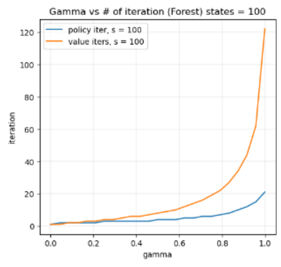


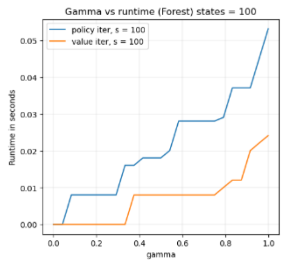


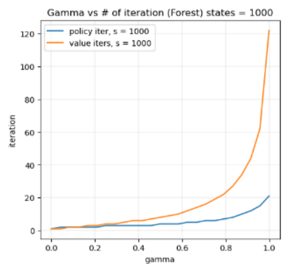
It appears that in the case of the gridworld MDP, the value iteration take more iteration as well as the runtime, although it usually has been the case from previous experiments that the policy iteration takes less steps while requiring more runtime per steps compared to the value iteration. Despite this unexpected behavior, it holds true that both the runtime and iterations increase for both cases. The biggest runtime and iteration slope/jump occurred when the gamma reached 0.9, which suggests a certain point of being far-sighted enough to affect the runtime and the number of iterations significantly.

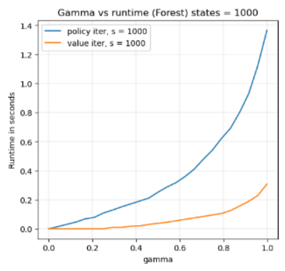
Forest Management MDP: Varying Gamma Values

Given the burn probability = 0.1 and reward structure of [r_waiting, r_cutting] = [4, 2], the runtime and the number of iterations to converge are evaluated. The state sizes of s = 100 and s = 1000 are considered to see increasing gamma's effects.









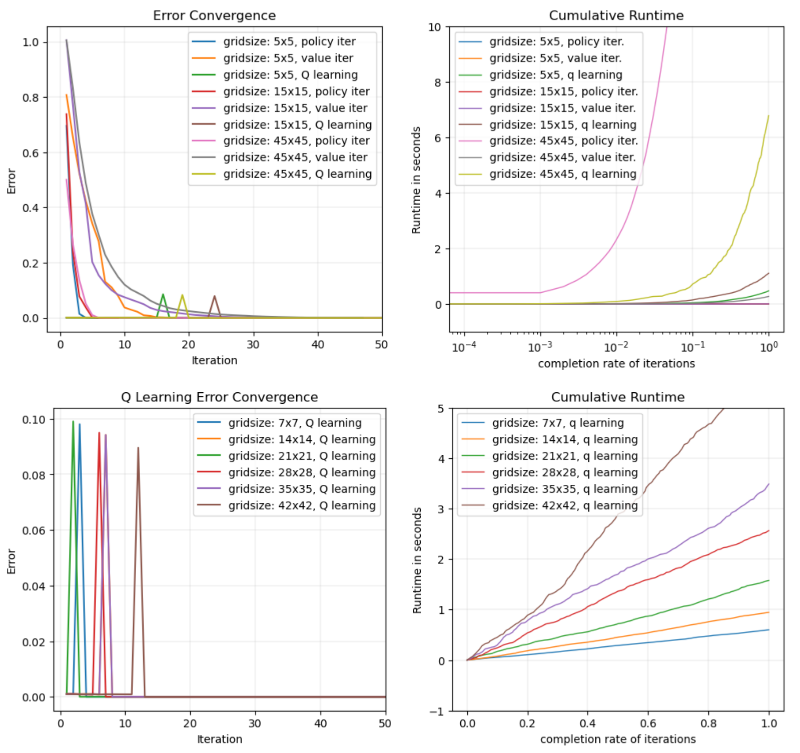
The increase of gamma clearly results in the increase in the amount of iterations for both the value iteration and the policy iterations. It also clearly results in the increase of the runtime also. These occurences are consistent across all state space sizes. Intuitively, the future rewards are valued greater when gamma increases, so the planning procedure opts for a far-sighted strategy that requires more iteration and runtime. For the state size of 1000, the increase in runtime per the increase of gamma is smoother compared to the case where the state size is 100. Additionally, as opposed to the grid world case, the runtime increase has been more gradual for this MDP and does not seem to have a certain point of a large jump. However the number of iteration steeply increases as the gamma nears 1.

Comparison of Q-Learning against the model-based algorithms

The method's convergence is when the error of the method, which is predefined uniquely per algorithm, decreases below a certain threshold theta. Although it makes more sense to plot the performance curves separately to exclude curves from non-native methods, for the sake of comparison in runtime and convergence behavior per iteration completed, the three methods are plotted together initially. For the Q Learning, the default initial epsilon of 1.0, minimum epsilon of 0.1 and geometric epsilon decay rate of 0.99 are used. Additionally, the default initial alpha of 0.1, minimum alpha of 0.001 and the geometric alpha decay rate of 0.99 are used.

Gridworld MDP: Comparing Q Learning to VI/PI - Convergence vs Cumulative Runtime.

Now let's observe Q Learning with regards to the runtime per iteration and error convergence in the gridworld size of 5x5, 15x15 and 45x45. The model-based methods of policy iteration and value iteration are compared in its convergence behavior in terms of error defined by future utility and cumulative runtime.



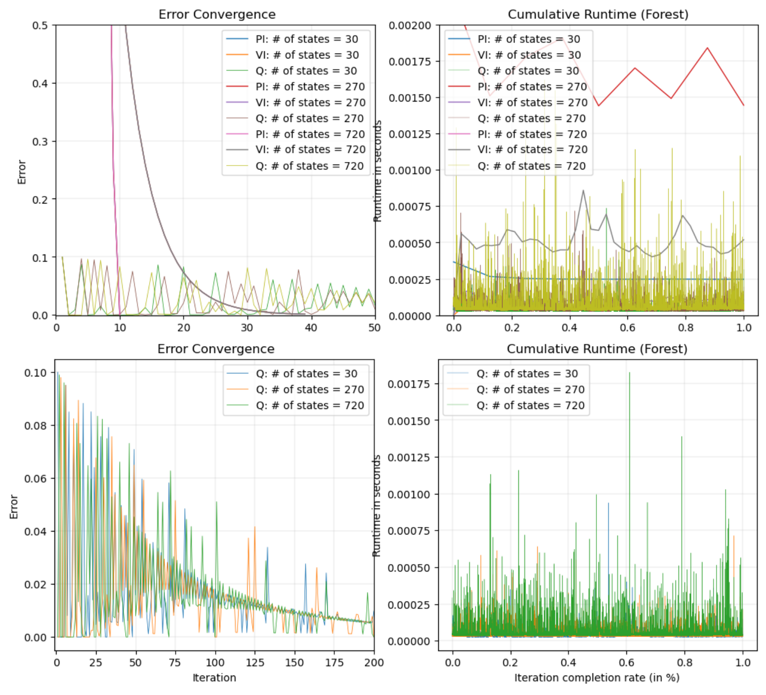
The Q-learning algorithm shows a different pattern of error as the iteration continues and appears it must be compared separately. The figure above is limited in x-axis to zoom in on the behaviors of the policy and the value iteration, however the Q-learning algorithm requires a comparison using different MDP sizes within only this algorithm to examine the behaviors. It can be shown that there are spikes rather than a curve given the definition of error for the Q learning family of algorithms. Since this is the amount of maximum value updated to the state-action pairs in the Q function, the spikes indicate the moments when the Q function learns in the episodes. Nonetheless, the Q-learning converged way later in terms of cumulative runtime.

This shows the unique behavior of the Q-learner's episodic learning process rather than showing gradual decrease for the error as the iteration goes on like in the case of model-based approaches. There actually appears to be regular spikes where the "error," or the learned value, jumps every fixed number of iterations (in below case 50~200 iterations depending on the size of the algorithm and the decay rate used). Initial spikes are the largest. The subsequent periodic spikes are out of focus in the plot above, which implies an episodic learning process where the Q-value is significantly and periodically updated after the agent finishes an episodic exploration of the MDP. After the largest initial spike, the degree of the spikes decrease as the iteration continues, suggesting potential to converge with the episodic learning reaches below a certain threshold.

The runtime is approximately directly proportional to the column size of the gridworld. It can be shown through additional experiments with the measurements of runtime per varying state sizes that the runtime is more closely related to the size of the gridworld in its width rather than in its area.

Figure on the bottom left zooms in on the first 30th iteration to observe a pattern to the initial spikes. For the initial spikes, it shows that the initial spikes of the larger sized MDPs occur in later iteration than the smaller sized MDPs, likely due to the larger numbers of state space to explore. To see this periodic occurrence, a different set of epsilon and alpha decay strategies can be viewed in the same-size MDP to be plotted for "error" convergence as well as the cumulative runtime of the varying strategies.

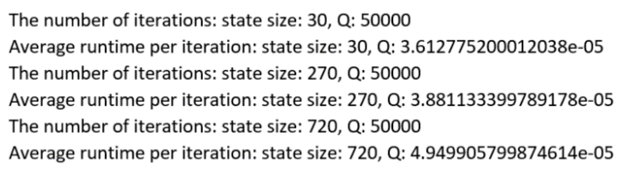
Forest MDP: Comparing Q Learning to VI/PI - Convergence vs Cumulative Runtime.



The different state space sizes of 30, 270, 720 are plotted using Q Learning, value iteration and policy iteration. A similar spiky pattern is observed for the Q learning and is the highest during the initial iterations. The spikiness levels down as the iteration continues. The runtime for the Q learning is not stable like that of value and policy iteration algorithms. The runtime does not have a narrowing or a widening trend. The run appears to spike every certain period without clear pattern generally with similar standard deviation throughout the iteration. The x axis is used to observe any linear increases. It is also observed that the value iteration and the Q learning are similar in the average runtime per iteration, suggesting that the Q learning takes the longest given the max_iter of 50000. The behavior of the errors and the runtime are completely different and cannot be compared in the same plot.

The Q learning runtime is generally constant as the iteration continues, showing infrequent surges. The runtime curve seems to suggest that a greater state size results in the greater mean runtime per iteration compared to lower iteration and showing slightly great infrequent surges. The larger state space of MDP requires more computation. Just as with the gridworld MDP, there is a need to view this with only Q learning error curves, to see what the long-term behaviors are like in varied focus. It shows that for larger state sizes the Q learning error's initial spikes are even higher. The Q learning's error decreases in volatility as the iteration continues and settles down in the early 100o iterations. Not only does the larger state space lead to a greater initial spike of the Q learning error.

So far, the probability of random action taken and the learning rate are initialized to 1.0 and 0.1 respectively and decayed geometrically with the rate of 0.99 for each. Perhaps above error and runtime behaviors are also sensitive to these exploratory and learning parameters of the family of algorithms. More visualization of the Q learning is showing in different settings with focus zoomed out.



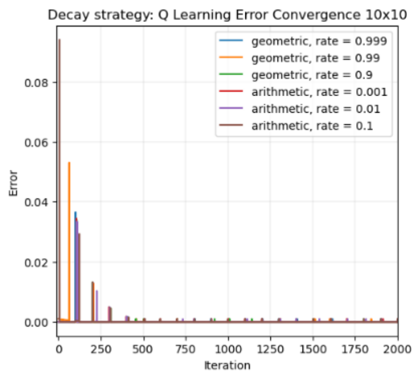
```
The number of iterations: state size: 30, Q: 50000
Average runtime per iteration: state size: 30, Q: 3.612775200012038e-05
The number of iterations: state size: 270, Q: 50000
Average runtime per iteration: state size: 270, Q: 3.881133399789178e-05
The number of iterations: state size: 720, Q: 50000
Average runtime per iteration: state size: 720, Q: 4.949905799874614e-05
```

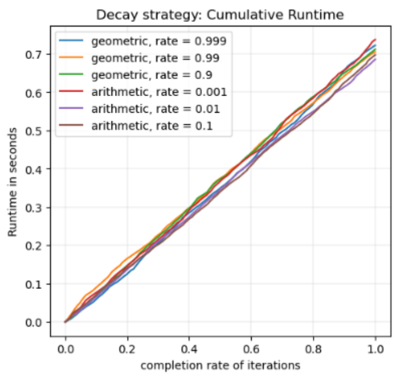
Q-Learning & the Exploration – Exploitation Dilemma

Just as randomized optimization algorithms can converge to a local optimum instead of a global optimum, the Q-learning family of algorithms can converge to the incorrect policy via the greedy action selection approach. For instance, if the Q function is initialized in a way that the exploration is limited to a certain set of actions per states, only this limited set is reinforced. The Q-learning convergence requires that all states, actions are visited infinitely often, and this is violated. There can be approaches to mitigate this, such as using random restarts every run or taking random actions based on a probability. This probability of random actions defined as epsilon can be decayed to exploit more than explore as the learner develops. The exploration approach taken here is the epsilon-greedy exploration whilst varying decay modes: arithmetic and geometric decay, plus with varying rates. Then it is compared with a different random initialization to see if there any better behaviors of convergence.

Gridworld MDP: Exploration using Epsilon-Greedy Approach

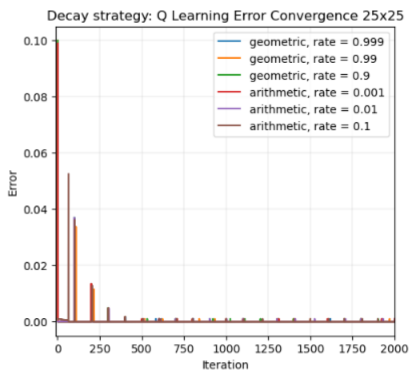
To balance exploration and exploitation, a simulated-annealing-like approach is used to perform the Q-learning where the probability of taking a random action to explore decays over the time horizon. Starting from the highest value of this probability, epsilon, the algorithm initially explores more than it exploits. As the number of episodes increases, the epsilon is decayed geometrically or arithmetically while depending on the decay rate and the algorithm gradually takes less random actions to exploit more than explore. To study the phenomenon, a varying set of epsilon decay strategy in terms of decay method and rate is studied in terms of convergence in episode reward as well as the runtime. To explore various epsilon decays, minimum epsilon value is set as 0.00001. To clearly see the effects of the different decays, the size of the MDP is limited to 10x10 and 25x25 only.

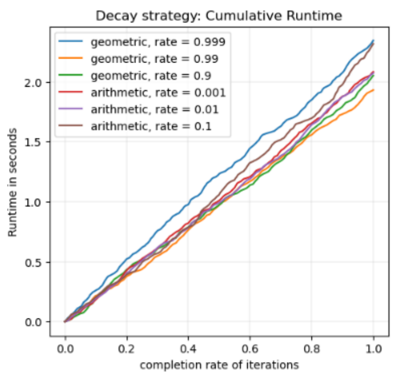




It turns out that regardless of the size the behaviors of the error are similar in the practical convergence within the first few hundred iterations. The clear difference shown is that there are varying runtime per iterations for the larger grid size MDP.

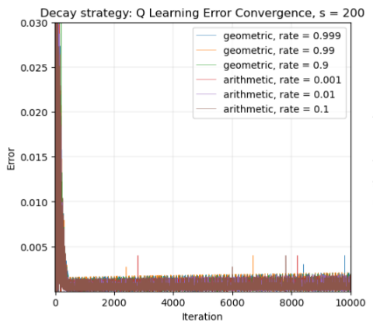
It is shown that regardless of the decay rate and type, the strategies follow a similar pattern. The Q learning algorithm produces spikes periodically and converges within the first few hundred iterations. Afterwards the periodic spikes of error persist but with less strength as the iteration continues. The runtimes for all of the strategies are approximately linear with some variances. There is no clear relationship between the runtime per iteration and iteration and the decay strategy as the slopes are high in varying rates. It appears to be largely dependent on the problem at hand, and in most cases that a tuning is necessary to see what kind of annealing might be necessary.

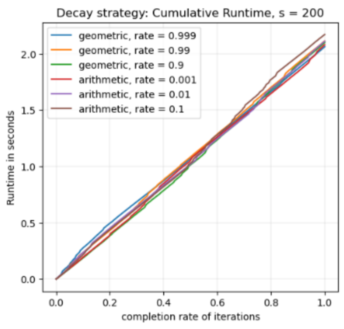


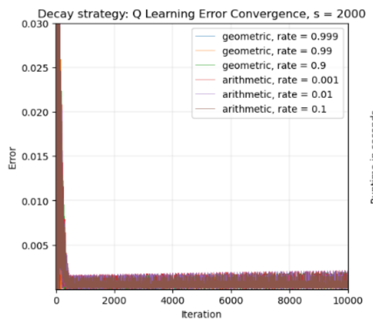


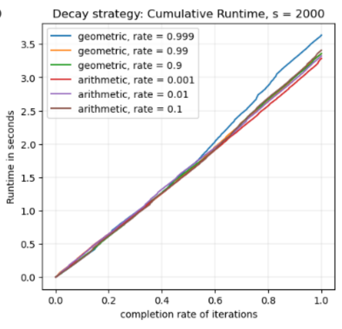
Forest Management MDP: Exploration using Epsilon-Greedy Approach

The forest management MDP shows a different behavior compared to the gridworld MDP in that the spikes occur more frequently. The most notable difference is that for any decay strategies the level of spikes gradually increases. It is similar in that the practical convergence occurs within the first few iterations. In this problem, the error convergence and the runtimes are also shown to be less sensitive to the decay strategy and the size of the problem as shown below. It appears that there are some unexpected spikes beyond the usual level for certain decay strategies when the state space size is smaller.









Mean runtime per iteration for geometric, r = 0.999: 4.1220400003330726e-05
Mean runtime per iteration for geometric, r = 0.99: 4.221874799739453e-05 (Increase)
Mean runtime per iteration for geometric, r = 0.9: 4.1950062004216306e-05 (Decrease)

Mean runtime per iteration for arithmetic, r = 0.001: 4.155618799952208e-05
Mean runtime per iteration for arithmetic, r = 0.01: 4.3381127998836746e-05 (Increase)
Mean runtime per iteration for arithmetic, r = 0.1: 4. 218555599753017e-05 (Decrease)

Total runtime for geometric, r = 0.999: 2.061020000166536
Total runtime for geometric, r = 0.99: 2.1109373998697265 (Increase)
Total runtime for geometric, r = 0.9: 2.0975031002108153 (Decrease)

Total runtime for arithmetic, r = 0.001: 2.077809399976104
Total runtime for arithmetic, r = 0.01: 2.1690563999418373 (Increase)
Total runtime for arithmetic, r = 0.1: 2.1092777998765087 (Decrease)

It is clarified from the runtime recorded for each decrease and increase in the decay rate for the geometric and the arithmetic decay method, respectively, there is no clear relationship between the decay rate and the mean runtime per iteration. Because of the similar mean runtime per iteration and the same number of iteration run, the overall runtime is also similar to one another and without any clear relationship to the decay method and the rate.

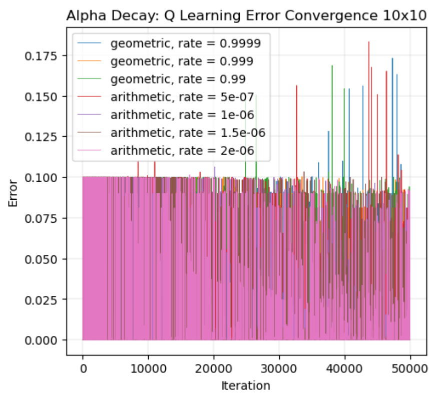
In the case of forest management MDP, due to that the forest MDP is more stochastic by design, it converges in the practical sense in the first 1000 iteration but continuing to run the iteration appears to show that the error, or the max amount the Q function updates in each iteration, gradually increases. Also, compared to the gridworld MDP that observes a periodic spike per episodes, this MDP shows much shorter "episodes."

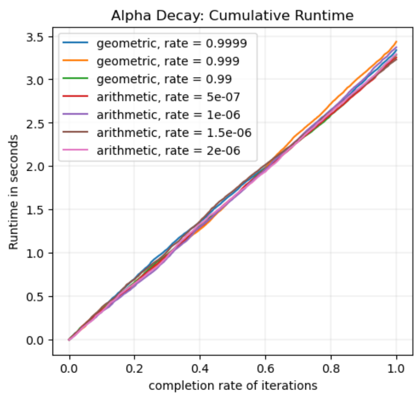
Learning Rate, Alpha

Lastly, the learning rate, or alpha, is compared over various values ranging between 0 and 1. Setting alpha to zero means that the Q function values are never updated whereas setting alpha to 1 means that the newly learned information completely overrides the old information. When the MDP is deterministic (without slip rate or other random probabilities) a learning rate of 1 is preferred. However, when the problem is stochastic like this one, there must be a decay in the learning rate for the algorithm to converge. When the alpha decays so as to allow the method to stop learning, it prevents the method from stochasticity-induced divergence.

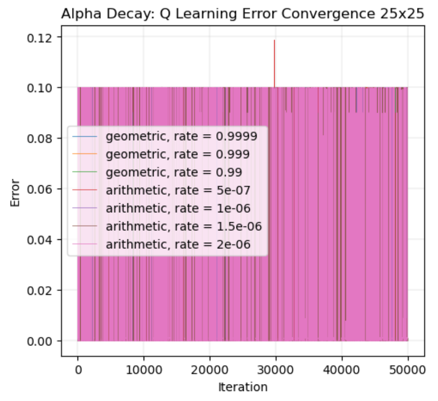
Gridworld MDP: Alpha Decay

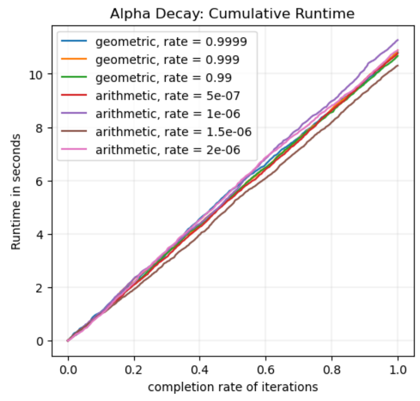
With similar initial settings to the epsilon decay, a 10x10 and a 25x25 gridworld MDP is studied via the comparison of varying alpha decay or non-decay strategies. As the decay continues and the alpha is set near-zero, it is hypothesized to make the algorithm's error converge to zero as that is how it is defined: *alpha*(R + gamma*Max[Q(s', A)] - Q[s, a])*





Strangely, it shows that by decaying alpha, the error function does not converge near zero at the 50000th iteration even for the largest alpha decay rate. This is strange since the alpha value is quite near zero at the 50000th iteration. But upon further analysis, the alpha value is not shown to be not zero enough due to the minimum alpha of 0.00001. Also when the plot's x-axis limit is expanded to 200,000, a clear decreasing of the error is visible for this gridworld MDP.





This occurrence of not converging near 50000th iteration is blatantly true for the curve with zero decay rate, i.e. with constant alpha, where the error practically converges to a non-zero value as shown on the bottom left figure. For other decay strategies it appears that continuing the iteration beyond the 50000th will eventually converge to zero according to the definition of the "error" of this Q-learning implementation when the size of the grid world is large at 25x25.

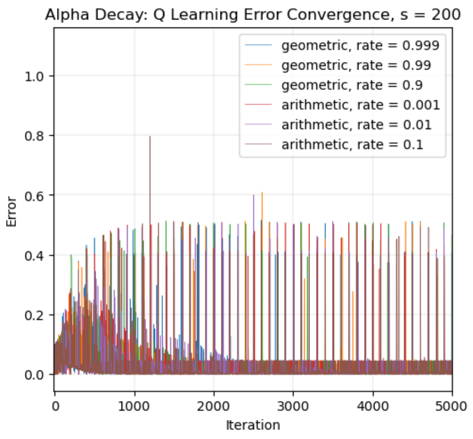
Again, by scrutinizing the mean runtime per iteration for the varying methods and varying rates, no clear ranking has been found, as shown on the print output on the right.

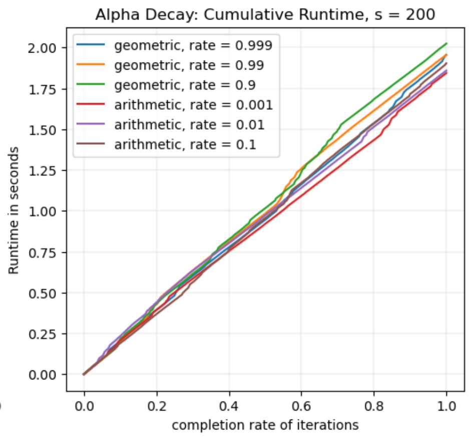
```
Mean runtime per iteration for geometric, r = 0.9999: 6.674493199990138e-05
Mean runtime per iteration for geometric, r = 0.999: 6.868227799944179e-05
Mean runtime per iteration for geometric, r = 0.99: 6.565776799930972e-05

Mean runtime per iteration for geometric, r = 0.9999: 0.0002160838480003804
Mean runtime per iteration for geometric, r = 0.999: 0.00021750600000118537
Mean runtime per iteration for geometric, r = 0.99: 0.00021337529000117682
```

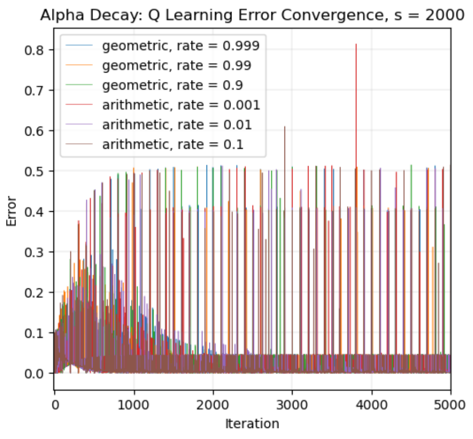
Forest Management MDP: Alpha Decay

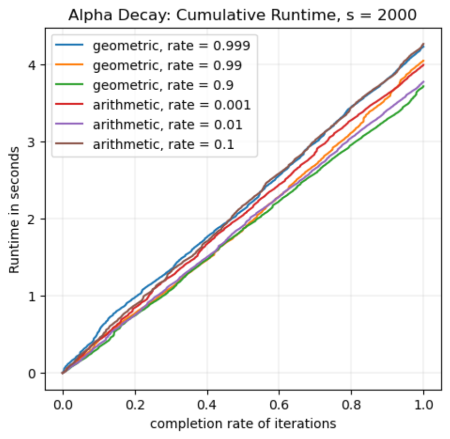
The forest MDP's alpha decay experiment shows a largely different behavior compared to the grid world MDP like the previous epsilon decay experiment's observation of the MDP where the signs of eventual convergence are nowhere to be found even when the iterations continue beyond the 50,000th.





In contrast to the previous behavior of divergence or a gradual increase in the error as the iterations continue in the case of epsilon decay experiment with re-initializations, the alpha decay appears to converge to a non-zero value when expanded the max_iter to 200,000 from 50000. The two different state space size shows no difference in the behaviors of the error functions as found in previous observations. Re-initializing the Q-function to repeat the experiment results in the same result each time.





Like the epsilon decay experiment, the varying rates of decay per the decay method does not show a clear relationship in the increase of the decrease of the mean runtime per iteration. This held true when tested over a larger log-span of rates, i.e. [0.1, 0.01,0.001,0.0001,0.0001], etc.

```
Mean runtime per iteration for geometric, r = 0.999: 3.909808799962775e-05
Mean runtime per iteration for geometric, r = 0.99: 3.9083601999700474e-05
Mean runtime per iteration for geometric, r = 0.9: 4.047241799191397e-05

Mean runtime per iteration for arithmetic, r = 0.001: 3.686344999958237e-05
Mean runtime per iteration for arithmetic, r = 0.01: 3.7171866000280716e-05
Mean runtime per iteration for arithmetic, r = 0.1: 3.805360200076393e-05
```

Conclusion

In solving stochastic MDP's, the Q-learning does perform better -- in the quality of convergence and the runtime considered -- compared to the model-based methods of value iteration and policy iteration. The experiment results observed here does not always support this, especially for smaller sized MDPs (size less than 10,000), since these "exact methods," particularly the value iteration, seem to converge in their notions of error faster than the Q-learning does despite the stochasticity introduced by the sliprate and outputs a sensible policy fast. However, it comes to mind that the concept of Q-learning is more robust – not only due to the good performance observed from the case where the state space size of the two MDPs increased exponentially -- but also because it seems to provide more flexibility in terms of dealing with problems that can arise from fitting more realistically conjured-up, complex MDPs. Perhaps the two MDPs studied here are not the best examples to confirm the power of the family of algorithms due to the relative simplicity in their structures.

Overall, it also turns out that the context of the problem drives the behaviors of these algorithms' performances at the parametric level in terms of the convergence of the pre-defined notion of "errors" implemented, runtime per iteration, number of implementations and the sensibility of the policy outputs. This is more reason to study and analyze the problem itself in terms of the MDP to deal with its unique workings of the exploration-exploitation dilemma.