Solutions to Selected Problems In: Reinforcement Learning: An Introduction by Richard S. Sutton and Andrew G. Barto.

John L. Weatherwax*

March 26, 2008

Chapter 1 (Introduction)

Exercise 1.1 (Self-Play):

If a reinforcement learning algorithm plays against itself it might develop a strategy where the algorithm facilitates winning by helping itself. In other words it might alternate between "good" moves and "bad" moves in such a way that the algorithm wins every game. Thus it would seem that the algorithm would not learn a min/max playing strategy.

Exercise 1.2 (Symmetries):

We could improve our reinforcement learning algorithm by taking advantage of symmetry by simplifying the definition of the "state" and "action" upon which the algorithm would works. By simplifying the state in such a way that the dimension decreases we can be more confident that our learned results will be statistically significant since the state space we operate in is reduced. If our opponent *was* taking advantage of symmetries in the game tic-tac-toe our algorithm should also since this fact, would enable us to be a better game player against this type of player. If our player does *not* use symmetries then our algorithm should not either (except to reduce the state space as discussed above) since enforcing a symmetry on our opponent (that is not in fact there) should decrease our performance when playing against this type of opponent.

*wax@alum.mit.edu

Exercise 1.3 (Greedy Play):

As will be discussed later in this book a greedy approach will not be able to learn more optimal moves as play unfolds. Thus there is a natural trade-off between attempting to explore the space of possibilities and selecting the action on this step that has the greatest reward. Problems with direct greedy play would be that our player fails to be able to capture moves that result in improved rewards because we never take a chance on unknown (or unexplored) actions.

Exercise 1.4 (Learning from Exploration):

If we *do not* learn from exploratory moves then the state probabilities learned would effectively be random in that we are not updating our undertaking of what happens when in a given state and a given action is taken. If we learn from our exploratory moves then our limiting probabilities should be those from the desired distribution of state and action selections. Obviously a more complete understanding of the probability densities should result in a better play since the player better understands the "game" he or she is playing.

Exercise 1.5 (Other Improvements):

One possible way would be to have a saved library of scripted plays. For example the logic would be something like, when in a set of known states always execute the following moves. This is somewhat like the game of chess where there are various "opening" positions that expert players have deemed good. Hopefully this might expedite the total learning process or at least improve our reinforcement players initial play.

Since the the tic-tac-toe problem is so simple we can solve this problem (using recursion) and computing *all* possible opponents moves and selecting at each step the move that optimizes our chance of winning. This is in fact a common introductory programming exercise.

Chapter 2 (Evaluative Feedback)

Action-Value Methods

See the Matlab file `n_armed_testbed.m` for code to generate the plots shown in Figure 2.1 of the book. We performed experiments with two different values for the standard deviation of the rewards received: 1 and 10.

Exercise 2.1 (Discussions on the 10-armed testbed):

Plotting both the cumulative average reward and the cumulative percent optimal action we see that in both cases the strategy with more exploration i.e. $\epsilon = 0.1$ will produce a larger cumulative reward and larger percentage of time the optimal reward is obtained. For 1000 plays the $\epsilon = 0.1$ strategy obtains a cumulative total reward of about 1300 while the greedy strategy obtains a total reward of about 1000 or an improvement of 30%. The above experiment is with the reward variance from each "arm" set at 1.0.

When we set the variance of each random draw to a larger number say $\sigma^2 = 10.0$ we see the same pattern.

I'll mention a very simple approach that greatly improves the performance of the greedy algorithm. This idea is motivated by the algorithms described in the book where they all start no knowledge of the true reward function $Q^*(a)$. These algorithms initially approximates this at $t = 0$ with zero values before beginning of each strategy. If instead, we allow our algorithm to take one draw on *each* arm and uses the observed values that result as its initial approximation for $Q^*(a)$ the performance is considerably improved. If the variance of the rewards on each arm is small then this set of initial trials will yield relatively good results since we have a very simple estimate (a single sample) of the rewards from each arm.

Softmax-Action-Selection

Exercise 2.2 (programming softmax-action selection):

See the Matlab file `n_armed_testbed_softmax.m` for code to generate plots similar to those shown in Figure 2.1 of the book but using softmax action selection.

Exercise 2.3 (the logistic or sigmoid function):

With two actions (say x and$y$) the Gibbs distribution discussed in this section requires comparing the following two probabilities

$$\frac{e^{x/\tau}}{e^{x/\tau} + e^{y/\tau}} \quad \text{and} \quad \frac{e^{y/\tau}}{e^{x/\tau} + e^{y/\tau}} .$$

By multiplying the first expression by $e^{-x/\tau}$ on the top and the bottom we obtain

$$\frac{1}{1 + e^{(y-x)/\tau}} ,$$

which is the definition of the logistic function. The temperature τ here plays the role of how quickly the incoming signal is "squashed". The larger τ is the more quickly the function squashes its inputs and the distribution becomes uniform.

Evaluation versus Instruction

See the Matlab files `binary_bandit_exps.m` and `binary_bandit_exps_Script.m` for code to generate the plots shown in Figure 2.3 of the book. In addition, we plot the results for two "easy" problems denoted as the upper left and lower right corners of the diagram in Figure 2.2. These easy problems are where $p_1 = 0.05$ and $p_2 = 0.85$ and $p_A = 0.9$ and $p_B = 0.1$.

Incremental Implementation

Exercise 2.5 (the n-armed bandit with $\alpha = 1/k$):

See the Matlab files `exercise_2_5.m` for code to simulate the n-armed bandit problem, with greedy action selection and incremental computation of action values (here $\alpha = 1/k$).

Tracking a Non-stationary Problem

Exercise 2.6:

We are told in this problem that the step sizes, $\alpha_k(a)$, are not constant with the timestep but have a k dependence. Following the discussion in the book we will derive an expression for Q_k in terms of its initial specification Q_0, the received rewards r_k, and the step size values α_k. Now for *each* action a our action-value estimate is updated using

$$Q_k = Q_{k-1} + \alpha_k(r_k - Q_{k-1})\,,$$

where we have dropped the dependence on the action taken (i.e. a) for notational simplicity. In the above formula we can recursively substitute for the earlier terms (e.g. replace Q_{k-1} with an expression involving Q_{k-2}) to obtain the desired expression. We will follow this prescription until the pattern is clear and at that point generalize. We find

$$\begin{aligned}
Q_k &= Q_{k-1} + \alpha_k(r_k - Q_{k-1}) \\
&= \alpha_k r_k + (1-\alpha_k)Q_{k-1} \\
&= \alpha_k r_k + (1-\alpha_k)\left[\alpha_{k-1}r_{k-1} + (1-\alpha_{k-1})Q_{k-2}\right] \\
&= \alpha_k r_k + (1-\alpha_k)\alpha_{k-1}r_{k-1} + (1-\alpha_k)(1-\alpha_{k-1})Q_{k-2} \\
&= \alpha_k r_k + (1-\alpha_k)\alpha_{k-1}r_{k-1} + (1-\alpha_k)(1-\alpha_{k-1})\left[\alpha_{k-2}r_{k-2} + (1-\alpha_{k-2})Q_{k-3}\right] \\
&= \alpha_k r_k + (1-\alpha_k)\alpha_{k-1}r_{k-1} + (1-\alpha_k)(1-\alpha_{k-1})\alpha_{k-2}r_{k-2} \\
&+ (1-\alpha_k)(1-\alpha_{k-1})(1-\alpha_{k-2})Q_{k-3} \\
&= \cdots \\
&= \alpha_k r_k + (1-\alpha_k)\alpha_{k-1}r_{k-1} + (1-\alpha_k)(1-\alpha_{k-1})\alpha_{k-2}r_{k-2} + \cdots
\end{aligned}$$

$$
\begin{aligned}
&+ \quad (1-\alpha_k)(1-\alpha_{k-1})(1-\alpha_{k-2})\cdots(1-\alpha_3)\alpha_2 r_2 \\
&+ \quad (1-\alpha_k)(1-\alpha_{k-1})(1-\alpha_{k-2})\cdots(1-\alpha_2)\alpha_1 r_1 \\
&+ \quad (1-\alpha_k)(1-\alpha_{k-1})(1-\alpha_{k-2})\cdots(1-\alpha_2)(1-\alpha_1)Q_0 \,.
\end{aligned}
$$

Thus in the general case the weight on the prior reward Q_0 is given by $\prod_{i=1}^{k}(1-\alpha_i)$.

Exercise 2.7:

See the Matlab files `exercise_2_7.m` and `exercise_2_7_Script.m` for code to simulate the n-armed bandit problem on a non-stationary problem. As suggested in the problem given an initial set of action values $Q^*(a)$ on each timestep the action values take a random walk. That is at the next timestep they are given by

$$
Q^*(a) = Q^*(a) + \sigma_{\text{RW}} dW \,,
$$

where dW is a random variable drawn from a standard normal. We have introduced a variable σ_{RW} that determines how much non-stationary is introduced at each step. We expect that as σ_{RW} increases the sample-average method will perform increasingly worse than the action-value method with a constant step size α. This can be seen from the experiments performed with the code.

Optimistic Initial Values

See the Matlab files `opt_initial_values.m` and `opt_initial_values_Script.m` for code to generate the plots shown in Figure 2.4 of the book.

Exercise 2.8:

If the initial play selected when using the optimistic method are by chance ones of the *better* choices then the action value estimates $Q(a)$ for these plays will be magnified resulting in an emphasis to continue playing this action. This results in large actions values being received on the initial draws and consequently very good initial play. In the same way, if the algorithm initially selects poor plays then initially the algorithm will perform poorly resulting in very poor initial play.

Reinforcement Comparison

See the Matlab files `reinforcement_comparison_methods.m` and the driver "Script" file for code to generate the plots shown in Figure 2.5 of the book.

Exercise 2.9:

The fact that the temperature parameter is not found in the softmax relationship for this problem is not a problem since we introduce another constant β in our action preference update equation. This parameter β has the affect of various temperature settings in a Gibbs like probability update.

Exercise 2.10:

The α parameter in the reference reward update equation determines how fast we converge to a reference value $\alpha = 0$ and convergence is immediate where when $\alpha = 1$ the sequence $\bar{r}_t$ changes with each reward. The same comments apply to the action preferences $p_t(a)$ and the parameter β. Since at the beginning of our experiments we don't know either of these unknowns the reference reward value $\bar{r}$ or the action preferences $p(a)$ to appropriately learn these we need two parameters. If we set $\alpha = \beta$ we would effectively be enforcing that the rate of convergence of each of the estimates should be the same.

Exercise 2.11:

The action preference update equation is modified as follows

$$p_{t+1}(a_t) = p_t(a_t) + \beta(r_t - \bar{r}_t)(1 - \pi_t(a_t)) \,.$$

This modification is implemented in the Matlab file `exercise_2_11.m`, and the driver "Script" file. The plots produced by this code indeed show that the addition of this factor does indeed improve the performance of the algorithm.

Pursuit Methods

See the Matlab files `persuit_method.m` and `persuit_method_Script.m` for code to generate the plots shown in Figure 2.6 of the book.

Exercise 2.12:

The persuit algorithm is *not* guaranteed to always select the optimal action. For example if the initial rewards are such that we incorrectly update the wrong actions, causing $\pi(a)$ to become biased away from its true value we could never actually draw from the optimal arm. Setting the initial action probabilities equal to a uniform distribution helps this. In the case where the prior distributions are set such that $\pi = 1$ for a specific arm and $\pi = 0$ for all others emphasizes the fact that the algorithm does not fully explore the state space.

Exercise 2.15:

One idea is to have the algorithm perform an ϵ-persuit algorithm where ϵ percent of the time we randomly try an arm. The remaining $1-\epsilon$ amount of the time we follow the standard persuit algorithm. Thus even if the action probabilities $\pi_t(a)$ converge to something incorrect the fact that ϵ percent of the time we are selecting a random arm means that in enough trials we will explore the entire space.

As a practical detail, we could code our methods so that on the ϵ percent of the time it is exploring it is explicitly forbidden from drawing on the "greedy" arm. This would be the arm that would be most likely be selected using the action probabilities π_t. That is, don't draw on the arm a^* given by

$$a^* = \text{ArgMax}_a(\pi_t(a)) \,.$$

This modification may help in the exploration process since we don't waste an exploratory draw.

Associative Search

Exercise 2.16 (unknown bandits):

If you are not told which of the cases of the problem one faces, we can assume that on average half of the time we will be facing a case A instance and half of the time a case B instance. In general if we knew which case we were facing we would like to pull different arms (the optimal play would be to pull the second arm in the case of A instance and pull the first arm in the case of a case B instance). Since we have no knowledge the best we can obtain is the average reward. The expectation of playing each type of game (assuming that we have no way to select which arm to pull and we select randomly) is given by

$$\begin{aligned} E[A] &= 0.5(0.1) + 0.5(0.9) = 0.5 \\ E[B] &= 0.5(0.2) + 0.5(0.8) = 0.5 \,. \end{aligned}$$

So assuming a uniform distribution over which type of case we are given and where we play many games is given by

$$0.5(0.5) + 0.5(0.5) = 0.5 \,.$$

In the case where one is *told* which case one is playing one could learn the *optimal* arm to pull in each case and then play that arm repeatedly. In this case the expected reward for playing many times is given by

$$0.5(0.2) + 0.5(0.9) = 0.55 \,,$$

thus we see that when we know the type of the game we are playing our expected reward increases as one would expect.

Chapter 3 (The Reinforcement Learning Problem)

The Agent-Environment Interface

Exercise 3.1 (examples of reinforcement learning tasks):

One example might be the robot learning of how to escape a maze. The state could be the robots position (and directional heading) in the maze. The actions could be one of the following: to move forward, backwards, or sideways. The reward could be negative one if the robot collided with a wall and/or plus one once the solution to the maze is found.

Another example might be a reinforcement algorithm aimed a learning how to drive a car around a race track. The state could be a vector representing the distance to each of the lateral sides (the walls to the left and right of the car), the directional heading, the velocity and acceleration of the car. The actions could be to change any of the state vectors i.e. to accelerate or decelerate or to change heading. The rewards could be zero if the car is proceeding comfortably along and minus one if the car collides with a wall.

A third example might be learning the optimal time to buy or sell securities given some information about the current state of the securities return. In this case the state would be the magnitude and direction of the instantaneous return, the action might be the size of a position to take (how much of a security to buy or sell) and whether to buy or sell. The reward might be the corresponding profit or loss from this action. One would hope that over time the reinforcement learning algorithm would learn how much and the given direction to take to maximize a profit.

Exercise 3.2 (other goal directed learning tasks):

One difficulty could be with learning tasks that result in a *vector* of rewards rather than just a scalar. In this case one is trying to maximize simultaneously several things. If there is no clear way to reduce this to a scalar problem it might be unclear how to apply reinforcement learning ideas. One way to produce a scalar problem is to come up with a set of "weights" and take the inner product of the rewards with these weights.

Exercise 3.3 (driving as a reinforcement task):

The correct level at which to define the agent and the environment depends on what task one wants to use the algorithm for. For example if one wants to model the car interacting with the road it maybe simpler to use the situation where we model the implied torques on the wheels. If one however want to model interacting drivers on various roads then it maybe better to model everything at a much higher level, where we are considering "where"

to drive. For any given task some information maybe easier to obtain and may guide the modeler in selecting the level to work at.

Returns

Exercise 3.4 (pole-balancing as an episodic task):

In this case the returns at each time will be related to $-\gamma^k$ as before but only up to some K less than the episode end time T. This differs in that they is a ceiling ($-\gamma^T$) above which the returns cannot get larger than.

Exercise 3.5 (running in a maze):

You have not communicated a "time limit" to the robot in this formulation. Since the agent suffers no loss while exploring the maze one possible solution is to explore until our episode of time elapses. Since we want our robot to actively seek out a solution to the maze problem better performance may result if we prescribe a reward of -1 (or a penalty) for each timestep the robot spends in the maze without finding the exit.

The Markov Property

Exercise 3.6 (broken vision system):

In a general principles, to have a Markov state means that the knowledge of this state is sufficient to predict the next state and the expected reward given some action. Since we cannot see everything the photograph of the objects does not contain the entirety of information needed to next predict what state (image) we will be presented with next. For example, we don't know if something will pass over our head from behind us and land in our field of view. If we assume that everything we will ever see is in front of us then not knowing the objects velocities prevents us from having a full Markov state. If we assume that everything is *stationary* then we would might say that we had a Markov state. Obviously if our camera was broken for some length of time we would not be able to predict fully in the future what we might see and the last known image would not be a Markov state.

Markov Decision Processes

Exercise 3.7 (linking transition probabilities and expected rewards):

Consider first the expression for the expected next reward $R^a_{s,s'}$. This is defined as the expectation of the reward r_{t+1} given the current state, the action taken, and the next state i.e.

$$R^a_{s,s'} = E\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\}\,.$$

By the definition of expectation this is equal to

$$\sum_{r'} r' P\{r_{t+1} = r'|s_t = s, a_t = a, s_{t+1} = s'\}\,.$$

Where the sum is over all of the finite possible rewards r'. By the definition of conditional probability, the probability in the above can be written in terms of the joint conditional distribution as

$$P\{r_{t+1} = r'|s_t = s, a_t = a, s_{t+1} = s'\} = \frac{P\{s_{t+1} = s', r_{t+1} = r'|s_t = s, a_t = a\}}{P\{s_{t+1} = s'|s_t = s, a_t = a\}}\,.$$

In the above we recognize the denominator as equal to the definition of our transition probabilities for our finite Markov decision process namely $P^a_{s,s'}$. Thus we obtain

$$R^a_{s,s'} = \sum_{r'} r' \frac{P\{s_{t+1} = s', r_{t+1} = r'|s_t = s, a_t = a\}}{P^a_{s,s'}}\,.$$

Multiplying both sides by $P^a_{s,s'}$ we finally obtain the desired relationship

$$R^a_{s,s'} P^a_{s,s'} = \sum_{r'} r' P\{s_{t+1} = s', r_{t+1} = r'|s_t = s, a_t = a\}\,.$$

Value Functions

I found it necessary to consider the derivation of Bellman's equation in more detail, which I do here. Remembering that the definition of $V^\pi(s)$ is given by $E_\pi\{R_t|s_t = s\}$, and $R_t = \sum_{k=0}^\infty \gamma^k r_{t+k+1}$ we have the first set of equations given in the book. That is

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t|s_t = s\} \\ &= E_\pi\{\sum_{k=0}^\infty \gamma^k r_{t+k+1}|s_t = s\} \end{aligned}$$

To evaluate this expectation we will use the conditional expectation theorem which states that one can evaluate the expectation of a random variable say X by conditioning on all possible other events (here Y and Y^c). Mathematically, this statement is

$$E[X] = E[X|Y]P\{Y\} + E[X|Y^c]P\{Y^c\}\,.$$

In the above expectation we now condition over each chosen action at timestep t i.e. a_t. This gives

$$V^\pi(s) = \sum_a E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} P\{a_t = a | s_t = s\}$$

Since the probabilities $P\{a_t = a|s_t = s\}$ are our policy which have been denoted $\pi(s, a)$ with this change in notation the above becomes

$$V^\pi(s) = \sum_a \pi(s, a) E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} .$$

Again using the conditional expectation theorem on the policy expectation E_π but this time conditioning on the state s' that the environment puts us into we find the above equal to

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a, s_{t+1} = s'\} P\{s_{t+1} = s' | s_t = s, a_t = a\} .$$

Since $P\{s_{t+1} = s'|s_t = s, a_t = a\}$ is probability of the next state begin s' or $P^a_{s,s'}$ and with this change in notation we have

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a, s_{t+1} = s'\} P^a_{s,s'} .$$

Now releasing the first reward r_{t+1} (or the $k = 0$ term) and using the linearity of the expectation we find

$$\begin{aligned} V^\pi(s) &= \sum_a \pi(s, a) \sum_{s'} P^a_{s,s'} E_\pi\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\} \\ &+ \sum_a \pi(s, a) \sum_{s'} P^a_{s,s'} E_\pi\{\sum_{k=0}^{\infty} \gamma^{k+1} r_{t+k+2} | s_t = s, a_t = a, s_{t+1} = s'\} . \end{aligned}$$

Remembering the definition of $R^a_{s,s'}$ as $E_\pi\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\}$ the above becomes

$$\begin{aligned} V^\pi(s) &= \sum_a \pi(s, a) \sum_{s'} P^a_{s,s'} R^a_{s,s'} \\ &+ \gamma \sum_a \pi(s, a) \sum_{s'} P^a_{s,s'} E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a, s_{t+1} = s'\} . \end{aligned}$$

Now by the Markov property of our states this second expected reward does not depend on the previous state s or the action taken then a and we have

$$E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a, s_{t+1} = s'\} = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s'\} ,$$

also this last expectation is the definition of our state policy function V^π at the state s' or $V^\pi(s')$. Putting everything together we have

$$\begin{aligned} V^\pi(s) &= \sum_a \pi(s, a) \sum_{s'} P^a_{s,s'} R^a_{s,s'} + \gamma \sum_a \pi(s, a) \sum_{s'} P^a_{s,s'} V^\pi(s') \quad (1) \\ &= \sum_a \pi(s, a) \sum_{s'} P^a_{s,s'} \left[R^a_{s,s'} + \gamma V^\pi(s') \right] . \end{aligned}$$

Which is Bellman's equation as presented in the book but just derived in more (perhaps excruciating) detail. Before continuing, as discussed in the book Bellman's equation represents the trade off between immediate rewards and future rewards that occur from being in future states. In the Equation 1 the first term represents the average of the *immediate* rewards while the second term represents the average rewards obtainable by transitioning states.

Exercise 3.8 (the Bellman equation for the action-value function $Q^\pi(s,a)$):

Following the derivation of Bellman's equation for the state-value function $V^\pi(s)$ we begin with the definition of $Q^\pi(s,a)$

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{R_t|s_t=s,a_t=a\} \\ &= E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s,a_t=a\}\,. \end{aligned}$$

Again using the conditional expectation theorem, by conditioning on the state s' that the environment puts us into when our agent starts in state s and takes action a we find that $Q^\pi(s,a)$ is given by

$$Q^\pi(s,a)=\sum_{s'}E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s,a_t=a,s_{t+1}=s'\}P\{s_{t+1}=s'|s_t=s,a_t=a\}\,.$$

Changing notation by introducing $P^a_{s,s'}$ as the second probability factor above, releasing the $k=0$ term, and factoring out a γ in the second summation we find that we now have $Q^\pi(s,a)$ given by

$$\begin{aligned} Q^\pi(s,a) &= \sum_{s'}E_\pi\{r_{t+1}|s_t=s,a_t=a,s_{t+1}=s'\}P^a_{s,s'} \\ &+ \gamma\sum_{s'}E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t=s,a_t=a,s_{t+1}=s'\}P^a_{s,s'}\,. \end{aligned}$$

Introducing $R^a_{s,s'}$ in the first expectation above and the fact that we are assuming Markov states the above becomes

$$Q^\pi(s,a)=\sum_{s'}R^a_{s,s'}P^a_{s,s'}+\gamma\sum_{s'}E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1}=s'\}P^a_{s,s'}\,.$$

We now use the conditional expectation theorem again this time on the expectation term conditioning on the possible actions we can take from state s'. We have

$$\begin{aligned} E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1}=s'\} &= E_\pi\{R_{t+1}|s_{t+1}=s'\} \\ &= \sum_{a'}E_\pi\{R_{t+1}|s_{t+1}=s',a_{t+1}=a'\}P\{a_{t+1}=a'|s_{t+1}=s'\} \\ &= \sum_{a'}E_\pi\{R_{t+1}|s_{t+1}=s',a_{t+1}=a'\}\pi(s',a') \\ &= \sum_{a'}Q^\pi(s',a')\pi(s',a')\,. \end{aligned}$$

When this is substituted into the expression for $Q^\pi(s,a)$ we obtain

$$Q^\pi(s,a) = \sum_{s'}R^a_{s,s'}P^a_{s,s'}+\gamma\sum_{s'}\sum_{a'}Q^\pi(s',a')\pi(s',a')P^a_{s,s'}\,.$$

as Bellman's equation for the action-value function Q^π. Again we two terms which representing the trade off that exists between immediate rewards and future rewards that can be obtained by from alternative states s'.

Exercise 3.9 (verification of Bellman's equation for gridworld):

We want to check that at the center state we have equality in Bellman's equation. Since the left hand side is given by $V^\pi(s) = +0.7$ we check that the right hand side is equal to this. Letting the letters n, s, w, and e denote the squares to the north, south, west, and east respectively we see that the right hand side of Bellman's equation is given by

$$\begin{aligned} & \pi(s,a=n)\sum_{s'} P_{s,s'}^{a=n}(R_{s,s'}^{a=n}+\gamma V^\pi(s')) + \pi(s,a=e)\sum_{s'} P_{s,s'}^{a=e}(R_{s,s'}^{a=e}+\gamma V^\pi(s')) \\ + & \pi(s,a=s)\sum_{s'} P_{s,s'}^{a=s}(R_{s,s'}^{a=s}+\gamma V^\pi(s')) + \pi(s,a=w)\sum_{s'} P_{s,s'}^{a=w}(R_{s,s'}^{a=w}+\gamma V^\pi(s')) \\ = & \frac{1}{4}(R_{s,s'=n}^{a=n}+\gamma V^\pi(s'=n)) + \frac{1}{4}(R_{s,s'=e}^{a=e}+\gamma V^\pi(s'=e)) \\ + & \frac{1}{4}(R_{s,s'=s}^{a=s}+\gamma V^\pi(s'=s)) + \frac{1}{4}(R_{s,s'=w}^{a=w}+\gamma V^\pi(s'=w))\,. \end{aligned}$$

This is because $R=0$ for all of the these steps the above becomes

$$\begin{aligned} & \frac{\gamma}{4}(V^\pi(s'=n)+V^\pi(s'=e)+V^\pi(s'=s)+V^\pi(s'=w)) \\ = & \frac{0.9}{4}(2.3+0.4-0.4+0.7) = \frac{2.7}{4} \approx 0.7\,, \end{aligned}$$

to the accuracy of the numbers given.

Exercise 3.10 (a constant reward increase):

Considering the definition of $V^\pi(s)$ the state value function under policy π, we have

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t|s_t=s\} \\ &= E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s\}\,. \end{aligned}$$

We assume that each reward has a constant added to it. That is we consider the reward $\hat{r}_{t+k+1}$ defined in terms of r_{t+k+1} by

$$\hat{r}_{t+k+1} = r_{t+k+1} + C\,,$$

then the state value function for this new sequence of rewards is given by

$$\begin{aligned} \hat{V}^\pi(s) &= E_\pi\{\sum_{k=0}^{\infty}\gamma^k \hat{r}_{t+k+1}|s_t=s\} \\ &= E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s\} + E_\pi\{\sum_{k=0}^{\infty}\gamma^k C|s_t=s\} \\ &= V^\pi(s) + C\sum_{k=0}^{\infty}\gamma^k \\ &= V^\pi(s) + \frac{C}{1-\gamma}\,. \end{aligned}$$

Thus the value of K that is added to $V^\pi(s)$ is $\frac{C}{1-\gamma}$.

Exercise 3.11 (adding a constant to episodic tasks):

It would seem that if there were only a finite number of terms then one could perform the same manipulation as above and make the same type of argument as in Exercise 3.10. Thus I don't see how adding a constant to each reward in an episodic task would change the resulting policy.

Exercise 3.12 (the state value function backup diagram):

The intuition in this backup diagram is that the state value function at s must be the "average" of the possible action value functions for each possible action a. In equation form this this

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t|s_t = s\} \\ &= \sum_a E_\pi\{R_t|s_t = s, a_t = a\}P\{a_t = a|s_t = s\} \\ &= \sum_a E_\pi\{R_t|s_t = s, a_t = a\}\pi(s,a)\,. \end{aligned}$$

Since the *definition* of the expectation term above (i.e. $E_\pi\{R_t|s_t = s, a_t = a\}$) is $Q^\pi(s,a)$ we see that the above is equal to

$$V^\pi(s) = \sum_a Q^\pi(s,a)\pi(s,a)$$

Exercise 3.13 (the action value function backup diagram):

The intuition in using the backup diagram is that we recognize the action value function, $Q^\pi(s,a)$, as the expected sum of the next reward r_{t+1} plus the expected sum of the remaining rewards (which depend on the next state and the policy). Using the backup diagram we have

$$Q^\pi(s,a) = E_\pi\{r_{t+1}|s_t = s, a_t = a\} + \gamma E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s, a_t = a\}\,.$$

Now the first term simplifies by conditioning on the next state s' we find our self in

$$\begin{aligned} E_\pi\{r_{t+1}|s_t = s, a_t = a\} &= \sum_{s'} E_\pi\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\}P\{s_{t+1} = s'|s_t = s, a_t = a\} \\ &= \sum_{s'} R^a_{s,s'}P^a_{s,s'} \end{aligned}$$

The second term (without the γ) is exactly

$$\begin{aligned} E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s, a_t = a\} &= \sum_{s'} E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s, a_t = a, s_{t+1} = s'\} \\ &\times \; P\{s_{t+1} = s'|s_t = s, a_t = a\} \\ &= \sum_{s'} E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1} = s'\}P^a_{s,s'}\,, \end{aligned}$$

since the sum of rewards ($\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}$) are received at times $t+1$ onward at which point, after conditioning we are in state s' and the previous state s and action a give no information (assuming our states are Markov). Finally, replacing the expectation

$$E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s'\}$$

with $V^\pi(s')$ we have shown that $Q^\pi(s,a)$ is equal to

$$Q^\pi(s,a) = \sum_{s'} (R^a_{s,s'} + \gamma V^\pi(s')) P^a_{s,s'} \,,$$

Optimal Value Functions

See the Matlab code `rr_state_bellman.m` for code to numerically evaluate the optimal Bellman equations for the recycling robot.

Exercise 3.14 (the optimal state-value function for golf):

If we assume a penalty (reward) of -1 for each stroke we take to get the ball in the hole, the optimal state-value function $V^*(s)$ gives the expected number of swings to get the ball in the hole using any type of club. Thus where we expect it to take many swings to get the ball in the hole using just a putter, we expect that using *both* a putter and a driver would allow one to get the ball in the hole in a quicker amount of time. Thus the optimal state-value function for golf would qualitatively like that given in Figure 3.6 but would have smaller numbers on the outlying contours. This is because using both types of clubs we expect to be able to get the ball in the hole in fewer swings than just using the putter.

Exercise 3.15 (the contours of the optimal action-value for putting):

The optimal action-value function $Q^*(s, \text{putter})$ means that we take the action of using a putter and then follow the *optimal* policy from that point onward. I would expect that taking a putter first would be the optimal action for region of state space that are close to the hole and this state value function would give an expected number of swings that were quite small. For regions of state space further away from the hole I would expect the choice of using a putter as the first club to *not* be optimal and this would give action-value states that are slightly lower (and requiring more swings) in these regions of state space.

Exercise 3.16 (the Bellman equation for the recycling robot):

Bellman's optimal equation for $Q^*(s,a)$ is given by

$$Q^*(s,a) = \sum_{s'} P^a_{s,s'}(R^a_{s,s'} + \gamma \max_{a'} Q(s',a'))\,.$$

We will evaluate this function for all the possible states the robot can be in and all the corresponding actions the robot can take when in those states. To begin, if the robots is in the state of "high" (h) (with any action a) the above becomes

$$Q^*(h,a) = \sum_{s'} P^a_{h,s'}(R^a_{h,s'} + \gamma \max_{a'} Q^*(s',a'))\,.$$

In the high state the allowable actions are to either search (s) or wait (w). If we search, the possible next states the environment would put us in are either high or low (l). Evaluating the above assuming an action of searching and summing over all possible resulting states gives the following

$$\begin{aligned} Q^*(h,s) &= \sum_{s' \in \{h,l\}} P^s_{h,s'}(R^s_{h,s'} + \gamma \max_{a'} Q^*(s',a')) \\ &= P^s_{h,h}(R^s_{h,h} + \gamma \max_{a'} Q^*(h,a')) + P^s_{h,l}(R^s_{h,l} + \gamma \max_{a'} Q^*(l,a'))\,. \end{aligned}$$

Using the definition of the various transition probabilities and one step rewards we define $P^s_{h,h} = \alpha$, $P^s_{h,l} = 1-\alpha$, $R^s_{h,h} = R^s$, and $R^s_{h,l} = R^s$ so that the above becomes

$$\begin{aligned} Q^*(h,s) &= \alpha(R^s + \gamma \max_{a'} Q^*(h,a')) + (1-\alpha)(R^s + \gamma \max_{a'} Q^*(l,a')) \\ &= R^s + \alpha\gamma \max\{Q^*(h,s), Q^*(h,w)\} + (1-\alpha)\gamma \max\{Q^*(l,s), Q^*(l,w), Q^*(l,rc)\}\,. \end{aligned}$$

In the same way, we can evaluate the action-value function $Q^*(h,w)$ which is the situation where we begin in a high state and perform the action of waiting (w). In this case we find

$$\begin{aligned} Q^*(h,w) &= P^w_{h,h}(R^w_{h,h} + \gamma \max_{a'}(Q^*(h,a')) + P^w_{h,l}(R^w_{h,l} + \gamma \max_{a'} Q^*(l,a')) \\ &= R^w + \gamma \max\{Q^*(h,s), Q^*(h,w)\}\,. \end{aligned}$$

Since $P^w_{h,h} = 1$ and $P^w_{h,l} = 0$. Continuing the process we have action-value function when we are in a low state and we search, wait, or recharge (rc) given by as the following. For $Q^*(l,s)$ we have the following

$$\begin{aligned} Q^*(l,s) &= P^s_{l,h}(R^s_{h,l} + \gamma \max_{a'}(Q^*(h,a')) + P^s_{l,l}(R^s_{l,l} + \gamma \max_{a'} Q^*(l,a')) \\ &= (1-\beta)(-3 + \gamma \max\{Q^*(h,s), Q^*(h,w)\}) \\ &+ \beta(R^s + \gamma \max\{Q^*(l,s), Q^*(l,w), Q^*(l,rc)\}\,. \end{aligned}$$

For $Q^*(l,w)$ the following

$$\begin{aligned} Q^*(l,w) &= P^w_{l,h}(R^w_{l,h} + \gamma \max_{a'} Q^*(h,a')) + P^w_{l,l}(R^w_{l,l} + \gamma \max_{a'} Q^*(l,a')) \\ &= R^w + \gamma \max\{Q^*(l,s), Q^*(l,w), Q^*(l,rc)\}\,. \end{aligned}$$

Since $P^w_{l,h} = 0$ and $P^w_{l,l} = 1$. Finally, we find that $Q^*(l, rc)$ given by

$$\begin{aligned} Q^*(l, rc) &= P^{rc}_{l,h}(R^{rc}_{l,h} + \gamma \max_{a'} Q^*(h, a')) + P^{rc}_{l,l}(R^{rc}_{l,l} + \gamma \max_{a'} Q^*(l, a')) \\ &= \gamma \max\{Q^*(h, s), Q^*(h, w)\} . \end{aligned}$$

Here for future reference we will list the optimal entire Bellman's system for the recycling robot example. We have that

$$\begin{aligned} Q^*(h, s) &= R^s + \alpha\gamma \max\{Q^*(h, s), Q^*(h, w)\} \\ &+ (1 - \alpha)\gamma \max\{Q^*(l, s), Q^*(l, w), Q^*(l, rc)\} \\ Q^*(h, w) &= R^w + \gamma \max\{Q^*(h, s), Q^*(h, w)\} \\ Q^*(l, s) &= (1 - \beta)(-3 + \gamma \max\{Q^*(h, s), Q^*(h, w)\}) \\ &+ \beta(R^s + \gamma \max\{Q^*(l, s), Q^*(l, w), Q^*(l, rc)\} \\ Q^*(l, w) &= R^w + \gamma \max\{Q^*(l, s), Q^*(l, w), Q^*(l, rc)\} \\ Q^*(l, rc) &= \gamma \max\{Q^*(h, s), Q^*(h, w)\} . \end{aligned}$$

These could be solved by iteration once appropriate constants are specified. See the file `rr_action_bellman.m` for Matlab code to numerically evaluate the optimal Bellman equations for the recycling robot.

Exercise 3.17 (evaluation of the gridworld optimal state-value function):

By definition, $V^*(s)$ is given by

$$\begin{aligned} V^*(s) &= \max_{a \in A(s)} E_{\pi^*}\{R_t | s_t = s, a_t = a\} \\ &= \max_{a \in A(s)} E_{\pi^*}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} . \end{aligned}$$

We begin by assuming that the optimal policy for this gridworld example is given in Figure 3.8c. In addition, the rewards structure for this problem is known. Specifically, it is minus one if we step off the gridworld, $+10$ at the position $(1, 2)$ (in matrix notation), a $+5$ at the position $(1, 4)$ and zero otherwise. Using this information we can explicitly compute the right hand side of the above expression for $V^*(s)$. We have that V^* at the state $(1, 2)$ is given by

$$V^*((1, 2)) = +10 + \gamma \cdot 0 + \gamma^2 \cdot 0 + \gamma^3 \cdot 0 + \gamma^4 \cdot 0 + \gamma^5 \cdot 10 + \cdots$$

In the above formula, the pattern of terms above will continue indefinitely. We now describe how each term comes about so that the pattern is clear. The term $+10$ is the constant reward given by the environment when we are moved from the position $(1, 2)$ to the position $(5, 2)$. The term $\gamma \cdot 0$ comes from the reward given when the optimal step from position $(5, 2)$ to $(4, 2)$ is taken. The term $\gamma^2 \cdot 0$ comes from the optimal step from the position $(4, 2)$ to $(3, 2)$ with a zero reward. The term $\gamma^3 \cdot 0$ comes from the optimal step from the position $(3, 2)$ to $(2, 2)$ with a zero reward. The term $\gamma^4 \cdot 0$ comes from the optimal step from the position $(2, 2)$ to $(1, 2)$ with a zero reward. This final step places us at the position $(1, 2)$ and lets us receive the next reward of $+10$ and gives the term $+10\gamma^5$.

We can in fact explicitly evaluate the above sum. We find

$$\begin{aligned} V^*((1,2)) &= 10 + 10\gamma^5 + 10\gamma^{10} + \cdots \\ &= 10\sum_{k=0}^{\infty}\gamma^{5k} = \frac{10}{1-\gamma^5}\,. \end{aligned}$$

if $\gamma = 0.9$ the above sum to three decimal places is given by 24.419.

Chapter 4 (Dynamic Programming)

Policy Evaluation

See the Matlab code `iter_poly_gw_inplace.m` (for inplace) and `iter_poly_gw_not_inplace.m` (for otherwise) to duplicate the example from this section on policy evaluation for the grid-world example.

Exercise 4.1 (evaluate Q^π for some states and actions):

Using the result derived by considering the backup diagram between the action-value and state-value functions in Exercise 3.13 we recall that our action-value function Q can be written in terms of our state-value function V as

$$Q^\pi(s,a) = \sum_{s'} R^a_{s,s'}P^a_{s,s'} + \gamma\sum_{s'} P^a_{s,s'}V^\pi(s')\,.$$

For first case of interest (denoting the terminal state by T) we have

$$Q^\pi(11,\text{down}) = R^{\text{down}}_{11,T}\cdot 1 + \gamma\cdot 1\cdot V^\pi(T) = 0\,.$$

Since we are assuming that all of our actions are deterministic, the reward for transitioning to the terminal state is zero, and that the state-value function evaluated at the terminal state is zero.

For the second case of interest we have

$$\begin{aligned} Q^\pi(7,\text{down}) &= R^{\text{down}}_{7,11}\cdot 1 + \gamma\cdot 1\cdot V^\pi(11) \\ &= -1 - 14\gamma = -13.6\,. \end{aligned}$$

Here have assumed that movement to any non terminal state gives a reward of -1, $\gamma = 0.9$, and $V^\pi(11) = 14$ (which is the value found from the policy iterations given in Figure 4.2).

Exercise 4.2 (the addition of a new state):

I'll assume that the transitions from the original states being unchanged means that the state-value function values do not change for the original grid locations when this newly added state is appended. Then we can iterate Equation 4.5 to obtain the state value function at this new state i.e. $V^\pi(15)$. We find that

$$\begin{aligned} V^\pi(15) &= \sum_a \pi(15,a) \sum_{s'} P^a_{15,s'}(R^a_{15,s'} + \gamma V(s')) \\ &= \frac{1}{4}(R^{\text{left}}_{15,12} + \gamma V(12) + R^{\text{up}}_{15,13} + \gamma V(13) \\ &+ R^{\text{right}}_{15,14} + \gamma V(14) + R^{\text{down}}_{15,15} + \gamma V(15)) \,. \end{aligned}$$

Putting in what we know for the values of $V^\pi(\cdot)$ as given by the numbers in Figure 4.2 from the book we see that $V^\pi(15)$ should equal

$$\begin{aligned} V^\pi(15) &= \frac{1}{4}(-1 - 22\gamma - 1 - 20\gamma - 1 - \gamma 14 - 1 + \gamma V(15)) \\ &= -1 - 14\gamma + \frac{\gamma}{4} V(15) \,. \end{aligned}$$

When we solve the above for $V(15)$ we get $V(15) = -\frac{4(1+14\gamma)}{4-\gamma}$. If we take $\gamma = 0.9$ then we find $V(15) = -17.548$.

If the action from state 13 now takes the agent to state 15 the above analysis can be perform for both the values $V^\pi(15)$ and now $V^\pi(13)$. This would give *two* equations derived like the above containing and these two unknowns. These two equations would then be solved for the two values of V. We now discuss this implementation. The "steady state" equation that V^π must satisfy is given by

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{s,s'}(R^a_{s,s'} + \gamma V^\pi(s')) \,.$$

With a uniform probability policy $\pi(s,a) = \frac{1}{4}$ we have that the above is given by

$$V^\pi(s) = \frac{1}{4} \sum_a \sum_{s'} P^a_{s,s'}(R^a_{s,s'} + \gamma V^\pi(s')) \,,$$

and since the steps are deterministic

$$P^a_{s,s'} = \begin{cases} 1 & \text{if} s' = a \\ 0 & \text{if} s' \neq a \end{cases} ,$$

i.e. we end up at the state our action desires, our state value function V^π is given by

$$\begin{aligned} V^\pi(s) &= \frac{1}{4}(R^{\text{up}}_{s,\text{up}} + \gamma V^\pi(\text{up}) + R^{\text{down}}_{s,\text{down}} + \gamma V^\pi(\text{down}) \\ &+ R^{\text{right}}_{s,\text{right}} + \gamma V^\pi(\text{right}) + R^{\text{left}}_{s,\text{left}} + \gamma V^\pi(\text{left})) \,. \end{aligned}$$

In the above, the notation $R^{\text{up}}_{s,\text{up}}$ means that in state s we choose the action "up" which deterministically transfers us to the state one square above that of s. Next, since the rewards

are negative one (unless we enter a terminal state which cannot happen for the states 13 or 15 we are considering) the equation that $V^\pi(13)$ must satisfy is given by

$$\begin{aligned} V^\pi(13) &= \frac{1}{4}(-4 + \gamma V^\pi(9) + \gamma V^\pi(15) + \gamma V^\pi(14) + \gamma V^\pi(12)) \\ &= -1 + \frac{\gamma}{4}(-20 + V^\pi(15) - 14 - 22) \\ &= -1 + \frac{\gamma}{4}(-56 + V^\pi(15))\,. \end{aligned}$$

Also $V^\pi(15)$ satisfies the earlier equation

$$\begin{aligned} V^\pi(15) &= \frac{1}{4}(-4 + \gamma V^\pi(13) + \gamma V^\pi(15) + \gamma V^\pi(14) + \gamma V^\pi(12)) \\ &= -1 + \frac{\gamma}{4}(V^\pi(13) + V^\pi(15) - 14 - 22) \\ &= -1 + \frac{\gamma}{4}(V^\pi(13) + V^\pi(15)) - 9\gamma\,. \end{aligned}$$

Thus the system of equations to be solved for $V^\pi(13)$ and $V^\pi(15)$ is

$$\begin{aligned} V^\pi(13) - \frac{\gamma}{4}V^\pi(15) &= -1 - 14\gamma \\ -\frac{\gamma}{4}V^\pi(13) + \left(1 - \frac{\gamma}{4}\right)V^\pi(15) &= -1 - 9\gamma\,. \end{aligned}$$

Since this is a linear system for $V^\pi(13)$ and $V^\pi(15)$ we can solve it in a number of ways. When we take $\gamma = 0.9$ solving the above system gives

$$V^\pi(13) = -17.3 \quad \text{and} \quad V^\pi(15) = -16.7\,.$$

Out of pure laziness this is solved in the Matlab file `ex_4_2_sys_solv.m`.

Exercise 4.3 (sequential action-value function approximations):

From the Bellman's equation for the action value function $Q^\pi(s,a)$ derived in Exercise 3.8 we have

$$Q^\pi(s,a) = \sum_{s'} R^a_{s,s'}P^a_{s,s'} + \gamma \sum_{s'}\sum_{a'} Q^\pi(s',a')\pi(s',a')P^a_{s,s'}\,.$$

Now an algorithm for *computing* $Q^\pi(s,a)$ can be obtained by using the rule of thumb: "turn equality into assignment". To do this we can let Q_k be the previous estimate of our action value function Q under policy π insert it into the right hand side of the above to obtain a new approximation of Q^π on the left hand side. As an iteration equation this is

$$Q^\pi_{k+1}(s,a) = \sum_{s'} R^a_{s,s'}P^a_{s,s'} + \gamma \sum_{s'}\sum_{a'} Q^\pi_k(s',a')\pi(s',a')P^a_{s,s'}\,.$$

Exercise 4.4 (avoiding infinite iterations with unbounded value functions):

If the value function becomes unbounded for particular states as a practical matter we can implement policy evaluation in such a way that we skip updating states whose value function is more negative than some specified threshold. When this is done our policy evaluation algorithm will simply ignore these states and process the more important ones. If it happens that after policy evaluation *all* values of V are truncated by this threshold this is an indication that our threshold was too small and should be increased.

Policy Iteration

See the Matlab codes `jcr_example.m`, `jcr_policy_evaluation.m`, `jcr_policy_improvement.m`, and `jcr_rhs_state_value_bellman.m` for results that duplicate Figure 4.4 from the book.

Exercise 4.5 (jacks car rental programming):

Please see the Matlab files: `ex_4_5_policy_evaluation.m`, `ex_4_5_policy_improvement.m`, `ex_4_5_rhs_state_value_bellman.m`, and `ex_4_5_Script.m`. The script `ex_4_5_Script.m` is the main driver and produces plots similar (but modified for the specifics of this problem) to those shown in the book for the default jack's car rental problem.

Exercise 4.6 (policy iteration for action value functions Q):

The algorithms for action value functions are the *same* in principle as those for state value functions except that the action value function algorithms are considering a *combined* state action variable (s, a) rather than just a state variable s alone. With this observation, policy iteration for the action-value function Q then becomes

1. Initialize $Q(s, a)$ and $\pi(s)$ arbitrarily for every $s \in S$ and $a \in \mathcal{A}(s)$.
2. *Policy Evaluation*
   - Repeat
     - $\Delta \leftarrow 0$
     - For every (s, a) pair perform
       * $q \leftarrow Q(s, a)$
       * Update $Q(s, a)$ using

$$Q(s,a) \leftarrow \sum_{s'} R^a_{s,s'} P^a_{s,s'} + \gamma \sum_{s'} \sum_{a'} Q^\pi(s', a') \pi(s', a') P^a_{s,s'} .$$

        * $\Delta \leftarrow \max(\Delta, |q - Q(s,a)|)$
    - Until $\Delta < \Theta$ (small positive number)

3. *Policy Improvement:*
    - policy_stable $\leftarrow$ True
    - For every (s,a) pair perform
        - $p \leftarrow \pi(s,a)$
        - $\pi(s,a) \leftarrow \text{ArgMax}_{s,a}(Q^{\pi}(s,a))$
        - if $p \neq \pi(s,a)$ then policy_stable $\leftarrow False$
    - if policy_stable then stop; else go to *Policy Evaluation* step.

Here Θ is a numerical convergence parameter that determines how accurate our ultimate solution is. To see that by using the algorithm we will continue to obtain better policies recall that V^{π} must equal

$$V^{\pi} = \sum_{a} Q^{\pi}(s,a)\pi(s,a)\,.$$

By picking our policy $\pi(s,a)$ to be the one that is largest with respect to Q we will result in a policy π' such that $V^{\pi'}(s) \geq V^{\pi}(s)$ i.e. we have found a better policy. This final update equation of $\pi(s,a0$ is also Equation 5.1 from the book.

Value Iteration

Exercise 4.8 (discussion of the gamblers problem):

The gamblers problem is particularly unstable, but for the configuration specified in the book the policy seems to try to get to 100 in the fewest number of total flips.

Exercise 4.9 (programming the gamblers problem):

Please see the Matlab files: `gam_rhs_state_value_bellman.m`, and `gam_Script.m`. The script `gam_Script.m` is the main driver and produces plots similar (but modified for the specifics of this problem).

Chapter 5 (Monte Carlo Methods)

Monte Carlo Policy Evaluation

See the Matlab code `cmpt_bj_value_fn.m`, `determineReward.m`, `shufflecards.m`, `handValue.m` and `stateFromHand.m` for a set of functions that duplicates the results presented in Figure 5.2.

Exercise 5.1 (observations about the value function for blackjack):

The value function jumps for the last two rows because these rows correspond to a player with a hand that sums to 20 or 21. These hand values are great enough that with a large probability the player will win the game and receive a reward of one. The drop of on the last row on the left is due to the fact that the dealer is showing an ace and that because of this has a finite probability of getting blackjack or a large hand value and consequently winning the game which would yield a result of negative one. The front most values in the case where the player has a usable ace are greater than when the player does not have a usable ace because in the former case the player knows that one ace is held by himself leaving only three aces left and correspondingly lowering the probability that the dealer has one.

Monte Carlo Control

See the Matlab file `mc_es_bj_Script.m` for code that uses Monte Carlo exploring starts to solve the blackjack problem and to reproduce the plots in Figure 5.5 from the book.

On-Policy Monte Carlo Control

See the Matlab file `soft_policy_bj_Script.m` for the computation of the optimal policy for blackjack using soft policy evaluation.

Off-Policy Monte Carlo Control

Exercise 5.4 (programming the racetrack example):

See the Matlab files `ex_5_4_Script.m`, `mk_rt.m`, `gen_rt_episode.m`, `init_unif_policy.m`, `mcEstQ.m`, `rt_pol_mod.m` and `velState2PosActions.m` for code to solve the racetrack example using on-policy Monte Carlo control method.

Exercise 5.6 (a recursive formulation of a weighted average):

Given Eq 5.4 in the book i.e.

$$V_n = \frac{\sum_{k=1}^{n} w_k R_k}{\sum_{k=1}^{n} w_k} ,$$

We can derive recursive formulas for V_n in the following way. We begin by incrementing n by one to get V_{n+1} and releasing the last term w_{n+1} as

$$\begin{aligned} V_{n+1} &= \frac{\sum_{k=1}^{n+1} w_k R_k}{\sum_{k=1}^{n+1} w_k} = \frac{\sum_{k=1}^{n} w_k R_k + w_{n+1} R_{n+1}}{\sum_{k=1}^{n+1} w_k} \\ &= \frac{\sum_{k=1}^{n} w_k}{\sum_{k=1}^{n+1} w_k} \left(\frac{\sum_{k=1}^{n} w_k R_k}{\sum_{k=1}^{n} w_k} \right) + \left(\frac{w_{n+1}}{\sum_{k=1}^{n+1} w_k} \right) R_{n+1} . \end{aligned}$$

Defining the sum of the weights as $W_n = \sum_{k=1}^{n} w_k$ we see that the above is equal to

$$V_{n+1} = \frac{W_n}{W_{n+1}} V_n + \frac{w_{n+1}}{W_{n+1}} R_{n+1} .$$

Writing the first expression $\frac{W_n}{W_{n+1}}$ as $\frac{W_{n+1} - w_{n+1}}{W_{n+1}} = 1 - \frac{w_{n+1}}{W_{n+1}}$ the above becomes

$$\begin{aligned} V_{n+1} &= V_n - \frac{w_{n+1}}{W_{n+1}} V_n + \frac{w_{n+1}}{W_{n+1}} R_{n+1} \\ &= V_n - \frac{w_{n+1}}{W_{n+1}} \left(R_{n+1} - V_n \right) , \end{aligned}$$

as expected.

Exercise 5.7 (recursive formulation of the off-policy control algorithm):

To modify the off-policy control algorithm Figure 5.7 to use incremental weight computation we need to modify the lines that compute $N(s,a)$, $D(s,a)$, and $Q(s,a)$ with the direct algorithm. In the *non-incremental* algorithm these are given by direct update equations

$$\begin{aligned} N(s,a) &\leftarrow N(s,a) + w R_t \\ D(s,a) &\leftarrow D(s,a) + w \\ Q(s,a) &\leftarrow \frac{N(s,a)}{D(s,a)} . \end{aligned}$$

While in the *incremental* weight algorithm these quantities are computed (with $D(s,a)$ initialized to zero) as

$$\begin{aligned} D(s,a) &\leftarrow D(s,a) + w \\ Q(s,a) &\leftarrow Q(s,a) + \frac{w}{D(s,a)} (R_t - Q(s,a)) . \end{aligned}$$

Chapter 6 (Temporal Difference Learning)

Advantages of TD Prediction Methods

See the files `mk_fig_6_6.m` and `mk_arms_error_plt.m` for Matlab code to duplicate the two figures from this section.

Exercise 6.1 (TD learning v.s. MC learning):

I would expect that TD learning would be much better than MC learning in the case where we are moved to a new building since this is just a change in our initial route and some of the state encountered during the general episode will be the same. For example, on our drive home many of the states are the *same* once we enter the highway and the value function estimates for these states obtained when we worked in the original building should be very close to what we will compute starting from our new building. Staring with a very good initial guess should result in faster convergence.

I would expect this same phenomena to happen in our *original* learning task if our initial guess at the value function is very close to that of the true value function.

Exercise 6.2 (discussions on TD learning on a random walk):

Since this problem in undiscounted, we have $\lambda = 1$, and taking $\alpha = 0.1$ for the TD(0) update expression we obtain the following

$$V(s_t) \leftarrow V(s_t) + 0.1(r_{t+1} + V(s_{t+1}) - V(s_t)) \,.$$

For transitions among states that do not end in one of the terminal states we receive a zero reward, and since initially our value function begins as a constant our first update is given by $V(s_t) \leftarrow V(s_t)$ or no change in the value function. If we terminate on the left our reward is (taking $V(s_{t+1}) = 0$ when s_{t+1} is the left most terminal state) that the state A is updated as

$$V(A) \leftarrow V(A) + 0.1(0 + 0 - V(A)) = 0.9V(A) = 0.45 \,.$$

Which agrees with the plotted value of $V(A)$ for the first iteration. Thus on the first episode the random walk took us off the left end of the domain and we decreased our state value function by 0.05.

Exercise 6.4 (grown of the RMS error):

Large values of α imply that more of $V(s)$ is updated at each timestep. This in turn makes the TD(0) algorithm solution depend more heavily on the *specific* returns received at each step

of the specific random walk sequence. What we are seeing is probably the perturbations introduced into V at every step due to the randomness of the specific step taken in the random walk. At smaller values of α learning takes longer to do but is much less sensitive to any specific/individual random step.

Optimality of TD(0)

See the files `mk_batch_arms_error_plt.m`, `cmpt_arms_err.m`, `eg_6_2_learn.m`, and `eg_rw_batch_learn.m` for Matlab code to duplicate the two figure from this section.

Sarsa: On-Policy TD Control

See the code `run_all_gw_Script.m` to run all of the gridworld experiments (including the exercises below). Specifically, to duplicate the results from this section run the code `windy_gw_Script.m`, which calls `windy_gw.m`. All gridworld policies are plotted using the function `plot_gw_policy.m`.

Exercise 6.6 (windy gridworld with king's moves):

See the code `wgw_w_kings_Script.m` and `wgw_w_kings.m` to run the version of the gridworld experiments where our agent can take king's moves. To see the results of a ninth action (wind movement only) see the code `wgw_w_kings_n_wind_Script.m` and `wgw_w_kings_n_wind.m`.

Exercise 6.7 (stochastic wind)

See the code `wgw_w_stoch_wind_Script.m` and `wgw_w_stoch_wind.m` to run the gridworld experiments where our wind can be stochastic.

Q-learning: Off-Policy TD Control

See the Matlab code `learn_cw_Script.m`, `learn_cw.m`, and `plot_cw_policy.m` to perform the cliff walk experiments from this section.

Exercise 6.9 (Q-learning is an off-policy control method):

Q-learning is an off-policy control method because the action selection is taken with respect to an ϵ-greedy algorithm while the action value function update is determined using a *different* policy i.e. one that is directly greedy with respect to the action value function.

Exercise 6.10 (taking the expectation over actions):

The new method is off-policy. The action selection is ϵ-greedy while the action value update is not it is an expectation. If the learning task is stationary i.e. does not change over time, I would expect this algorithm to perform slightly worse. The trade off is again withholding greedy action selection (which would seem to get the best immediate reward) with exploration (which might give better rewards in the future). Here our action value function update explicitly continues to explore by sampling all action value functions in its neighborhood i.e. the expression

$$\sum_a \pi(s_t, a) Q(s_{t+1}, a) \,,$$

by using an average rather than taking the explicitly greedy selection.

R-Learning for Undiscounted Continual Tasks

See the code `R_learn_acq_Script.m`, and `R_learn_acq.m` to perform the experiments with the access-control queuing task from this section.

Exercise 6.11 (an on-policy method for undiscounted continual tasks)

We can get an on-policy method by not taking the maximum in the algorithm given in this section and replacing that maximum with an appropriate policy specific action selection. This is analogous to the difference between Q-learning and SARSA. For example, the an on-policy algorithm would be

- Initialize ρ and $Q(s, a)$ for all s and a arbitrarily
- Repeat Forever
  - $s \leftarrow$ current state
  - Choose action a from s using behavior policy say an ϵ greedy policy with respect to the current action value function $Q(s, a)$
  - Take action a, observe reward r and new state s'.
  - Choose action a' according to the *same* policy used above i.e. an ϵ greedy policy

- $Q(s,a) \leftarrow Q(s,a) + \alpha[r - \rho + Q(s',a') - Q(s,a)]$
- If $Q(s,a) = \max_a Q(s,a)$ then
  - $\rho \leftarrow \rho + \beta[r - \rho + \max_{a'} Q(s',a') - \max_a Q(s,a)]$

Note that to make the algorithm in the book on-policy we only need to remove the maximum in the Q update rule. This will cause Q to limit to Q^π where π is our policy i.e. say an the ϵ greedy policy. The second two lines (the ones for updating ρ do not change because we want our value of ρ to limit to ρ^π.

Chapter 7 (Eligibility Traces)

n-step TD Prediction

Example 7.1 (n-step TD Methods on the Random Walk):

Here are some notes I created to better understand this example. Under the specific episode described where we starting in state C and take our first step to D we have the one step return update to apply to C of

$$\begin{aligned} R_t^{(1)} &= r_{t+1} + \gamma V_t(s_{t+1}) \\ &= 0 + \gamma V_t(D) = \gamma V_t(D)\,. \end{aligned}$$

From this we would change $V_t(C)$ by

$$\begin{aligned} V_{t+1}(C) &= V_t(C) + \alpha(R_t^{(1)} - V_t(C)) \\ &= V_t(C) + \alpha(\gamma V_t(C) - V_t(C)) = V_t(C)\,, \end{aligned}$$

if we assume that this is an undiscounted problem i.e. $\gamma = 1$. So we see that under a one-step TD method there is no update to state C for this random walk. The same arguments show that $V_{t+1}(D)$ is the same as $V_t(D)$ i.e. there is no update to V for state D under the given episode. For the update to state E on this episode since $R_t^{(1)} = +1$ we have that

$$\begin{aligned} V_{t+1}(E) &= V_t(E) + \alpha(R_t^{(1)} - V_t(E)) \\ &= V_t(E) + \alpha(1 - V_t(E)) \\ &= V_t(E) + \alpha(0.5 + 0.5 - V_t(E)) \\ &= V_t(E) + \alpha(0.5)\,, \end{aligned}$$

Since $V(E)$ was initially set to be 0.5. So we see that the value of V at E is incremented upwards as claimed in the book. In a two-step method, for the state D we have

$$\begin{aligned} R_t^{(2)} &= r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2}) \\ &= 0 + \gamma(+1) + \gamma^2(0) = \gamma\,. \end{aligned}$$

Since in our terminal state we take the state value function to be zero. Thus the two-step TD method would update $V_t(D)$ as

$$\begin{aligned} V_{t+1}(D) &= V_t(D) + \alpha(R_t^{(2)} - V_t(D)) \\ &= V_t(D) + \alpha(\gamma - V_t(D)) \\ &= V_t(D) + \alpha(0.5 + 0.5 - V_t(D)) \\ &= V_t(D) + \alpha(0.5)\,, \end{aligned}$$

or an increment upwards. The increment to $V(E)$ with two-step TD would be the same as with one-step TD so is given by the formula above. Thus with two-step TD we see that in addition to incrementing $V(E)$ we increment $V(D)$ towards one also.

See `rw_online_ntd_learn.m`, `rw_offline_ntd_learn.m`, `rw_episode.m` and the associated driver scripts for code to duplicate the computational experiments for the n-step random walk.

The forward view of TD(λ)

Here I derive the expression presented in the book for the λ-return for an episodic tasks given in this section of the book. We recall that t is a state index for the states $s_0, s_1, s_2, \cdots, s_T$ which precede the returns $r_1, r_2, \cdots, r_{T-1}$. Now in state s_t we will receive rewards

$$r_{t+1}, r_{t+2}, r_{t+3}, \cdots, r_{T-1}\,,$$

before entering terminal state. The total number of rewards received then (starting in state s_t) is given by

$$T - 1 - (t+1) - 1 = T - t - 1\,.$$

Thus the n-step back up reward $R_t^{(n)}$ starting at state s_t can only at most include $T-t-1$ rewards before $R_t^{(n)} = R_t$, i.e. we have that

$$R_t^{(n)} = R_t \quad \text{when} \quad n \geq T - t\,.$$

The defining equation for the lambda return R_t^λ is given by (and can be manipulated as)

$$\begin{aligned} R_t^\lambda &= (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \\ &= (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + (1-\lambda)\sum_{n=T-t}^{\infty} \lambda^{n-1} R_t^{(n)}\,. \end{aligned}$$

where we have broken the sum above at the point where the n-step returns equal the total return. Evaluating the second summation we find

$$\begin{aligned} \sum_{n=T-t}^{\infty} \lambda^{n-1} R_t^{(n)} &= R_t \sum_{n=T-t}^{\infty} \lambda^{n-1} \\ &= R_t \sum_{n=0}^{\infty} \lambda^{n+T-t-1} \end{aligned}$$

$$
\begin{aligned}
&= R_t \lambda^{T-t-1} \sum_{n=0}^{\infty} \lambda^n \\
&= R_t \frac{\lambda^{T-t-1}}{1-\lambda} .
\end{aligned}
$$

Combining this with the above is Equation 7.3 in the book.

Example 7.2 (λ-return on the random walk task):

See `rw_offline_tdl_learn_Script.m` and `rw_offline_tdl_learn.m` for code to duplicate the computational experiments showing performance of off-line $TD(\lambda)$ return algorithm.

Exercise 7.4 (the n-step TD methods half life):

The λ-return is defined as

$$
R_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^n R_t^{(n)} .
$$

Defining τ to be the value of n such that

$$
\lambda^{\tau} \approx \frac{1}{2} \Rightarrow \tau \approx \frac{\ln(1/2)}{\ln(\lambda)} .
$$

Thus given λ, the half life τ is given by this expression. For example if $\lambda = 0.8$, then we compute that $\tau \approx 3$ so we are looking that many steps ahead before our weighting drops by one half.

The backwards view of TD(λ)

See `rw_online_w_et_Script.m`, and `rw_online_w_et.m` for code to solve the random walk task using eligibility traces.

Equivalence of forward and backward views

See `rw_online_tdl_learn_Script.m`, `rw_online_tdl_learn.m` for code to duplicate the computational experiments showing performance of on-line $TD(\lambda)$ return algorithm.

SARSA(λ)

See the code `gw_w_et.m` and `gw_w_et_Script.m` for an implementation of the gridworld example with eligibility traces.

Replacing Traces

See the code `rw_accumulating_vs_replacing_Script.m` and `rw_online_w_replacing_traces.m` for an implementation of the comparison between accumulating v.s. replacing eligibility traces from this section.

Exercise 7.7 (learning a one-directional Markov chain):

See the code `eg_7_5_episode.m`, `eg_7_5_learn_at.m`, `eg_7_5_learn_rt.m`, and `eg_7_5_Script.m` for code that compares accumulating and replacing eligibility traces on this Markov chain problem.

Implementation issues

Example 7.9 (thresholding eligibility traces):

Before beginning we not that the *online* $TD(\lambda)$ algorithm is given in the section entitled: The Backwards view of $TD(\lambda)$. The algorithm we desire to create will update $V(S)$ only if in that state S our eligibility trace is greater than ϵ. The modification needed for the online algorithm are given by

- Initialize V arbitrarily and $e(S) = 0$ for all $s \in \mathcal{S}$
- Repeat For Every Episode
  - Initialize s, the state to the start of this episode
  - Repeat for every step in the episode
    - $a \leftarrow$ action given by π for s
    - Take action a observe reward r and next state s'
    - $\delta \leftarrow r + \gamma V(s') - V(s)$
    - $e(s) \leftarrow e(s) + 1$
    - For all s, if $e(s) > \epsilon$
      - $V(s) \leftarrow V(s) + \alpha\delta e(s)$
      - $e(s) \leftarrow \gamma\lambda e(s)$
  - $s \leftarrow s'$
  - Until s is the terminal state

Here we will update $V(s)$, if our eligibility trace in that state is greater than ϵ. A possible modification to this algorithm would be to modify the "if $e(s) > \epsilon$" statement above to instead be "if $\alpha\delta e(s) > \epsilon$" since that is the actual expression used to update $V(\cdot)$.

Chapter 8 (Generalization and Function Approximation)

Linear Methods

See the code `linAppFn.m`, `targetF.m`, `stp_fn_approx_Script.m` for a duplication of the figure from this section of the book i.e. the reconstructed step function.

Example 8.7 (the placement of tiles for proper generalization):

We would want to run the tilings in a direction perpendicular to the direction where one expects tiling to help with generalization. Thus if we expect that x_1, to be the dimension where the value function depends then we run our tilings *perpendicular* to this dimension i.e. parallel to the dimension x_2. This is similar to the *stripes* example in the figure in this section.

Control with Function Approximation

See the codes `GetTiles_Mex.C`, `GetTiles_Mex_Script.m`, `tiles.C`, `tiles.h`, `mnt_car_learn.m`, `get_ctg.m`, `ret_q_in_st.m`, `next_state.m`, `mnt_car_learn_Script.m`, and `do_mnt_car_Exps.m` for some code to solve the mountain car example that is presented in this section.

Chapter 9 (Planning and Learning)

Integrated Planning, Acting, and Learning

Example 9.1 (Dyna Maze):

For code to perform the DynaMaze experiments from this section see the Matlab files `mk_ex_9_1_mz.m`, `dynaQ_maze.m`, `plot_mz_policy.m`, `dynaQ_maze_Script.m`, and `do_ex_9_1_exps.m`.

Exercise 9.1 (eligibility traces v.s. planning):

An eligibility trace would update the action value function Q along the specific trace taken in state space, while the addition of the planning steps in the dynaQ algorithm can update

a great number of states, specifically ones that are not only on the last pass through state space. In general this is many more than the direct eligibility traces could modify.

When the Model is Wrong

Example 9.2 (The blocking maze):

For code to perform the blocking maze experiment from this section see the Matlab files `mk_ex_9_2_mz.m`, `dynaQplus_maze.m`, `dynaQplus_maze_Script.m`, and `do_ex_9_1_exps.m`.

Example 9.3 (The shortcut maze):

For code to perform the blocking maze experiment from this section see the Matlab files `mk_ex_9_3_mz.m`

setting the value of κ in dynaQplus

For appropriate exploration and planning, in dynaQplus, the model estimated reward r is taken to be $r+\kappa\sqrt{n}$ where n is the number of timesteps since this particular action was tried in this state. Since this additional factor is used to modify Q for it to make a difference, we must pick κ such that $\kappa\sqrt{n}$ is on the order of r. If we have $r \in [-1, 0]$, representing a penalty for each timestep during which we have not solved our problem and if $n \in [0, 100]$, where 100 is taken to be the largest number of timesteps that elapse before this state/action pair is visited. Then $\sqrt{n} \in [0, 10]$ and $\kappa \approx 0.1$ to have $\kappa\sqrt{n}$ the same order as r. In the same way we see that if $n \in [0, N_{\max}]$ then we should take

$$\kappa = O\left(\frac{1}{\sqrt{N_{\max}}}\right) ,$$

here $N_{\max}$ is the maximum number of timesteps that could possibly occur before we updated this state/action pair. Since we may not know this number beforehand, to simplify things, we can take this to be the maximum number of timesteps that we use in training our algorithm with.

Exercise 9.4 (a modified dynaQplus):

The suggested algorithm for this example was coded up and can be found as the Matlab function `ex_9_4_dynaQplus.m`. The one drawback that this algorithm has is that by *only* modifying the action selection we are only able to make local exploratory excursions about the given point in state space. The dynaQplus algorithm, on the other hand, by modifying

the action value function Q *many times* during planning is able to modify the action values to a point where (hopefully) long range exploration of state space can take place. The hopefully statement above is because there is a delicate balance between the number of planning steps, this size of κ, and the size of the state space to achieve a given amount of exploration. If we don't have enough planning steps (or κ is too small, or the state space is too large) we won't be able to fully explore and find better policies through the states space. This means that on tasks where the environment becomes *better* this suggested algorithm should have trouble "finding" this better solution, at least in comparison with other algorithms such as dynaQplus. In the shortcut task from the book we would expect this algorithm to not work as well as dynaQplus. In Figure 1 we see the found policies for both this algorithm and the dynaQplus algorithm. We see that while this algorithm finds a successful policy initially, it is not able to correct this decision when the environment changes. This is similar to the process of finding a local minimum when one desires to find a global minimum, we are not able to reinitialize the search for the better minimum. Both plots were generated with 5000 timesteps and five planning steps. See the Matlab script `ex_9_4_dynaQplus_Script` for the code to run this example and to reproduce these plots.

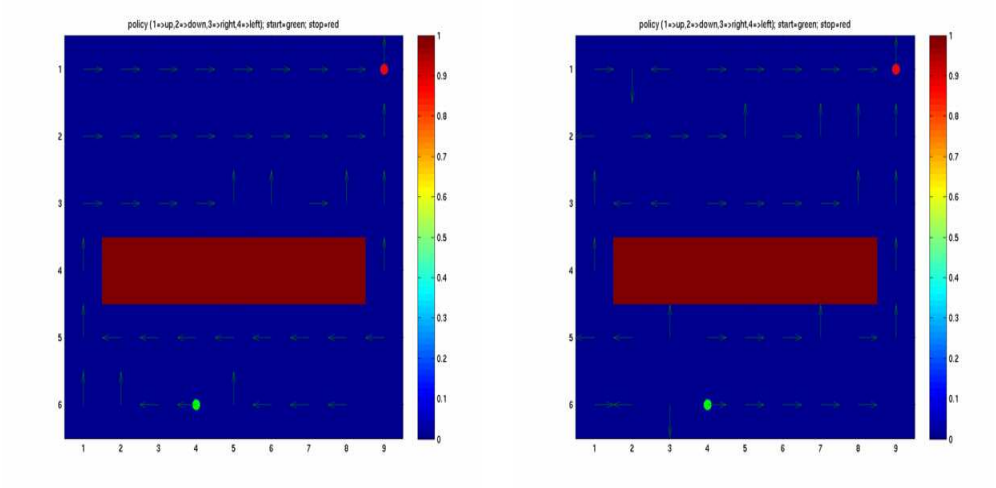

Figure 1: The suggested algorithm (left) and the dynaQplus (right) algorithm after 5000 timesteps. We see that the suggested algorithm is not able to find the newly opened (and better) path. The dynaQplus algorithm is able to.