A Tutorial for Reinforcement Learning

Abhijit Gosavi
Department of Engineering Management and Systems Engineering
Missouri University of Science and Technology
219 Engineering Management, Rolla, MO 65409
Email:gosavia@mst.edu

November 22, 2014

If you find this tutorial useful, or the codes in C at

`http://web.mst.edu/~gosavia/bookcodes.html`

useful, or the MATLAB codes at

`http://web.mst.edu/~gosavia/mrrl_website.html`

useful, please do cite my book (for which this material was prepared), now in its second edition.

A. Gosavi. *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning*, Springer, New York, NY, Second edition, 2014.

Book website: http://web.mst.edu/~gosavia/book.html

Contents

1 Introduction **3**

2 MDPs and SMDPs **3**

3 RL **6**

3.1 Average reward . . . 7

3.2 Selecting the learning rate or step size . . . 10

3.3 Discounted reward . . . 11

3.4 Codes . . . 12

4 Example **12**

5 Conclusions **12**

1 Introduction

The tutorial is written for those who would like an introduction to reinforcement learning (RL). The aim is to provide an intuitive presentation of the ideas rather than concentrate on the deeper mathematics underlying the topic.

RL is generally used to solve the so-called Markov decision problem (MDP). In other words, the problem that you are attempting to solve with RL should be an MDP or its variant. The theory of RL relies on dynamic programming (DP) and artificial intelligence (AI). We will begin with a quick description of MDPs. We will discuss what we mean by "complex" and "large-scale" MDPs. Then we will explain why RL is needed to solve complex and large-scale MDPs. The semi-Markov decision problem (SMDP) will also be covered.

The tutorial is meant to serve as an *introduction* to these topics and is based mostly on the book: "Simulation-based optimization: Parametric Optimization techniques and reinforcement learning" [3]. The book discusses this topic in greater detail in the context of simulators. There are at least two other textbooks that I would recommend you to read: (i) Neuro-dynamic programming [1] (lots of details on convergence analysis) and (ii) Reinforcement Learning: An Introduction [8] (lots of details on underlying AI concepts). A more recent tutorial on this topic is [7]. This tutorial has 2 sections:

- Section 2 discusses MDPs and SMDPs.
- Section 3 discusses RL.

By the end of this tutorial, you should be able to

- Identify problem structures that can be set up as MDPs / SMDPs.
- Use some RL algorithms.

We will not discuss how to use function approximation, but will provide some general advice towards the end.

2 MDPs and SMDPs

The framework of the MDP has the following elements: (1) state of the system, (2) actions, (3) transition probabilities, (4) transition rewards, (5) a policy, and (6) a performance metric. We assume that the system is modeled by a so-called abstract stochastic process called the Markov chain. When we observe the system, we observe its Markov chain, which is defined by the states. We explain these ideas in more detail below.

State: The "state" of a system is a parameter or a set of parameters that can be used to describe a system. For example the geographical coordinates of a robot can be used to describe its "state." A system whose state changes with time is called a *dynamic* system. Then it is not hard to see why a moving robot produces a dynamic system.

Another example of a dynamic system is the queue that forms in a supermarket in front of the counter. Imagine that the state of the queuing system is defined by the number of

people in the queue. Then, it should be clear that the state fluctuates with time, and then this is dynamic system.

It is to be understood that the transition from one state to another in an MDP is usually a random affair. Consider a queue in which there is one server and one waiting line. In this queue, the state x, defined by the number of people in the queue, transitions to $x + 1$ with some probability and to $x - 1$ with the remaining probability. The former type of transition occurs when a new customer arrives, while the latter event occurs when one customer departs from the system because of service completion.

Actions: Now, usually, the motion of the robot can be controlled, and in fact we are interested in controlling it in an optimal manner. Assume that the robot can move in discrete steps, and that after every step the robot takes, it can go North, go South, go East, or go West. These four options are called *actions* or *controls* allowed for the robot.

For the queuing system discussed above, an action could be as follows: when the number of customers in a line exceeds some prefixed number, (say 10), the remaining customers are diverted to a new counter that is opened. Hence, two actions for this system can be described as: (1) Open a new counter (2) Do not open a new counter.

Transition Probability: Assume that action a is selected in state i. Let the next state be j. Let $p(i, a, j)$ denote the probability of going from state i to state j under the influence of action a in one step. This quantity is also called the transition probability. If an MDP has 3 states and 2 actions, there are 9 transition probabilities per action.

Immediate Rewards: Usually, the system receives an immediate reward (which could be positive or negative) when it transitions from one state to another. This is denoted by $r(i, a, j)$.

Policy: The policy defines the action to be chosen in every state visited by the system. Note that in some states, no actions are to be chosen. States in which decisions are to be made, i.e., actions are to be chosen, are called *decision-making* states. In this tutorial, by states, we will mean decision-making states.

Performance Metric: Associated with any given policy, there exists a so-called performance metric — with which the performance of the policy is judged. Our goal is to select the policy that has the best performance metric. We will first consider the metric called the average reward of a policy. We will later discuss the metric called ~~average~~ discounted reward. We will assume that the system is run for a long time and that we are interested in a metric measured over what is called the infinite time horizon.

Time of transition: We will assume for the MDP that the time of transition is unity (1), which means it is the *same* for every transition. Hence clearly 1 here does not have to mean 1 hour or minute or second. It is some fixed quantity fixed by the analyst. For the SMDP, this quantity is *not* fixed as we will see later.

Let us assume that a policy named π is to be followed. Then $\pi(i)$ will denote the action selected by this policy for state i. Every time there is a jump in the Markov chain (of the system under the policy in consideration), we say a transition (or jump) has occurred. *It is*

important to understand that during a transition we may go from a state to itself!

Let x_s denote the state of the system before the sth transition. Then, the following quantity, in which $x_1 = i$, is called the average reward of the policy π starting at state i.

$$\rho_i = \lim_{k \to \infty} \frac{\mathsf{E}\left[\sum_{s=1}^{k} r(x_s, \pi(x_s), x_{s+1}) | x_1 = i\right]}{k} \quad (1)$$

This average reward essentially denotes the sum of the total immediate rewards earned divided by the number of jumps (transitions), calculated over a very long time horizon (that is k assumes a large value.) In the above, the starting state is i and $\pi(x_s)$ denotes the action in state x_s. Also note that $\mathsf{E}[.]$ denotes the average value of the quantity inside the square brackets.

It is not hard to show that the limit in (1) is such that its value is the *same* for any value of x_1, if the underlying Markov chains in the system satisfy certain conditions (related to the regularity of the Markov chains); in many real-world problems such conditions are often satisfied. Then

$$\rho_i = \rho \text{ for any value of } i.$$

The objective of the average-reward MDP is to find the policy that maximizes the performance metric (average reward) of the policy.

Another popular performance metric, commonly used in the literature, is discounted reward. The following quantity is called the discounted reward of a policy π. Again, let x_s denote the state of the system before the sth jump (transition).

$$\psi_i = \lim_{k \to \infty} \mathsf{E}\left[\sum_{s=1}^{k} \gamma^{s-1} r(x_s, \pi(x_s), x_{s+1}) \bigg| x_1 = i\right], \quad (2)$$

where γ denotes the discount factor; this factor, γ, is less than 1 but greater than 0. Eqn. (2) has a simple interpretation:

$$\psi_i = \mathsf{E}[r(x_1, \pi(x_1), x_2) + \gamma r(x_2, \pi(x_2), x_3) + \gamma^2 r(x_3, \pi(x_3), x_4) + \cdots]$$

The discounted reward essentially measures the present value of the sum of the rewards earned in the future over an infinite time horizon, where γ is used to discount money's value. We should point out that:

$$\gamma = \left(\frac{1}{1+\mu}\right)^1, \quad (3)$$

where μ is the rate of interest; the rate is expressed as a fraction here and not in percent. When $\mu > 0$, we have that $0 < \gamma < 1$. It is worthwhile pointing out that in the MDP, we have $1/(1+\mu)$ raised to the power 1 because in the MDP we assume a fixed rate of discounting and that the time duration of each transition is fixed at 1. This mechanism thus captures within the MDP framework the notion of time value of money.

The objective of the discounted-reward MDP is to find the policy that maximizes the performance metric (discounted reward) of the policy starting from every state.

Note that for average reward problems, the immediate reward in our algorithms can be earned during the entire duration of the transition. However, for the discounted problems, we will assume that the immediate reward is earned immediately after the transition starts.

The MDP can be solved with the classical method of dynamic programming (DP). However, DP needs all the transition probabilities (the $p(i, a, j)$ terms) and the transition rewards (the $r(i, a, j)$ terms) of the MDP.

For Semi-Markov decision problems (SMDPs), an additional parameter of interest is the time spent in each transition. The time spent in transition from state i to state j under the influence of action a is denoted by $t(i, a, j)$. To solve SMDPs via DP one also needs the transition times (the $t(i, a, j)$ terms). For SMDPs, the average reward that we seek to maximize is defined as:

$$\rho_i = \lim_{k\to\infty} \frac{\mathsf{E}\left[\sum_{s=1}^{k} r(x_s, \pi(x_s), x_{s+1})|x_1 = i\right]}{\mathsf{E}\left[\sum_{s=1}^{k} t(x_s, \pi(x_s), x_{s+1})|x_1 = i\right]}. \tag{4}$$

(Technically, lim should be replaced by lim inf everywhere in this tutorial, but we will not worry about such technicalities here.) It can be shown that the quantity has the same limit for any starting state (under certain conditions). A possible unit for average reward here is $ per hour.

For discounted reward, we will, as stated above, assume the immediate reward is earned *immediately* after the transition starts and does not depend on the duration of the transition. Thus, the immediate reward is a lumpsum reward earned at the start of the transition (when the immediate reward is a function of the time interval, see instead the algorithm in [2]). Also, because of the variability in time, we will assume continuously compounded rate of interest. Then, we seek to maximize:

$$\psi_i = \lim_{k\to\infty} \mathsf{E}\left[r(x_1, \pi(x_1), x_2) + \sum_{s=2}^{k} r(x_s, \pi(x_s), x_{s+1}) \int_{\tau_s}^{\tau_{s+1}} e^{-\mu\tau} d\tau \middle| x_1 = i \right],$$

where $e^{-\mu\tau}$ denotes the discounting factor over a period of length τ (under continuous compounding) and τ_s is the time of occurrence of the sth jump (transition). Note that since the immediate reward does not depend on the time τ, it can be taken out of the integral. Essentially, what we have above is the sum of discounted rewards with the discount factor in each transition appropriately calculated using the notion of continuous compounding.

Curses: For systems which have a large number of governing random variables, it is often hard to derive the exact values of the associated transition probabilities. This is called the *curse of modeling*. For large-scale systems with millions of states, it is impractical to store these values. This is called the *curse of dimensionality*.

DP breaks down on problems which suffer from any one of these curses because it needs all these values.

Reinforcement Learning (RL) can generate near-optimal solutions to large and complex MDPs. In other words, RL is able to make inroads into problems which suffer from one or more of these two curses and cannot be solved by DP.

3 RL

We will describe a basic RL algorithm that can be used for average reward SMDPs. Note that if $t(i, a, j) = 1$ for all values of i, j, and a, we have an MDP. Hence our presentation will

be for an SMDP, but it can easily be translated into that of an MDP by setting $t(i, a, j) = 1$ in the steps.

It is also important to understand that the transition probabilities and rewards of the system are not needed if any one of the following is true:

1. we can play around in the real world system choosing actions and observing the rewards
2. if we have a simulator of the system.

The simulator of the system can usually be written on the basis of the knowledge of some other easily accessible parameters. For example, the queue can be simulated with the knowledge of the distribution functions of the inter-arrival time and the service time. Thus the transition probabilities of the system are usually **not** required for writing the simulation program.

Also, it is important to know that the RL algorithm that we will describe below requires the updating of certain quantities (called Q-factors) in its database whenever the system visits a new state.

When the simulator is written in C or in any special package such as ARENA, it is possible to update certain quantities that the algorithm needs whenever a new state is visited.

Usually, the updating that we will need has to be performed immediately after a new state is visited. In the simulator, or in real time, it IS possible to keep track of the state of the system so that when it changes, one can update the relevant quantities.

The key idea in RL is store a so-called Q-factor for each state-action pair in the system. Thus, $Q(i, a)$ will denote the Q-factor for state i and action a. The values of these Q-factors are initialized to suitable numbers in the beginning (e.g., zero or some small number to all the Q-factors). Then the system is simulated (or controlled in real time) using the algorithm. In each state visited, some action is selected and the system is allowed to transition to the next state. The immediate reward (and the transition time) that is generated in the transition is recorded as the feedback. The feedback is used to update the Q-factor for the action selected in the previous state. Roughly speaking if the feedback is good, the Q-factor of that particular action and the state in which the action was selected is increased (rewarded) using the Relaxed-SMART algorithm. If the feedback is poor, the Q-factor is punished by reducing its value.

Then the same reward-punishment policy is carried out in the next state. This is done for a large number of transitions. At the end of this phase, also called the learning phase, the action whose Q-factor has the highest value is declared to be the optimal action for that state. Thus the optimal policy is determined. Note that this strategy does not require the transition probabilities.

3.1 Average reward

We begin with average reward. The algorithm that we will describe is called R-SMART (Relaxed Semi-Markov Average reward Technique). The original version in [5] required that the average reward be initialized to a value close to its optimal value. We present here a modified version from [4] that does not require this. We first present the CF-version of R-SMART, where CF stands for the contraction factor. We will later present another version

of this algorithm.

Steps in CF-version of R-SMART:

The steps in the Learning Phase are given below.

- Step 1: Set the Q-factors to some arbitrary values (e.g. 0), that is:

$$Q(i, a) \leftarrow 0 \text{ for all } i \text{ and } a.$$

Set the iteration count, k, to 1. Let ρ^k denote the average reward in the kth iteration of the algorithm. Set ρ^1 to 0 or any other value. Let the first state be i. Let $\mathcal{A}(i)$ denote the set of actions allowed in state i. Let α^k denote the main learning rate in the kth iteration. Selecting a value for α (or any learning rate/step size) is a topic that we discuss below. Let β^k denote the secondary learning rate. Also set the following two quantities to 0: total_reward and total_time. $ITERMAX$ will denote the number of iterations for which the algorithm is run. It should be set to a large number. Set η, a scaling constant, to a small positive value, which is close to but less than 1, e.g., 0.99.

- Step 2: Determine the action associated to the Q-factor that has the highest value in state i. (Imagine there are two actions in a state i and their values are $Q(i, a) = 19$ and $Q(i, 2) = 45$; the clearly action 2 has the greatest Q-factor.) This is called the **greedy** action. Select the greedy action with probability $(1 - \mathsf{p}(k))$, where for instance $\mathsf{p}(k)$ could be defined as follows:

$$\mathsf{p}(k) = \frac{G_1}{G_2 + k}$$

in which $G_2 > G_1$, and G_1 and G_2 are large positive constants, e.g., 1000 and 2000 respectively. With a probability of $\mathsf{p}(k)$, choose one of the other actions. This can be done easily. The non-greedy actions are called exploratory actions, and selecting an exploratory action is called exploration. It should be clear that our probability of exploration will decay with k, the iteration number, and it should. (For the two-action case, you can generate a random number between 0 and 1. If the number is less than or equal to $(1 - \mathsf{p}^k)$, choose the greedy action; otherwise, choose the other action.)

Let the action selected be denoted by a. If a is a greedy action, set $\phi = 0$ (this means that the action selected is greedy with respect to the Q-factor.) Otherwise, set $\phi = 1$. Simulate action a.

- Step 3: Let the next state be denoted by j. Let $r(i, a, j)$ denote the transition reward and $t(i, a, j)$ denote the transition time. Then update $Q(i, a)$ as follows:

$$Q(i, a) \leftarrow (1 - \alpha^k)Q(i, a) + \alpha^k \left[r(i, a, j) - \rho^k t(i, a, j) + \eta \max_{b \in \mathcal{A}(j)} Q(j, b) \right].$$

In the above, $\max_{b \in \mathcal{A}(j)} Q(j, b)$ is essentially the maximum numeric value of all the Q-factors in state j. The value of α^k should be selected appropriately. See the next subsection for a discussion on learning rate rules.

- Step 4: If $\phi = 1$, that is the action was non-greedy, go to Step 5. Otherwise, update total_reward and total_time as follows.

$$\text{total_reward} \leftarrow \text{total_reward} + r(i, a, j),$$
$$\text{total_time} \leftarrow \text{total_time} + t(i, a, j).$$

  Then update the average reward as:

$$\rho^{k+1} \leftarrow (1 - \beta^k)\rho^k + \beta^k \left[\frac{\text{total_reward}}{\text{total_time}} \right],$$

  where β is chosen using a rule that decays to 0 faster than α. The next subsection discusses how appropriate values for α and β can be chosen.

- Step 5: Increment k by 1. Set: $i \leftarrow j$. If $k < ITERMAX$, return to Step 2. Otherwise, declare the action for which $Q(i, .)$ is maximum to be the optimal action for state i (do this for all values of i, i.e., for all states to generate a policy), and STOP.

Note that in the above, the exploration probability $\mathsf{p}(k)$ gradually decays to 1, and in the limit, the algorithm selects the greedy actions. However, this decay should be gradual. If the decay is very quick (e.g., you use small values for G_1 and G_2), the algorithm will most likely converge to an incorrect solution. In other words, the algorithm should be allowed to explore sufficiently.

Further note: η is a positive scalar selected by the user such that $0 < \eta < 1$. The positive scalar η has to be less than 1; it is the contracting factor (CF) that enables the algorithm to converge gracefully to the optimal solution. Its value should be as close to 1 as possible and should not be changed during the learning. An example could be $\eta = 0.99$. For a sufficiently large value of η, the algorithm is guaranteed to converge to the optimal solution. In practice, $\eta = 1$ may also generate convergent behavior, but there is no known proof of this.

The next phase is called the frozen phase because the Q-factors are *not* updated here. This phase is performed to estimate the average reward of the policy declared by the frozen phase to be the optimal policy. (By optimal, of course, we only mean the best that RL can generate; it may not necessarily be optimal, but hopefully is very similar to the optimal in its performance.) Steps in the Frozen Phase are as follows.

- Step 1: Use the Q-factors learned in the Learning Phase. Set iteration count k to 0. $ITERMAX$ will denote the number of iterations for which the frozen phase is run. It should be set to a large number. Also set the following two quantities to 0: total_reward and total_time.

- Step 2: Select for state i the action which has the maximum Q-factor. Let that action be denoted by u. Simulate action b.

- Step 3: Let the next state be denoted by j. Let $r(i, u, j)$ denote the transition reward and $t(i, u, j)$ denote the transition time.

  Then update total_reward and total_time as follows.

$$\text{total_reward} \leftarrow \text{total_reward} + r(i, u, j),$$
$$\text{total_time} \leftarrow \text{total_time} + t(i, u, j).$$

- Step 4: Increment k by 1. Set: $i \leftarrow j$. If $k < ITERMAX$, return to Step 2. Otherwise, calculate the average reward of the policy learned in the learning phase as follows:

$$\rho = \frac{\text{total_reward}}{\text{total_time}},$$

and STOP.

The value of ρ^k in the learning phase can also be used as an estimate of the actual ρ while the learning is on. But typically a frozen phase is carried out to get a cleaner estimate of the actual average reward of the policy learned.

Steps in the SSP-version of R-SMART:
We now present the SSP-version of the algorithm, where SSP stands for the stochastic shortest-path problem, a problem that we do not consider in this tutorial. The problem we solve is not an SSP; the algorithm will use a connection to the SSP to solve our average reward problem. Thus, this SSP-version is essentially going to solve the average reward problem that we have considered above. The algorithm's steps are identical to those of the CF-version described above with two differences:

- Step 1: There is no need to select any value for η because η is not needed in this algorithm. Further, choose any state in the system to be a distinguished state, and then denote it by i^*.
- The main update in Step 3 will be as follows:

$$Q(i,a) \leftarrow (1-\alpha^k)Q(i,a) + \alpha^k \left[r(i,a,j) - \rho^k t(i,a,j) + I(j \neq i^*) \max_{b \in \mathcal{A}(j)} Q(j,b) \right].$$

  In the above, $I(j \neq i^*)$ is the indicator function whose value can be calculated as follows: $I(j \neq i^*) = 1$ when $j \neq i^*$ and $I(j \neq i^*) = 0$ when $j = i^*$. Note that the indicator function replaces η used in the CF-version.

Note that, in practice, the CF-version of R-SMART often outperforms the SSP-version, although the SSP version can be shown to converge under milder conditions.

3.2 Selecting the learning rate or step size

The learning rate (α or β) should be a positive value typically less than 1. It is also a function of k. We need to ensure some conditions on these rules for RL within simulators. These are described in detail in [1]. Some commonly used examples of step-sizes are:

$$\alpha^k = a/(b+k)$$

where for instance $a = 90$ and $b = 100$. Another rule is the log rule: $log(k)/k$ (make sure k starts at 2; otherwise use $log(k+1)/(k+1)$ for k starting at 1) studied in [6]. The log rule converges to 0 slower than the $a/(b+k)$ rule for many values of a and b.

The rule $1/k$ where $a = 1$ and $b = 0$ may not lead to good behavior [6], although it is still widely used and was also used by this author in some of his own prior work. (We, obviously, do not recommend it any more, especially after the findings in [6] in the context of Q-Learning).

When we have two step sizes like in R-SMART above, we must make sure that β converges to 0 faster. We can then use $\alpha = log(k)/k$ and $\beta = 90/(100+k)$. Ideally, as k tends to ∞, β^k/α^k should tend to 0.

3.3 Discounted reward

For discounted reward, the learning phase is sufficient. The steps we describe are from the famous Q-Learning algorithm of Watkins [11]. They apply to the MDP; we will discuss the SMDP extension later.

- Step 1: Set the Q-factors to some arbitrary values (e.g., 0), that is:

  $$Q(i,a) \leftarrow 0 \text{ for all } i \text{ and } a.$$

  Let $\mathcal{A}(i)$ denote the set of actions allowed in state i. Let α^k denote the main learning rate in the kth iteration. Set $k = 1$. $ITERMAX$ will denote the number of iterations for which the algorithm is run. It should be set to a large number.

- Step 2: Select an action a in state i with probability $1/|\mathcal{A}(i)|$, where $|\mathcal{A}(i)|$ denotes the number of elements in the set $\mathcal{A}(i)$. Simulate action a.

- Step 3: Let the next state be denoted by j. Let $r(i,a,j)$ denote the transition reward and $t(i,a,j)$ denote the transition time. Then update $Q(i,a)$ as follows:

  $$Q(i,a) \leftarrow (1-\alpha^k)Q(i,a) + \alpha^k \left[r(i,a,j) + \gamma \max_{b \in \mathcal{A}(j)} Q(j,b) \right],$$

  where you should compute α^k using one of the rules discussed above. Also, γ denotes the discount factor.

- Step 4: Increment k by 1. Set: $i \leftarrow j$. If $k < ITERMAX$, return to Step 2. Otherwise, for each state i declare the action for which $Q(i,.)$ is maximum to be the optimal action, and STOP.

Note that in the algorithm above, there is no decay of exploration. This is because in Q-Learning, the exploration need not decay. However, in practice, to get the algorithm to converge faster, people do employ decay of exploration. It is only recommended in online applications and not in simulators.

For the SMDP extension, you can use the following update in Step 3 of the above algorithm (see pages 245-246 of [3] for more details):

$$Q(i,a) \leftarrow (1-\alpha^k)Q(i,a) + \alpha^k \left[r(i,a,j) + e^{-\mu t(i,a,j)} \max_{b \in \mathcal{A}(j)} Q(j,b) \right],$$

where the exponential term arises as follows:

$$\gamma^\tau = \left(\frac{1}{1+\mu}\right)^\tau \approx e^{-\mu\tau}.$$

in which τ denotes the time and the discounting mechanism was explained above in Equation (3). Note that in the above approximation to obtain the exponential term, we use the fact that μ is quite small.
Note on the vanishing discount approach: You can actually use a discounted RL algorithm to solve the average reward problem via the vanishing discount approach. In this heuristic approach, one uses a discounted algorithm with γ very close to 1 (for MDPs) and μ very close to 0 (for SMDPs). This can work very well in practice.

3.4 Codes

Some MATLAB codes for these algorithms have been placed on the following website [9]:

`http://web.mst.edu/~gosavia/mrrl_website.html`

Codes in C have been placed at the following website:

`http://web.mst.edu/~gosavia/bookcodes.html`

4 Example

We end with a simple example from Gosavi [3]. Figure 1 shows a simple MDP; the legend in the figure explains the transition rewards and probabilities. This is an average reward problem on which you could use R-SMART. The optimal policy for this MDP is: action 2 in state 1 and action 1 in state 2. The average reward of the optimal policy is 10.56.

5 Conclusions

The tutorial presented above showed you one way to solve an MDP provided you have the simulator of the system or if you can actually experiment in the real-world system. Transition probabilities of the state transitions were not needed in this approach; this is the most attractive feature of this approach.

We did *not* discuss what is to be done for large-scale problems. That is beyond the scope of this tutorial. What was discussed above is called the *lookup-table* approach in which each Q-factor is stored explicitly (separately). For large-scale problems, clearly it is not possible to store the Q-factors explicitly because there is too many of them. Instead one stores a few scalars, *called basis functions*, which on demand can generate the Q-factor for any state-action pair. Function approximation when done improperly can become unstable [3].

We can provide some advice to beginners in this regard. First, attempt to cluster states so that you obtain a manageable state space for which you can actually use a lookup table.

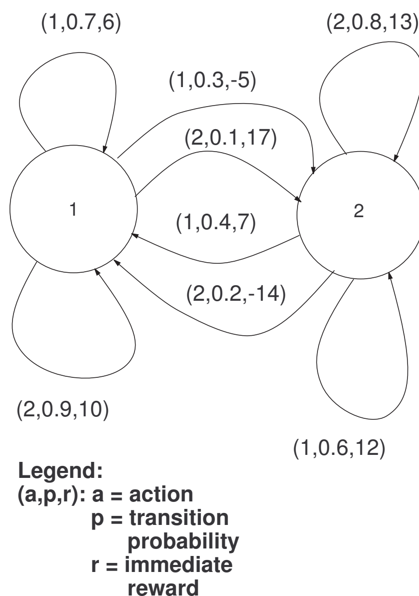

Figure 1: A two state MDP

In other words, divide the state space for each action into grids and use only one Q-factor for all the states in each grid. The total number of grids should typically be a manageable number such as $10,000$. If this does not work well, produce a smaller number of grids but use an incremental Widrow-Hoff algorithm, that is, a *neuron* (see [3]), in each grid. If you prefer using linear regression, go ahead because that will work just as well. If this works well, and you want to see additional improvement, do attempt to use *neural networks*, either neurons or those based on back-propagation [10]. It is with function approximation that you can scale your algorithm up to realistic problem sizes.

I wish you all the best for your new adventures with RL, but cannot promise any help with homework or term papers — sorry :-(

References

[1] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming.* Athena, MA, 1996.

[2] S. J. Bradtke and M. Duff. Reinforcement learning methods for continuous- time Markov decision problems. In *Advances in Neural Information Processing Systems* 7. MIT Press, Cambridge, MA, USA, 1995.

[3] A. Gosavi. *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning*, Springer, New York, NY, 2014. http://web.mst.edu/~gosavia/book.html

[4] A. Gosavi. Target-Sensitive Control of Markov and Semi-Markov Processes. *International Journal of Control, Automation, and Systems*, 9(5):1-11, 2011.

[5] A Gosavi. Reinforcement Learning for Long-run Average Cost. *European Journal of Operational Research.* Vol 155, pp. 654-674, 2004. Can be found at: http://web.mst.edu/~gosavia/rsmart.pdf

[6] A Gosavi. On Step Sizes, Stochastic Shortest Paths, and Survival Probabilities in Reinforcement Learning, *Conference Proceedings of the Winter Simulation Conference*, 2009. Available at: http://web.mst.edu/~gosavia/wsc_2008.pdf.

[7] A Gosavi. Reinforcement Learning: A Tutorial Survey and Recent Advances. *INFORMS Journal on Computing.* Vol 21(2), pp. 178–192, 2009. Available at: http://web.mst.edu/~gosavia/joc.pdf

[8] R. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, MA, USA, 1998.

[9] MATLAB repository for Reinforcement Learning, created by A. Gosavi at Missouri University of Science and Technology, http://web.mst.edu/~gosavia/mrrl_website.html.

[10] P. J. Werbos. *Beyond regression: New tools for prediction and analysis of behavioral sciences.*. Ph.D. thesis, Harvard University, USA, 1974.

[11] C. J. Watkins. *Learning from delayed rewards.* Ph.D. thesis, Kings College, Cambridge, England, May 1989.