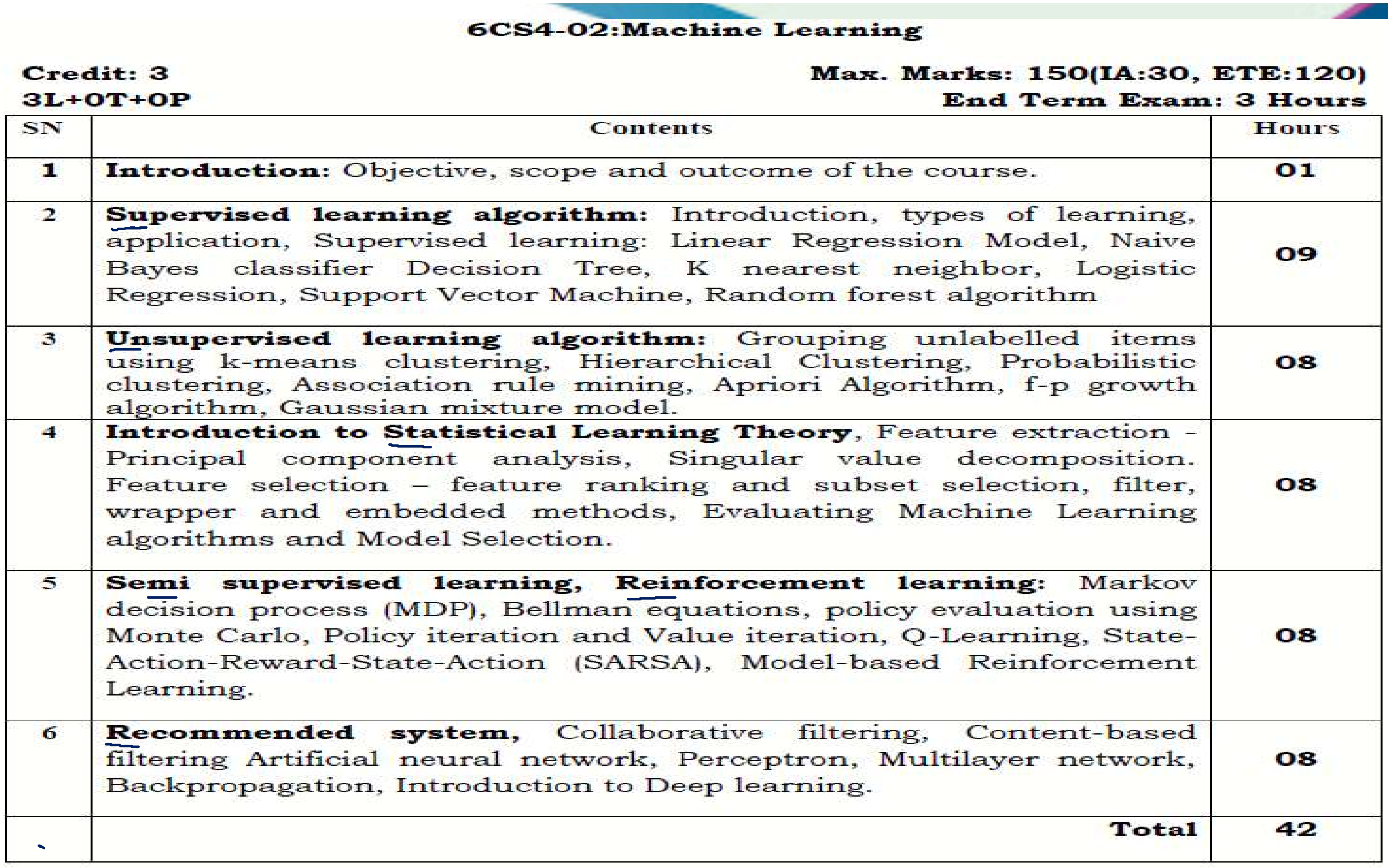


6CS4-02:Machine Learning

Credit: 3 **Max. Marks: 150(IA:30, ETE:120)**
3L+0T+0P **End Term Exam: 3 Hours**

SN	Contents	Hours
1	**Introduction:** Objective, scope and outcome of the course.	**01**
2	**Supervised learning algorithm:** Introduction, types of learning, application, Supervised learning: Linear Regression Model, Naive Bayes classifier Decision Tree, K nearest neighbor, Logistic Regression, Support Vector Machine, Random forest algorithm	**09**
3	**Unsupervised learning algorithm:** Grouping unlabelled items using k-means clustering, Hierarchical Clustering, Probabilistic clustering, Association rule mining, Apriori Algorithm, f-p growth algorithm, Gaussian mixture model.	**08**
4	**Introduction to Statistical Learning Theory**, Feature extraction - Principal component analysis, Singular value decomposition. Feature selection – feature ranking and subset selection, filter, wrapper and embedded methods, Evaluating Machine Learning algorithms and Model Selection.	**08**
5	**Semi supervised learning, Reinforcement learning:** Markov decision process (MDP), Bellman equations, policy evaluation using Monte Carlo, Policy iteration and Value iteration, Q-Learning, State-Action-Reward-State-Action (SARSA), Model-based Reinforcement Learning.	**08**
6	**Recommended system,** Collaborative filtering, Content-based filtering Artificial neural network, Perceptron, Multilayer network, Backpropagation, Introduction to Deep learning.	**08**
	Total	**42**

Machine Learning

Unit-4

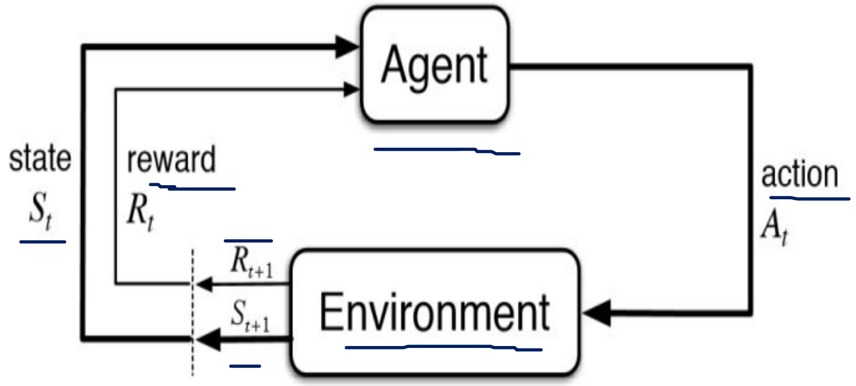
Semi supervised learning and Reinforcement learning

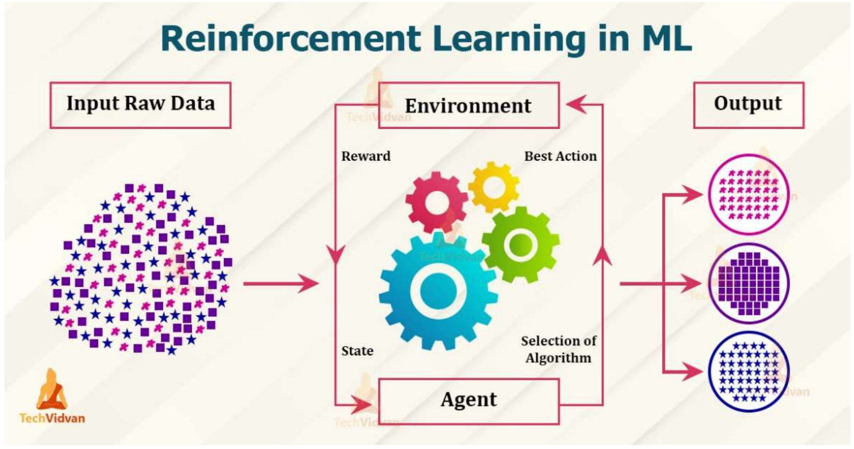
B K BIRLA INSTITUTE OF ENGINEERING & TECHNOLOGY, PILANI

Reinforcement Learning :

Reinforcement Learning is a type of Machine Learning. It allows machines and software agents to automatically determine the ideal behavior within a specific context, in order to maximize its performance. Simple reward feedback is required for the agent to learn its behavior; this is known as the reinforcement signal.

There are many different algorithms that tackle this issue. As a matter of fact, Reinforcement Learning is defined by a specific type of problem, and all its solutions are classed as Reinforcement Learning algorithms. In the problem, an agent is supposed to decide the best action to select based on his current state. When this step is repeated, the problem is known as a Markov Decision Process.

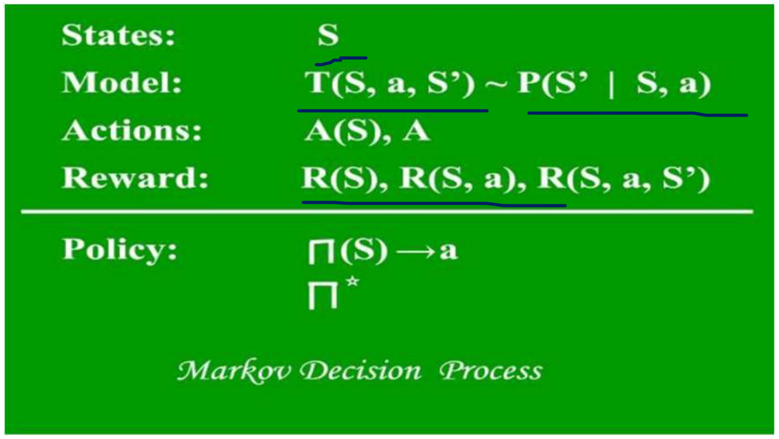




Markov Decision Process:

A Markov Decision Process (MDP) model contains:

- A set of possible world states S.
- A set of Models.
- A set of possible actions A.
- A real valued reward function R(s,a).
- A policy the solution of Markov Decision Process.



States:	S
Model:	$T(S, a, S') \sim P(S' \mid S, a)$
Actions:	$A(S), A$
Reward:	$R(S), R(S, a), R(S, a, S')$
Policy:	$\Pi(S) \rightarrow a$
	Π^*

Markov Decision Process

Markov Decision Process:

What is a State?

A State is a set of tokens that represent every state that the agent can be in.

What is a Model?

A Model (sometimes called Transition Model) gives an action's effect in a state. In particular, T(S, a, S') defines a transition T where being in state S and taking an action 'a' takes us to state S' (S and S' may be same). For stochastic actions (noisy, non-deterministic) we also define a probability P(S'|S,a) which represents the probability of reaching a state S' if action 'a' is taken in state S. Note Markov property states that the effects of an action taken in a state depend only on that state and not on the prior history.

What is Actions?

An Action A is set of all possible actions. A(s) defines the set of actions that can be taken being in state S.

What is a Reward?

A Reward is a real-valued reward function. R(s) indicates the reward for simply being in the state S. R(S,a) indicates the reward for being in a state S and taking an action 'a'. R(S,a,S') indicates the reward for being in a state S, taking an action 'a' and ending up in a state S'.

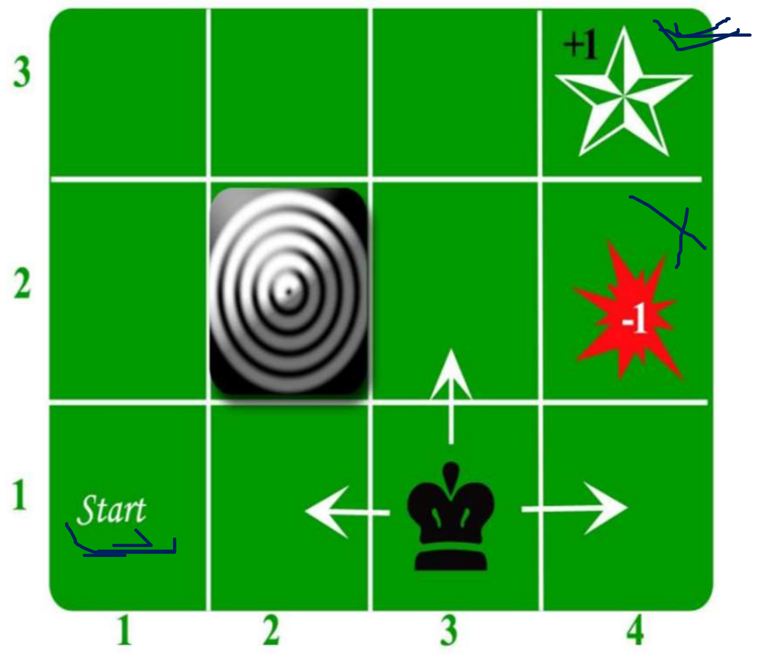
What is a Policy?

A Policy is a solution to the Markov Decision Process. A policy is a mapping from S to a. It indicates the action 'a' to be taken while in state S.

Markov Decision Process:

Let us take the example of a grid world:

An agent lives in the grid. The example is a 3*4 grid. The grid has a START state(grid no 1,1). The purpose of the agent is to wander around the grid to finally reach the Blue Diamond (grid no 4,3). Under all circumstances, the agent should avoid the Fire grid (orange color, grid no 4,2). Also the grid no 2,2 is a blocked grid, it acts like a wall hence the agent cannot enter it.
The agent can take any one of these actions: UP, DOWN, LEFT, RIGHT

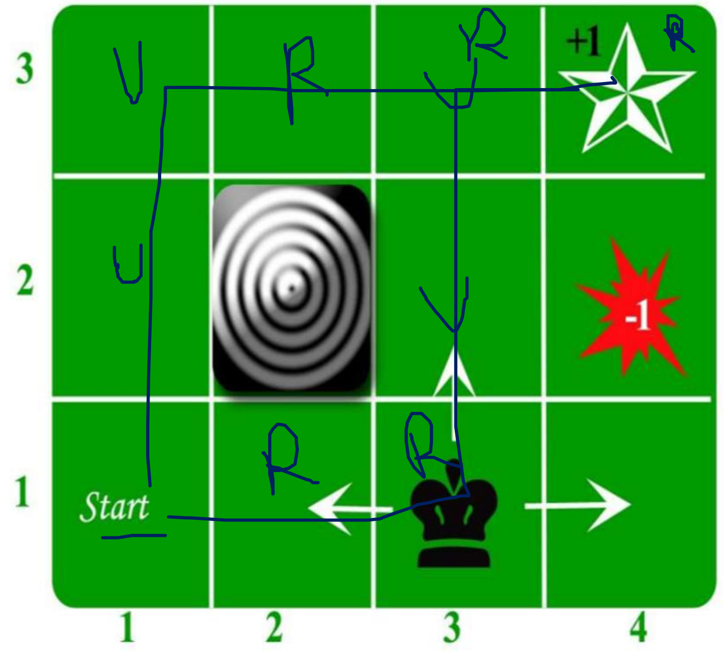


Markov Decision Process:

Walls block the agent path, i.e., if there is a wall in the direction the agent would have taken, the agent stays in the same place. So for example, if the agent says LEFT in the START grid he would stay put in the START grid.

First Aim: To find the shortest sequence getting from START to the Diamond. Two such sequences can be found:

- RIGHT RIGHT UP UP RIGHT
- UP UP RIGHT RIGHT RIGHT



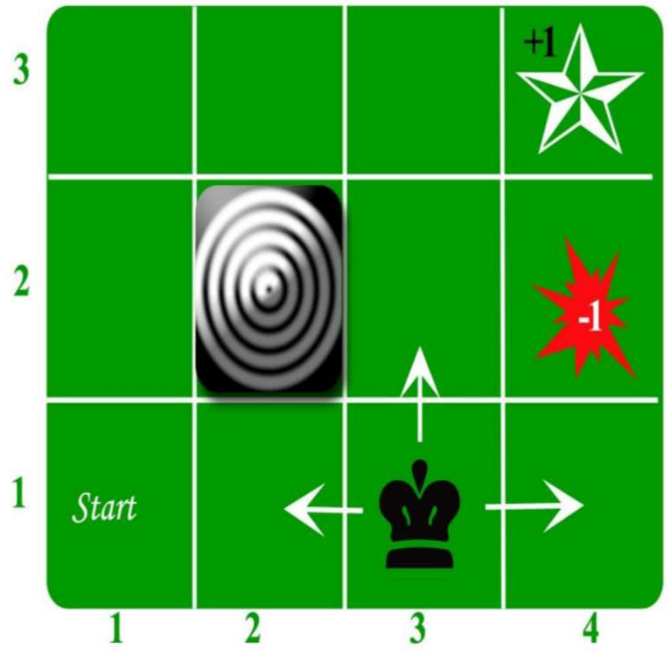
Markov Decision Process:

Let us take the second one (UP UP RIGHT RIGHT RIGHT) for the subsequent discussion. The move is now noisy. 80% of the time the intended action works correctly. 20% of the time the action agent takes causes it to move at right angles. For example, if the agent says UP the probability of going UP is 0.8 whereas the probability of going LEFT is 0.1 and probability of going RIGHT is 0.1 (since LEFT and RIGHT is right angles to UP).

The agent receives rewards each time step:-

 Small reward each step (can be negative when can also be term as punishment, in the above example entering the Fire can have a reward of -1).

- **Big rewards come at the end (good or bad).**
- **The goal is to Maximize sum of rewards.**



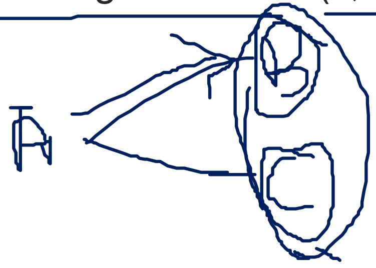
Equation

Bellman Ford ~~Algorithm~~:

Bellman equation is the basic block of solving reinforcement learning and is omnipresent in RL. It helps us to solve MDP. To solve means finding the optimal policy and value functions.
The optimal value function V*(S) is one that yields maximum value.
The value of a given state is equal to the max action (action which maximizes the value) of the reward of the optimal action in the given state and add a discount factor multiplied by the next state's Value from the Bellman Equation.

$$V(s) = \max_a(R(s,a) + \gamma V(s'))$$

Let's understand this equation, V(s) is the value for being in a certain state. V(s') is the value for being in the next state that we will end up in after taking action a. R(s, a) is the reward we get after taking action a in state s.



Bellman Ford Algorithm:

As we can take different actions so we use maximum because our agent wants to be in the optimal state. γ is the discount factor as discussed earlier. This is the bellman equation in the deterministic environment. It will be slightly different for a non-deterministic environment or stochastic environment.

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s') \right)$$

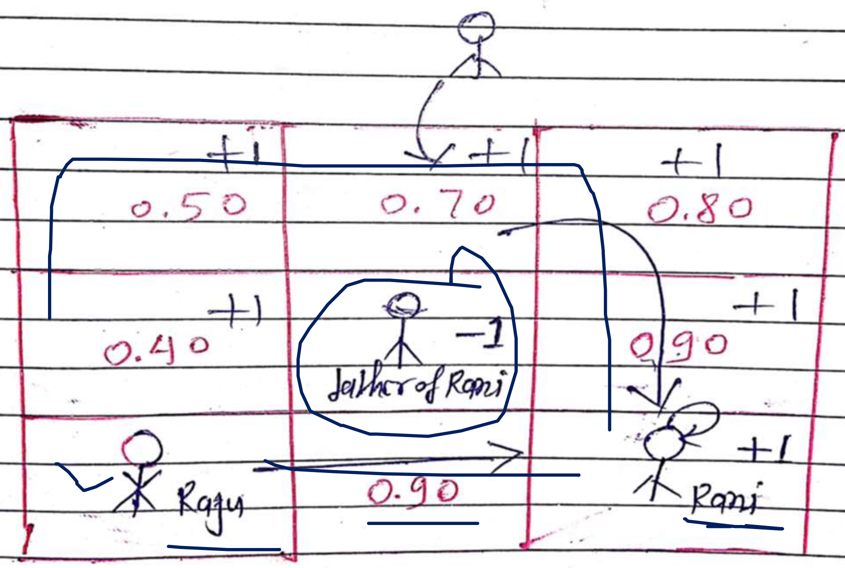
In a stochastic environment when we take an action it is not confirmed that we will end up in a particular next state and there is a probability of ending in a particular state. P(s, a,s') is the probability of ending is state s' from s by taking action a. This is summed up to a total number of future states. For example, if by taking an action we can end up in 3 states s_1,s_2, and s_3 from state s with a probability of 0.2, 0.2 and 0.6. The Bellman equation will be

$V(s) = \max_a(R(s,a) + \gamma(0.2*V(s_1) + 0.2*V(s_2) + 0.6*V(s_3)))$

We can solve the Bellman equation using a special technique called dynamic programming.

Bellman Ford Algorithm:

$$V_\pi(s) = r(s,a) + \gamma \sum V_\pi(s')$$



$$\begin{tikzcd} & & S_3 \\ S \arrow[r] & S \arrow[ur] \arrow[r] & S_2 \end{tikzcd}$$

policy evaluation using Monte Carlo:

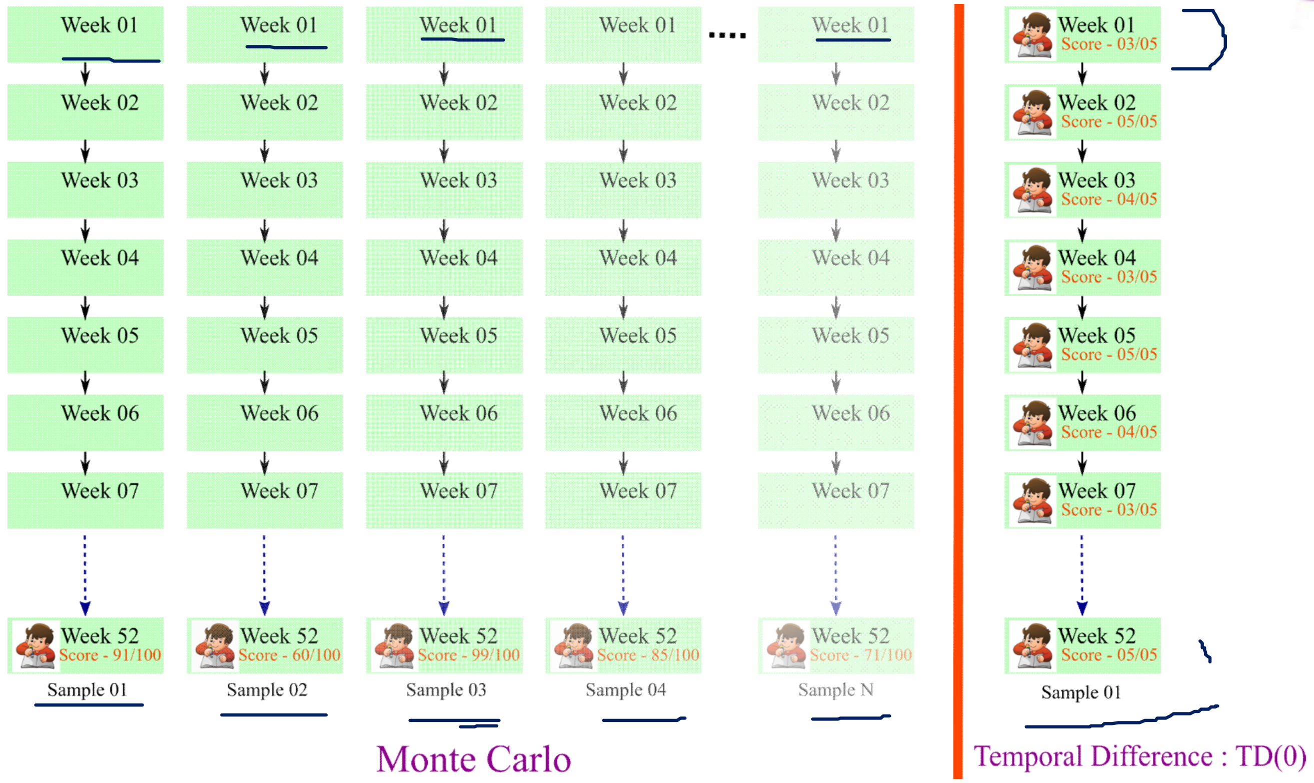
In Dynamic programming we need a model(agent knows the MDP transition and rewards) and agent does planning (once model is available agent need to plan its action in each state). There is no real learning by the agent in Dynamic programming method.
Monte Carlo method on the other hand is a very simple concept where agent learn about the states and reward when it interacts with the environment. In this method agent generate experienced samples and then based on average return, value is calculated for a state or state-action. Below are key characteristics of Monte Carlo (MC) method:

1. There is no model (agent does not know state MDP transitions)
2. agent learns from sampled experience
3. learn state value vπ(s) under policy π by experiencing average return from all sampled episodes (value = average return)
4. only after a complete episode, values are updated (because of this algorithm convergence is slow and update happens after a episode is Complete)
5. There is no bootstrapping
6. Only can be used in episodic problems

policy evaluation using Monte Carlo:

Consider a real life analogy; Monte Carlo learning is like annual examination where student completes its episode at the end of the year. Here, the result of the annual exam is like the return obtained by the student. Now if the goal of the problem is to find how students score during a calendar year (which is a episode here) for a class, we can take sample result of some student and then calculate mean result to find score for a class (don’t take the analogy point by point but on a holistic level I think you can get the essence of MC learning). Similarly we have TD learning or temporal difference learning (TD learning is like updating value in every time step and does not require wait till end of episode to update the values) that we will cover in future blog, can be thought like a weekly or monthly examination (student can adjust their performance based on this score (reward received) after every small interval and final score is accumulation of the all weekly tests (total rewards)).

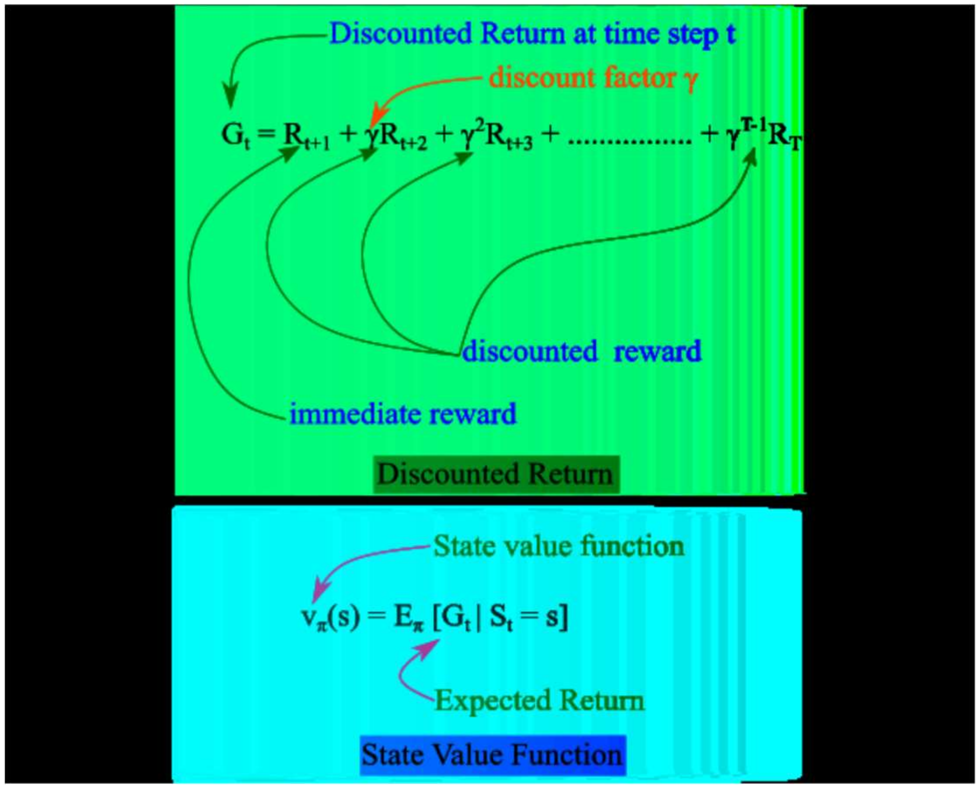
policy evaluation using Monte Carlo:



Value function = Expected Return
Expected return is equal to discounted sum of all rewards.

policy evaluation using Monte Carlo:

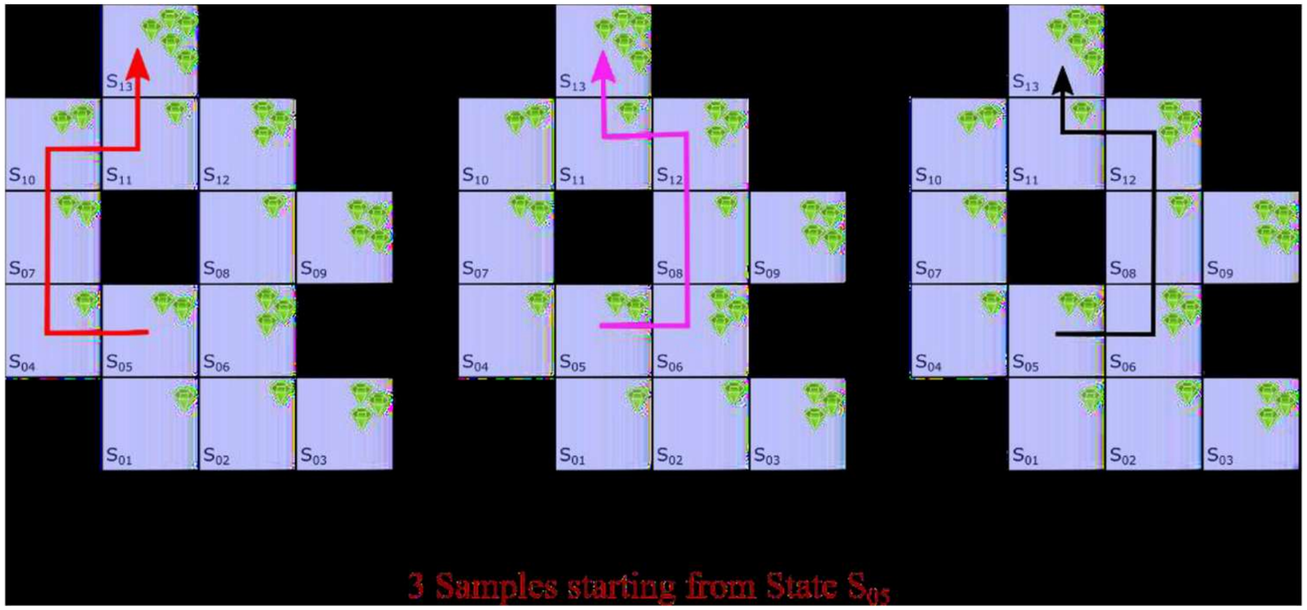
In Monte Carlo Method instead of expected return we use empirical return that agent has sampled based following the policy



Discounted Return at time step t
discount factor γ
$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots\ldots\ldots\ldots + \gamma^{T-1} R_T$
discounted reward
immediate reward
Discounted Return
State value function
$v_\pi(s) = E_\pi [G_t \mid S_t = s]$
Expected Return
State Value Function

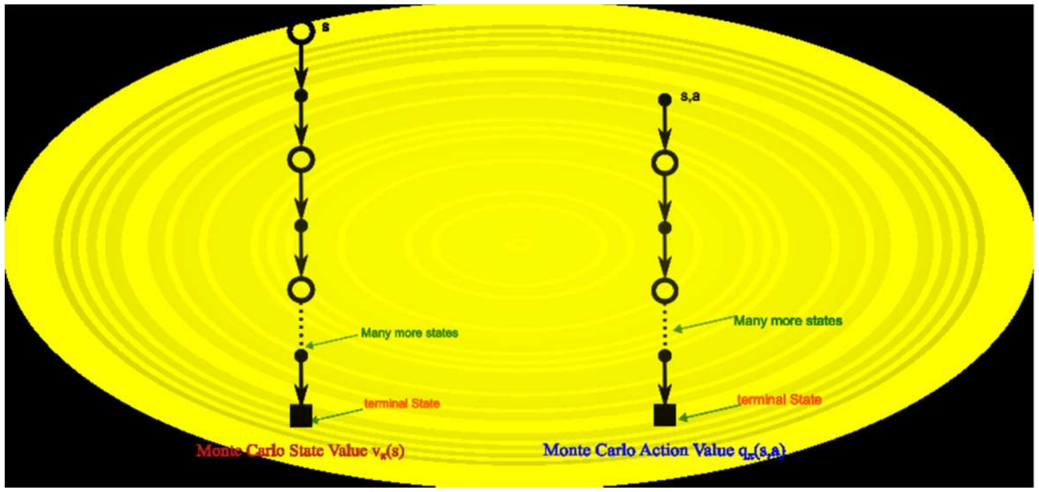
policy evaluation using Monte Carlo:

If we go back to our very first example of gem collection, agent follows policy and complete an episode, along the way in each step it collects rewards in the form of gem. To get state value agent sum-up all the gems collected after each episode starting from that state. Refer to below diagram where 3 samples collected starting from State S 05. Total reward collected (discount factor is considered as 1 for simplicity) in each episode as follows:



policy evaluation using Monte Carlo:

Return(Sample 01) = 2 + 1 + 2 + 2 + 1 + 5 = 13 gems
Return(Sample 02) = 2 + 3 + 1 + 3 + 1 + 5 = 15 gems
Return(Sample 03) = 2 + 3 + 1 + 3 + 1 + 5 = 15 gems
Observed mean return (based on 3 samples) = (13 + 15 + 15)/3 = 14.33 gems
Thus state value as per Monte Carlo Method, v π(S 05) is 14.33 gems based on 3 samples following policy π.
Monte Carlo Backup diagram
Monte Carlo Backup diagram would look like below


s
s,a
Many more states
Many more states
terminal State
terminal State
Monte Carlo State Value $v_\pi(s)$
Monte Carlo Action Value $q_\pi(s,a)$

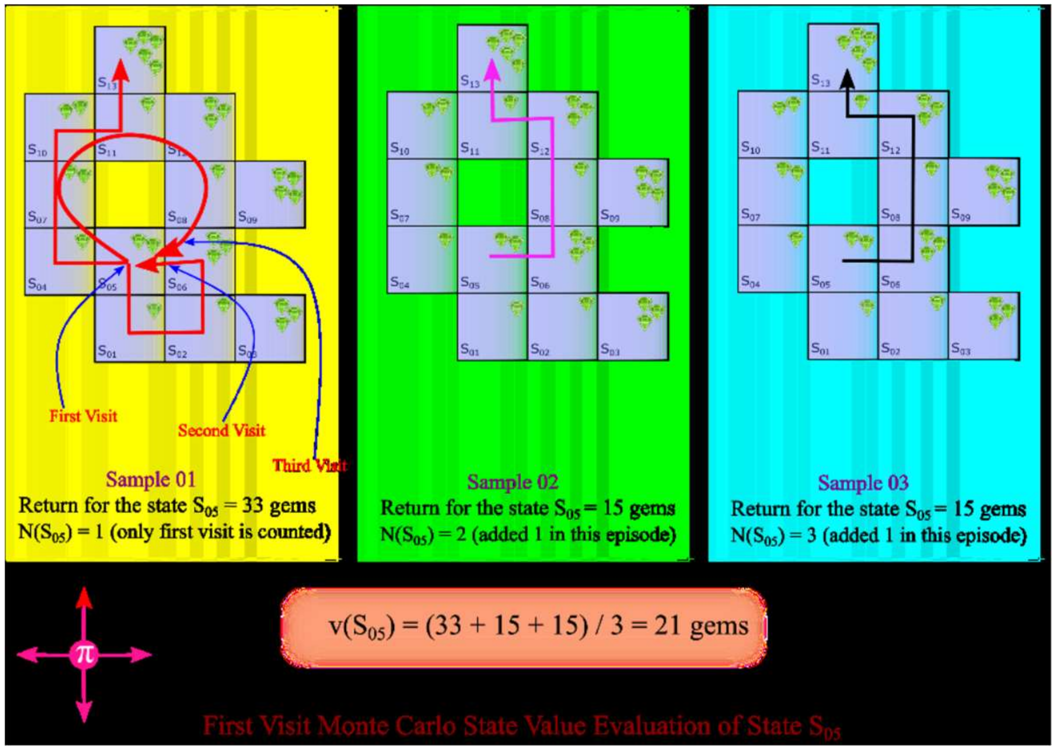
policy evaluation using Monte Carlo:

There are two types of MC learning policy evaluation (prediction) methods:
First Visit Monte Carlo Method
In this case in an episode first visit of the state is counted (even if agent comes-back to the same state multiple time in the episode, only first visit will be counted). Detailed step as below:

1. To evaluate state s, first we set number of visit, N(s) = 0, Total return TR(s) = 0 (these values are updated across episodes)
2. The first time-step t that state s is visited in an episode, increment counter N(s) = N(s) + 1
3. Increment total return TR(s) = TR(s) + Gt
4. Value is estimated by mean return V(s) = TR(s)/N(s)
5. By law of large numbers, V(s) -> vπ(s) (this is called true value under policy π) as N(s) approaches infinity

Refer to below diagram for better understanding of counter increment.

policy evaluation using Monte Carlo:



S13
S10
S11
S12
S07
S08
S09
S04
S05
S06
S01
S02
S03
First Visit
Second Visit
Third Visit
Sample 01
Return for the state S_{05} = 33 gems
$N(S_{05})$ = 1 (only first visit is counted)
Sample 02
Return for the state S_{05} = 15 gems
$N(S_{05})$ = 2 (added 1 in this episode)
Sample 03
Return for the state S_{05} = 15 gems
$N(S_{05})$ = 3 (added 1 in this episode)
π
$v(S_{05}) = (33 + 15 + 15) / 3 = 21$ gems
First Visit Monte Carlo State Value Evaluation of State S_{05}

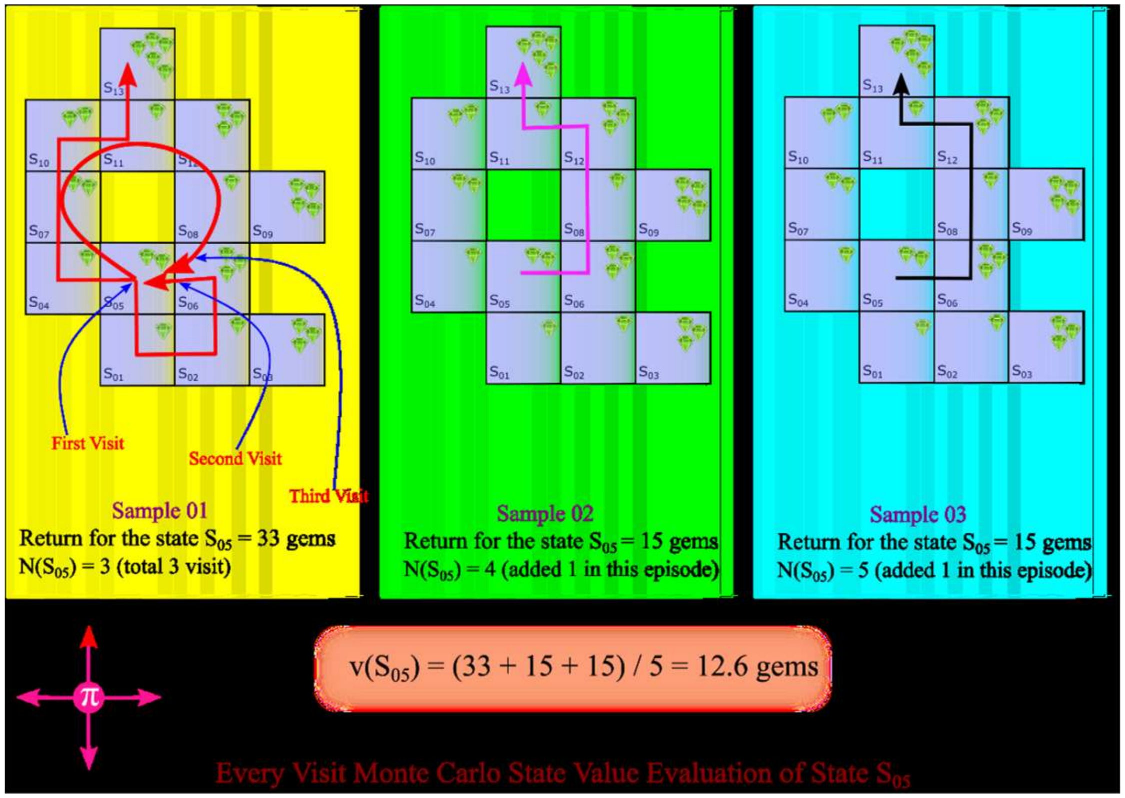
policy evaluation using Monte Carlo:

Every Visit Monte Carlo Method
In this case in an episode every visit of the state is counted. Detailed step as below:

1. To evaluate state s, first we set number of visit, N(s) = 0, Total return TR(s) = 0 (these values are updated across episodes)
2. every time-step t that state s is visited in an episode, increment counter N(s) = N(s) + 1
3. Increment total return TR(s) = TR(s) + Gt
4. Value is estimated by mean return V(s) = TR(s)/N(s)
5. By law of large numbers, V(s) -> vπ(s) (this is called true value under policy π) as N(s) approaches infinity

Refer to below diagram for better understanding of counter increment.

policy evaluation using Monte Carlo:



S_{13}
S_{10}
S_{11}
S_{12}
S_{07}
S_{08}
S_{09}
S_{04}
S_{05}
S_{06}
S_{01}
S_{02}
S_{03}
First Visit
Second Visit
Third Visit
Sample 01
Return for the state S_{05} = 33 gems
$N(S_{05})$ = 3 (total 3 visit)
Sample 02
Return for the state S_{05} = 15 gems
$N(S_{05})$ = 4 (added 1 in this episode)
Sample 03
Return for the state S_{05} = 15 gems
$N(S_{05})$ = 5 (added 1 in this episode)
π
$v(S_{05}) = (33 + 15 + 15) / 5 = 12.6$ gems
Every Visit Monte Carlo State Value Evaluation of State S_{05}

policy evaluation using Monte Carlo:

Usually MC is updated incrementally after every episode (no need to store old episode values, it could be a running mean value for the state updated after every episode).
Update V(s) incrementally after episode S 1, A 2, R 3,….,S T For each state S t with return G t
Usually in place of 1/N(S t) a constant learning rate (α) is used and above equation becomes :
For Policy improvement, Generalized Policy Improvement concept is used to update policy using action value function of Monte Carlo Method.

Monte Carlo Methods have below advantages:

- zero bias
- Good convergence properties (even with function approximation)
- Not very sensitive to initial value
- Very simple to understand and use

But it has below limitations as well:

- MC must wait until end of episode before return is known
- MC has high variance
- MC can only learn from complete sequences
- MC only works for episodic (terminating) environments

Policy Iteration:

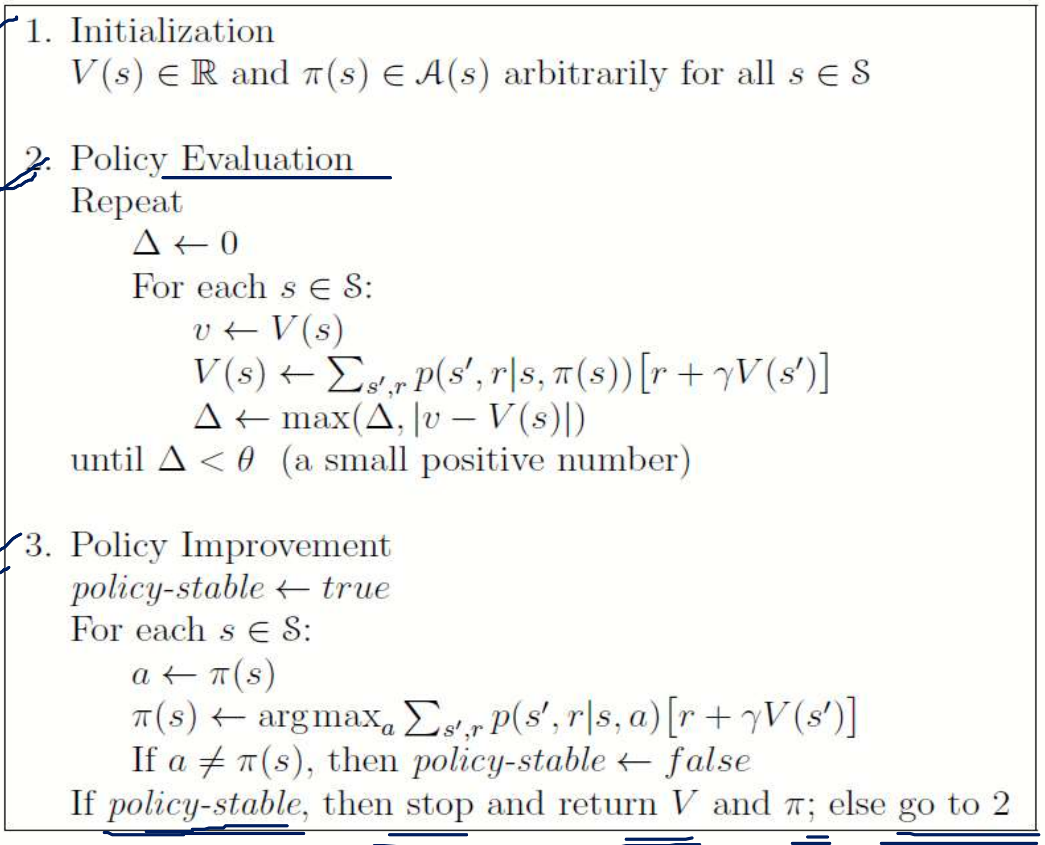
Once a policy, π, has been improved using v_π to yield a better policy, π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*,$$

where $\xrightarrow{E}$ denotes a policy *evaluation* and $\xrightarrow{I}$ denotes a policy *improvement*. Each policy is guaranteed to be a strict improvement over the previous one

$\boxed{0 \; 00}$ 1 → 2 → 3 → 10 [illegible]

Policy Iteration:



```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
       $\Delta \leftarrow 0$
       For each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
       $a \leftarrow \pi(s)$
       $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
       If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V$ and $\pi$; else go to 2
```

Policy Iteration:

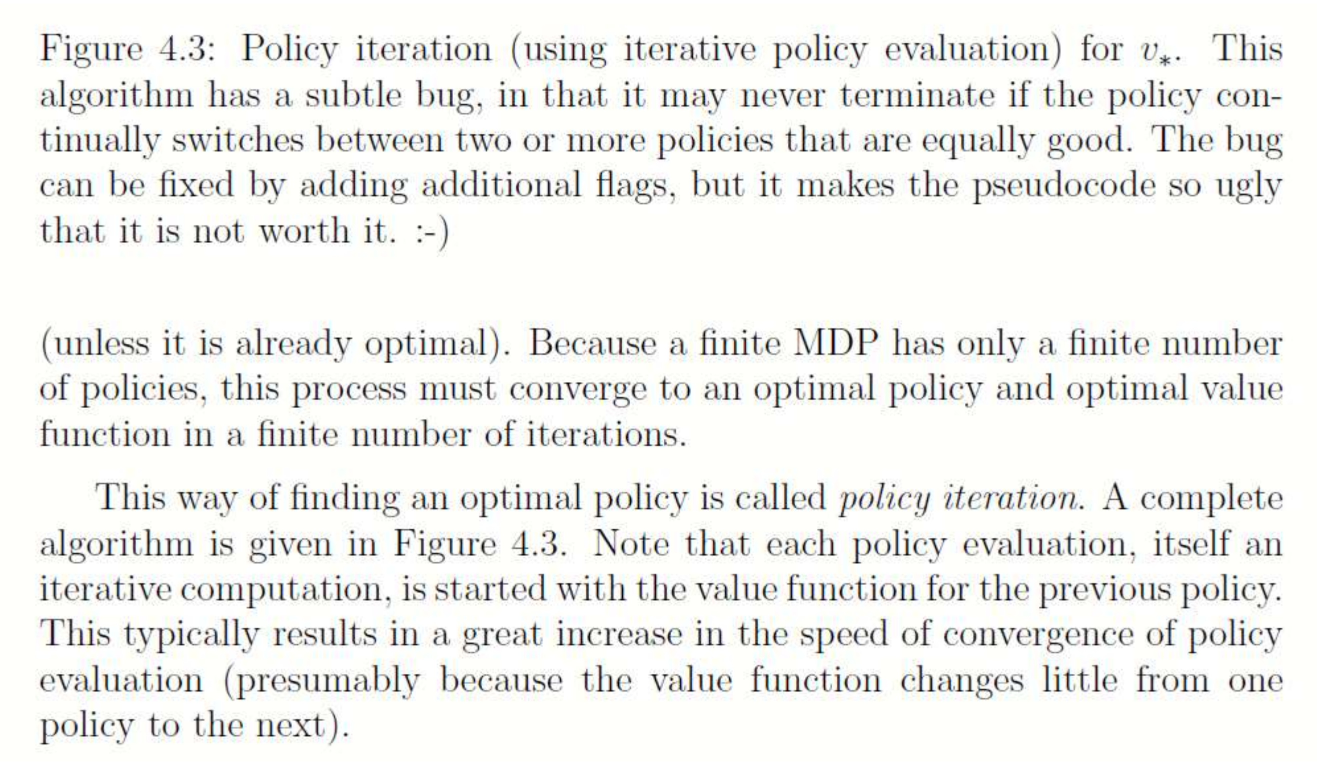
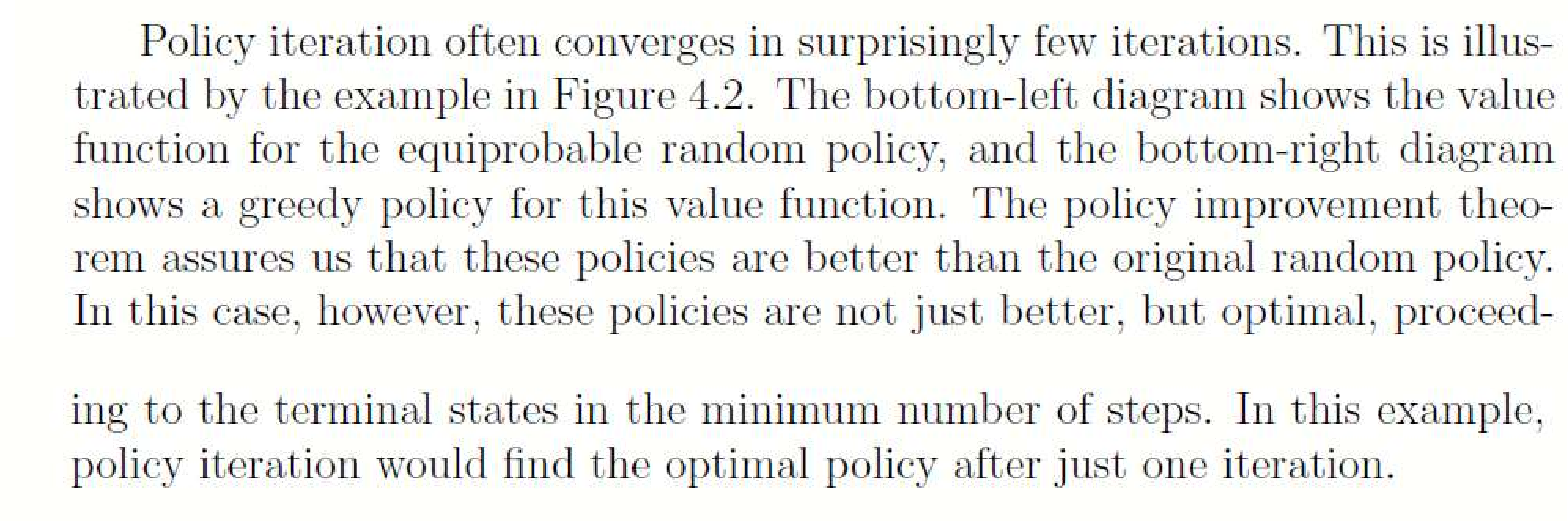


Figure 4.3: Policy iteration (using iterative policy evaluation) for v_*. This algorithm has a subtle bug, in that it may never terminate if the policy continually switches between two or more policies that are equally good. The bug can be fixed by adding additional flags, but it makes the pseudocode so ugly that it is not worth it. :-)

(unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.

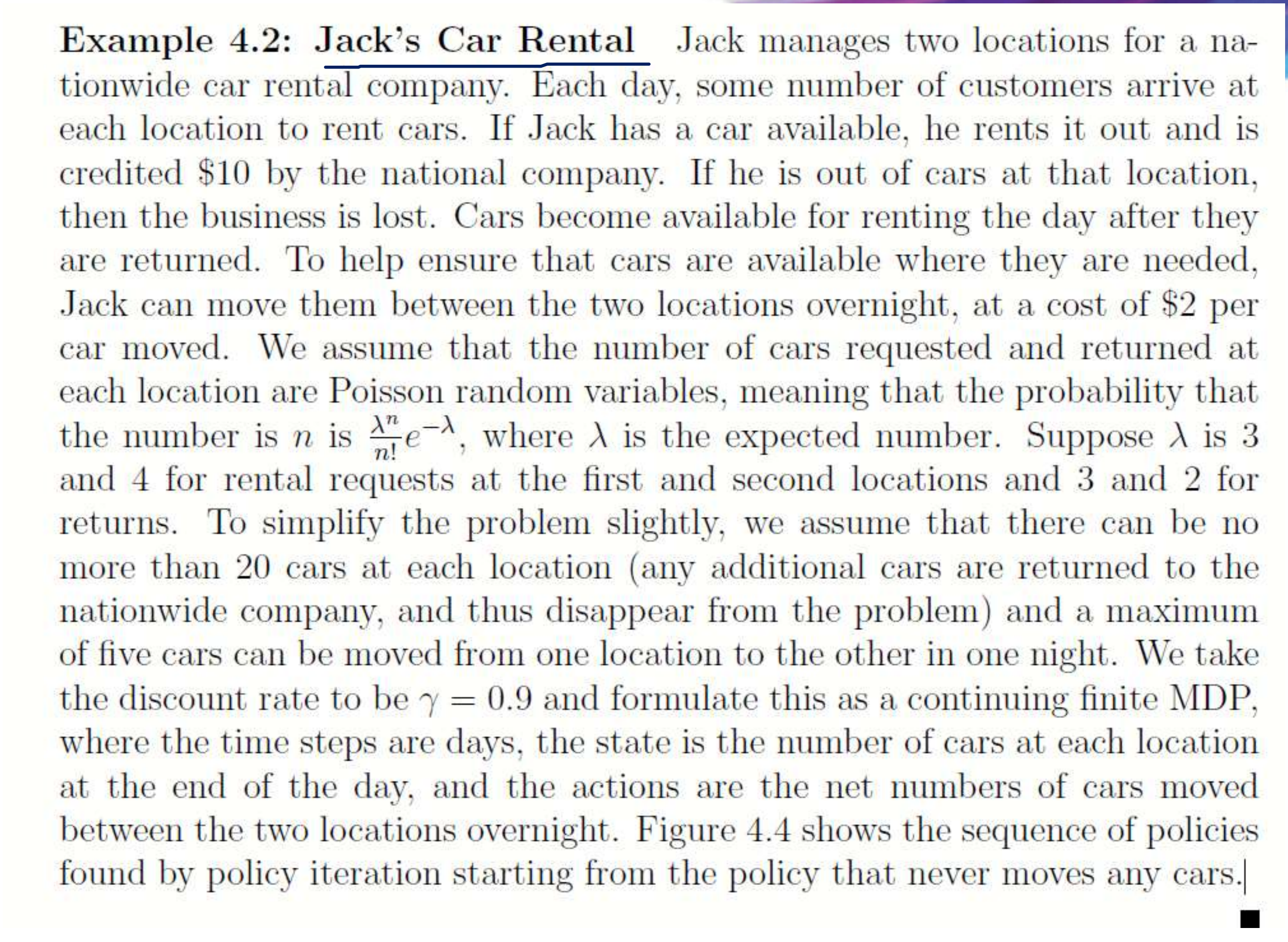
This way of finding an optimal policy is called *policy iteration*. A complete algorithm is given in Figure 4.3. Note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next).

Policy Iteration:



Policy iteration often converges in surprisingly few iterations. This is illustrated by the example in Figure 4.2. The bottom-left diagram shows the value function for the equiprobable random policy, and the bottom-right diagram shows a greedy policy for this value function. The policy improvement theorem assures us that these policies are better than the original random policy. In this case, however, these policies are not just better, but optimal, proceeding to the terminal states in the minimum number of steps. In this example, policy iteration would find the optimal policy after just one iteration.

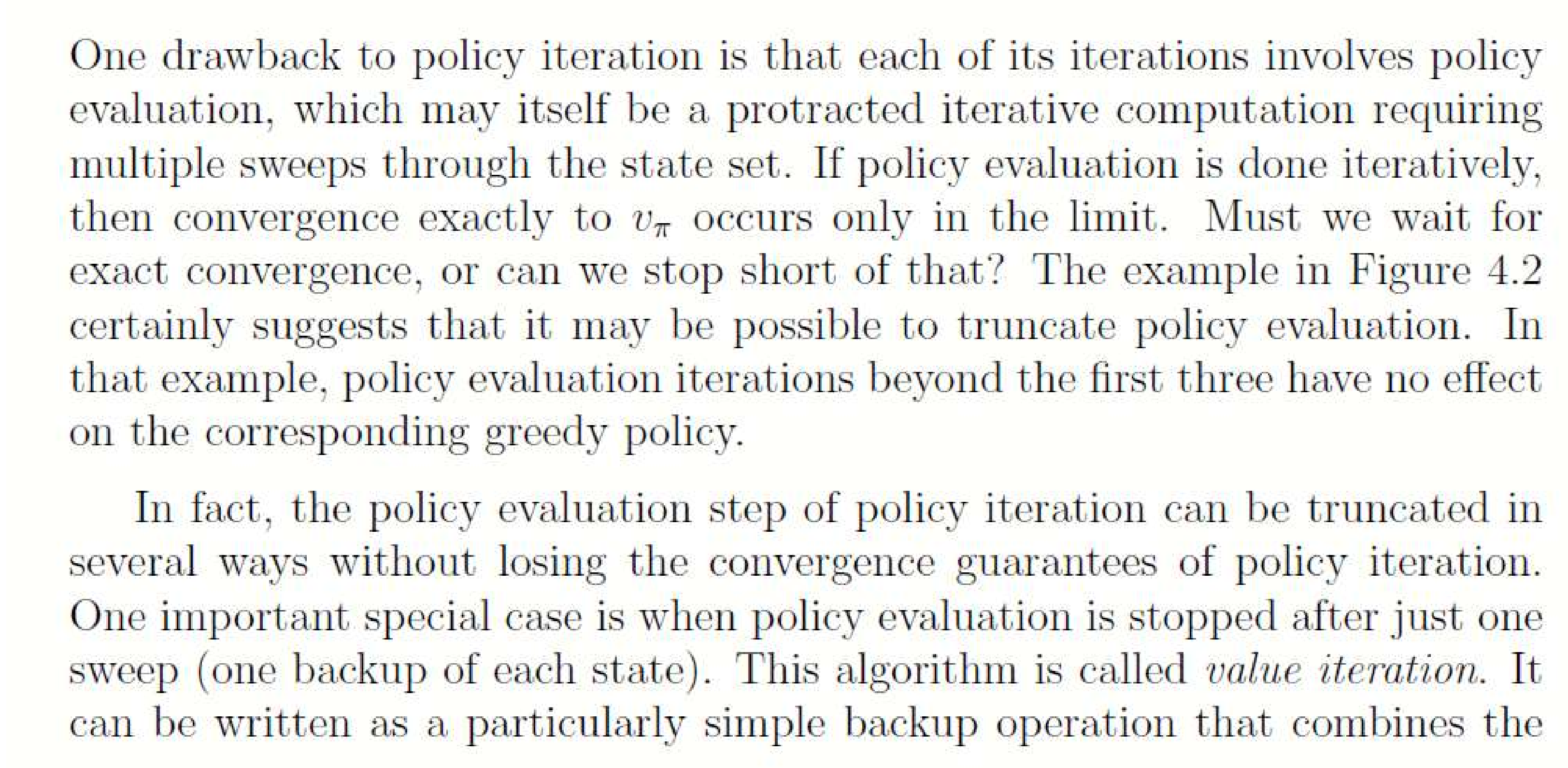
Policy Iteration:



Example 4.2: Jack's Car Rental Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited \$10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of \$2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is n is $\frac{\lambda^n}{n!}e^{-\lambda}$, where λ is the expected number. Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = 0.9$ and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations overnight. Figure 4.4 shows the sequence of policies found by policy iteration starting from the policy that never moves any cars.

■

Value Iteration:



One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to v_π occurs only in the limit. Must we wait for exact convergence, or can we stop short of that? The example in Figure 4.2 certainly suggests that it may be possible to truncate policy evaluation. In that example, policy evaluation iterations beyond the first three have no effect on the corresponding greedy policy.

In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one backup of each state). This algorithm is called *value iteration*. It can be written as a particularly simple backup operation that combines the

Value Iteration:

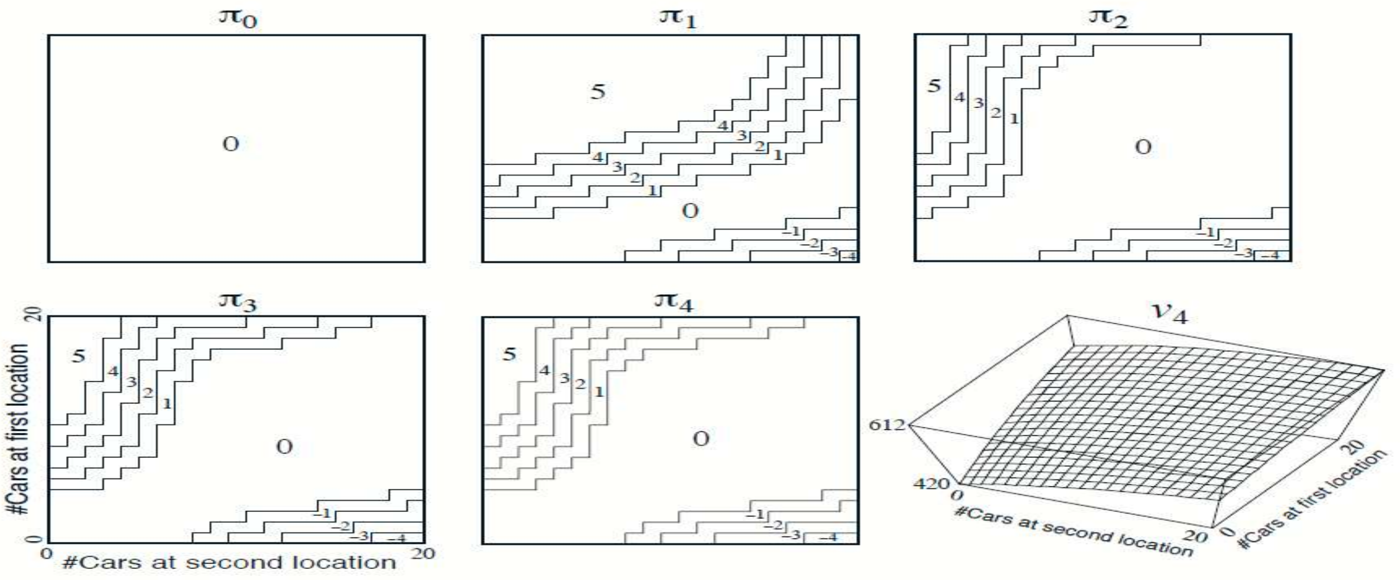
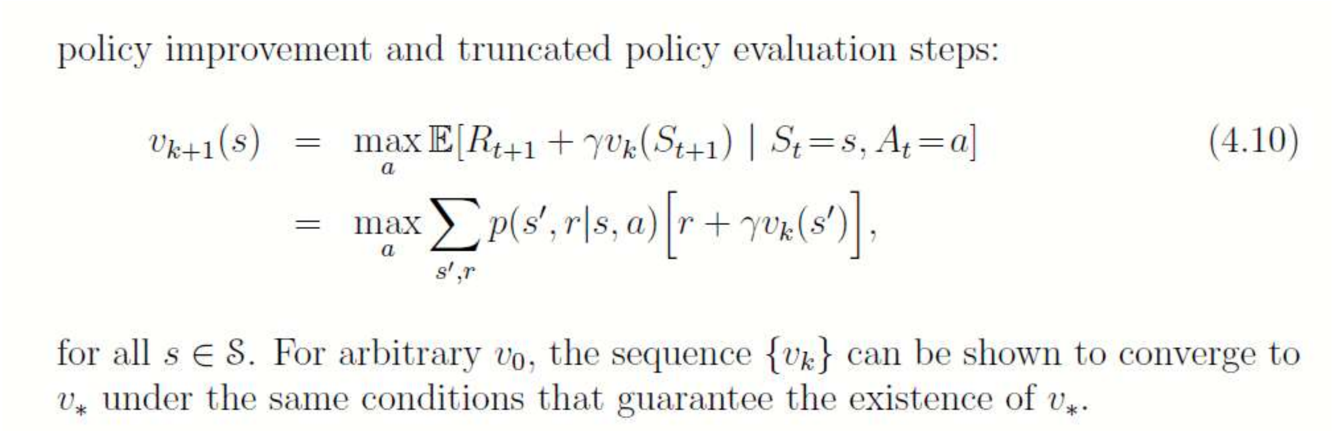


Figure 4.4: The sequence of policies found by policy iteration on Jack's car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous policy, and the last policy is optimal.

Value Iteration:

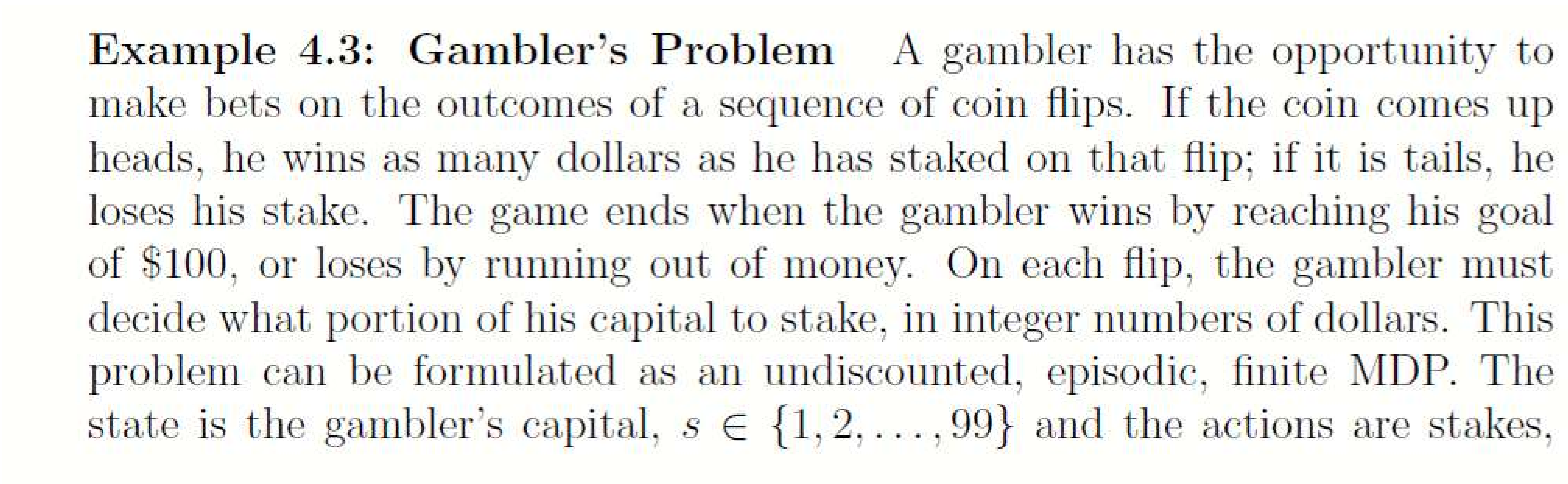


policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_k(s')\Big], \end{aligned} \tag{4.10}$$

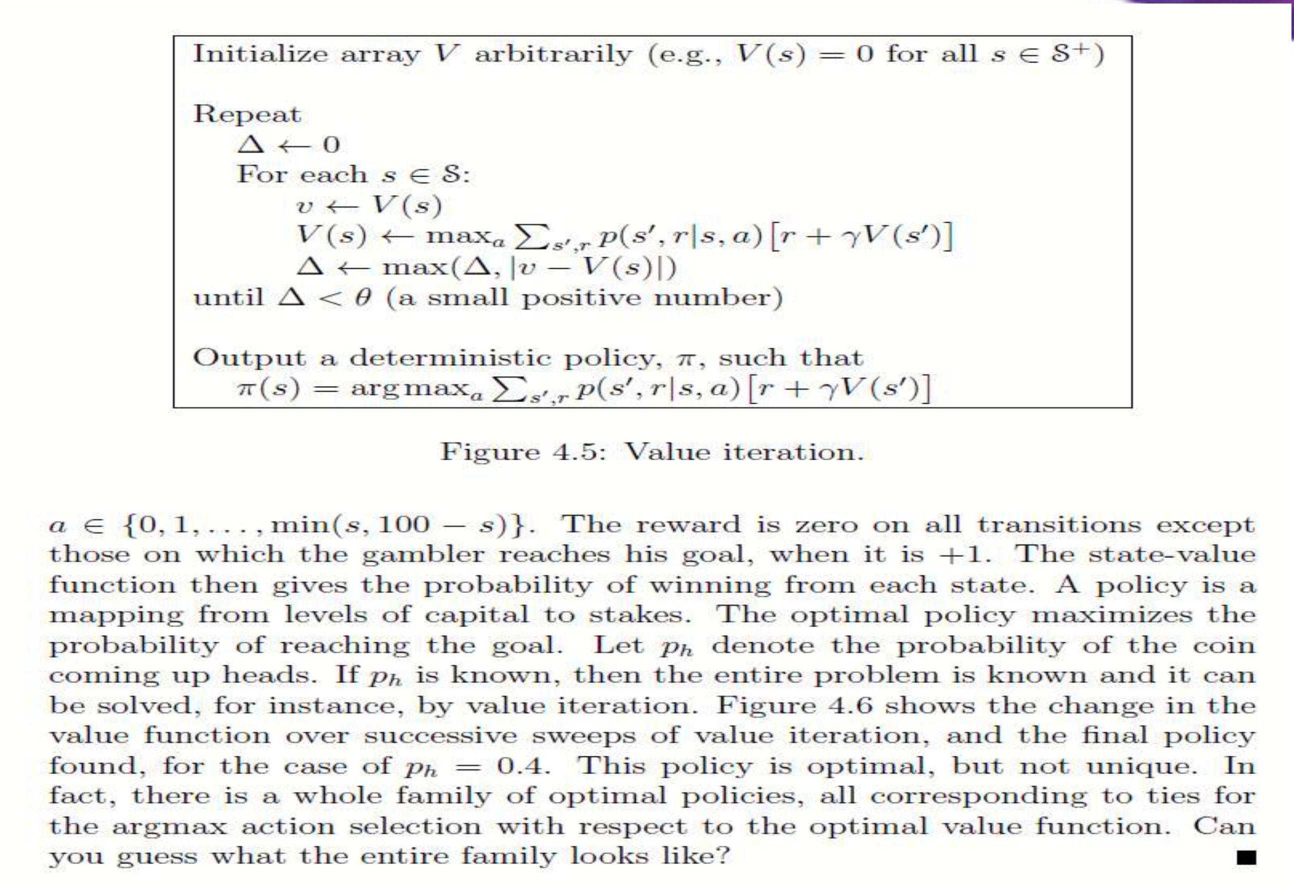
for all $s \in \mathcal{S}$. For arbitrary v_0, the sequence $\{v_k\}$ can be shown to converge to v_* under the same conditions that guarantee the existence of v_*.

Value Iteration:



Example 4.3: Gambler's Problem A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of \$100, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars. This problem can be formulated as an undiscounted, episodic, finite MDP. The state is the gambler's capital, $s \in \{1, 2, \ldots, 99\}$ and the actions are stakes,

Value Iteration:

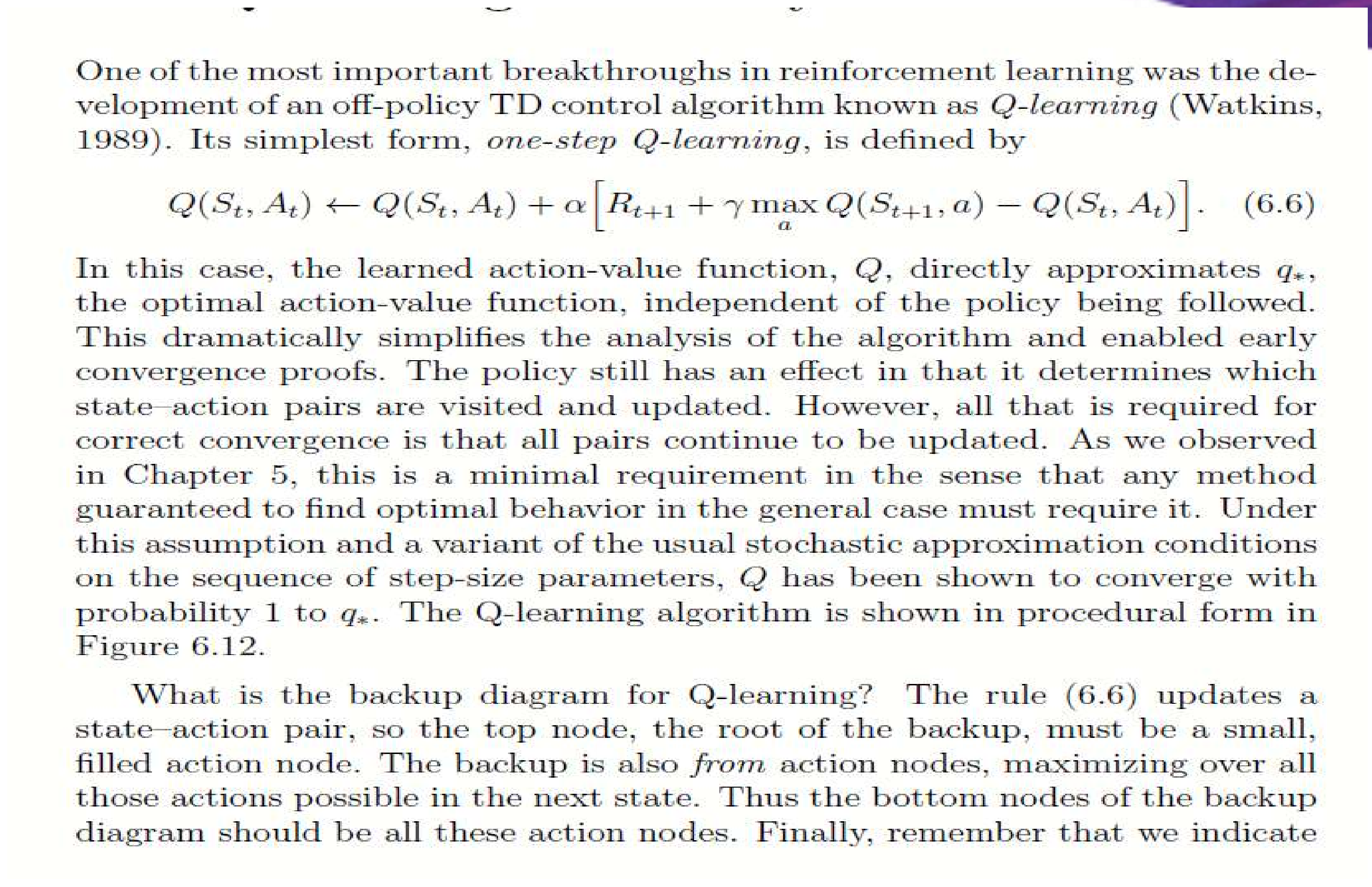


```
Initialize array $V$ arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
       $v \leftarrow V(s)$
       $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
   $\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
```

Figure 4.5: Value iteration.

$a \in \{0, 1, \ldots, \min(s, 100 - s)\}$. The reward is zero on all transitions except those on which the gambler reaches his goal, when it is $+1$. The state-value function then gives the probability of winning from each state. A policy is a mapping from levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal. Let p_h denote the probability of the coin coming up heads. If p_h is known, then the entire problem is known and it can be solved, for instance, by value iteration. Figure 4.6 shows the change in the value function over successive sweeps of value iteration, and the final policy found, for the case of $p_h = 0.4$. This policy is optimal, but not unique. In fact, there is a whole family of optimal policies, all corresponding to ties for the argmax action selection with respect to the optimal value function. Can you guess what the entire family looks like? ■

Q-Learning:



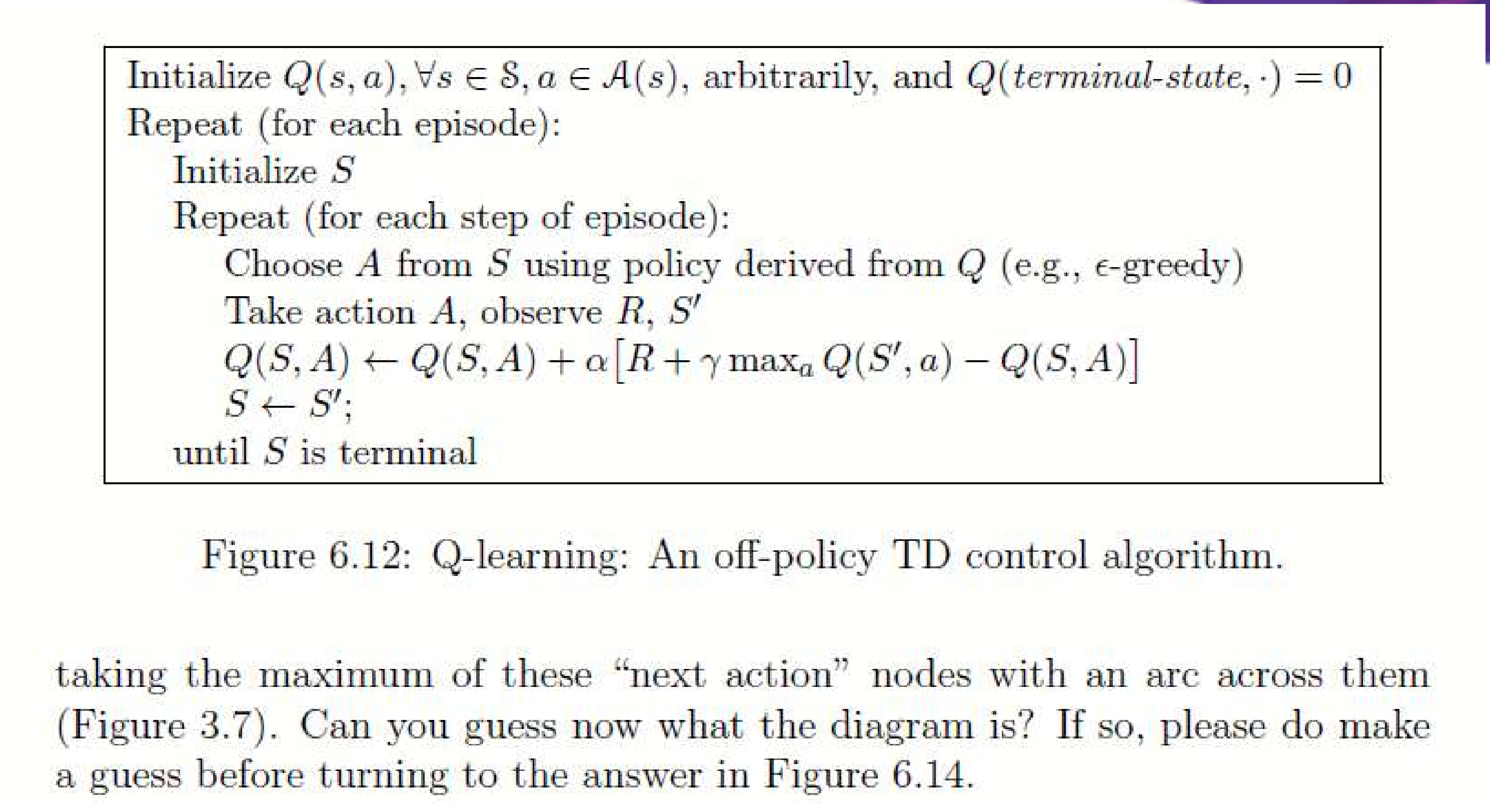
One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as *Q-learning* (Watkins, 1989). Its simplest form, *one-step Q-learning*, is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]. \quad (6.6)$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we observed in Chapter 5, this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_*. The Q-learning algorithm is shown in procedural form in Figure 6.12.

What is the backup diagram for Q-learning? The rule (6.6) updates a state–action pair, so the top node, the root of the backup, must be a small, filled action node. The backup is also *from* action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate

Q-Learning:

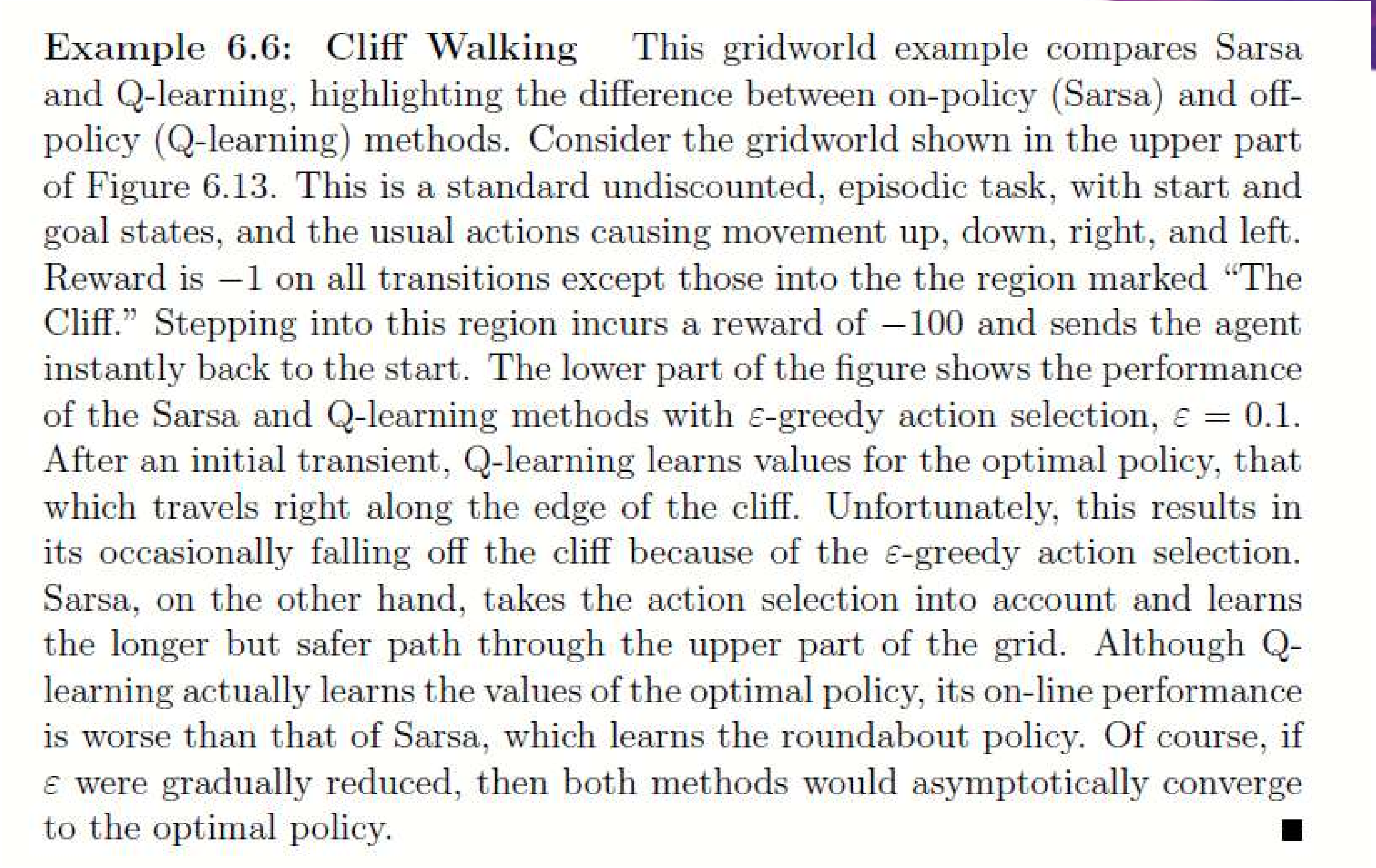


```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$;
    until $S$ is terminal
```

Figure 6.12: Q-learning: An off-policy TD control algorithm.

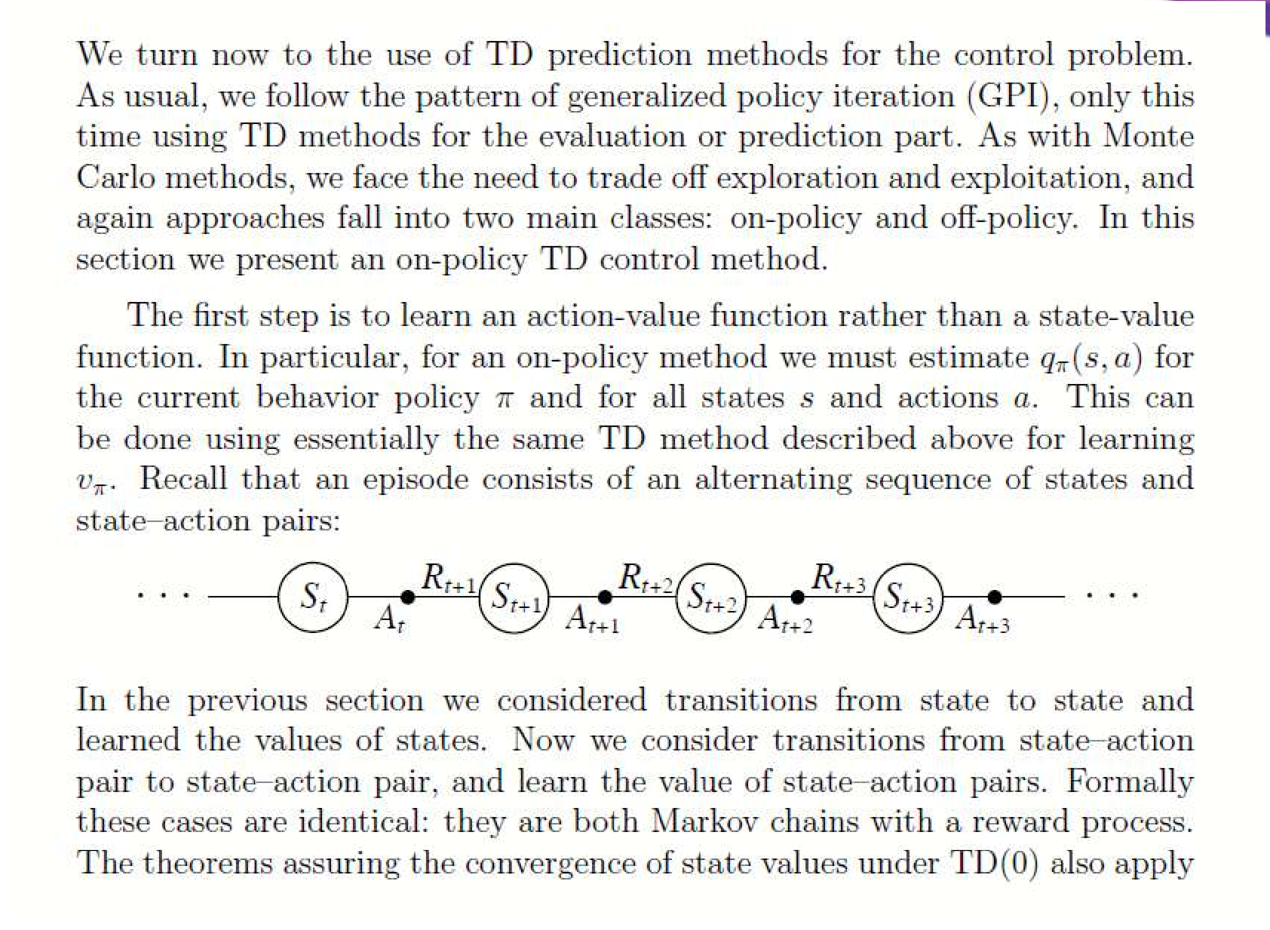
taking the maximum of these “next action” nodes with an arc across them (Figure 3.7). Can you guess now what the diagram is? If so, please do make a guess before turning to the answer in Figure 6.14.

Q-Learning:



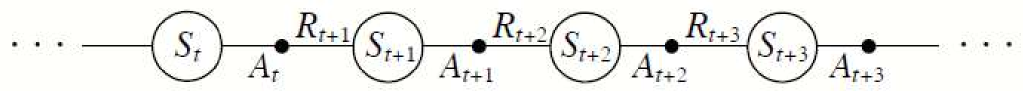
Example 6.6: Cliff Walking This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the gridworld shown in the upper part of Figure 6.13. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the the region marked "The Cliff." Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start. The lower part of the figure shows the performance of the Sarsa and Q-learning methods with ε-greedy action selection, $\varepsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε-greedy action selection. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its on-line performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if ε were gradually reduced, then both methods would asymptotically converge to the optimal policy. ■

State-Action-Reward-State-Action (SARSA):



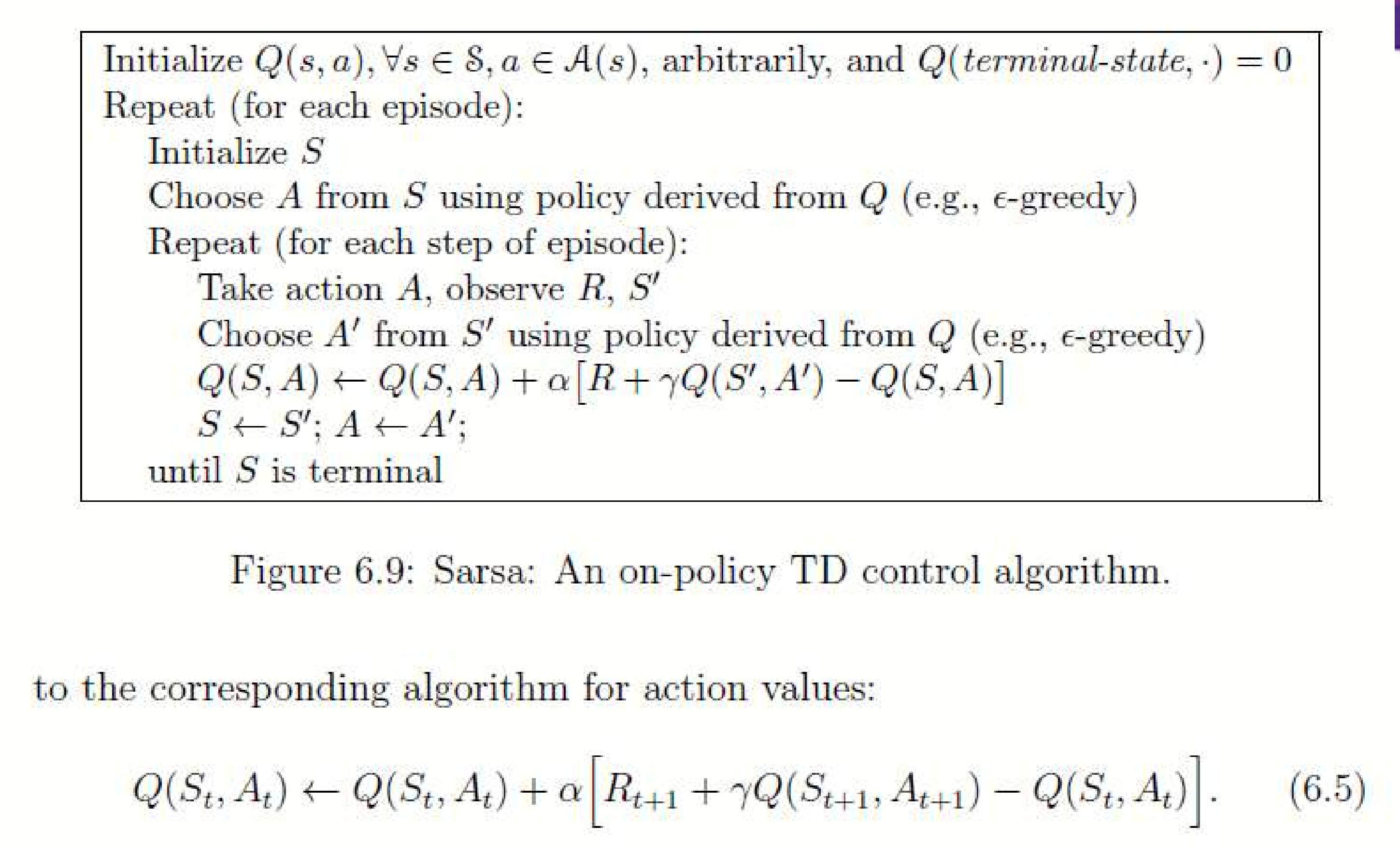
We turn now to the use of TD prediction methods for the control problem. As usual, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part. As with Monte Carlo methods, we face the need to trade off exploration and exploitation, and again approaches fall into two main classes: on-policy and off-policy. In this section we present an on-policy TD control method.

The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a. This can be done using essentially the same TD method described above for learning v_π. Recall that an episode consists of an alternating sequence of states and state–action pairs:



In the previous section we considered transitions from state to state and learned the values of states. Now we consider transitions from state–action pair to state–action pair, and learn the value of state–action pairs. Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also apply

State-Action-Reward-State-Action (SARSA):



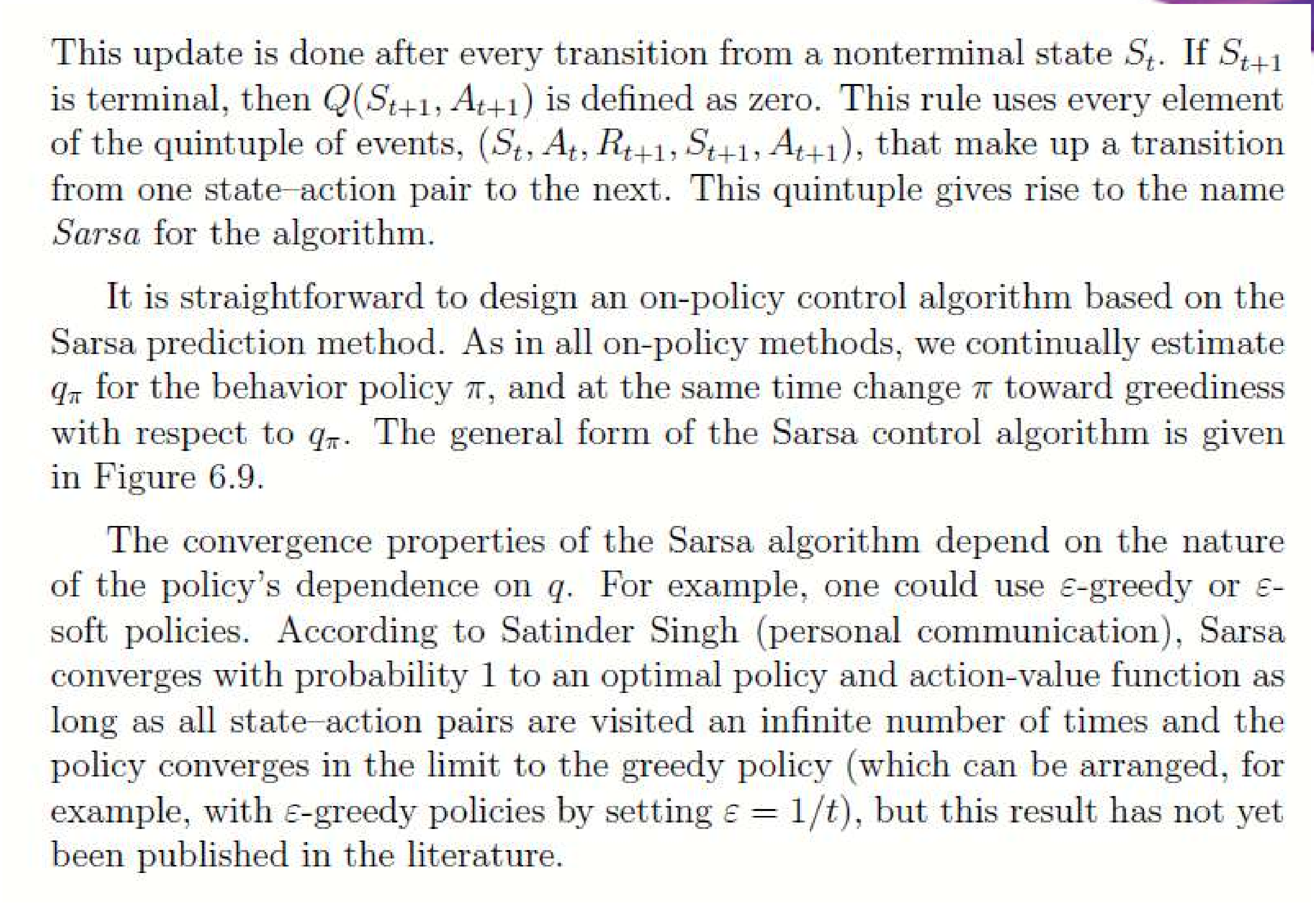
```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
   Initialize $S$
   Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
   Repeat (for each step of episode):
      Take action $A$, observe $R$, $S'$
      Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
      $S \leftarrow S'$; $A \leftarrow A'$;
   until $S$ is terminal
```

Figure 6.9: Sarsa: An on-policy TD control algorithm.

to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]. \qquad (6.5)$$

State-Action-Reward-State-Action (SARSA):



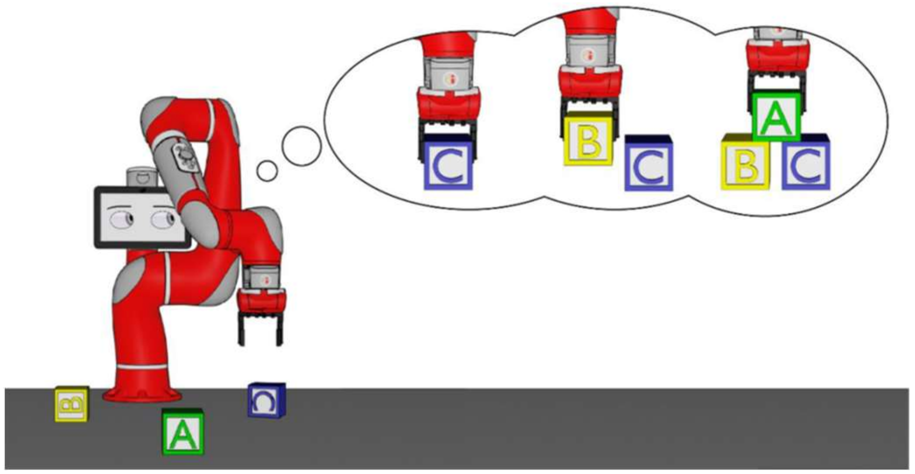
This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state–action pair to the next. This quintuple gives rise to the name *Sarsa* for the algorithm.

It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π. The general form of the Sarsa control algorithm is given in Figure 6.9.

The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on q. For example, one could use ε-greedy or ε-soft policies. According to Satinder Singh (personal communication), Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state–action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ε-greedy policies by setting $\varepsilon = 1/t$), but this result has not yet been published in the literature.

Model-based Reinforcement Learning:

Reinforcement learning systems can make decisions in one of two ways. In the model-based approach, a system uses a predictive model of the world to ask questions of the form "what will happen if I do x?" to choose the best x1. In the alternative model-free approach, the modeling step is bypassed altogether in favor of learning a control policy directly. Although in practice the line between these two techniques can become blurred, as a coarse guide it is useful for dividing up the space of algorithmic possibilities.



Predictive models can be used to ask "what if?" questions to guide future decisions.

Model-based Reinforcement Learning:

The natural question to ask after making this distinction is whether to use such a predictive model. The field has grappled with this question for quite a while, and is unlikely to reach a consensus any time soon. However, we have learned enough about designing model-based algorithms that it is possible to draw some general conclusions about best practices and common pitfalls. In this post, we will survey various realizations of model-based reinforcement learning methods. We will then describe some of the tradeoffs that come into play when using a learned predictive model for training a policy and how these considerations motivate a simple but effective strategy for model-based reinforcement learning. The latter half of this post is based on our recent paper on model-based policy optimization, for which code is available here.

Model-based techniques
Below, model-based algorithms are grouped into four categories to highlight the range of uses of predictive models. For the comparative performance of some of these approaches in a continuous control setting, this benchmarking paper is highly recommended.

Model-based Reinforcement Learning:

Analytic gradient computation
Assumptions about the form of the dynamics and cost function are convenient because they can yield closed-form solutions for locally optimal control, as in the LQR framework. Even when these assumptions are not valid, receding-horizon control can account for small errors introduced by approximated dynamics. Similarly, dynamics models parametrized as Gaussian processes have analytic gradients that can be used for policy improvement. Controllers derived via these simple parametrizations can also be used to provide guiding samples for training more complex nonlinear policies.

Sampling-based planning
In the fully general case of nonlinear dynamics models, we lose guarantees of local optimality and must resort to sampling action sequences. The simplest version of this approach, random shooting, entails sampling candidate actions from a fixed distribution, evaluating them under a model, and choosing the action that is deemed the most promising. More sophisticated variants iteratively adjust the sampling distribution, as in the cross-entropy method (CEM; used in PlaNet, PETS, and visual foresight) or path integral optimal control (used in recent model-based dexterous manipulation work).

Model-based Reinforcement Learning:

Model-based data generation

An important detail in many machine learning success stories is a means of artificially increasing the size of a training set. It is difficult to define a manual data augmentation procedure for policy optimization, but we can view a predictive model analogously as a learned method of generating synthetic data. The original proposal of such a combination comes from the Dyna algorithm by Sutton, which alternates between model learning, data generation under a model, and policy learning using the model data. This strategy has been combined with iLQG, model ensembles, and meta-learning; has been scaled to image observations; and is amenable to theoretical analysis. A close cousin to model-based data generation is the use of a model to improve target value estimates for temporal difference learning.

Value-equivalence prediction

A final technique, which does not fit neatly into model-based versus model-free categorization, is to incorporate computation that resembles model-based planning without supervising the model's predictions to resemble actual states. Instead, plans under the model are constrained to match trajectories in the real environment only in their predicted cumulative reward. These value-equivalent models have shown to be effective in high-dimensional observation spaces where conventional model-based planning has proven difficult.

Thanks!