Module 12 - Markov Decision Process and Reinforcement Learning Algorithms

SIT320 - Advanced Algorithms

Dr. Nayyar Zaidi

Today

- Introduction to Reinforcement Learning

- Markov Decision Process
    - How to solve an MDP?
    - Dynamic Programming

- Unfolding terms:
    - Learning vs. Planning
    - Policy Evaluation and Policy Control
    - Value Iteration Algorithm

- Monte-Carlo Algorithms
- Temporal Difference Learning Algorithms

- Planning
    - Monte Carlo Tree Search (MCTS)

Today

- How did you learn to ride a Bike?
  - Supervised Learning
  - Unsupervised Learning
  - Reinforcement Learning

- Earliest papers in behavioural psychology by Sutton & Barto
- 1983 paper brought RL into mainstream CS

- **What is Reinforcement Learning (RL)?**
  - Mathematic formulation of Trial and Error
  - Learning from minimal feedback
  - Learning about the system through interacting with the system, can't be done completely offline (some notion of interaction)
  - Inspired by studies in Behavioural Psychology
    - Pavlov's Dog Experiment
    - Signal: Bell, Reward: Food
    - **Behavioural Conditioning**

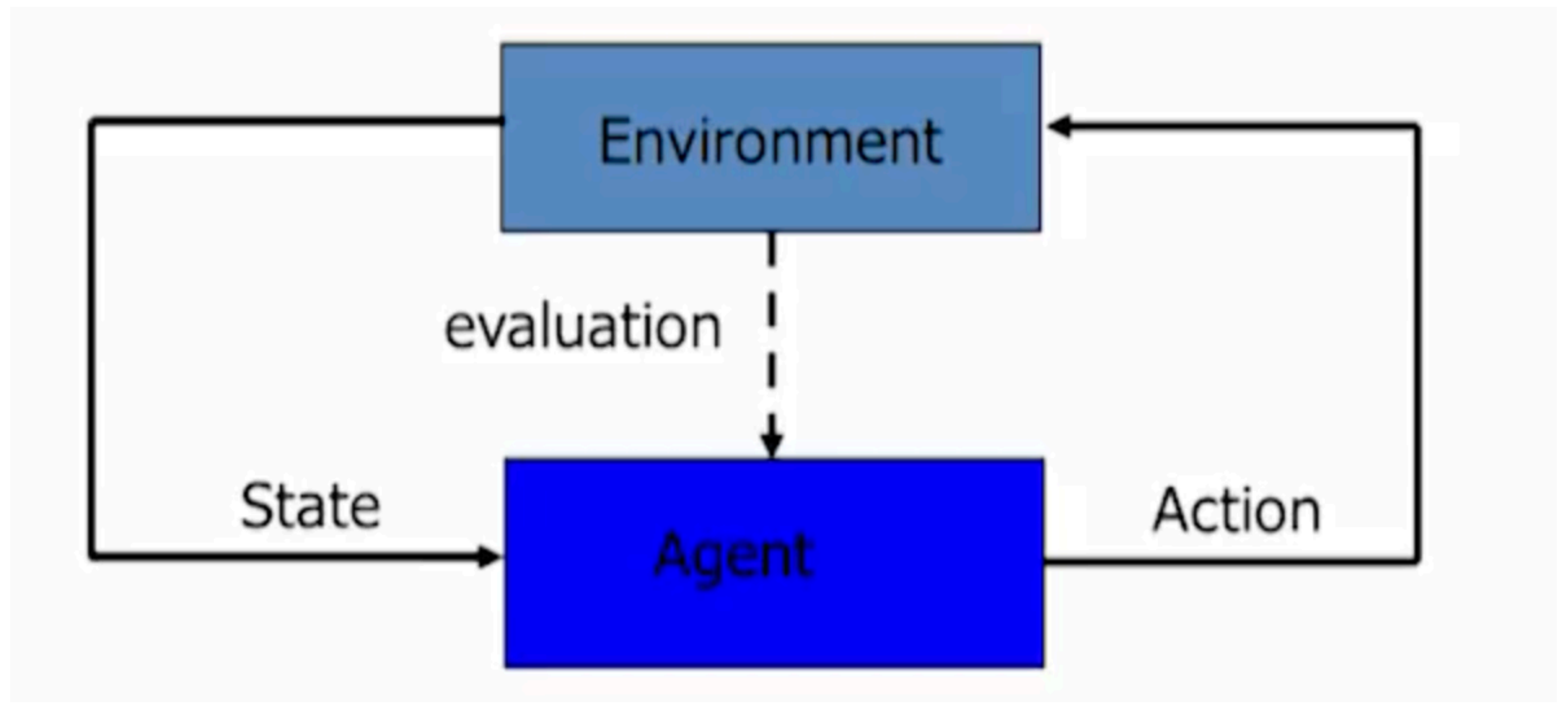
- **How to play a game of chess?**
  - **Detailed supervision:**
    - Look at board position, and then look at an Oracle which tells you what to do, and then make a move. Technically mapping from input to an output, and then learn to generalised
  - **Trial & Error:**
    - You play/make moves, and at some point you get a reward if you win. No body tells you given the position to do this or don't do this

Introduction

- **Salient Features of RL**
  - **(1)** Rewards/Punishments can be delayed from the actions that led to rewards/punishments to happen
    - You might need sequence of actions before you get a reward
      - E.g., in chess you will make a sequence of actions before you win/lose — **Temporal Disconnect**
      - Learning associations between input (states) to actions through a policy
  - **(2)** Noisy/stochastic world
  - **(3)** Trade-off between Exploration and Exploitation
  - **(4)** Goal-Directed

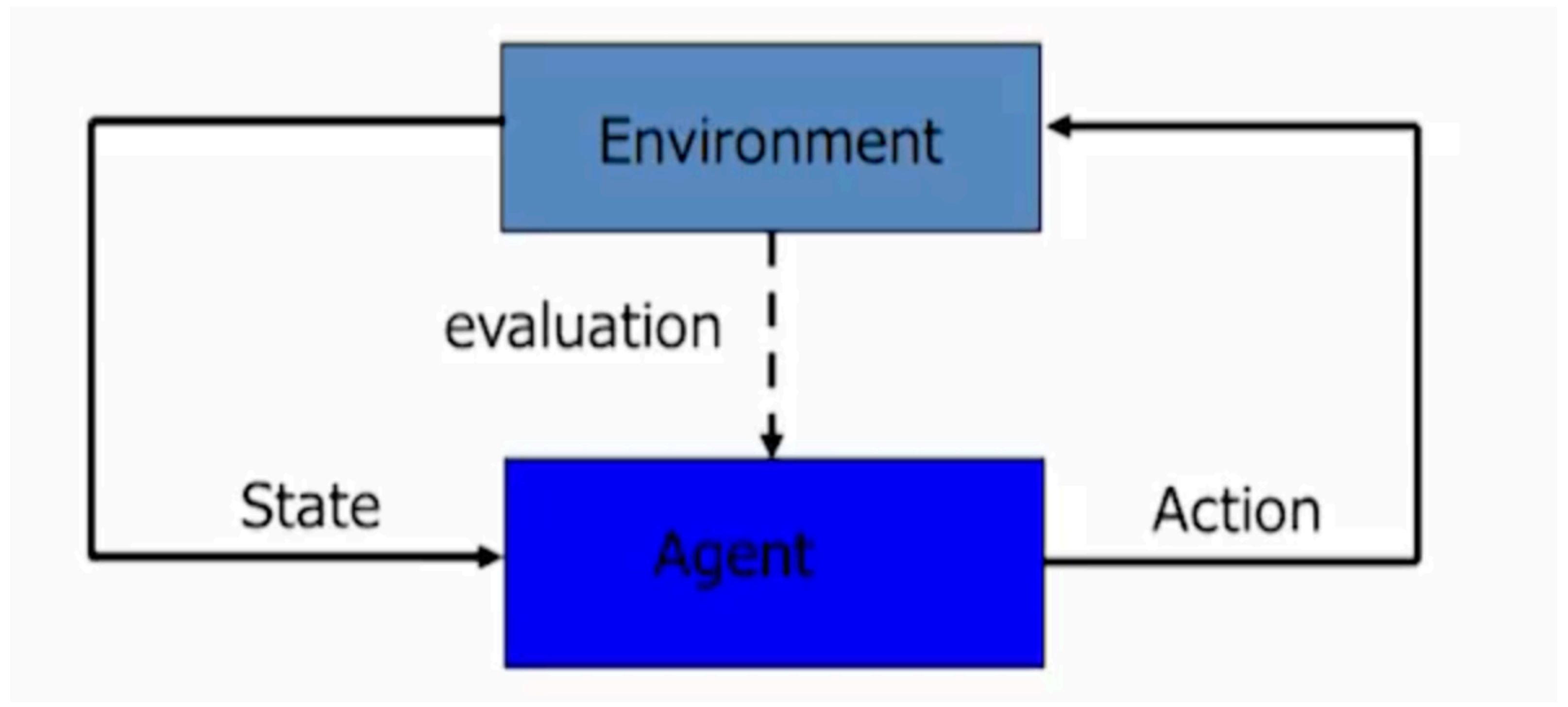
- **Applications**
  - Game Playing
    - Chess, Go
  - Online Advertising
    - Immediate RL, Recommender Systems
  - A mobile robot decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station
  - An adaptive controller adjusts parameters of a petroleum refinery's operation in real time. The controller optimizes the yield/cost/quality trade-off on the basis of specified marginal costs without sticking strictly to the set points originally suggested by engineers.

Elements of Reinforcement Learning



- Reinforcement Learning:
    - Learn from close interactions
    - Stochastic Environment
    - Noisy Delayed scalar evaluation
    - Maximize a measure of long-term performance

Elements of Reinforcement Learning



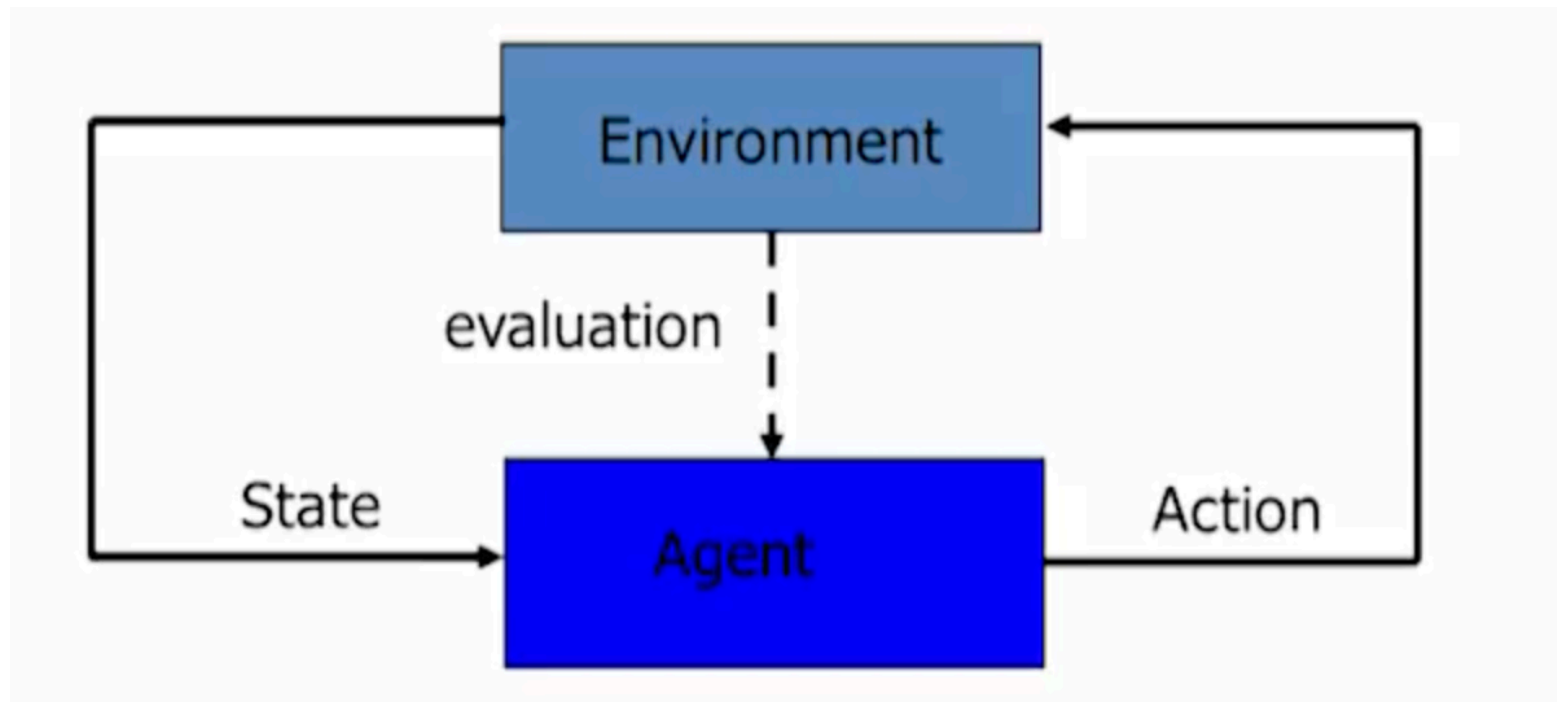
- Agent / Environment

1. Policy,
2. Reward Signal,
3. Value Function,
4. (Optionally) Model

- **Policy:**
  - A policy defines the learning agent's way of behaving at a given time,
    - mapping from perceived states of the environment to actions to be taken when in those states.
    - In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as a search process.
    - The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behaviour.
    - In general, policies may be **stochastic**, specifying probabilities for each action for each state.
    - Or **deterministic**, specifying an action for each state

$$\pi$$

Elements of Reinforcement Learning



- Agent / Environment

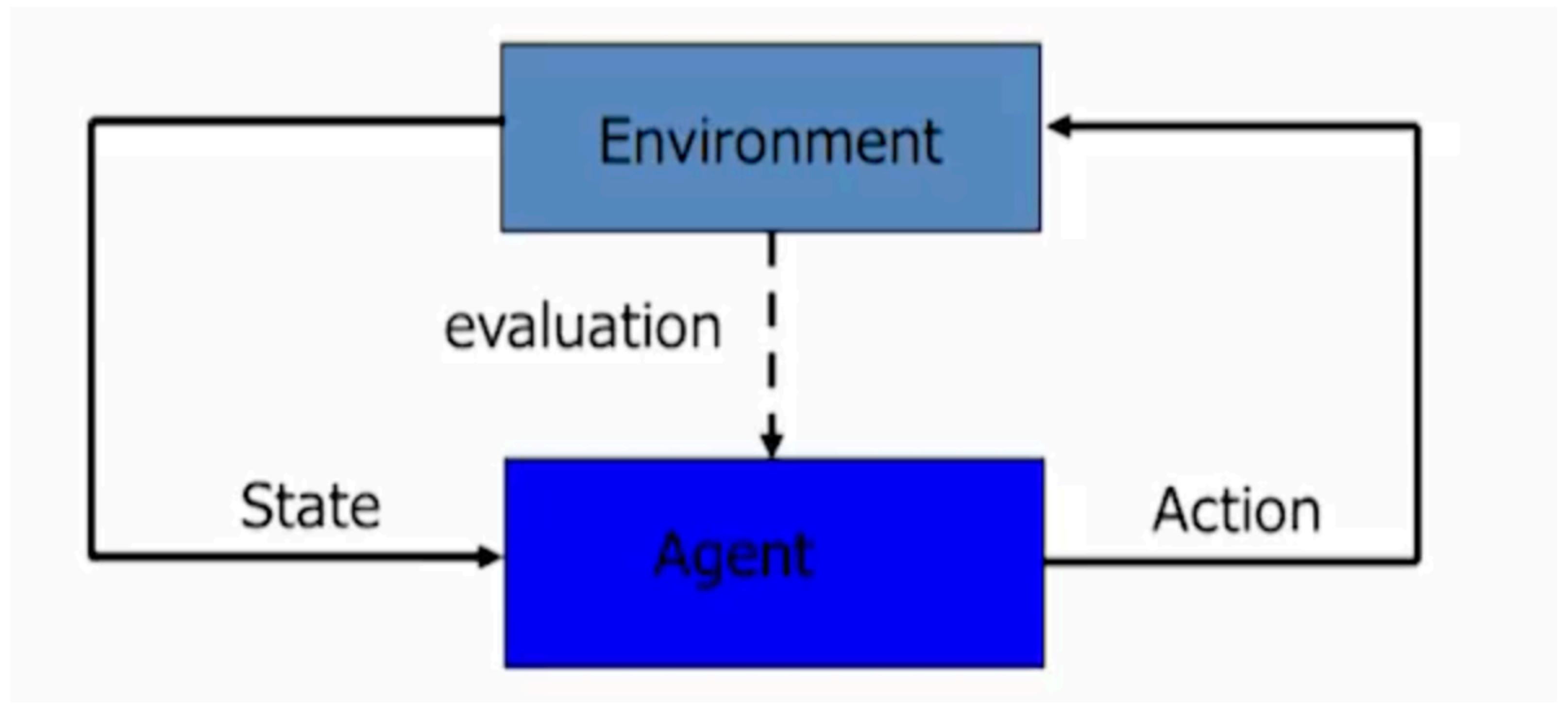
1. Policy,
2. Reward Signal (Evaluation),
3. Value Function,
4. (Optionally) Model

- **On each time step, the environment sends to the reinforcement learning agent a single number called the reward. The agent's sole objective is to maximize the total reward it receives over the long run.**

- **Evaluation:**
  - Comes form Environment when you take an action along with transition to another state
  - Reality/biological inputs are sensory inputs which can be interpreted as rewards or punishments
  - We consider a scalar value as a reward/punishment:
    - Why scalars?
      - Well, they are easy to optimise
    - Note: you are getting a scalar value say +5, -3, etc. you don't know if it is out of 10 or 1000 or 1 Million
    - How can we use these rewards
      - If you had target, you can compute the gradients, but you are training to predict 3 but 3 might be bad not good
      - Also, we do not have targets, and only we can learn is trial and error

$\mathscr{R}$

Elements of Reinforcement Learning



- Agent / Environment

1. Policy,
2. Reward Signal (Evaluation),
3. Value Function,
4. (Optionally) Model

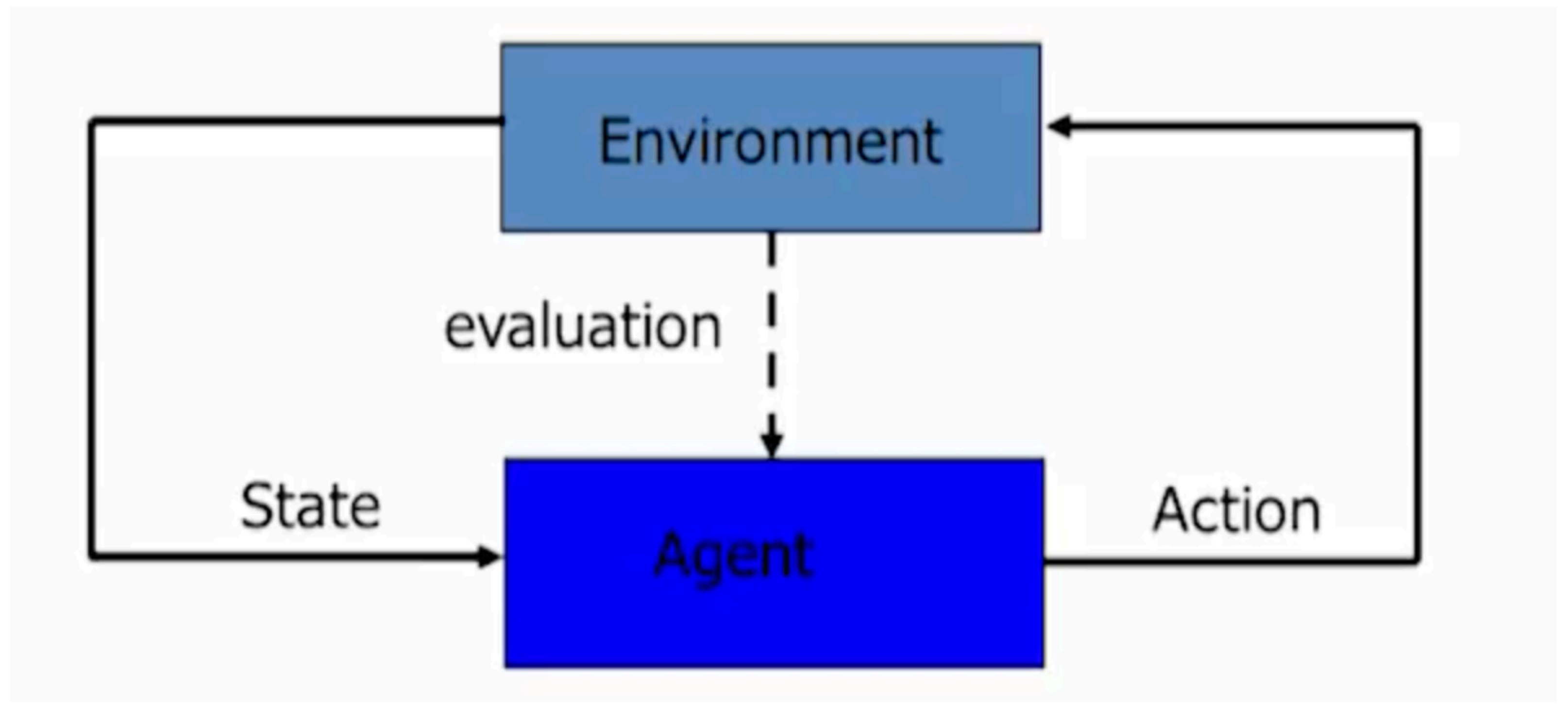
- **Value Function:**
    - Whereas the reward signal indicates what is good in an immediate sense, a value function specifies what is good in the long run.
    - Roughly speaking, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.
    - Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states.

- **To make a human analogy, rewards are somewhat like pleasure (if high) and pain (if low), whereas values correspond to a more refined and farsighted judgment of how pleased or displeased we are that our environment is in a particular state**

$$V(S)$$

$$Q(S, A)$$

Elements of Reinforcement Learning



- Agent / Environment

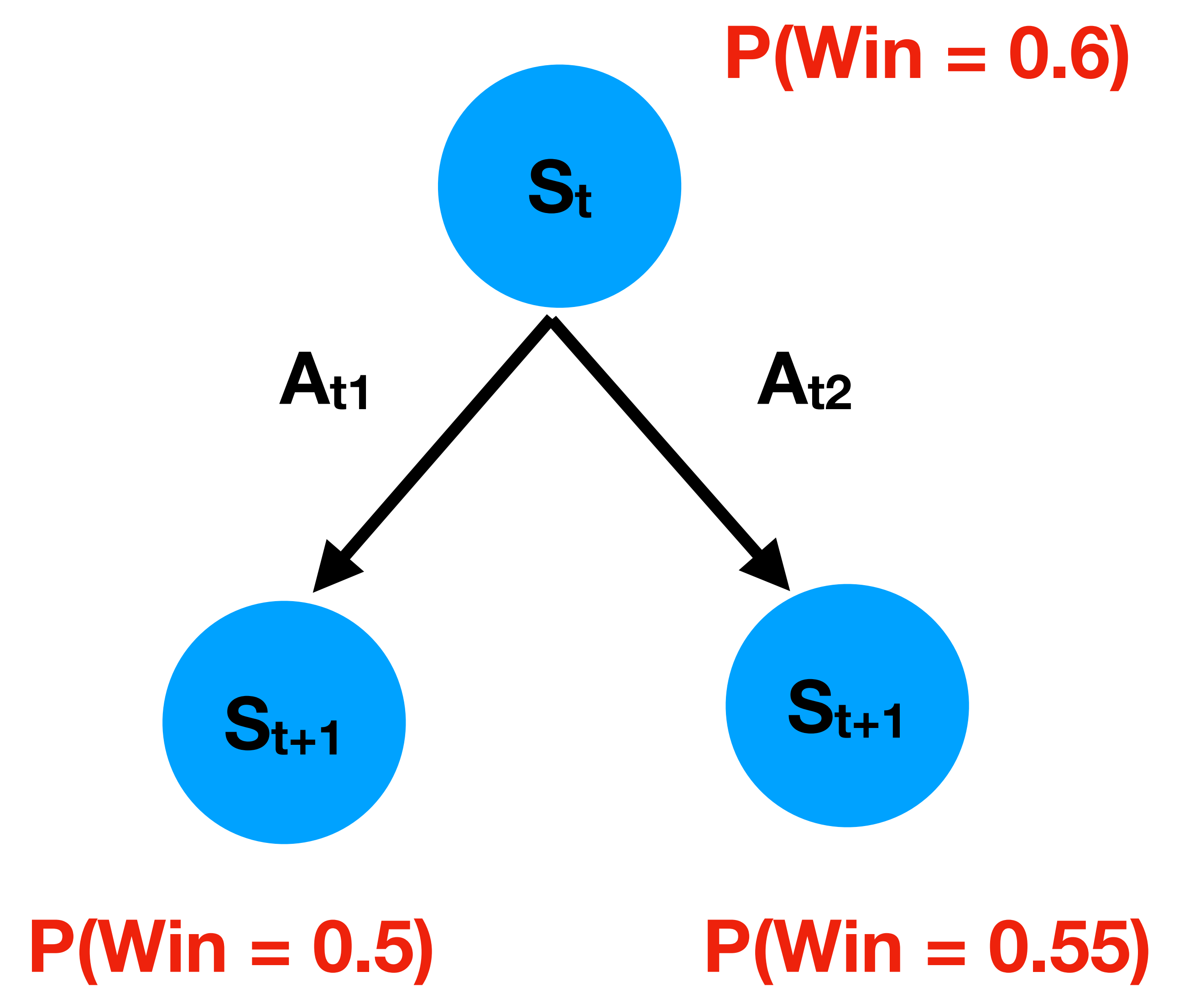
1. Policy,
2. Reward Signal (Evaluation),
3. Value Function,
4. (Optionally) Model

- **Model:**
  - This is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave.
    - For example, given a state and action, the model might predict the resultant next state and next reward.
    - Models are used for **planning**, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced.
  - Methods for solving reinforcement learning problems that use models and planning are called **model-based methods**, as opposed to simpler **model-free methods** that are explicitly <u>trial-and-error learners</u>—viewed as almost the opposite of planning.
  - Modern reinforcement learning spans the spectrum from low-level, trial-and-error learning to high-level, deliberative planning.

Soul of RL

- Prediction of outcomes time t+1 is better than the prediction at time t — we can use the later prediction adjust earlier predictions

- **Temporal Difference (TD) Learning**
  - Claim: ‘you are going to win’ — you can say with more confidence closer to the end of the game than at start of the game
  - Prediction at time t+1 is more accurate than at time t
    - **To improve prediction at time t —**
      - why don’t we just let the clock tick over, look at your prediction at time t+1 (based on additional knowledge you get as you are one step closer to the end), adjust your prediction at time t
    - Note, you are not taking back a bad move, you are modifying the probabilities so next time you come this state, you have better estimate of probabilities



Modify P(Win in S_t) based on P(Win in S_{t+1} | A_{t2} is taken)

If you game is deterministic: Modify 0.6 to 0.55, otherwise if is stochastic, change it little bit towards 0.6.

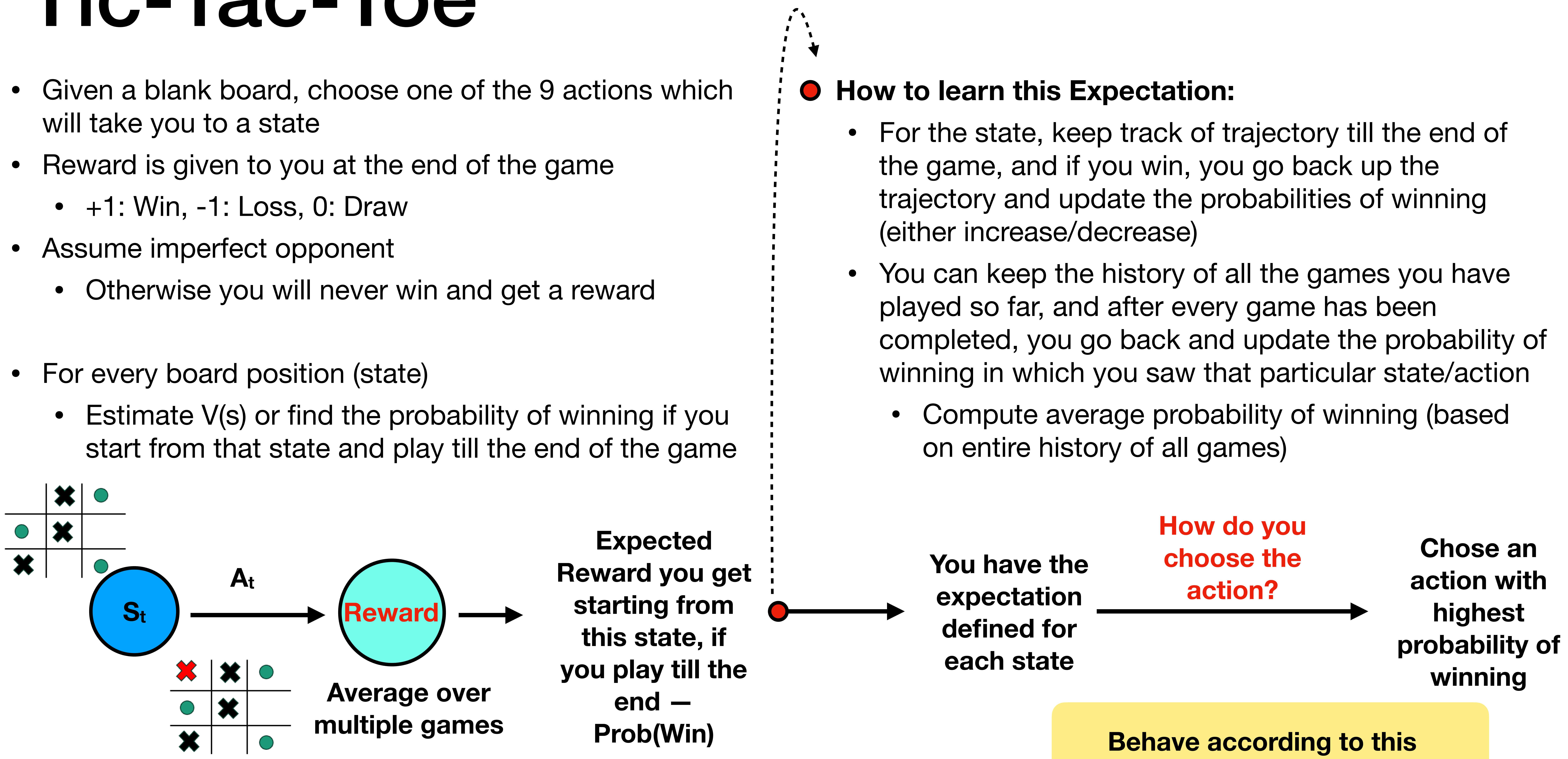
Tic-Tac-Toe

- Given a blank board, choose one of the 9 actions which will take you to a state
- Reward is given to you at the end of the game
  - +1: Win, -1: Loss, 0: Draw
- Assume imperfect opponent
  - Otherwise you will never win and get a reward

- For every board position (state)
  - Estimate V(s) or find the probability of winning if you start from that state and play till the end of the game

How to learn this Expectation:

- For the state, keep track of trajectory till the end of the game, and if you win, you go back up the trajectory and update the probabilities of winning (either increase/decrease)
- You can keep the history of all the games you have played so far, and after every game has been completed, you go back and update the probability of winning in which you saw that particular state/action
  - Compute average probability of winning (based on entire history of all games)

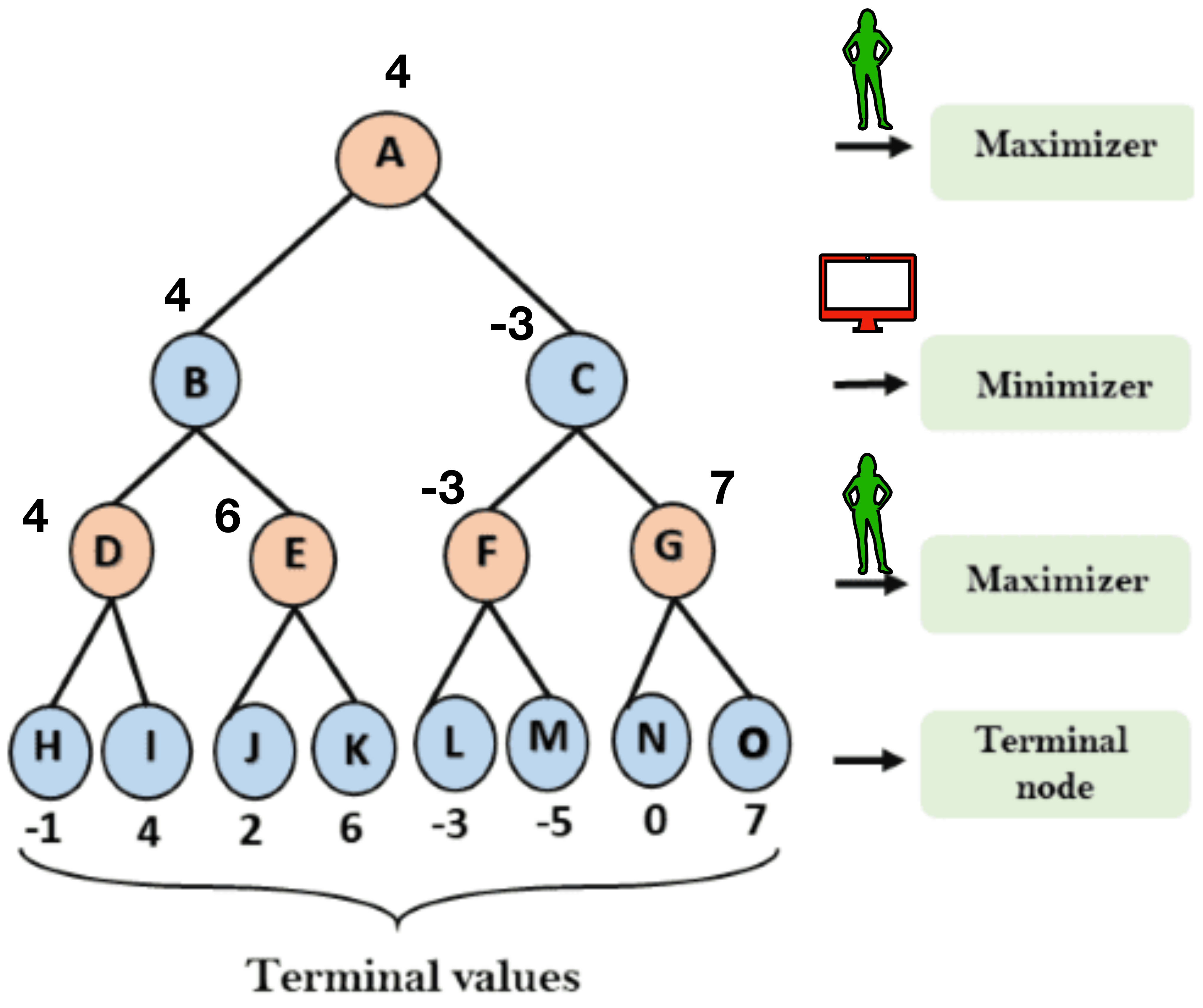


S_t
A_t
Reward
Average over multiple games
Expected Reward you get starting from this state, if you play till the end — Prob(Win)
You have the expectation defined for each state
How do you choose the action?
Chose an action with highest probability of winning

Behave according to this Expectation

Minmax Algorithm

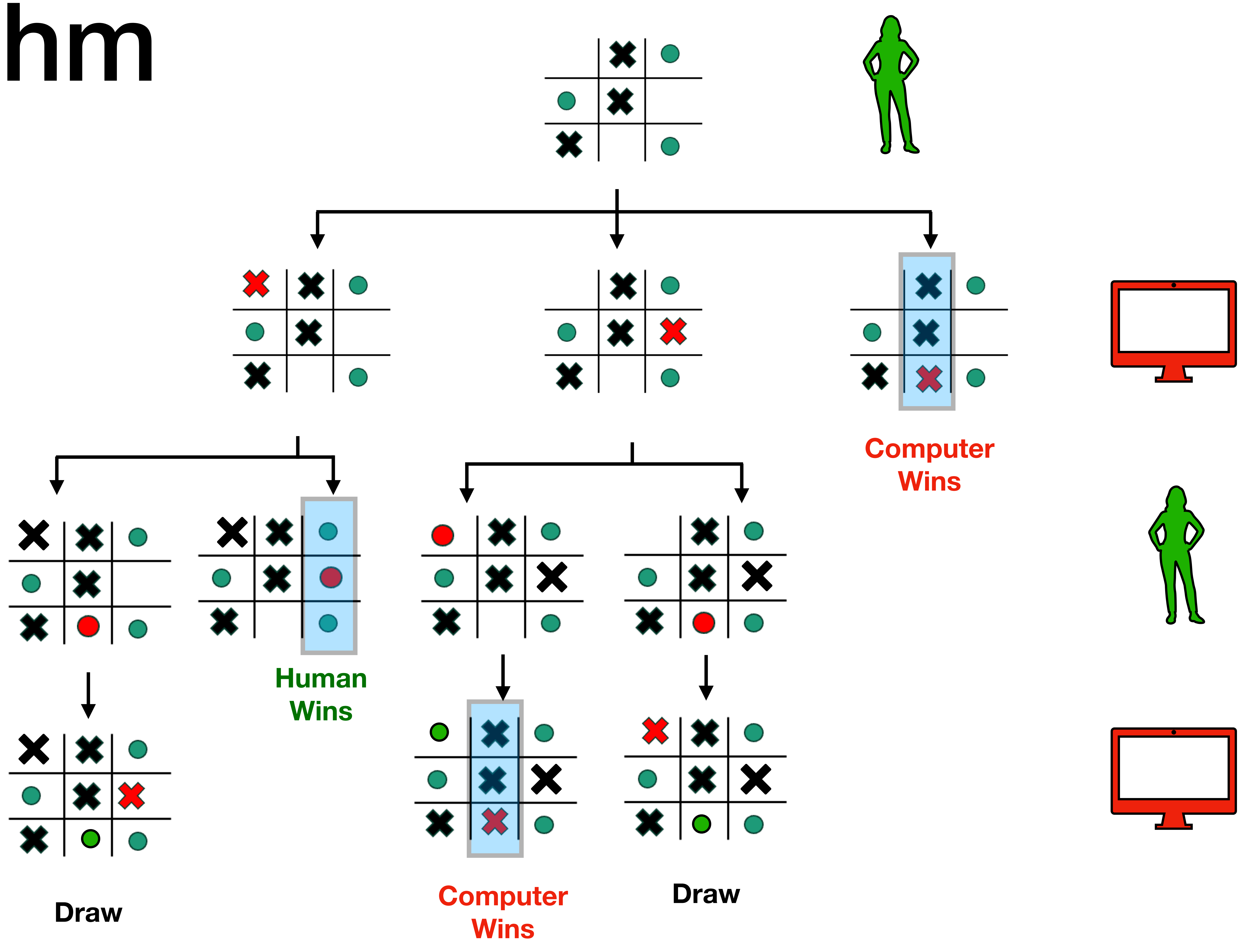
- Complete:
  - The algorithm will locate the optimal solution
- Optimal:
  - If both opponents play optimally, the min-max is optimal
- Time Complexity:
  - Executes a depth-first search — **O(b^m)**, where b is the game branching tree factor, and m is the tree's maximum depth
- Space Complexity:
  - **O(bm)**



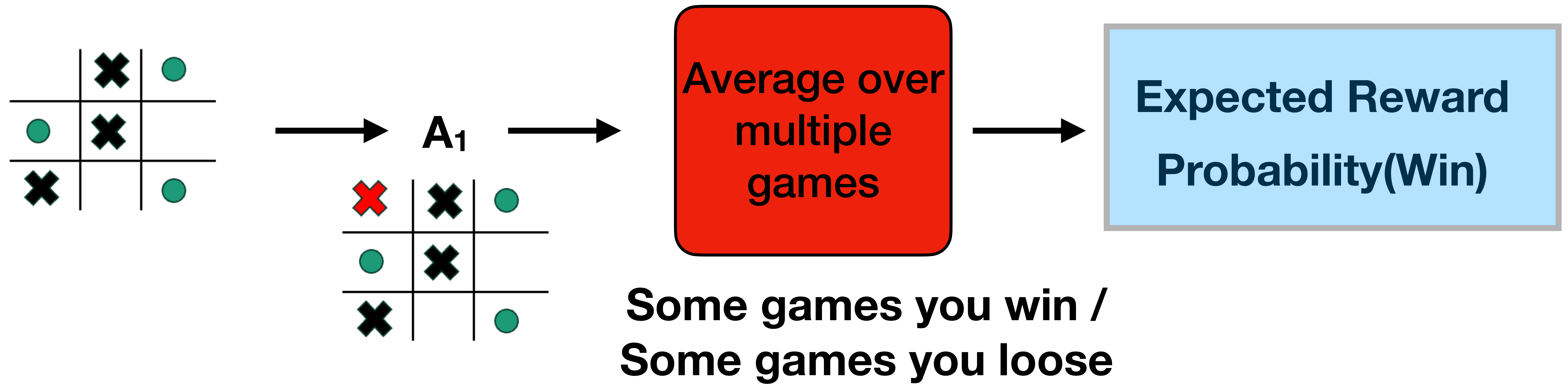
Minimax Algorithm

✖ Computer

● Human



Solving Tic-tac-toe with RL



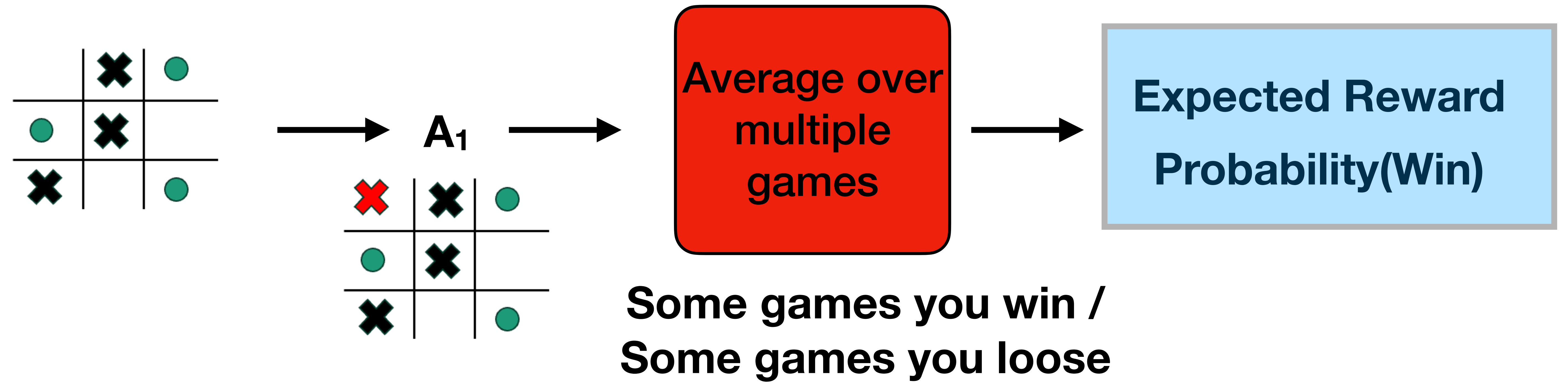
Estimating this is the crux of all RL algorithms

Once you have the expectation well defined — you will choose an action that has the highest probability of winning

- **Minimax Solution**
  - Assumes a particular way of playing by the opponent. E.g., a minimax player would never reach a game state from which it could lose, even if in fact it always won from that state because of incorrect play by the opponent

- **Dynamic Programming Solution**
  - Compute an optimal solution for any opponent, but require as input a complete specification of that opponent, including the probabilities with which the opponent makes each move in each board state
  - Let us assume that this information is not available a priori for this problem, as it is not for the vast majority of problems of practical interest
  - On the other hand, such information can be estimated from experience, in this case by playing many games against the opponent. About the best one can do on this problem is first to learn a model of the opponent’s behaviour, up to some level of confidence, and then apply dynamic programming to compute an optimal solution given the approximate opponent model. (**Costly Solution**)

Solving Tic-tac-toe with RL



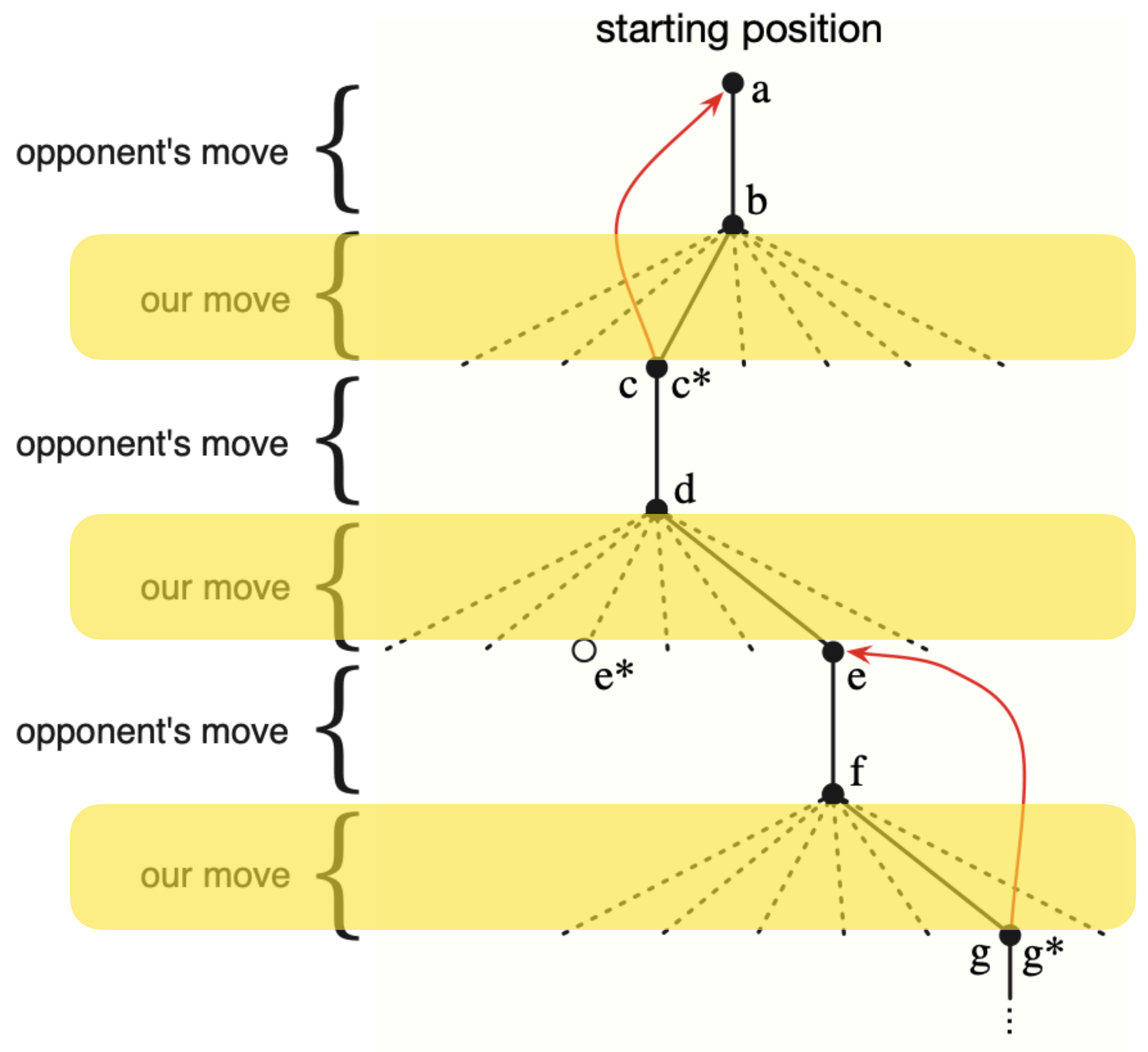
Estimating this is the crux of all RL algorithms

Once you have the expectation well defined — you will choose an action that has the highest probability of winning

- **Value Function based Solution**
  - Set up a table of numbers, one for each possible state of the game
  - Each number will be the latest estimate of the probability of our winning from that state.
  - We treat this estimate as the state's value, and the whole table is the learned value function
  - We then play many games against the opponent.

- To select our moves we examine the states that would result from each of our possible moves (one for each blank space on the board) and look up their current values in the table.
- Most of the time we move **greedily**, selecting the move that leads to the state with greatest value, that is, with the highest estimated probability of winning.
- Occasionally, however, we select randomly from among the other moves instead. These are called **exploratory** moves because they cause us to experience states that we might otherwise never see.

Solving Tic-tac-toe with RL



- While we are playing, we change the values of the states in which we find ourselves during the game
- We attempt to make them more accurate estimates of the probabilities of winning
- To do this, we "back up" the value of the state after each greedy move to the state before the move

- **The current value of the earlier state is updated to be closer to the value of the later state. This can be done by moving the earlier state's value a fraction of the way toward the value of the later state**

$$V(S_t) \leftarrow V(S_t) + \alpha \left[V(S_{t+1}) - V(S_t) \right]$$

- This update rule is an example of a temporal-difference learning method, so called because its changes are based on a difference, $V(S_{t+1})$ - $V(S_t)$, between estimates at two successive times

Computing Expected Rewards

- **In either case:**
  - One must Explore + Exploit, otherwise, you will not be able to run entire state-space
  - When to start exploring and exploiting?
    - Million dollar question
    - Explore-exploit dilemme
  - **Obvious question:**
    - If you are exploring, should you update the probabilities at state where you are exploring
      - No right answer

Markov Decision Process

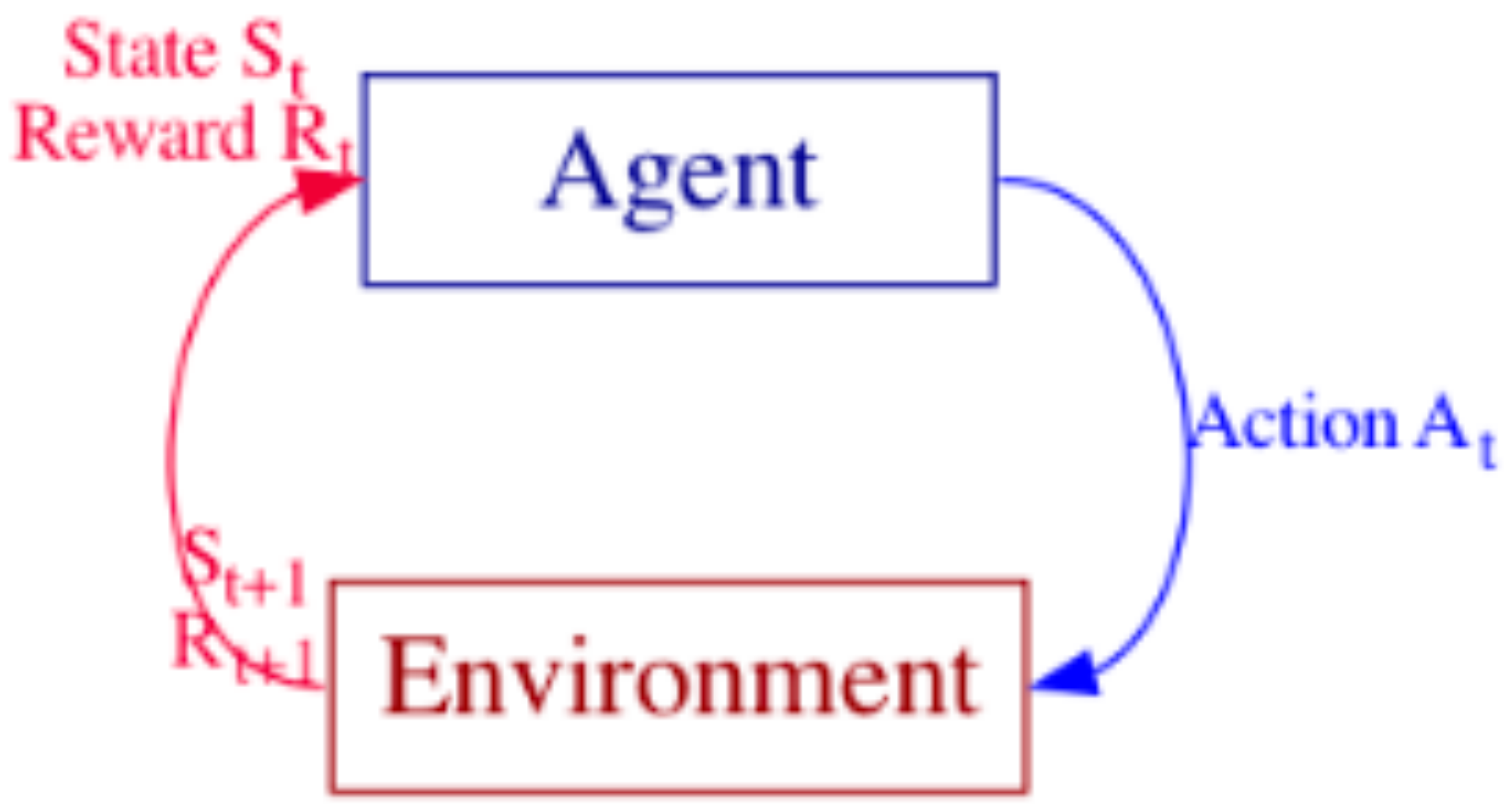
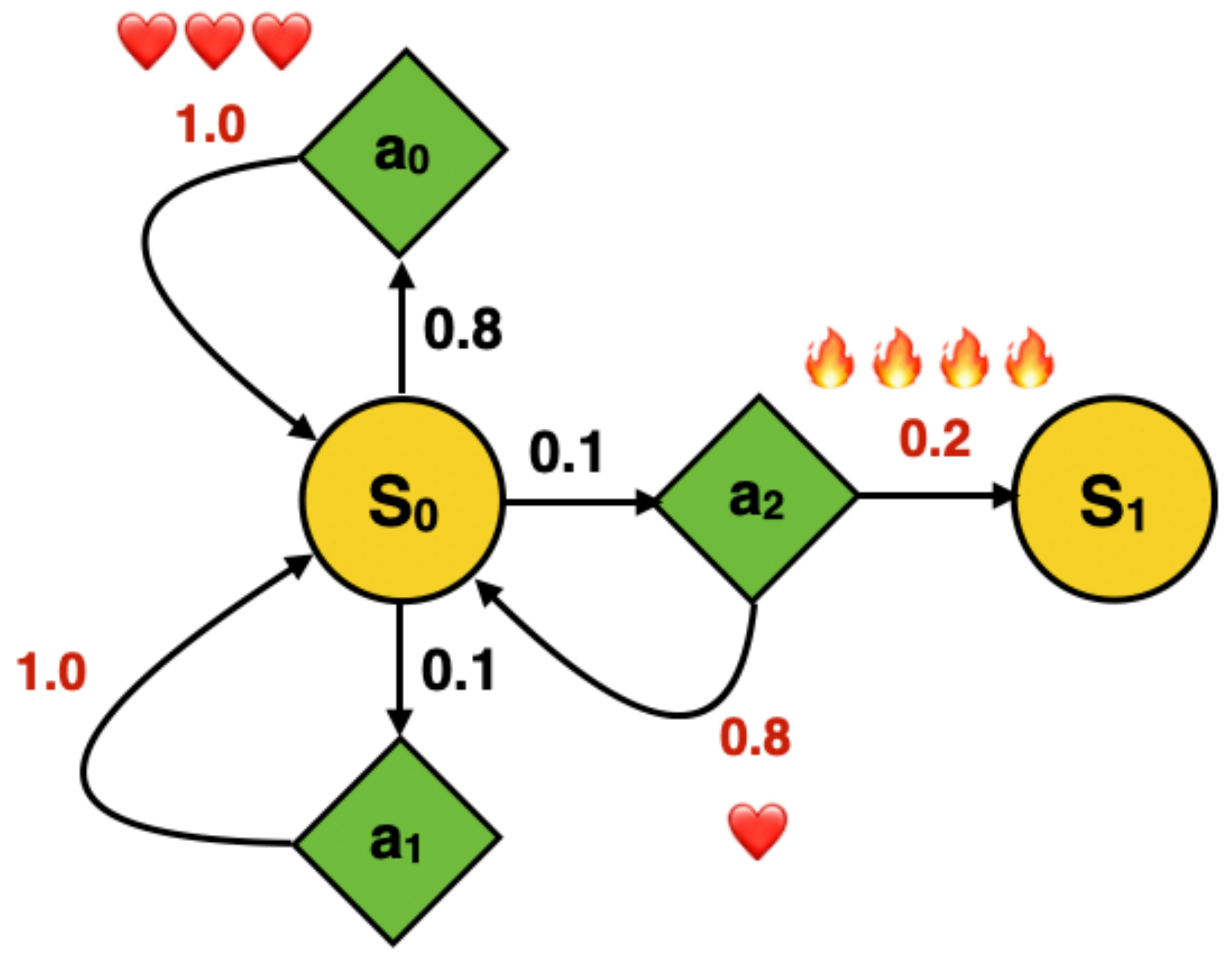


Figure 1.3.: The MDP Framework



Markov Decision Process

- **MDP**
  - The class of problems RL aims to solve can be described with a simple yet powerful mathematical framework known as Markov Decision Processes
  - State is assumed to have the **Markov Property**, which means:
    - The next State/Reward depends only on Current State (for a given Action)
    - The current State encapsulates all relevant information from the history of the interaction between the Agent and the Environment
    - The current State is a sufficient statistic of the future (for a given Action)
  - The goal of the Agent at any point in time is to maximize the Expected Sum of all future Rewards by controlling (at each time step) the Action as a function of the observed State (at that time step).
  - This function from a State to Action at any time step is known as the **Policy** function. So we say that the agent's job is exercise control by determining the **Optimal Policy function**

$$S_t \in S$$
$$\mathcal{A}_t \in \mathcal{A}$$
$$R_t \in \mathscr{R}$$

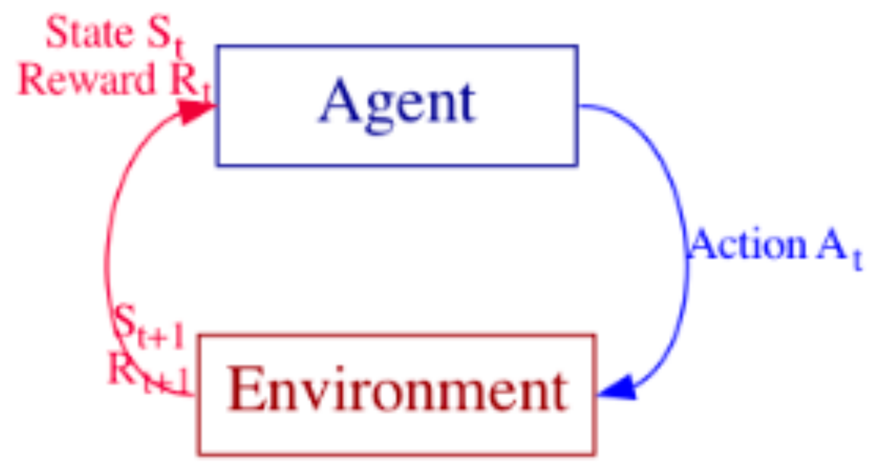
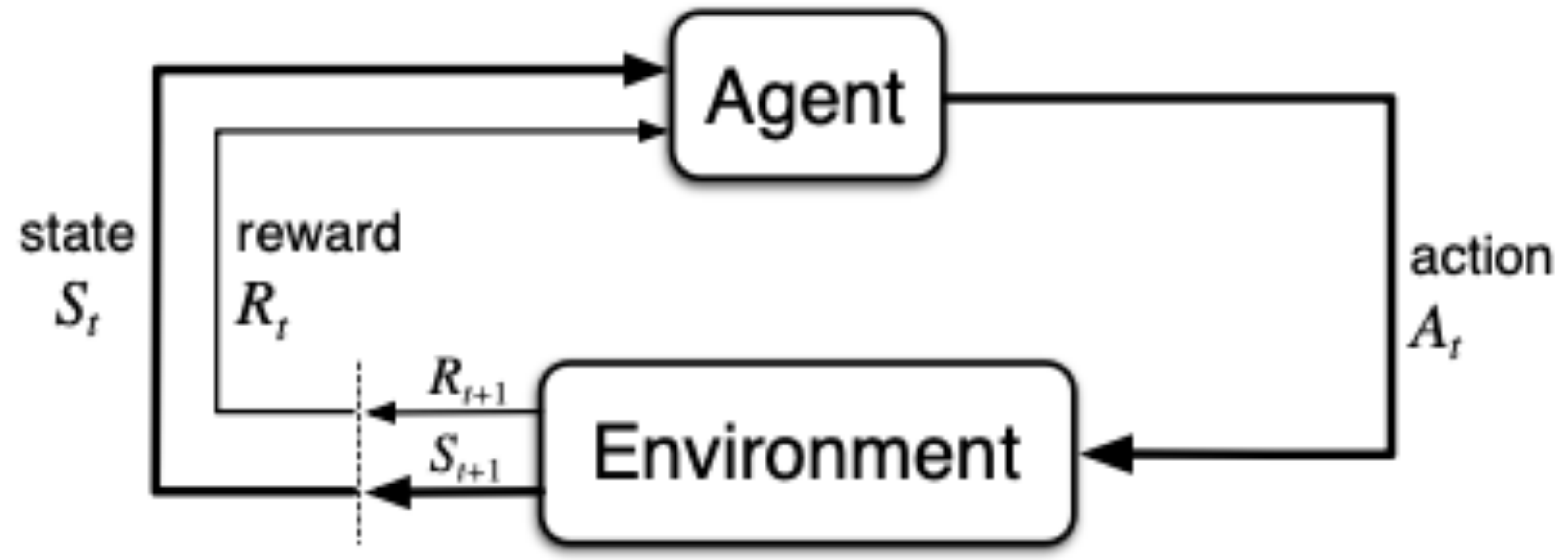


Figure 1.3.: The MDP Framework

$$p(r, s'|s, a) = \mathrm{P}(R_{t+1} = r, S_{t+1} = s'|S_t = s, A_t = a)$$
$$G_t = R_{t+1} + \gamma R_{t+1} + \gamma^2 R_{t+3} + \ldots$$
$$\gamma \in [0, 1]$$
$$\pi : S \to \mathcal{A} \qquad \mathrm{E}[G_t|S_t = s] \text{ for all } s \in S$$

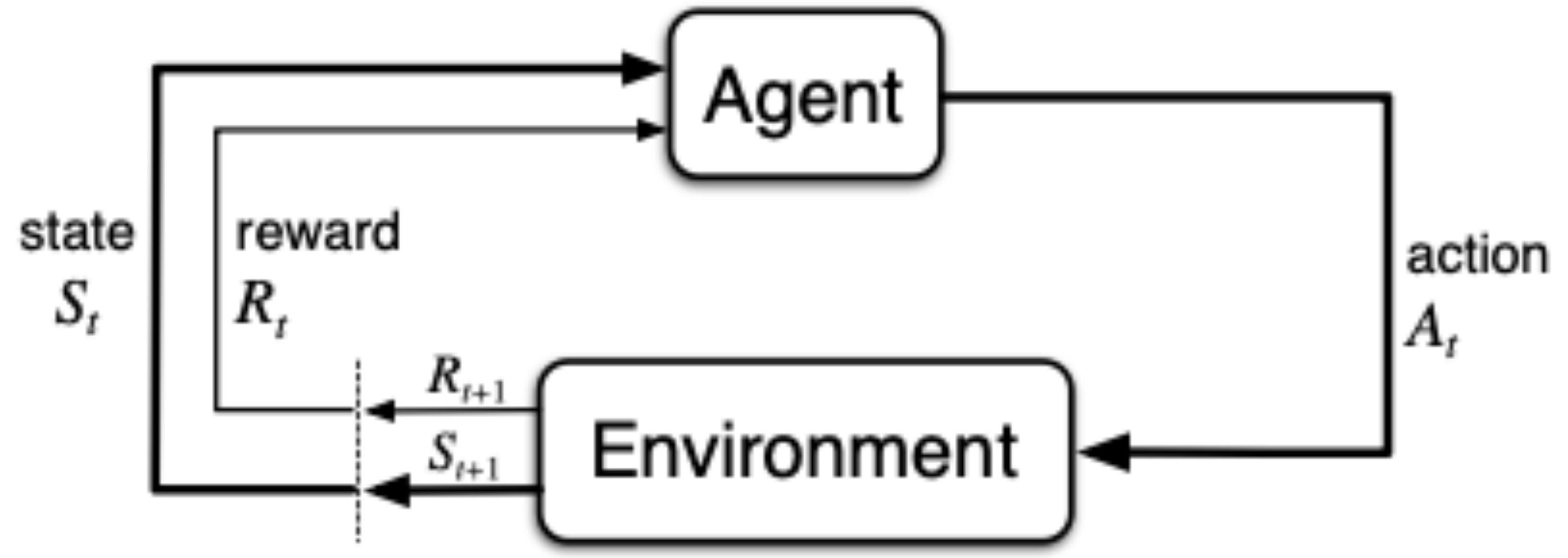
Difficulties in Solving MDP

- The challenge is simply that the State Space is very large (involving many variables) or complex (elaborate data structure), and hence, is computationally intractable. Like- wise, sometimes the Action Space can be quite large or complete
- No direct feedback on what the “correct” Action is for a given State
- The linkage between Actions and Rewards is further complicated by the fact that there is time-sequenced complexity in an MDP, meaning an Action can influence future States, which in turn influences future Actions.
- Agent often doesn’t know the model of the environment. By model, we are referring to the probabilities of state-transitions and rewards (the function p).
    - This means the Agent has to simultaneously learn the model (from the real-world data stream) and solve for the optimal policy.
- Agent has to find the balance between exploitation (retrying actions that have yielded good rewards so far) and exploration (trying actions that have either not been tried enough or not been tried at all)



Markov Decision Process

- In MDP, the probabilities given by p completely characterizes the environment dynamics, that is, the probability of each possible value for S_t and R_t depends on the immediately preceding state and action — S_{t-1} and A_{t-1}, and given them, not at all on the earlier states
  - From p — one can compute anything else one might want to know about the environment, such as
    - the state-transition probabilities
    - Expected rewards for state-action pairs
    - Expected rewards for state-action-next state triples



$$S_t \in S$$
$$\mathcal{A}_t \in \mathcal{A}$$
$$R_t \in \mathcal{D}$$

$$p(r, s'|s, a) = \mathrm{P}(R_{t+1} = r, S_{t+1} = s'|S_t = s, A_t = a)$$

$$p(s'|s, a) = \mathrm{P}(S_t = s'|S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} p(s', r|s, a)$$

$$G_t = R_{t+1} + \gamma R_{t+1} + \gamma^2 R_{t+3} + \dots$$

$$r(s, a) = \mathrm{E}(R_t|S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a)$$

$$\gamma \in [0, 1]$$

$$\pi : S \to \mathcal{A}$$

$$r(s, a, s') = \mathrm{E}(R_t|S_{t-1} = 2, A_{t-1} = a, S_t = s') = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)}$$

$$\mathrm{E}[G_t|S_t = s] \text{ for all } s \in S$$

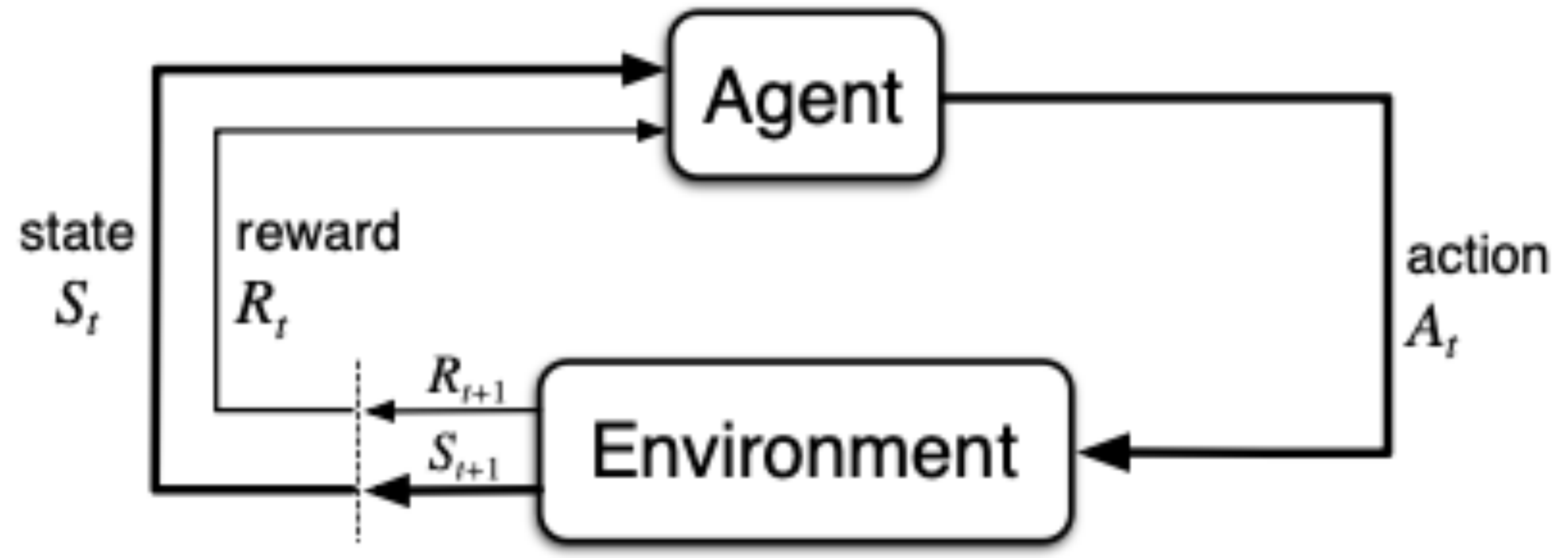
Solution of an MDP Problem

- The purpose or goal of an agent in an MDP is to maximise the total amount of reward it receives.
- This means maximizing not immediate rewards but cumulative rewards in the long term.
- **Return:**

$$G_t = R_{t+1} + R_{t+1} + \ldots + R_N$$

- **Episodic Tasks**
- **Continuing Taks**
- **Discounting:**
  -

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \ldots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma^1 R_{t+3} + \gamma^2 R_{t+2} + \ldots) \\ &= R_{t+1} + \gamma(G_{t+1}) \end{aligned}$$



Policies and Value Functions

- **What is a Policy?**
  - A policy is a mapping from states to probabilities of selecting each possible action

$$\pi(a \mid s)$$

- **Value Function:**
  - The value function of a state s under policy π denoted as vπ(s), is the expected return when starting in s and following π thereafter.
  - vπ(s) is called the **state-value function** for policy π
  - We define the value of taking action a in state s under a policy π, denoted as qπ(s,a), as the expected return starting from s, taking action a, and thereafter following policy π.
  - qπ is the **state-action-value function** for the policy π.

$$v_\pi(s) = \mathrm{E}_\pi[G_t | S_t = s]$$

$$q_\pi(s, a) = \mathrm{E}_\pi[G_t | S_t = s, A_t = a]$$

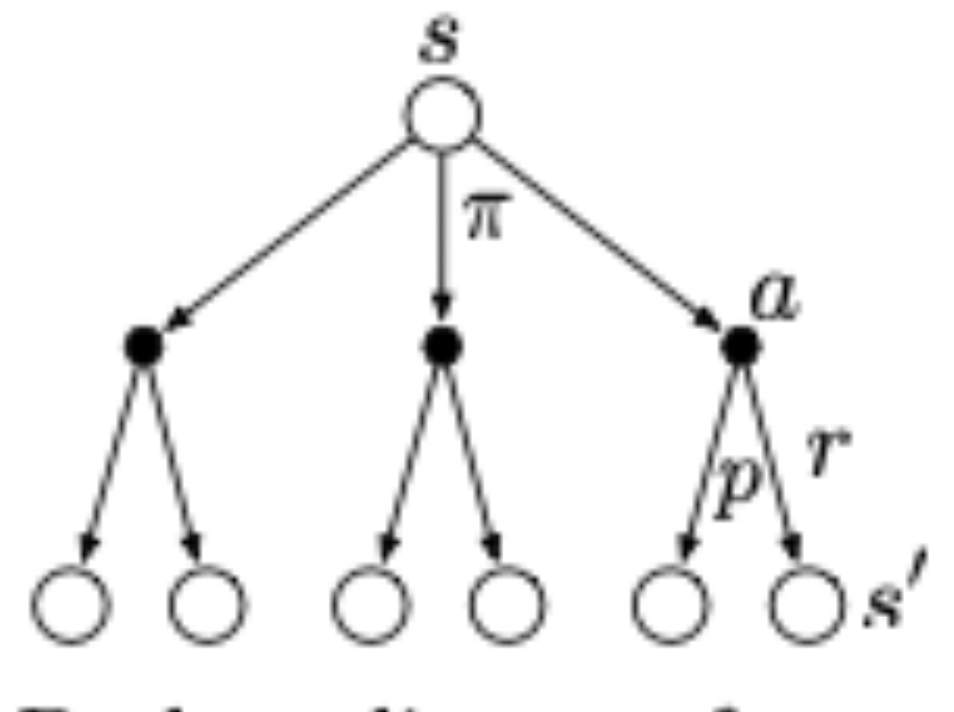
Policies and Value Functions

$\pi(a|s)$

$$v_\pi(s) = \mathrm{E}_\pi[G_t | S_t = s]$$

$$q_\pi(s,a) = \mathrm{E}_\pi[G_t | S_t = s, A_t = a]$$

- **How to compute vπ(s) and qπ(s,a)?**
  - Estimated from experience
  - For example, if an agent follow policy pi and maintain an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state's value, as the number of times that state is encountered approaches infinity
  - Such methods are called **Monte Carlo Methods**, because the involve averaging over many random samples of actual returns



Backup diagram for v_π

- **Fundamental Property of Value Functions**
  - Satisfy recursive relationship

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S}, \end{aligned}$$

Bellman Equation for vπ

$$\begin{aligned} q_\pi(s,a) &= \mathrm{E}_\pi[G_t | S_t = s, A_t = a] \\ &= \mathrm{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s,a)[r + \gamma q_\pi(s', a) | S_t = s, A_t = a] \end{aligned}$$

Bellman Equation for qπ(s,a)

Optimal Policies and Optimal Value Functions

$$\pi(a \mid s)$$

$$v_\pi(s) = \mathrm{E}_\pi[G_t | S_t = s]$$

$$q_\pi(s,a) = \mathrm{E}_\pi[G_t | S_t = s, A_t = a]$$

- **Solving an MDP / RL task means, roughly, finding a policy that achieves a lot of reward over the long run**
  - Aka optimal Policy

For all states (s)

$$v_*(s) = \max_\pi v_\pi(s)$$

For all states (s) and actions (a)

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

Bellman Optimality Equations

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma v_*(s')\big]. \end{aligned}$$

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1},a') \;\Big|\; S_t = s, A_t = a\Big] \\ &= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \max_{a'} q_*(s',a')\Big]. \end{aligned}$$

Optimal Policies and Optimal Value Functions

$$\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']\Big] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big], \quad \text{for all } s \in \mathcal{S},
\end{aligned}$$

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s', r} p(s', r \mid s, a)\big[r + \gamma v_*(s')\big].
\end{aligned}$$

$$\begin{aligned}
q_\pi(s, a) &= \mathrm{E}_\pi[G_t | S_t = s, A_t = a] \\
&= \mathrm{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a)[r + \gamma q_\pi(s', a) | S_t = s, A_t = a]
\end{aligned}$$

$$\begin{aligned}
q_*(s, a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \\
&= \sum_{s', r} p(s', r \mid s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big].
\end{aligned}$$

Prediction vs. Control

- **Prediction (or Evaluation):**
  - The problem of calculating $V^{\pi}(s)$ (Value Function for a given policy) is known as the **Prediction** problem (since this amounts to statistical estimation of the expected returns from any given state when following a policy π).
- **Control:**
  - The problem of calculating the Optimal Value Function V^* (and hence, optimal Policy π), is known as the **Control** problem (since this requires steering of the policy such that we obtain the maximum expected return from any state).
- Solving the Prediction problem is typically a stepping stone toward solving the (harder) problem of Control.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\Big] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big], \quad \text{for all } s \in \mathcal{S}, \end{aligned}$$

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r|s, a)\big[r + \gamma v_*(s')\big]. \end{aligned}$$

Planning vs. Learning

- **Planning**
  - Dynamic Programming Algorithms — assume that the agent knows of the transition probabilities p and the algorithm takes advantage of the knowledge of those probabilities (leveraging the Bellman Equation to efficiently calculate the Value Function)
  - DP algorithms are considered to be Planning and not Learning (in the language of A.I.) because the algorithm doesn't need to interact with the Environment and doesn't need to learn from the (states, rewards) data stream coming from the Environment
  - Rather, armed with the transition probabilities, the algorithm can reason about future probabilistic outcomes and perform the requisite optimization calculation to infer the Optimal Policy
    - So it plans its path to success, rather than learning about how to succeed

- **Learning**
  - Reinforcement Learning Algorithms
  - In typical real-world situations, one doesn't really know the transition probabilities p. This is the realm of Reinforcement Learning (RL)
  - RL algorithms interact with the Environment, learn with each new (state, reward) pair received from the Environment, and incrementally figure out the Optimal Value Function (with the "trial and error" approach that we outlined earlier)
  - However, note that the Environment interaction could be real interaction or simulated interaction

Road Ahead

- **Tabular Solution Methods**

  - All the core ideas of reinforcement learning algorithms in their simplest forms
    - that in which the state and action spaces are small enough for the approximate value functions to be represented as arrays, or tables.
    - In this case, the methods can often find exact solutions, that is, they can often find exactly the optimal value function and the optimal policy.
    - This contrasts with the approximate methods, which
      - only find approximate solutions,
      - but which in return can be applied effectively to much larger problems

Dynamic Programming	😀	**Mathematically well-developed**
	☹️	**Requires Model (Transition Probabilities)**
Monte Carlo Methods	😀	**No Model is required**
	☹️	**Not suited for step-by-step incremental computation**
TD Learning	😀	**No Model is required**
	😀	**Suited for step-by-step incremental computation**
	☹️	**Much complex to analyze**

Dynamic Programming

- Two step algorithm:
  - **Policy Evaluation (Prediction)**
    - How to compute the state-value function v(s) and q(s,a) for any arbitrary policy π

Strategy 1

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big], \end{aligned}$$

Strategy 2

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

  - **Policy Improvement (Control)**
    - Can policy π be improved? And, if yes, how can we modify it
    - Just be greedy to existing policy

$$\pi'(s) \doteq \arg\max_a q_\pi(s,a)$$

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

Dynamic Programming

- Two step algorithm:
  - **Policy Evaluation (Prediction)**
    - How to compute the state-value function v(s) and q(s,a) for any arbitrary policy π
  - **Policy Improvement (Control)**
    - Can policy π be improved? And, if yes, how can we modify it
    - Just be greedy to existing policy
    - The greedy policy takes the action that looks best in the short term—after one step of lookahead—according to v_π.
    - Greedy policy meets the conditions of the **policy improvement theorem**, so we know that it is as good as, or better than, the original policy.
    - The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called policy improvement.

Strategy 1

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big], \end{aligned}$$

Strategy 2

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

$$\pi'(s) \doteq \operatorname*{argmax}_a q_\pi(s,a)$$

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

Dynamic Programming

- **Policy Iteration:**
  - Repeat Policy Evaluation and Policy Improvement for some time until convergence

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

- **Value Iteration Algorithm**
  - Combines policy improvement and policy evaluation steps

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

- **Value Iteration is great, but we still need the transition probabilities, which in many (if not all) cases, is next to impossible to obtain**

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

+

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

Monte Carlo Method

- First learning methods for estimating value functions and discovering optimal policies.
- Do not assume complete knowledge of the environment
- Monte Carlo methods require only experience
  - sample sequences of states, actions, and rewards from **actual** or **simulated** interaction with an environment
  - **Actual Experience:**
    - Learning from actual experience is striking because it requires no prior knowledge of the environment's dynamics, yet can still attain optimal behaviour.
  - **Simulated Experience:**
    - Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming (DP).
    - In surprisingly many cases it is easy to generate experience sampled according to the desired probability distributions, but infeasible to obtain the distributions in explicit form.

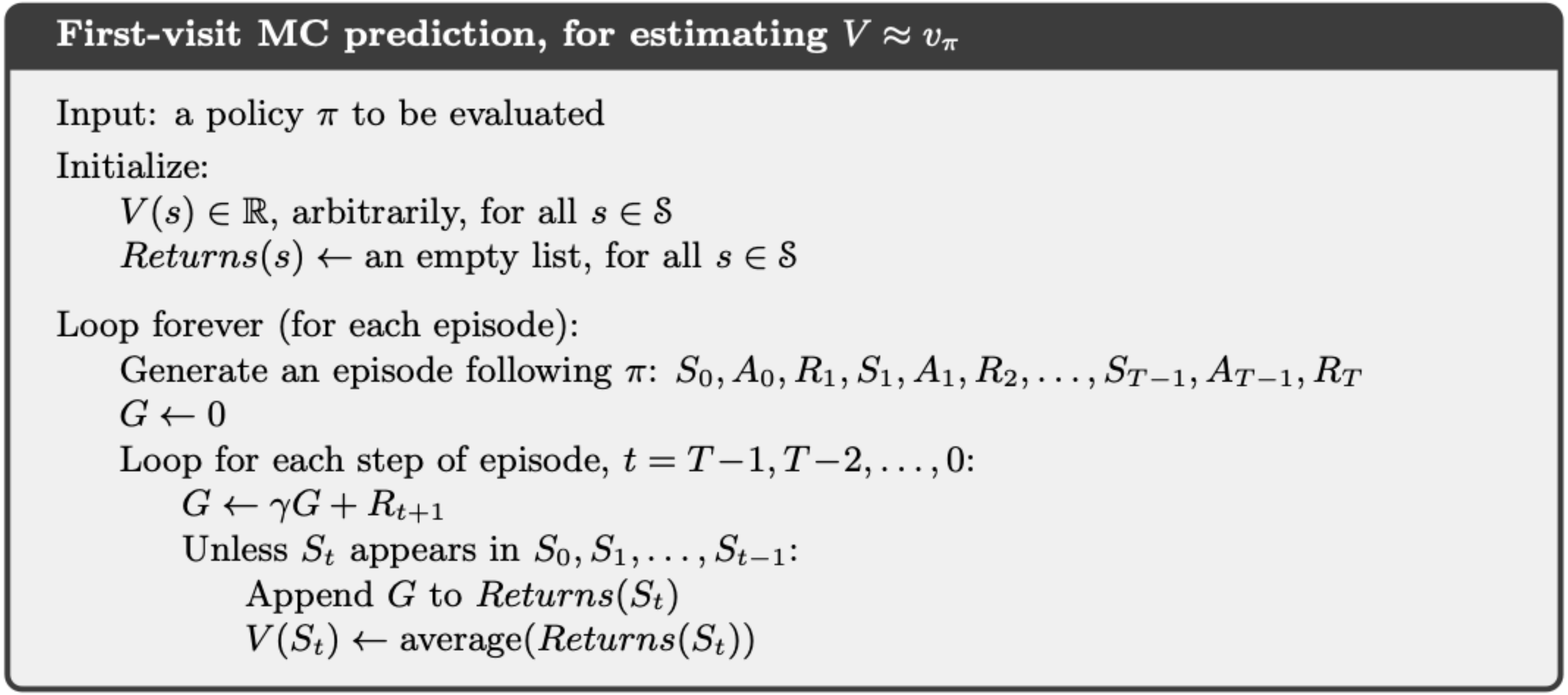
- **Issue 1:**
  - We constrain ourselves to episodic tasks:
  - We assume experience is divided into episodes
  - All episodes eventually terminate no matter what actions are selected.
  - Only on the completion of an episode are value estimates and policies changed
- **Issue 2:**
  - Primary goal of Monte Carlo Method is to estimate state-action q*(S,A) rather than v*(S)

Monte Carlo Evaluation

- Similar to value iteration algorithm that we discussed in Dynamic Programming, but now we do it for **episodes**
- The only complication is that many state–action pairs may never be visited.
- If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state.
- With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience.
- This is a serious problem because the purpose of learning action values is to help in choosing among the actions available in each state.
- To compare alternatives we need to estimate the value of all the actions from each state, not just the one we currently favour

- **Exploring Starts:**
  - For policy evaluation to work for action values, we must assure continual exploration.
  - One way to do this is by specifying that the episodes start in a state–action pair, and that every pair has a nonzero probability of being selected as the start.
  - This guarantees that all state–action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. We call this the assumption of exploring starts.

Monte Carlo Evaluation Algorithm



First-visit MC prediction, for estimating $V \approx v_\pi$

```
Input: a policy $\pi$ to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless $S_t$ appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append $G$ to $Returns(S_t)$
            $V(S_t) \leftarrow$ average($Returns(S_t)$)
```

Monte Carlo Control

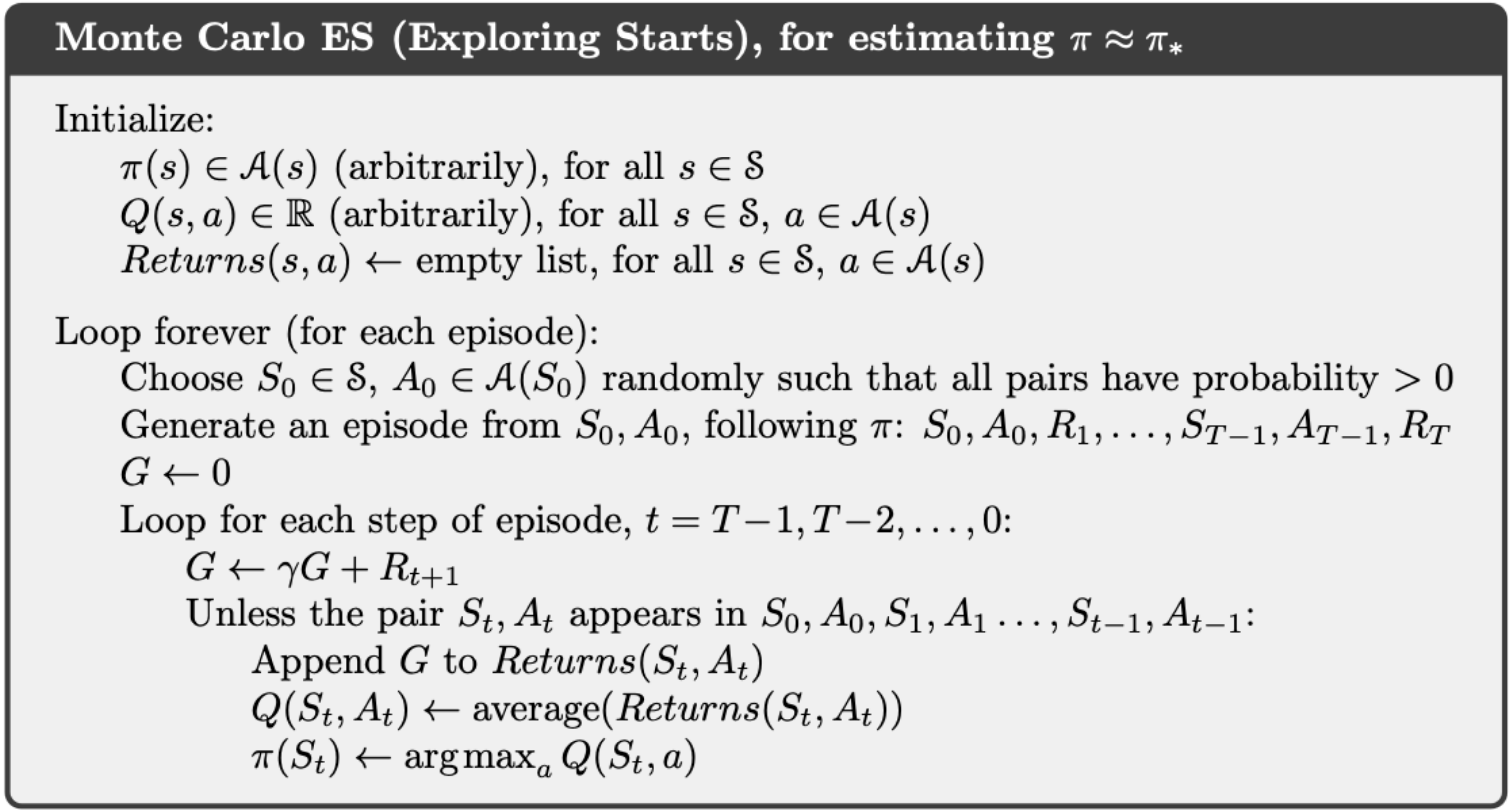
- Similar to value iteration algorithm that we discussed in Dynamic Programming
  - Act Greedy w.r.t the policy
- Then follow Policy Iteration
  - Repeat policy evaluation and policy control multiple times until convergence

$$\pi(s) \doteq \arg\max_a q(s,a).$$

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

- For Monte Carlo policy iteration it is natural to alternate between evaluation and improvement on an episode-by-episode basis.
  - After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode.

Monte Carlo Evaluation + Control



Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

```
Initialize:
    $\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
    Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability $> 0$
    Generate an episode from $S_0, A_0$, following $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow$ average($Returns(S_t, A_t)$)
            $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$
```

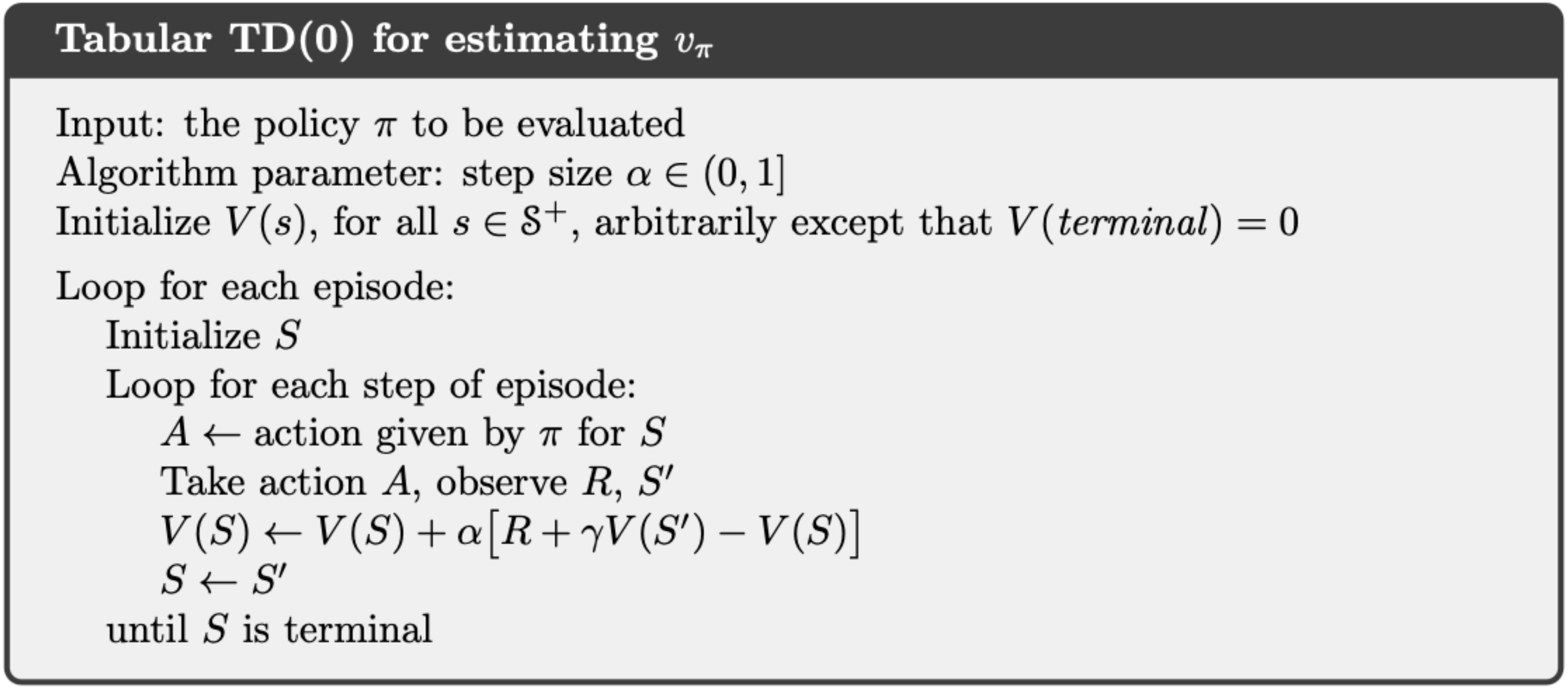
Temporal Difference Learning

- Both TD and Monte Carlo methods use experience to solve the prediction problem.
- TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas.
- Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics.
- Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).
- Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V\pi(S_t)$:
- TD methods need to wait only until the next time step. At time t+1 they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V\pi(S_{t+1})$.

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big],$$

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$$

TD Evaluation Algorithm

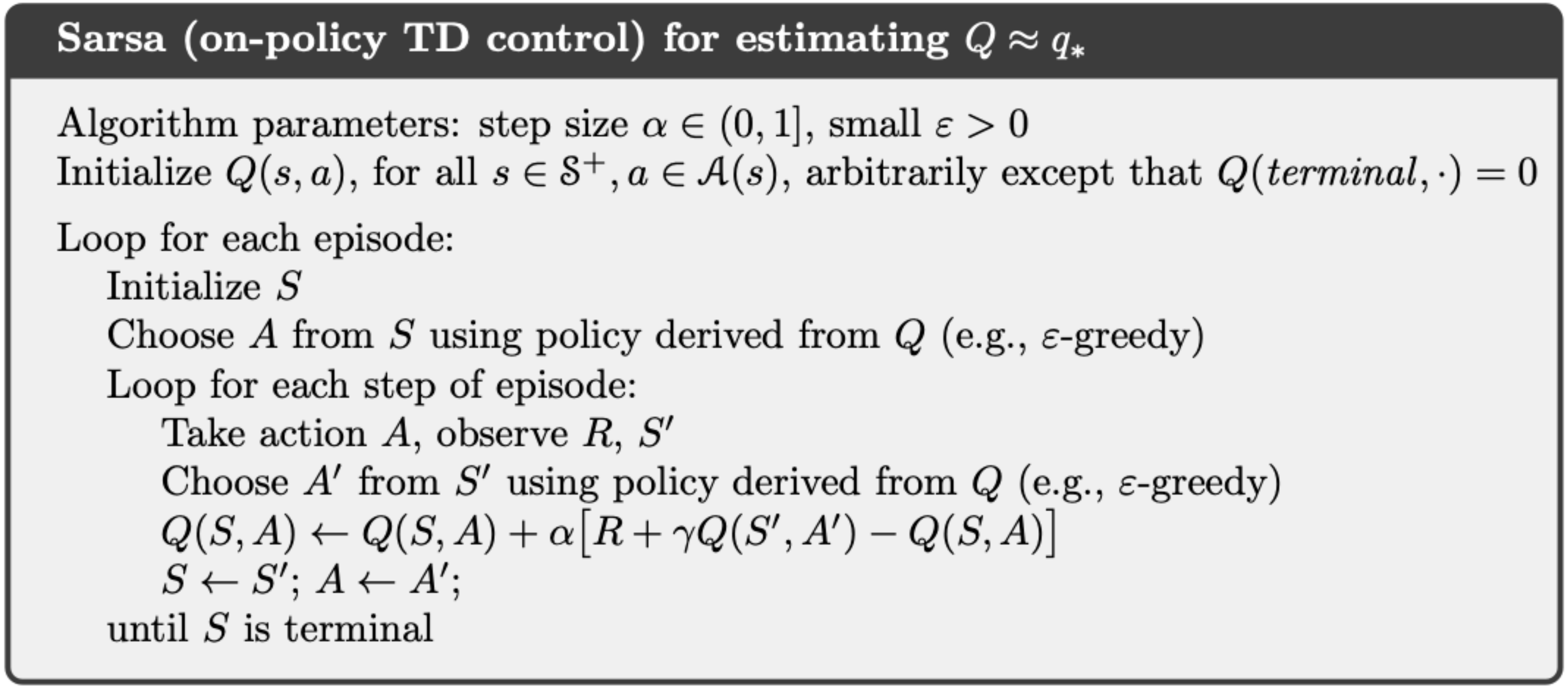


Tabular TD(0) for estimating v_π

```
Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

TD Control

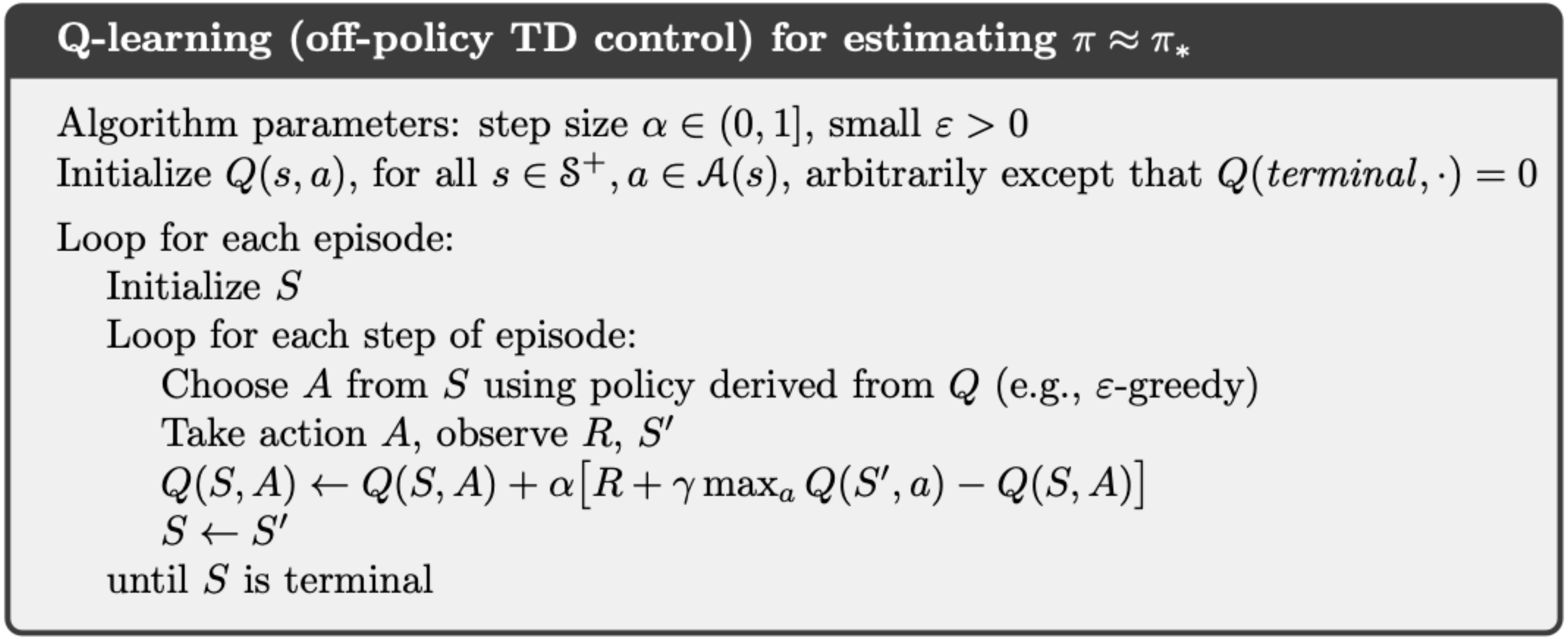


Sarsa (on-policy TD control) for estimating $Q \approx q_*$

```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-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
```

TD Evaluation + Control



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

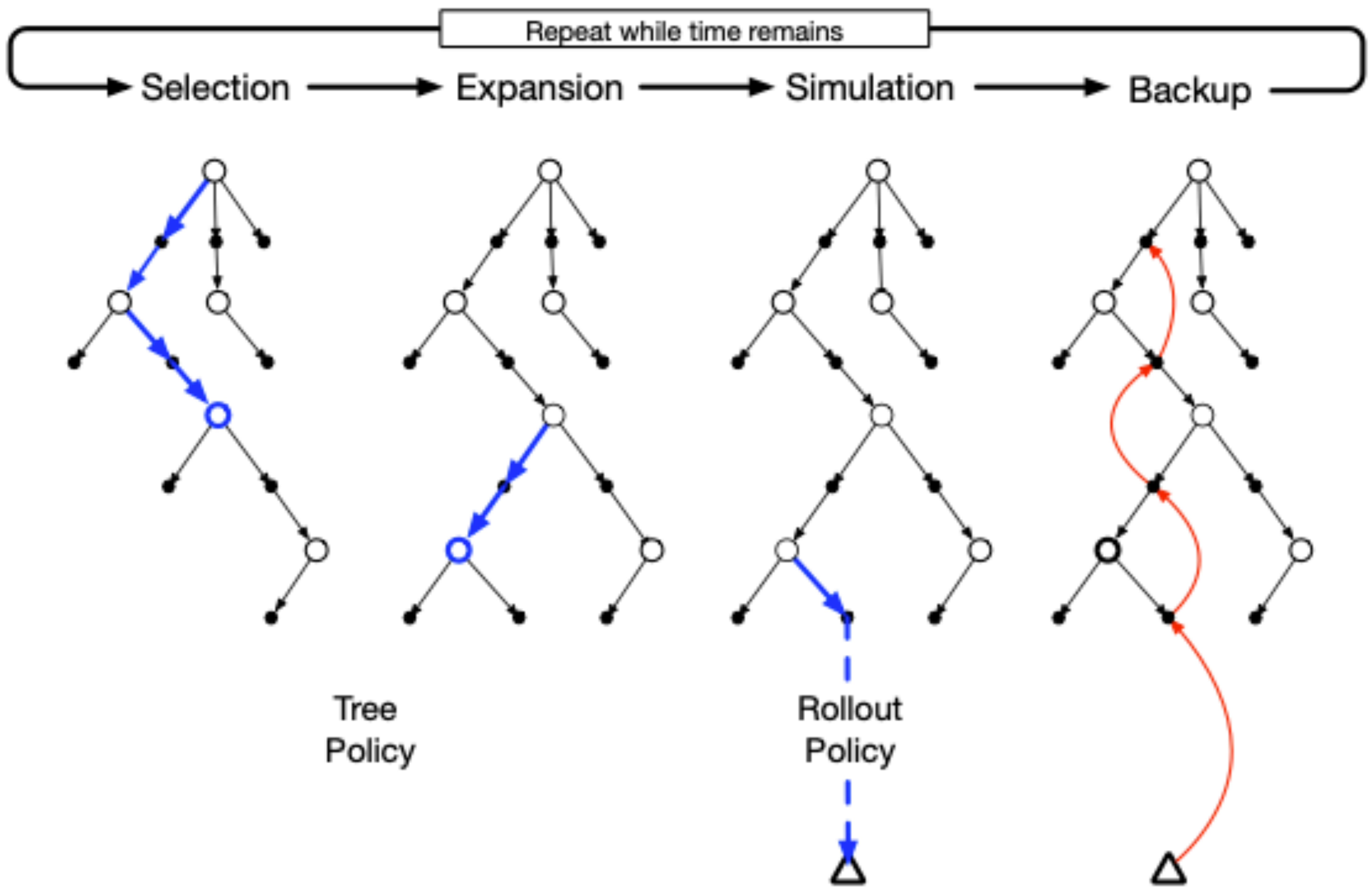
```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-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
```

Decision Time Planning

- Decision Time Planning
  - Heuristic Search
    - Idea of backing-upThe backing up stops at the state–action nodes for the current state
    - Once the backed-up values of these nodes are computed, the best of them is chosen as the current action, and then all backed-up values are discarded
  - Rollout Algorithms
    - Based on Monte Carlo Control applied to simulated trajectories that all begin at the current state
    - Estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy
    - **Hang on? How do we compute the trajectories in the first place? Do we have the model of the environment?**

Monte Carlo Tree Search



Summary