整合学习与规划（Integrating Learning and Planning）

吉建民

USTC
jianmin@ustc.edu.cn

2023 年 11 月 13 日

Used Materials

Disclaimer: 本课件大量采用了 Rich Sutton's RL class, David Silver's Deep RL tutorial 和其他网络课程课件，也采用了 GitHub 中开源代码，以及部分网络博客内容

Table of Contents

Policy Gradient

- 参数化 Policy $\pi_\theta(s,a) = P[a \mid s, \theta]$，定义目标函数 $J(\theta)$ 为策略 π_θ 的期望回报，再梯度上升计算 $\theta^* = argmax_\theta J(\theta)$，从而得到最优的策略 π_{θ^*}

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

- 策略梯度定理（Policy Gradient Theorem）

$$\nabla_\theta J(\theta) = E_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)\right]$$

- Score function: $\nabla_\theta \log \pi_\theta(s,a)$
- REINFORCE, Monte Carlo Policy Gradient

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi_\theta(s,t)$$

Actor-Critic Policy Gradient

- Actor-Critic Policy Gradient

$$\Delta\theta = \alpha Q_w(s,a)\nabla_\theta \log \pi_\theta(s,a)$$

- Actor-Critic Policy Gradient with Baseline

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$$

$$\Delta\theta = \alpha A^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(s,a)$$

  - For MC, $A^{\pi_\theta}(s,a) \approx G_t - V_v(s)$
  - For TD(0),

$$\delta_t = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

$$A^{\pi_\theta}(s,a) \approx \delta_t$$

  - For forward-view TD(λ), $A^{\pi_\theta}(s,a) \approx G_t^\lambda - V_v(s)$
  - For backward-view TD(λ),

$$\delta_t = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

$$e_t = \gamma\lambda e_{t-1} + \mathbf{1}(s,a)$$

$$A^{\pi_\theta}(s,a) \approx \delta_t e_t$$

Table of Contents

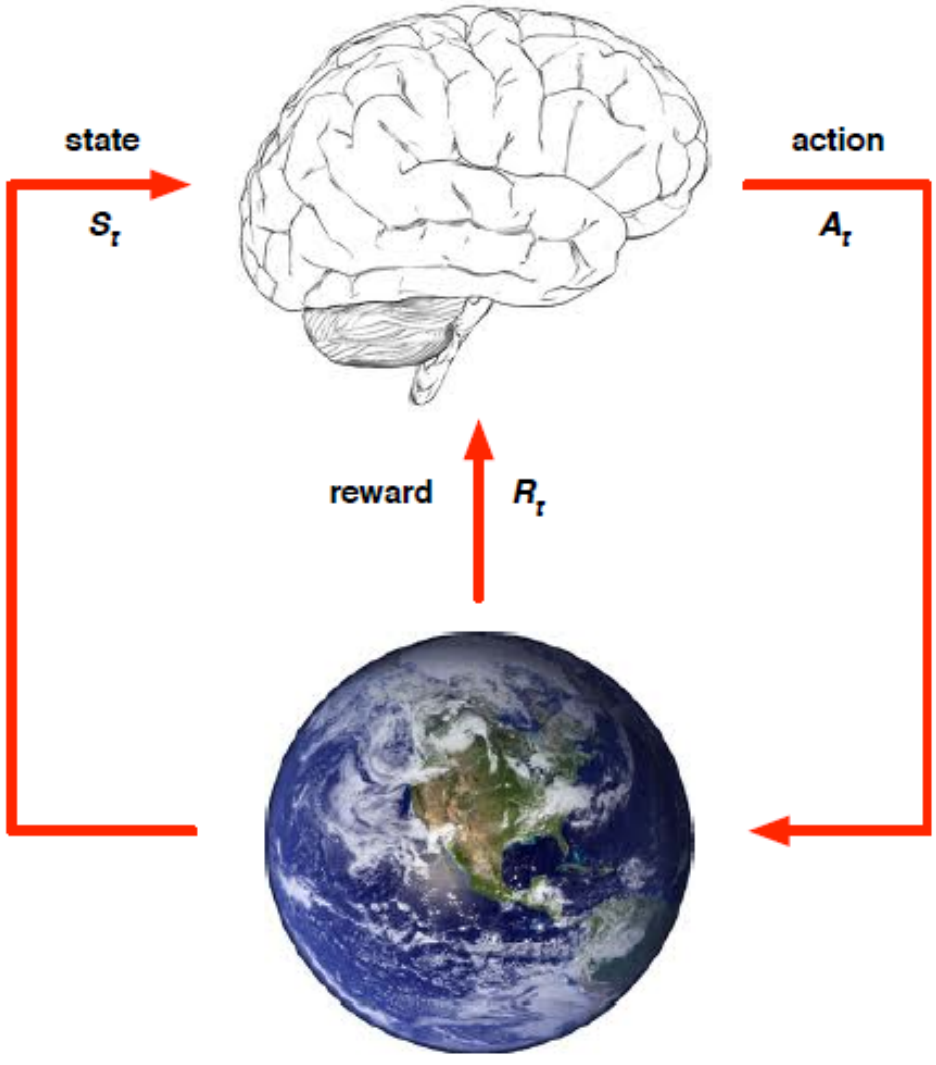
简介

- 本章主要介绍如何从经历中直接学习模型，如何构建一个模型，如何基于模型来进行“规划”，在此基础上将“学习”和“规划”整合起来
- 依赖于模型，个体可以通过模拟产生一系列虚拟的 Episodes，通过使用基于模拟的搜索方法，特别是蒙特卡罗树搜索方法，找到了一条解决诸如围棋等大规模 MDP 问题的有效可行的算法
- 解决这类问题的关键在于“前向搜索”和“采样”，通过将基于模拟的前向搜索与各种不依赖模型的强化学习算法结合，衍生出多个用来解决类似大规模问题的切实可行的算法

Model-Free and Model-Based RL

- Model-Free RL
  - No model
  - Learn value function (and/or policy) from experience
- Model-Based RL
  - Learn a model from experience
  - Plan value function (and/or policy) from model

Model-Free RL



Model-Based RL

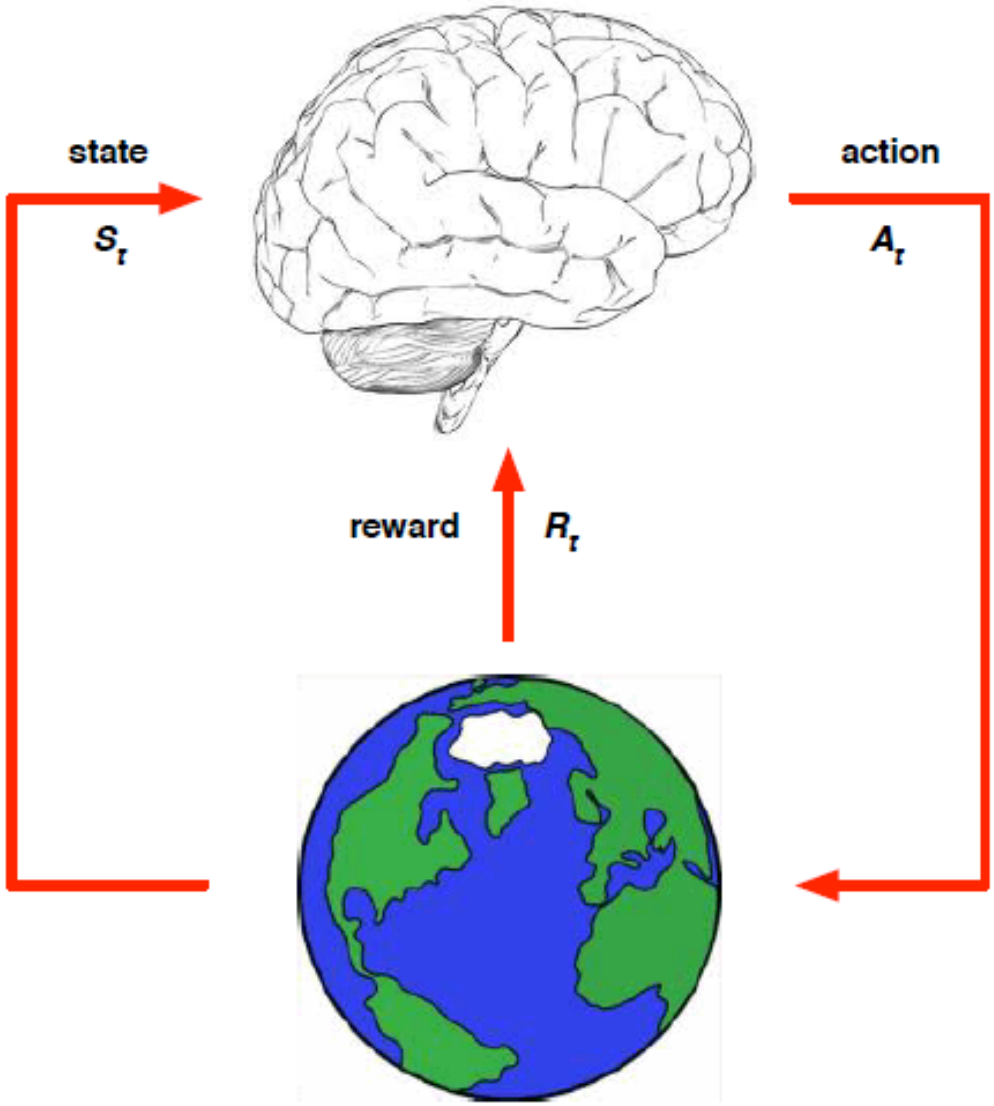
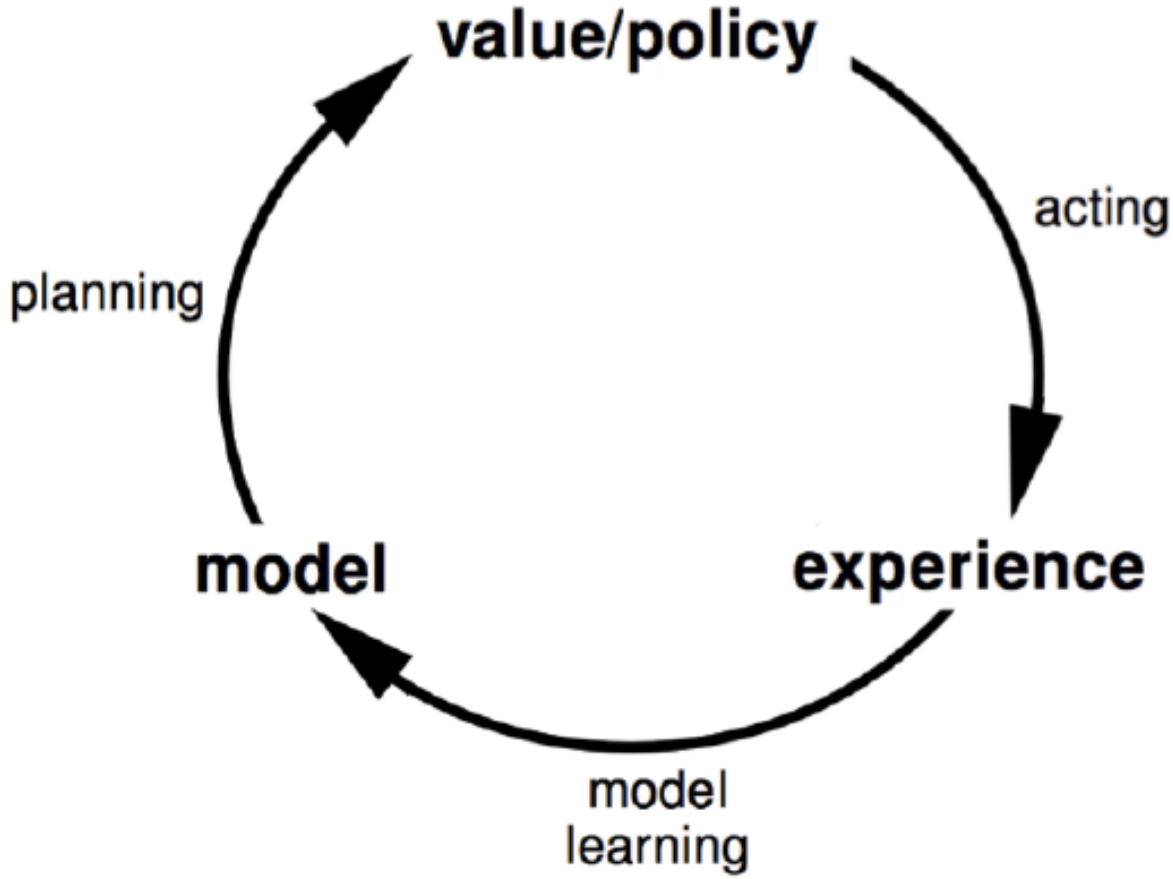


Table of Contents

基于模型的强化学习

模型学习在整个 RL 学习中的位置和作用：



基于模型的强化学习

- 基于模型学习的优点:
  - 当学习价值函数或策略变得很困难的时候，学习模型可能是一条不错的途径，像下棋这类活动，模型是相对直接的，相当于就是游戏规则
  - 可以使用监督式学习方法来获得模型
  - 模型能够从一个方面对个体理解环境提供帮助，通过模型，个体不再局限于如何最大化奖励本身，还能在一定程度上了解采取的动作为什么是好的或者不好，也就是具备一定的推理能力
- 基于模型学习的缺点:
  - 在学习模型的过程中，模型其实是一个个体对环境运行机制的描述，不完全是真实的环境运行机制，因此存在这一定程度的近似。另外当使用一个近似的模型去进行价值函数或策略函数的学习时，又会引入一次近似
  - 会带来双重的近似误差，有时候这会带来较为严重的错误

模型的数学描述

- 模型 M 是一个 MDP $\langle S, A, P, R \rangle$ 的参数化 (η) 表现形式
- 其中假定：状态空间 S 和行为空间 A 是已知的
- 模型 $M = \langle P_\eta, R_\eta \rangle$ 呈现了状态转移 $P_\eta \approx P$ 和奖励 $R_\eta \approx R$:

$$S_{t+1} \sim P_\eta(S_{t+1} \mid S_t, A_t)$$
$$R_{t+1} = R_\eta(R_{t+1} \mid S_t, A_t)$$

- 通常我们需要假设，状态转移函数和奖励函数是条件独立的：

$$P[S_{t+1}, R_{t+1} \mid S_t, A_t] = P[S_{t+1} \mid S_t, A_t] P[R_{t+1} \mid S_t, A_t]$$

用监督式学习来构建模型

- 目标：从经历 $\{S_1, A_1, R_2, ..., S_T\}$ 中估计一个模型 M_η
- 监督式学习的训练集：(X, y)

$$
\begin{aligned}
S_1, A_1 &\rightarrow R_2, S_2 \\
S_2, A_2 &\rightarrow R_3, S_3 \\
&\vdots \\
S_{T-1}, A_{T-1} &\rightarrow R_T, S_T
\end{aligned}
$$

- 从 s, a 学习 r 的过程是一个回归问题 (regression problem)
- 从 s, a 学习 s' 的过程是一个密度估计问题 (density estimation problem)
- 选择一个损失函数，比如均方差，KL 散度（相对熵）等，优化参数 η 来最小化经验损失 (empirical loss)。所有监督学习相关的算法都可以用来解决上述两个问题

训练模型

- 根据使用的算法不同，可以有如下多种模型：
  - 查表式 (Table lookup Model)
  - 线性期望模型 (Linear Expectation Model)
  - 线性高斯模型 (Linear Gaussian Model)
  - 高斯决策模型 (Gaussian Process Model)
  - 深信度神经网络模型 (Deep Belief Network Model)
  - ...
- 下面以查表模型来解释模型的构建

查表模型

- 通过经历得到各状态行为转移概率和奖励，把这些数据存入表中，使用时直接检索
- 状态转移概率和奖励计算方法如下

$$\hat{P}^a_{s,s'} = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbf{1}(S_t, A_t, S_{t+1} = s, a, s')$$

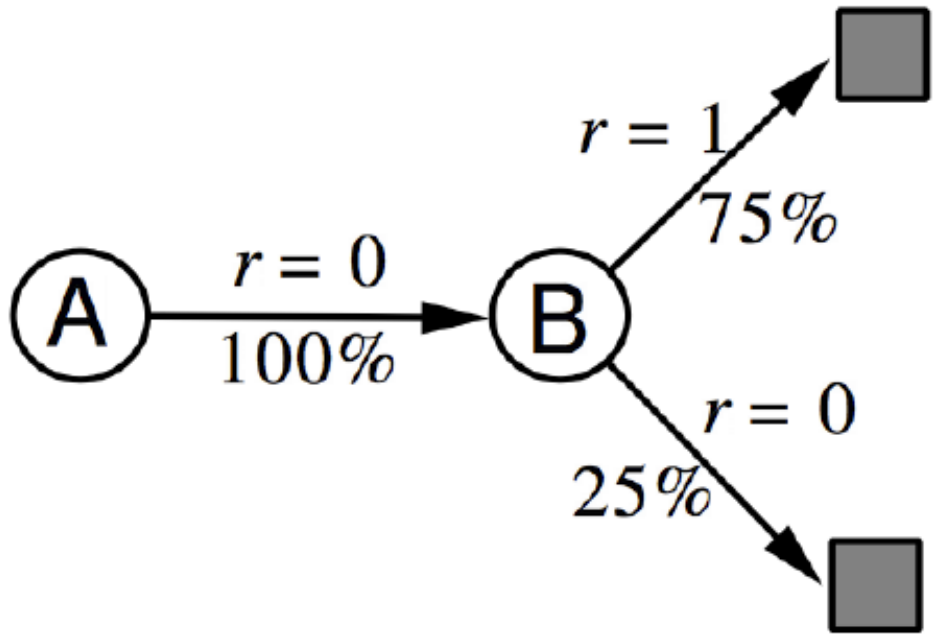
$$\hat{R}^a_s = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbf{1}(S_t, A_t = s, a) R_t$$

- 不过实际应用时，略有差别
  - 在学习的每一步（time-step），记录如下的状态转换（经历片段）：$\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$
  - 从模型采样构建虚拟经历时，从这些 tuples 中随机选择符合 $\langle s, a, \ldots \rangle$ 的一个 tuple，提供给个体进行价值或策略函数的更新
  - 这里的随机选择就体现了状态 s 的各种后续状态的概率分布

AB Example

Two states A, B; no discounting; 8 episodes of experience

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0



这相当于我们利用经历构建了一个查表模型。随后我们可以使用这个模型，进行虚拟采样，从而进行“规划”

利用模型进行“规划”

规划过程就是一个 MDP 的过程

- 给定一个模型 $M_\eta = \langle P_\eta, R_\eta \rangle$
- 求解 MDP $\langle S, A, P_\eta, R_\eta \rangle$，即找到基于该模型的最优价值函数或最优策略
- 多种规划算法：
  - 值迭代
  - 策略迭代
  - 树搜索方法（Tree Search）
  - ...

基于采样的规划（Sample-Based Planning）

- 基于学习得到的模型 M_η，可以用 DP 方法求解，也可以用 Sample-Based Planning
- 模型 M_η 只用来提供 sample（虚拟采样）

$$S_{t+1} \sim P_\eta(S_{t+1} \mid S_t, A_t)$$
$$R_{t+1} = R_\eta(R_{t+1} \mid S_t, A_t)$$

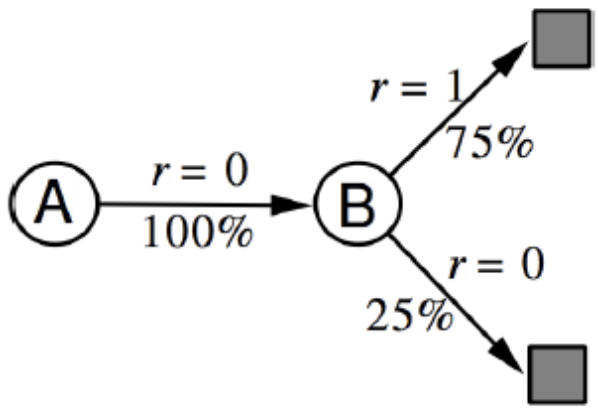
- 基于这些 sample 再采用 Model-free RL 来求解，如：
  - Monte-Carlo Control
  - Sarsa
  - Q-learning
- Sample-based planning 通常更有效

AB Example

- Construct a table-lookup model from real experience
- Apply model-free RL to sampled experience

Real experience

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0



Sampled experience

B, 1
B, 0
B, 1
A, 0, B, 1
B, 1
A, 0, B, 1
B, 1
B, 0

e.g. Monte-Carlo learning: $V(A) = 1, V(B) = 0.75$

不准确的模型（Inaccurate Model）

- 由于实际经历的不足或者一些无法避免的缺陷，我们通过实际经历学习得到的模型不可能是完美的模型：
  $\langle P_\eta, R_\eta \rangle \neq \langle P, R \rangle$
- Model-based RL 计算近似 MDP 模型 $\langle S, A, P_\eta, R_\eta \rangle$ 的最优策略，与真实 MDP 的最优策略有差距
- 使用近似的模型解决 MDP 问题与使用价值函数或策略函数的近似表达来解决 MDP 问题并不冲突，他们是从不同的角度来近似求解 MDP 问题，有时候构建一个模型来近似求解 MDP 比构建一个近似的价值函数或策略函数更方便
- 如果模型本身是不准确的，那么“规划”过程可能会得到一个次优的策略
  - Solution 1: 当模型不正确时，使用 Model-free RL
  - Solution 2: 显式的刻画模型的不确定性，并继续推理

Table of Contents

架构整合

- 本节将把基于模型的学习和不基于模型的学习结合起来，形成一个整合的架构，利用两者的优点来解决复杂问题
- 当构建了一个环境的模型后，个体可以有两种经历来源：实际经历 (real experience)、模拟经历 (simulated experience)
- Real experience Sampled from environment (true MDP)

$$S' \sim P^a_{s,s'}$$
$$R = R^a_s$$

- Simulated experience Sampled from model (approximate MDP)

$$S' \sim P_\eta(S' \mid S, A)$$
$$R = R_\eta(R \mid S, A)$$

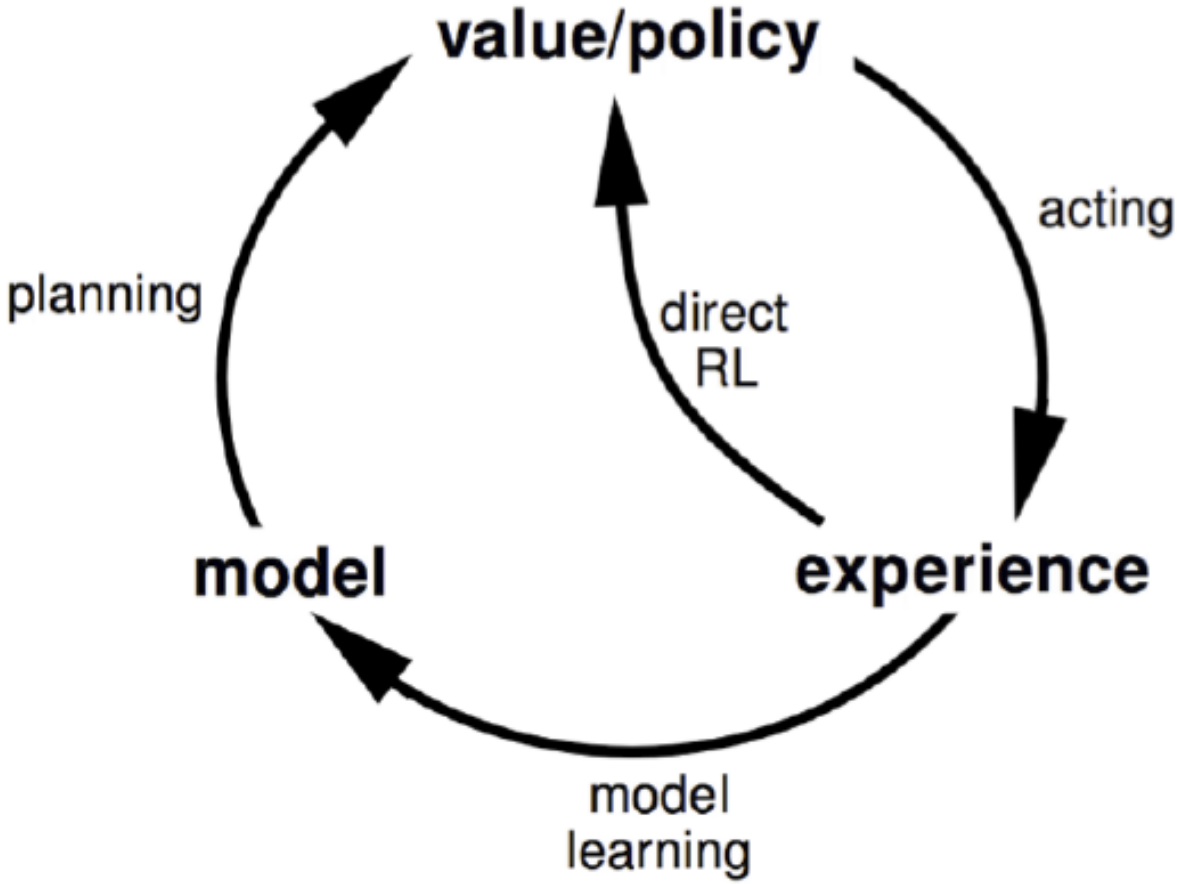
- 注：基于模拟的经历不能够用来指导个体选择合适探索方式。关于“探索”之后再介绍
- 基于这个思想，我们可以把不基于模型的真实经历和基于模型采样得到的模拟经历结合起来，提出一种新的架构：Dyna 算法

Integrating Learning and Planning

- Model-Free RL
  - No model
  - Learn value function (and/or policy) from real experience
- Model-Based RL (using Sample-Based Planning)
  - Learn a model from real experience
  - Plan value function (and/or policy) from simulated experience
- Dyna
  - Learn a model from real experience
  - Learn and plan value function (and/or policy) from real and simulated experience

Dyna Architecture

Dyna 算法从实际经历中学习得到模型，然后联合使用实际经历和模拟经历一边学习，一边规划更新价值和（或）策略函数：



Dyna-Q 算法

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Do forever:

(a) $s \leftarrow$ current (nonterminal) state

(b) $a \leftarrow \epsilon$-greedy(s, Q)

(c) Execute action a; observe resultant state, s', and reward, r

(d) $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$ ← direct RL

(e) $Model(s,a) \leftarrow s', r$ (assuming deterministic environment) ← model learning

(f) Repeat N times:

$s \leftarrow$ random previously observed state

$a \leftarrow$ random action previously taken in s

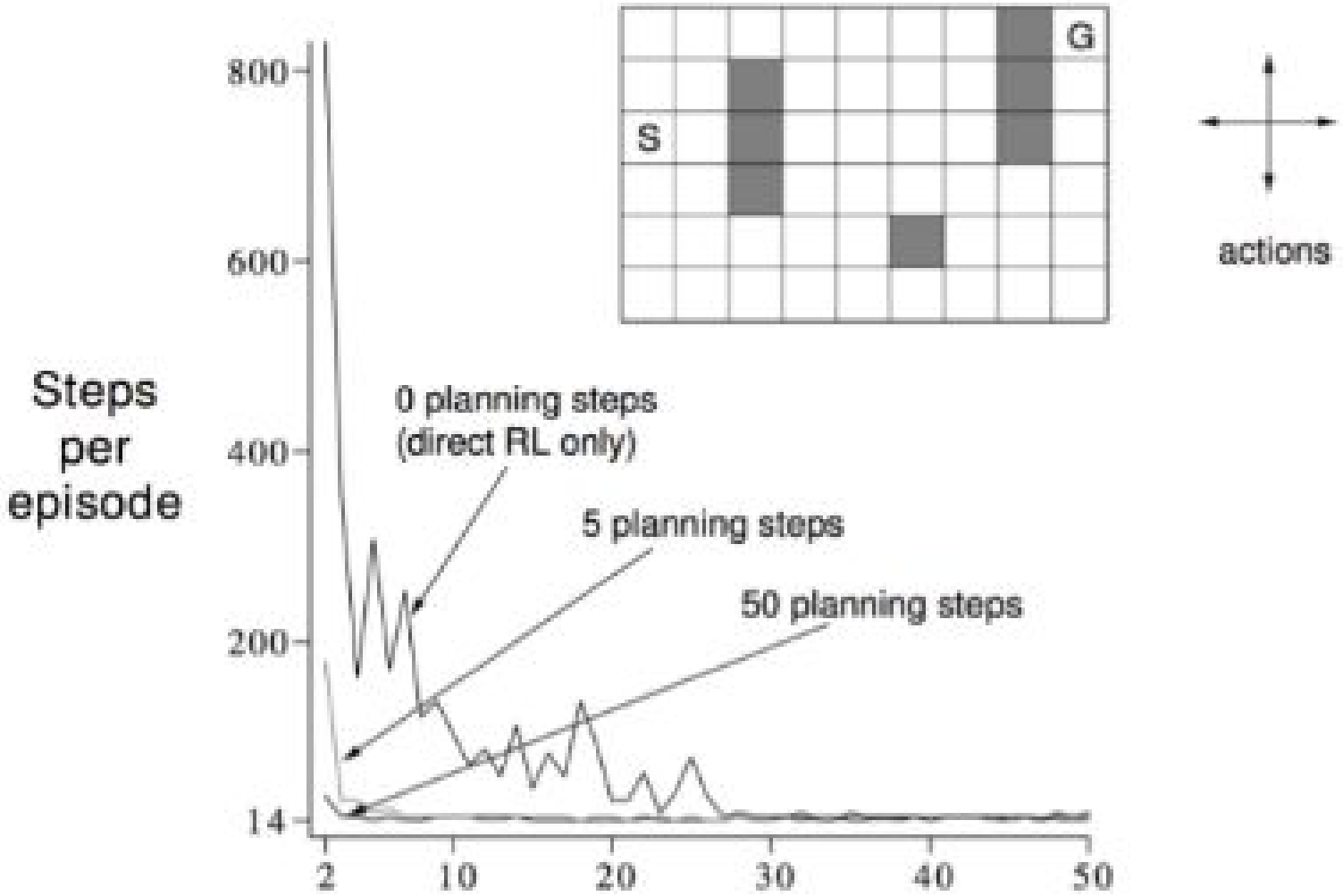
$s', r \leftarrow Model(s,a)$

$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$ ← planning

这个算法赋予了个体在与实际环境进行交互式时有一段时间用来思考的能力

在思考环节 (f 步)，个体将使用模型，在之前观测过的状态空间中随机采样一个状态，同时从这个状态下曾经使用过的行为中随机选择一个行为，将两者带入模型得到新的状态和奖励，依据这个来再次更新行为价值和函数

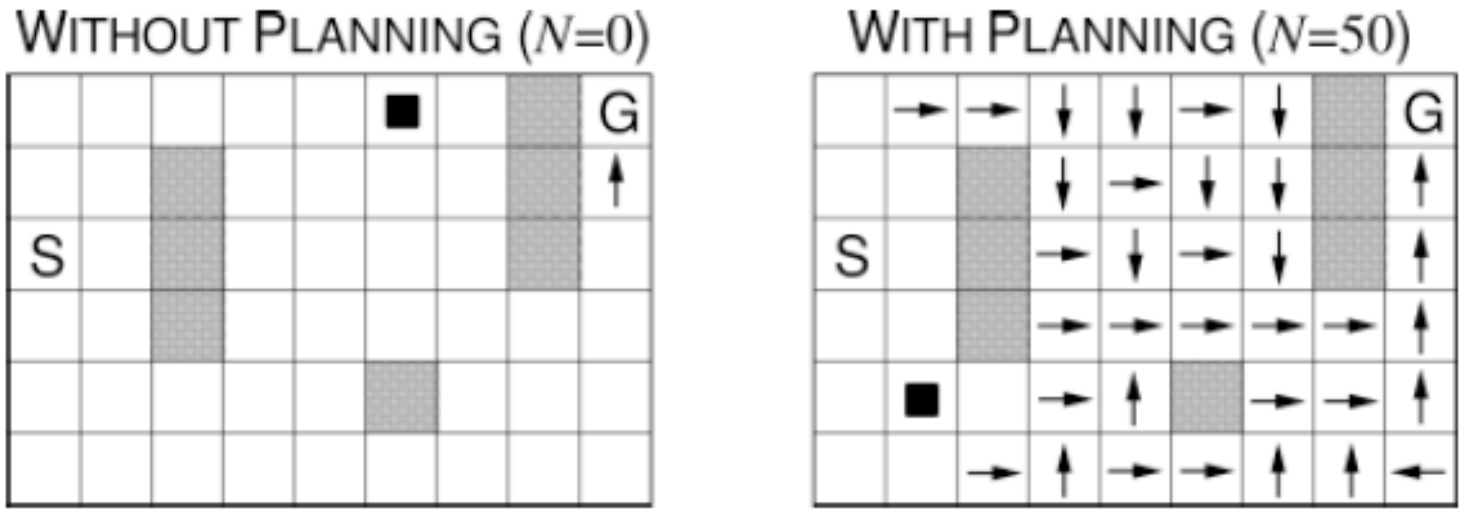
Dyna-Q on a Simple Maze



Dyna-Q on a Simple Maze

- Simple Maze 例子比较了在 Dyna 算法中一次实际经历期间采取不同步数的规划时的个体表现
- 横坐标是 Episode 序号，纵坐标是特定 Episode 所花费的步数
- 可以看出，当直接使用实际经历时，个体在早期完成一个 Episode 需要花费较多步数，而使用 5 步或 50 步思考步数时，个体完成一个 Episode 与环境发生的实际交互的步数要少很多
- 到后期基本无差别

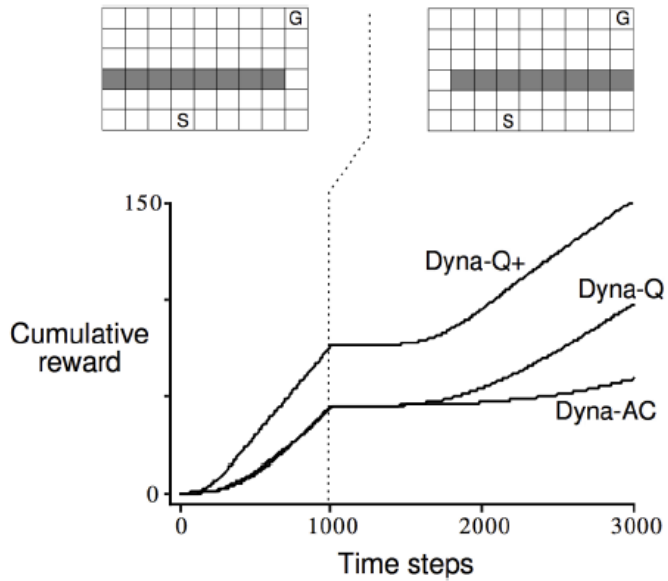
Dyna-Q on a Simple Maze



When the Model is Wrong: Blocking Maze

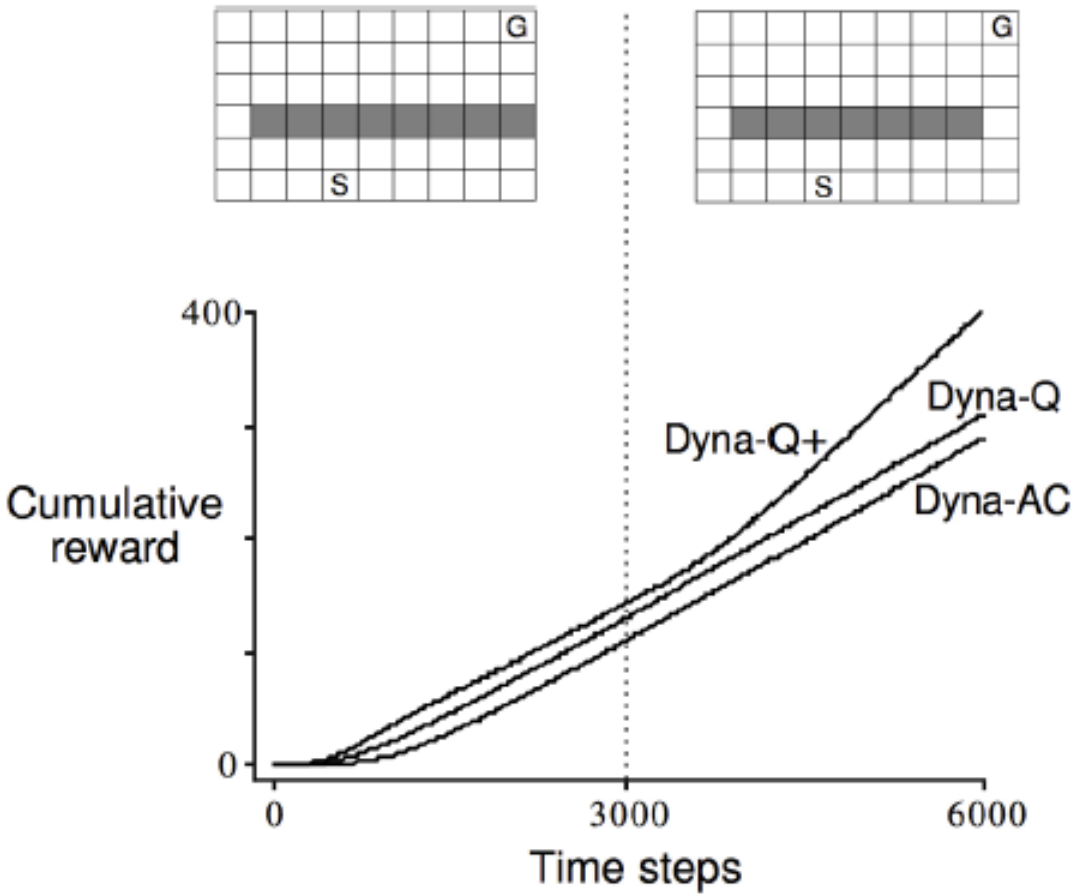
- 在虚线时刻环境发生了改变，变得更加困难，此时使用 Dyna-Q 算法在经历过一段时间的平台期后又找到了最优解决方案
- 在平台期，模型持续的给以个体原先的策略，也就是错误的策略，但个体通过与实际的交互仍然能够找到最优方案

- The changed environment is harder



When the Model is Wrong: Shortcut Maze

- The changed environment is easier



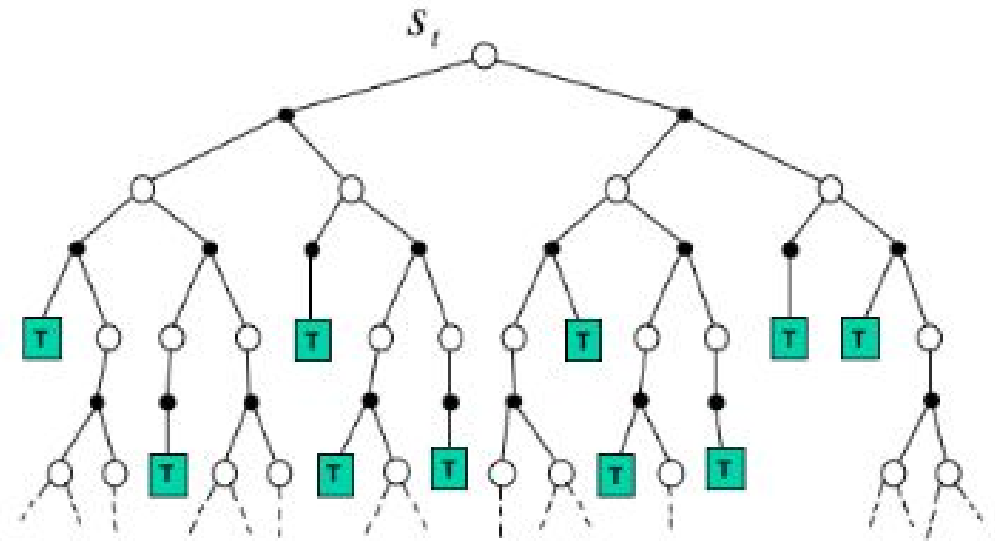
Dyna 算法

- 上面例子表明，Dyna-Q 算法赋予了个体一定的应对环境变化的能力，当环境发生一定改变时，个体一方面可以利用模型，一方面也可以通过与实际环境的交互来重新构建模型
- 例子中 Q+ 算法与 Q 的差别体现在：Q+ 算法鼓励探索，给以探索额外的奖励
- 到目前为止，我们尝试构建一个模型，然后联合实际经历与模拟经历来解决 MDP 问题
- 下面从一个不同的角度来思考“规划”：我们将专注于规划本身，如何高效地进行规划，把这些想法应用到基于模拟的搜索算法中，从模拟的经历中进行采样，进行较深层次的采样规划，来看看是否能够得到一些好的搜索类规划算法
  - 使用的关键思想是：通过采样来进行一定步数的前向搜索，也就是“基于模拟的搜索”(Simulation-Based Search)

Table of Contents

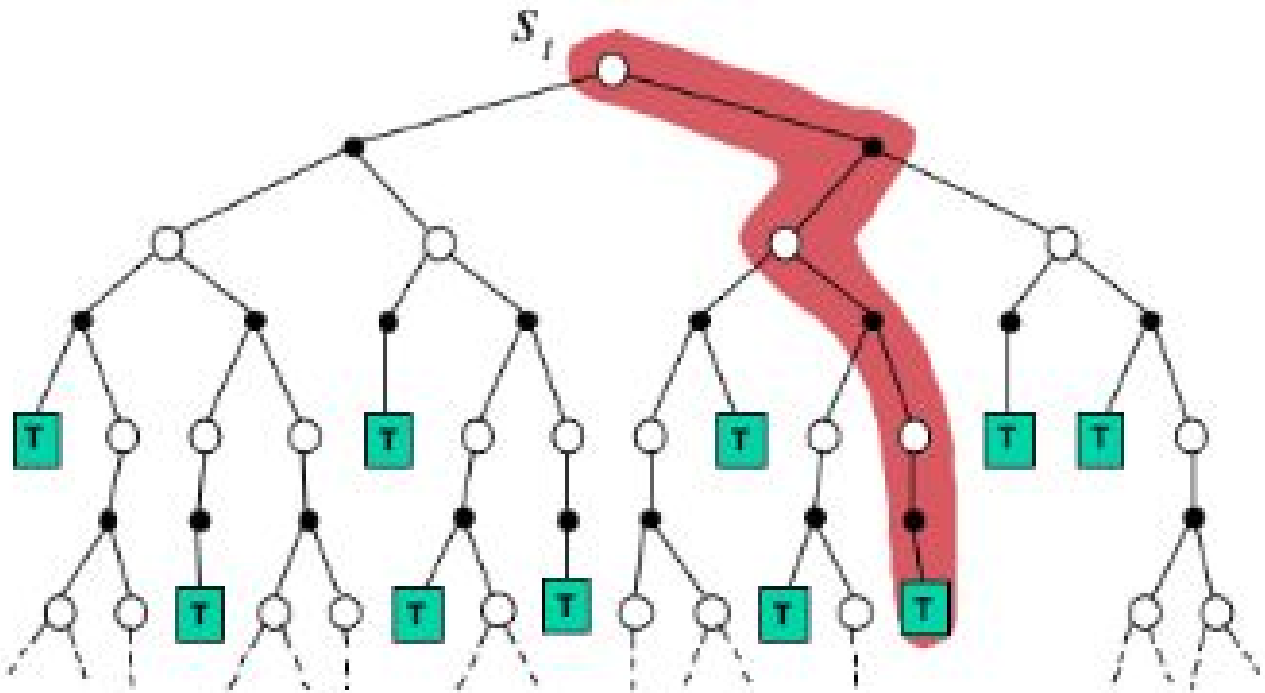
Forward Search

- 在介绍 Simulation-Based Search 前，先回顾下前向搜索 (Forward Search) 算法。该算法通过往前看来采取最好的行为
- 这种算法把当前状态 S_t 作为根节点构建了一个搜索树 (Search Tree)，使用 MDP 模型进行前向搜索
- 前向搜索不需要解决整个 MDP，而仅仅需要构建一个从当前状态开始与眼前的未来相关的次级（子）MDP



Simulation-Based Search

- Forward search paradigm using sample-based planning
  - Simulate episodes of experience from now with the model
  - Apply model-free RL to simulated episodes



Simulation-Based Search

具体步骤如下：

- Simulate episodes of experience from now with the model
  - 从当前时刻 t 开始，从模型中进行虚拟采样，形成从当前状态开始到终止状态 S_T 的一系列 Episode：

$$\{s_t^k, A_t^k, R_{t+1}^k, \ldots, S_T^k\}_{k=1}^K \sim M_\eta$$

- Apply model-free RL to simulated episodes
  - 如果我们使用蒙特卡罗控制，那么这个算法可以称作蒙特卡罗搜索 (Monte-Carlo Search)
  - 如果使用 Sarsa 方法，则称为 TD 搜索 (TD Search)

简单蒙特卡罗搜索 (Simple Monte-Carlo Search)

具体步骤如下：

1. 给定一个模型 M_η 和一个模拟策略（simulation policy）π
2. 针对行为空间里的每一个行为 $a \in A$
   - 2.1 Simulate 从这个当前（实际）状态 S_t 开始模拟出 K 个 Episodes:
   $$\{s_t, a, R^k_{t+1}, \ldots, S^k_T\}^K_{k=1} \sim M_{\eta,\pi}$$
   - 2.2 Evaluate (First Visit) 使用平均收获（蒙特卡罗评估）来评估当前行为 a 的行为价值 $Q(s_t, a)$
   $$Q(s_t, a) = \frac{1}{K}\sum_{k=1}^{K} G_t$$
3. 选择一个有最大 $Q(s_t, a)$ 的行为作为实际要采取的行为 a_t：
   $$a_t = argmax_{a\in A} Q(s_t, a)$$

- 上面就是简单模特卡洛搜索算法的主要思想，它基于一个特定的模拟策略 π
- 如果这个模拟策略 π 本身不是很好的话，那么基于该策略下产生的行为 a_t 可能就不是状态 s_t 下的最优行为

蒙特卡罗树搜索 (Monte-Carlo Tree Search)

- 蒙特卡罗树搜索是一种可以高效解决复杂问题的搜索方法。它使用当前的模拟策略（simulation policy）构建一个基于当前状态 s_t 的搜索树
- 和简单蒙特卡罗搜索不一样的是，蒙特卡罗树搜索方法将评估整个搜索树中每一个状态行为对的价值，并在此基础上改善 simulation policy
- 蒙特卡罗树搜索一个特点是在构建这个搜索树的过程中，更新了搜索树内状态行为对的价值，积累了丰富的信息，利用这些信息可以更新 simulation policy，使得 simulation policy 得到改进

Monte-Carlo Tree Search: Evaluation

具体做法如下：

1. 给定一个模型 M_η 和一个初始的模拟策略（simulation policy）π
2. Simulate 从当前状态 s_t 开始，使用当前的模拟策略 π，模拟生成 K 个 Episodes：

$$\{s_t, A_t^k, R_{t+1}^k, \ldots, S_T^k\}_{k=1}^K \sim M_{\eta,\pi}$$

3. 构建一个包括所有个体经历过的状态和行为的搜索树
4. Evaluate (Every Visit) 对于搜索树中每一个状态行为对 (s, a)，计算从该 (s, a) 开始的所有完整 Episode 收获的平均值，以此来估算该状态行为对的价值 $Q(s, a)$

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^{K} \sum_{u=t}^{T} \mathbf{1}(S_u, A_u = s, a) G_u$$

5. 当搜索过程结束，也就是所有 s, a 的 Q 值得到更新后，选择搜索树中状态 s_t 对应最大 Q 值的行为

$$a_t = argmax_{a \in A} Q(s_t, a)$$

Monte-Carlo Tree Search: Simulation

- 在 MCTS 中，simulation policy π improves
- 搜索树并不包括整个状态行为对空间的 Q 值，因此在改进策略时要分情况对待
- 每个 simulation，更新 simulation policy π 分为两个阶段 (in-tree, out-of-tree)：
  - 树内确定性策略（Tree Policy）(improves): pick actions to maximise $Q(S, A)$
  - 树外默认策略（Default Policy）: pick actions randomly
- Repeat (each simulation)
  - Evaluate states $Q(S, A)$ by Monte-Carlo evaluation
  - Improve tree policy, e.g. by ϵ—greedy(Q)
- Monte-Carlo control applied to simulated experience
- Converges on the optimal search tree, $Q(S, A) \rightarrow q^*(S, A)$

围棋中的蒙特卡罗树搜索 (MCTS in Go)

- 从强化学习的角度来看待围棋问题，可以认为棋盘中的每一个时刻黑白棋的位置构成一个状态 s_t ，当双方对弈未分出胜负时每一步的即时奖励都为 0 ，如果黑棋获胜则终止时刻奖励为 1，反之为 0：

$$R_t = 0 \text{ for all non-terminal steps } t < T$$

$$R_T = \begin{cases} 1 & \text{if Black wins} \\ 0 & \text{if White wins} \end{cases}$$

- 这是一个双方博弈的过程，策略可以被看成是双方策略的联合：$\pi = \langle \pi_B, \pi_W \rangle$
- 通常在针对双方博弈的游戏中，基于某一方（黑方）来设计某一状态的价值函数：

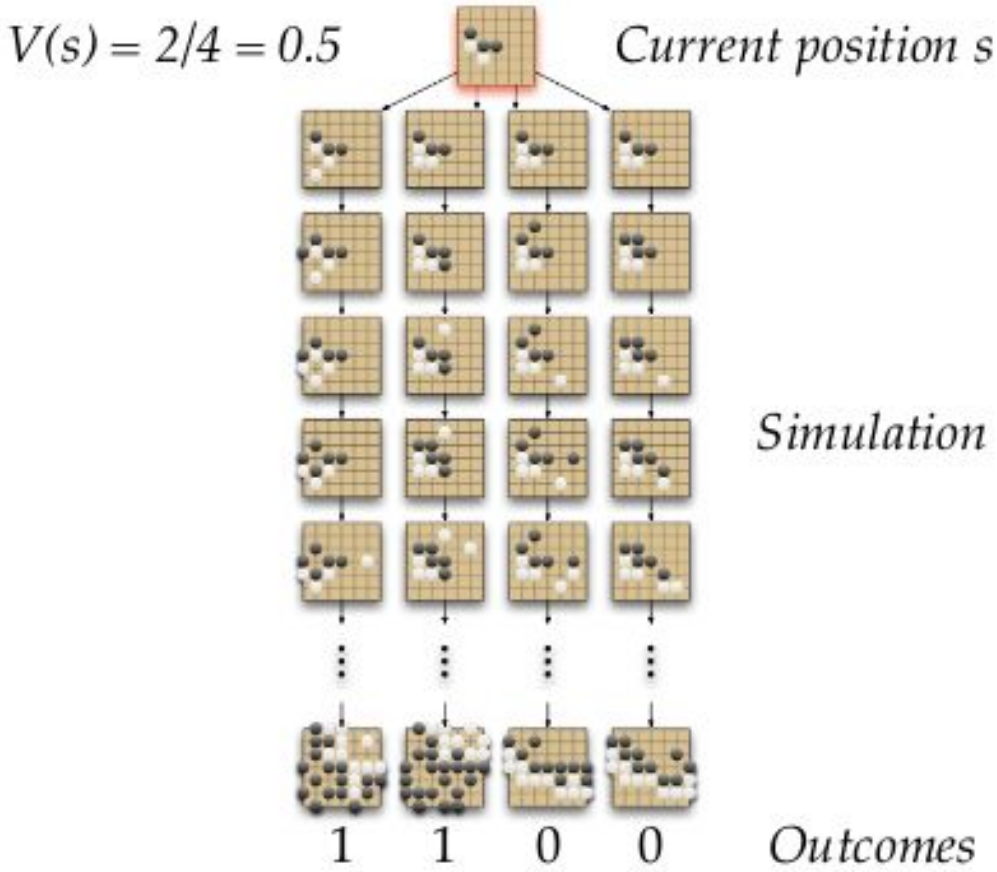
$$v_\pi(s) = E_\pi \left[R_t \mid S = s \right] = P \left[\text{Black wins} \mid S = s \right]$$

- 策略 π 下每一个状态的最优价值是：该策略下从白棋最小化状态价值列表中找到一个相对最大的价值

$$v^*(s) = \max_{\pi_B} \min_{\pi_W} v_\pi(s)$$

Monte-Carlo Evaluation in Go

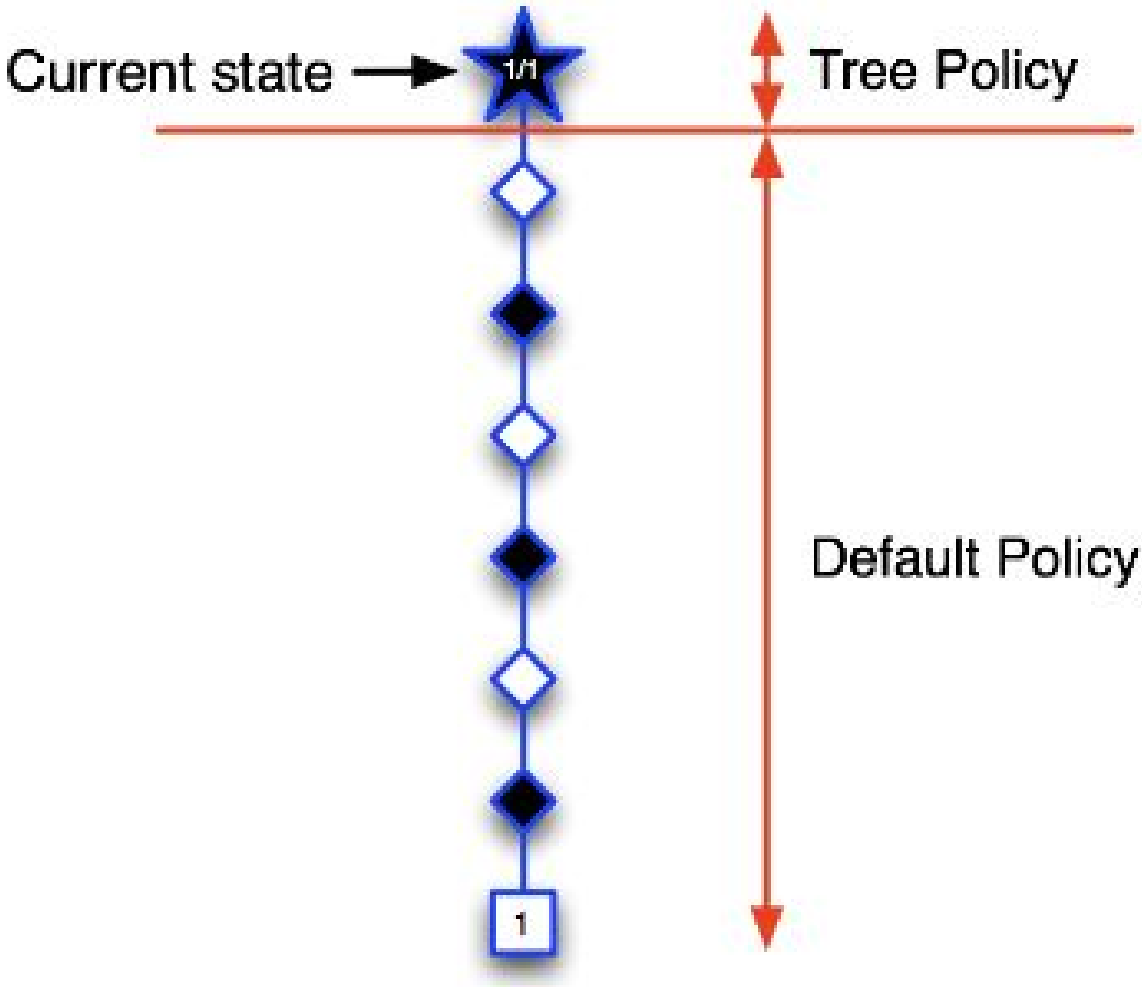
下图简单解释了简单蒙特卡罗评估的过程：



Monte-Carlo Evaluation in Go

- 从某一个状态开始，在当前模拟策略下产生了 4 个完整的 Episodes，根据这 4 个完整 Episode 最终的即时奖励得到针对初始状态时的 4 个收获值，分别为 1,1,0,0
- 平均后得到该策略下初始状态的价值估计为 0.5
- 简单蒙特卡罗评估是了解一个状态价值的一个非常方便的方法，在这种情况下使用价值函数的近似来评估一个状态的价值是非常困难的，得到的结果也不一定理想
- 下面我们结合实例和示意图，通过观察蒙特卡罗树搜索算法的几次迭代过程来实际了解该算法的运作过程

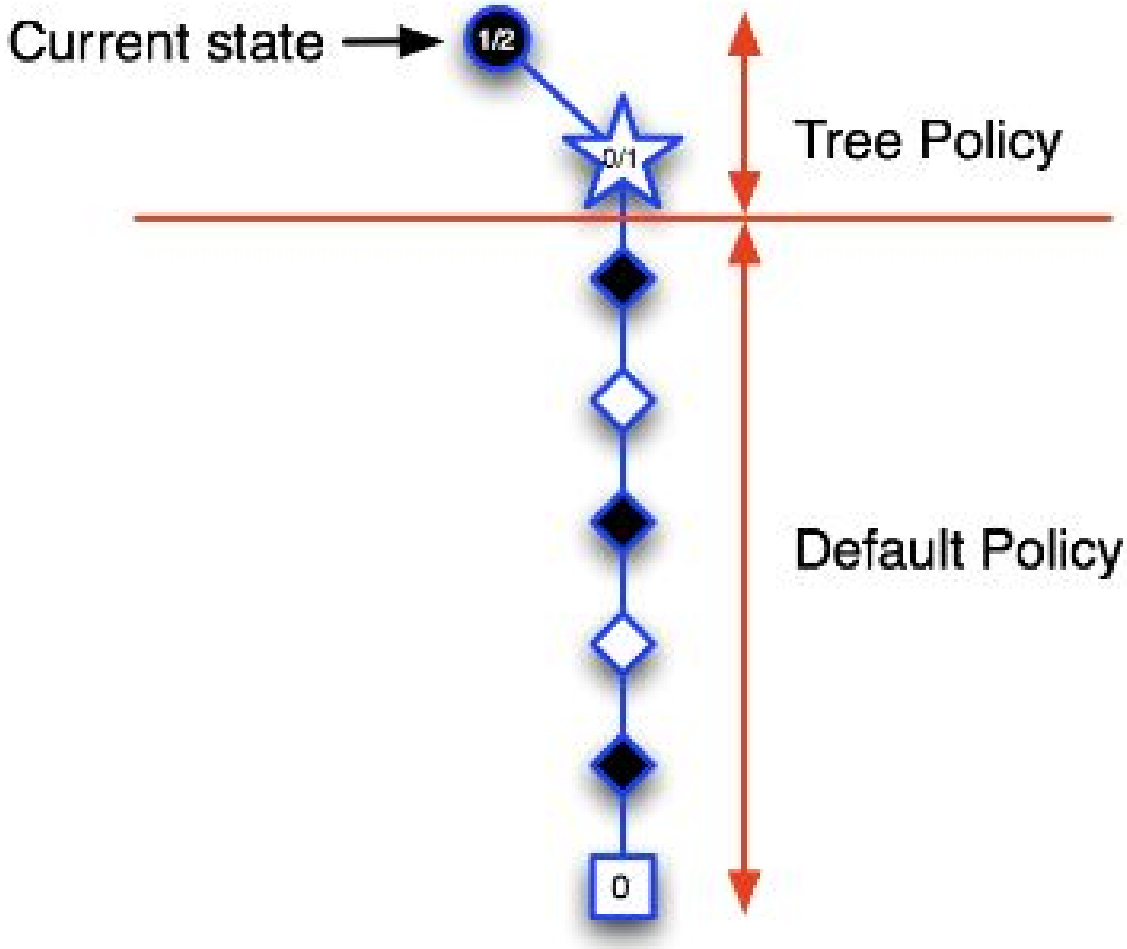
MCTS：第一次迭代



MCTS：第一次迭代

- 五角形表示的状态是个体第一次访问的状态，也是第一次被录入搜索树的状态
- 我们构建搜索树：将当前状态录入搜索树中。使用基于蒙特卡罗树搜索的策略（两个阶段），由于当前搜索树中只有当前状态，全程使用的应该是一个搜索第二阶段的默认随机策略，基于该策略产生一个直到终止状态的完整 Episode
- 图中那些菱形表示中间状态和方格表示的终止状态，在此次迭代过程中并不录入搜索树
- 终止状态方框内的数字 1 表示（黑方）在博弈中取得了胜利
- 此时我们就可以更新搜索树种五角形的状态价值，以分数 1/1 表示从当前五角形状态开始模拟了一个 Episode，其中获胜了 1 个 Episode
- 这是第一次迭代过程

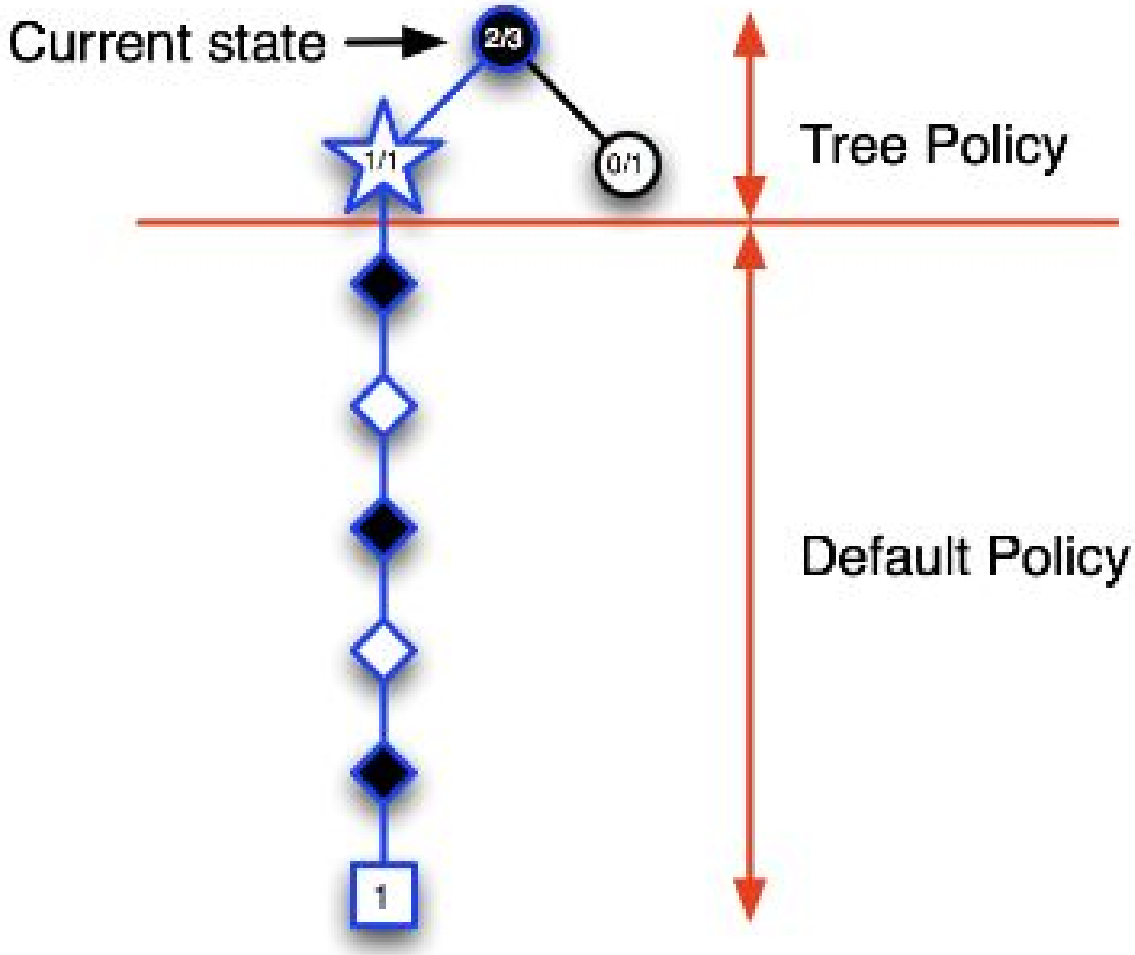
MCTS：第二次迭代



MCTS：第二次迭代

- 当前状态仍然是树内的圆形图标指示的状态，从该状态开始决定下一步动作
- 根据目前已经访问过的状态构建搜索树，依据模拟策略产生一个行为模拟进入白色五角形表示的状态，并将该状态录入搜索树，随后继续该次模拟的对弈直到 Episode 结束，结果显示黑方失败，因此我们可以更新新加入搜索树的五角形节点的价值为 0/1，而搜索树种的圆形节点代表的当前状态其价值估计为 1/2，表示进行了 2 次模拟对弈，赢得了 1 次，输了 1 次
- 第二次迭代结束
- 经过前两次的迭代，当位于当前状态（黑色圆形节点）时，当前策略会认为选择某行为进入上图中白色五角形节点状态对黑方不利，策略将得到更新：当前状态时会个体会尝试选择其它行为

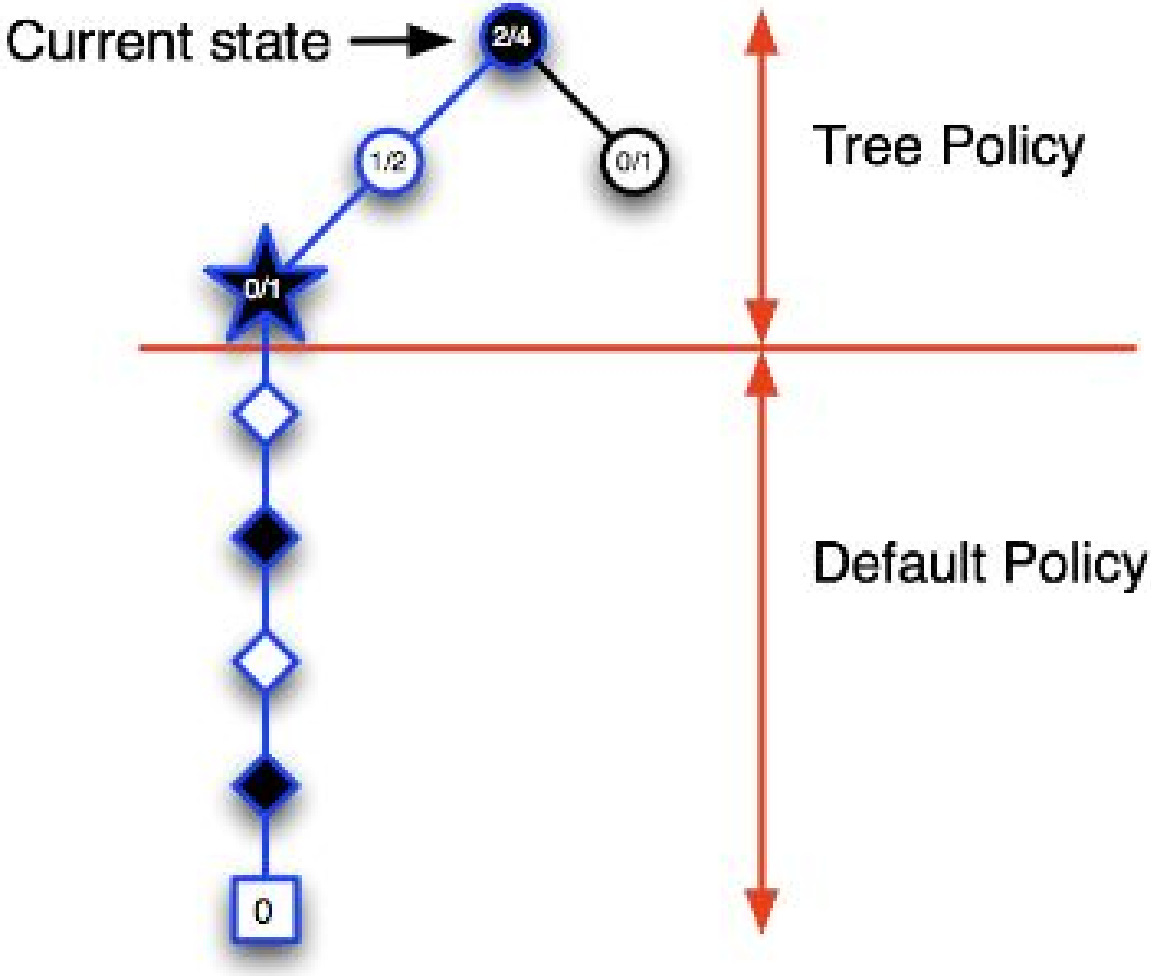
MCTS：第三次迭代



MCTS：第三次迭代

- 假设选择了一个行为进入白色五角形节点状态，将该节点录入搜索树，模拟一次完整的 Episode，结果显示黑方获胜，此时更新新录入节点的状态价值为 1/1，同时更新其上级节点的状态价值，这里需要更新当前状态的节点价值为 2/3，表明在当前状态下已经模拟了 3 次对弈，黑方获胜 2 次
- 随着迭代次数的增加，在搜索树里录入的节点开始增多，树内每一个节点代表的状态其价值数据也越来越丰富。在搜索树内依据 ϵ-greedy 策略会使得当个体出于当前状态（圆形节点）时更容易做出到达图中五角形节点代表的状态的行为

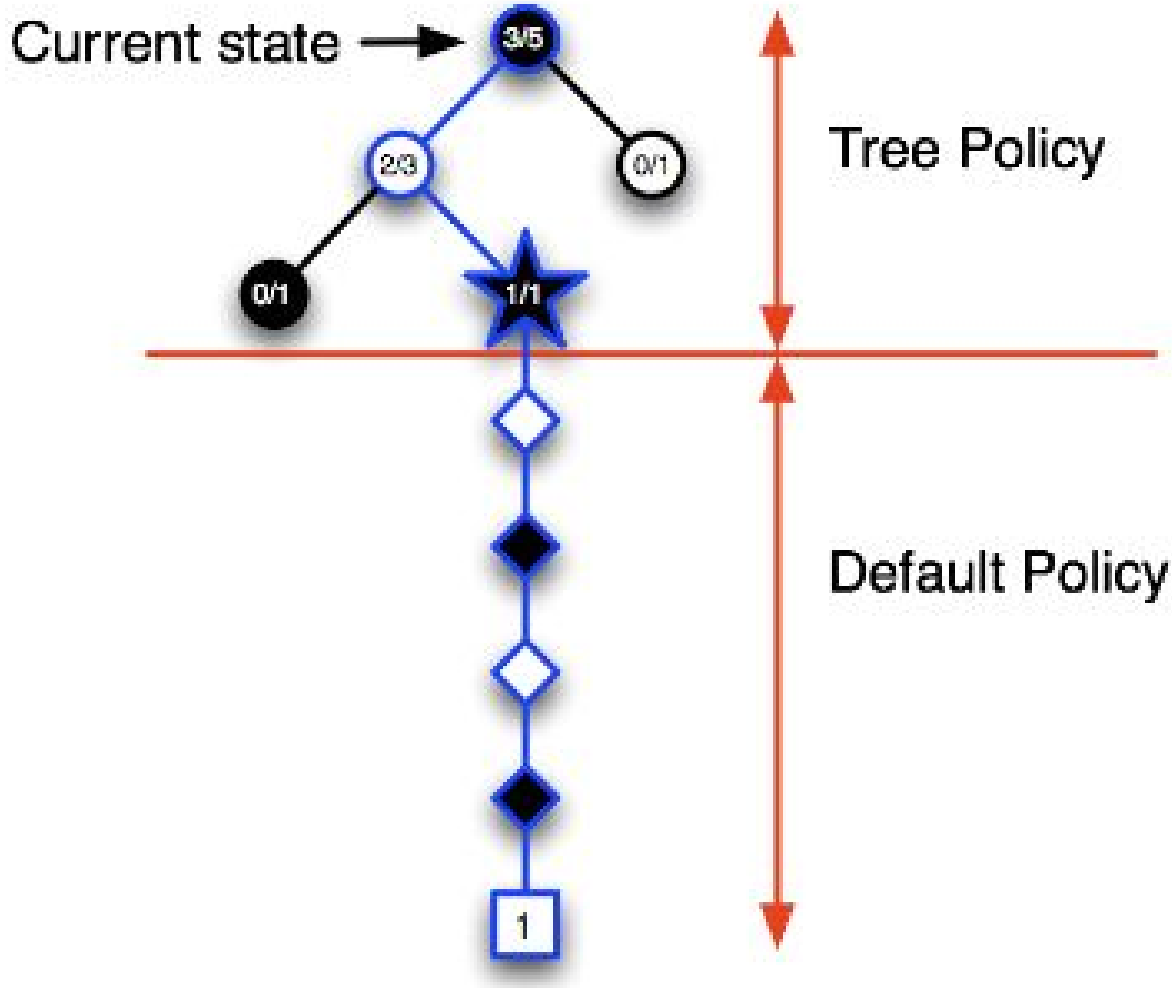
MCTS：第四次迭代



MCTS：第四次迭代

- 当个体位于当前（圆形节点）状态时，树内策略使其更容易进入左侧的蓝色圆形节点代表的状态，此时录入一个新的节点（五角形节点），模拟完 Episode 提示黑方失败，更新该节点以及其父节点的状态价值
- 该次迭代结束

MCTS：第五次迭代



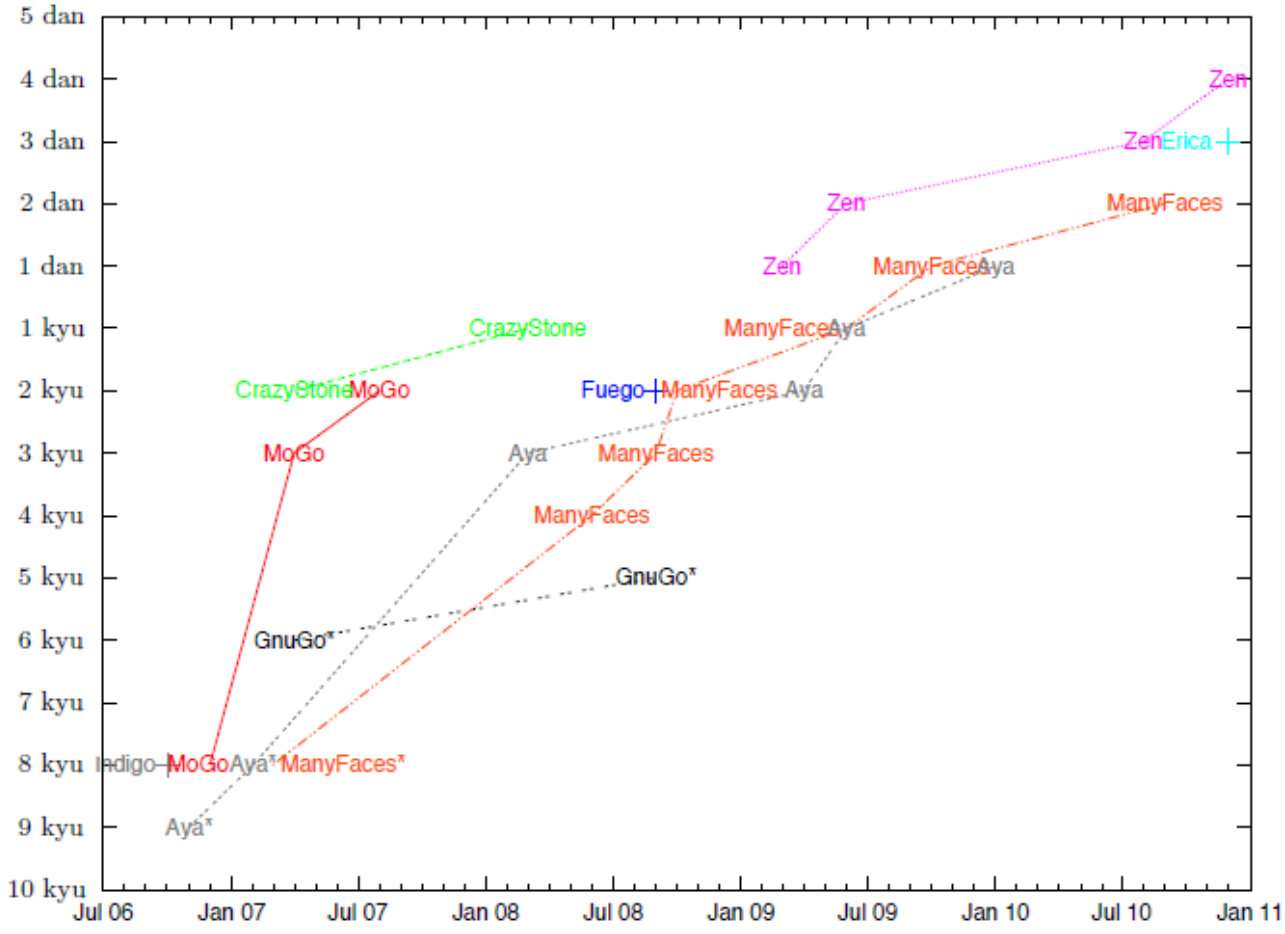
MCTS：第五次迭代

- 更新后的策略使得个体在当前状态时仍然有较大几率进入其左侧圆形节点表示的状态，在该节点，个体避免了进入刚才失败的那次节点，录入了一个新节点，基于模拟策略完成一个完整 Episode，黑方获得了胜利，同样的更新搜索树内相关节点代表的状态价值
- 如此反复，随着迭代次数的增加，当个体处于当前状态时，其搜索树将越来越深，那些能够引导个体获胜的搜索树内的节点将会被充分的探索，其节点代表的状态价值也越来越有说服力；同时个体也忽略了那些结果不好的一些节点（上图中当前状态右下方价值估计为 $0/1$ 的节点）。需要注意的是，仍然要对这部分节点进行一定程度的探索已确保这些节点不会被完全忽视
- 同样，随着迭代次数的增加，搜索树内的节点越来越多，代表着搜索树外的节点将逐渐减少，这意味着在模拟 Episode 的过程中随机策略发生的情况也越来越少。个体最终会得到一个基于当前状态一个非常有指导意义的搜索树。不过即时个体不能充分探索许多状态，少量的随机行为并不会影响个体整体的决策效果

Advantages of MC Tree Search

- 蒙特卡罗树搜索的优点
  - 蒙特卡罗树搜索是具有高度选择性的（Highly selective)、基于导致越好结果的行为越被优先选择（best-first）的一种搜索方法
  - 它可以动态的评估各状态的价值，这种动态更新价值的方法与动态规划不同，后者聚焦于整个状态空间，而蒙特卡罗树搜索是立足于当前状态，动态更新与该状态相关的状态价值
  - 它使用采样避免了维度灾难
  - 由于它仅依靠采样，因此适用于那些“黑盒”模型 (black-box models)
- 上述这些优点决定了其是可以高效计算的（computationally efficient)、实时（anytime) 及可以并行处理的 (parallelisable)

MC Tree Search in Computer Go



时序差分搜索 (Temporal-Difference Search)

- 前面讲解的都是蒙特卡罗搜索，它只是 Simulation-Based Search 的一种方法，实际上不应仅局限于特定的搜索树结构、模拟策略及蒙特卡罗搜索算法
- 解决这一大类问题的关键理念在于“前向搜索（forward sarch)”和“采样 (sampling)”
- 正确运用这两点，个体可以到达状态空间里许多非常优秀的状态，在此基础上把强化学习的一些不基于模型的算法应用到这些模拟产生的好的状态中，最终得到一个较优秀的策略
- 如果将其他强化学习算法应用于基于模拟的搜索，可以得到诸如 TD 搜索等其他搜索方法
- 如果说蒙特卡罗树搜索是对从当前状态开始的一个子 MDP 问题应用蒙特卡罗控制，那么 TD 搜索可以被看成时对从当前状态开始的一个子 MDP 问题应用 SARSA 学习
- 就像在不基于模型的强化学习里 TD 学习相比 MC 学习有众多优点一样，在基于模拟的搜索时，TD 搜索多数时候也是优于 MC 搜索的，特别是 TD(λ) 搜索
- 这主要是因为使用引导 (bootstrapping) 数据对解决基于模拟的搜索产生的子 MDP 问题同样是有积极意义的

MC vs. TD search

- For model-free reinforcement learning, bootstrapping is helpful
  - TD learning reduces variance but increases bias
  - TD learning is usually more efficient than MC
  - TD(λ) can be much more efficient than MC
- For simulation-based search, bootstrapping is also helpful
  - TD search reduces variance but increases bias
  - TD search is usually more efficient than MC search
  - TD(λ) search can be much more efficient than MC search

TD Search

对于 TD 搜索，其过程如下：

1. 从当前实际状态 s_t 开始，模拟一系列 Episodes，在这一过程中使用状态行为价值作为节点录入搜索树
2. 对搜索树内的每一个节点（状态行为对），估计其价值 $Q(s, a)$
3. 对于模拟过程中的每一步，使用 Sarsa 学习更新行为价值：

$$\Delta Q(S, A) = \alpha(R + \gamma Q(S', A') - Q(S, A))$$

4. 基于 Q 值，使用 ϵ-greedy 或其他探索方法来生成行为。

相比于 MC 搜索，TD 搜索不必模拟 Episode 到终止状态，其仅聚焦于某一个节点的状态，这对于一些有回路或众多旁路的节点来说更加有意义，这是因为在使用下一个节点估计当前节点的价值时，下一个节点的价值信息可能已经是经过充分模拟探索了的，在此基础上更新的当前节点价值会更加准确

Search tree vs. value function approximation

- Search tree is a table lookup approach
- Based on a *partial* instantiation of the table
- For model-free RL, table lookup is naive
  - Can't store value for all states
  - Doesn't generalize between similar states
- For simulation-based search, table lookup is less naive
  - Search tree stores value for easily reachable states
  - But still doesn't generalize between similar states
  - In huge search spaces, value function approximation is helpful

Linear TD Search

- Use linear value function approximation for action-value function $Q_\theta(s, a) \approx Q^\pi(s, a)$

$$Q_\theta(s, a) = \phi(s, a)^\top \theta$$

- Simulate episodes from the current (real) state s_t
- At each simulated step u, update action-values by linear Sarsa

$$\Delta\theta = \alpha(r_{u+1} + \gamma Q_\theta(s_{u+1}, a_{u+1}) - Q_\theta(s_u, a_u))\phi(s_u, a_u)$$

- Select actions based on action-values $Q_\theta(s, a)$, e.g. ϵ-greedy

Dyna-2 算法

- 我们再次回到 Dyna 算法中来。Dyna 算法一边从实际经历中学习，一边从模拟的经历中学习
- 如果我们把 simulation-based search 应用到 Dyna 算法中来，就变成了 Dyna-2 算法
- 使用该算法的个体维护了两套特征权重：
  - Long-term memory：一套反映了个体的长期记忆，该记忆是从真实经历中使用 TD 学习得到，它反映了个体对于某一特定强化学习问题的普遍性的知识、经验
  - Short-term (working) memory：另一套反映了个体的短期记忆，该记忆从基于模拟经历中使用 TD 搜索得到，它反映了个体对于某一特定强化学习在特定条件（比如某一 Episode、某一状态下）下的特定的、局部适用的知识、经验
- Long-term memory 学的是大局观，是从真实的以往的经验中学到的，要长期记住的东西，而 Short-term memory 学的是后续行为的局部引导，或者说是具体实施细节，是从虚拟的当前（构想）的经验中搜索得来的东西
- Dyna-2 算法最终将两套特征权重下产生的价值综合起来进行决策，以期得到更优秀的策略

Dyna-2 算法

Algorithm 1 Episodic Dyna-2

procedure LEARN
  Initialise A, B ▷ Transition and reward models
  $\theta \leftarrow 0$ ▷ Clear permanent memory
  loop
    $s \leftarrow s_0$ ▷ Start new episode
    $\bar{\theta} \leftarrow 0$ ▷ Clear transient memory
    $z \leftarrow 0$ ▷ Clear eligibility trace
    SEARCH(s)
    $a \leftarrow \pi(s; \bar{Q})$ ▷ e.g. ϵ-greedy
    while s is not terminal **do**
      Execute a, observe reward r, state s'
      $(A, B) \leftarrow$ UPDATEMODEL(s, a, r, s')
      SEARCH(s')
      $a' \leftarrow \pi(s'; \bar{Q})$
      $\delta \leftarrow r + Q(s', a') - Q(s, a)$ ▷ TD-error
      $\theta \leftarrow \theta + \alpha(s, a)\delta z$ ▷ Update weights
      $z \leftarrow \lambda z + \phi$ ▷ Update eligibility trace
      $s \leftarrow s', a \leftarrow a'$
    end while
  end loop
end procedure

Dyna-2 算法

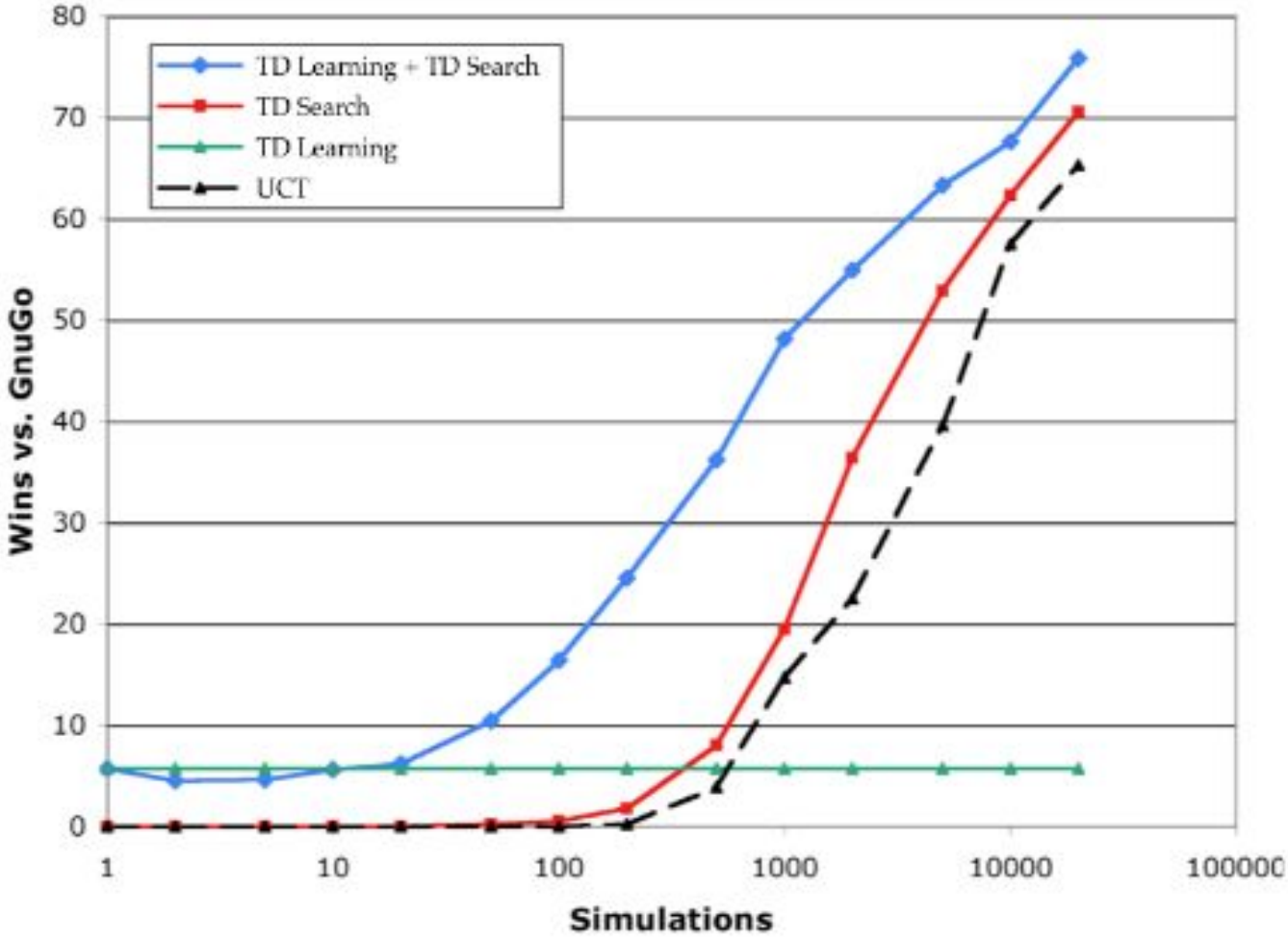
22: **procedure** SEARCH(s)
23: **while** time available **do**
24: $\bar{z} \leftarrow 0$ ▷ Clear eligibility trace
25: $a \leftarrow \bar{\pi}(s; \bar{Q})$ ▷ e.g. ϵ-greedy
26: **while** s is not terminal **do**
27: $s' \leftarrow A(s, a)$ ▷ Sample transition
28: $r \leftarrow B(s, a)$ ▷ Sample reward
29: $a' \leftarrow \bar{\pi}(s'; \bar{Q})$
30: $\bar{\delta} \leftarrow r + \bar{Q}(s', a') - \bar{Q}(s, a)$ ▷ TD-error
31: $\bar{\theta} \leftarrow \bar{\theta} + \bar{\alpha}(s, a)\bar{\delta}\bar{z}$ ▷ Update weights
32: $\bar{z} \leftarrow \bar{\lambda}\bar{z} + \bar{\phi}$ ▷ Update eligibility trace
33: $s \leftarrow s', a \leftarrow a'$
34: **end while**
35: **end while**
36: **end procedure**

$$Q(s, a) = \phi(s, a)^\top \theta$$

$$\bar{Q}(s, a) = \phi(s, a)^\top \theta + \bar{\phi}(s, a)^\top \bar{\theta}$$

Results of TD search in Go

下图展示了在围棋游戏中，使用不同算法的效果比较：



Results of TD search in Go

- 该图纵坐标表示的不同算法与使用基准算法（GnuGo）进行围棋对弈的获胜率，横坐标应该是个体实际或模拟运行的步数
- 黑色虚线是蒙特卡罗树搜索的表现
- 绿色是仅针对真实经历进行 TD 学习得到的结果，其效果非常不好
- 红色曲线表示的是 TD 搜索的效果，在围棋程序中表现比 MC 搜索要好一些
- 表现最好的是把 TD 学习和 TD 搜索结合起来的 Dyna-2 算法