

# Lecture 8: Integrating Learning and Planning

David Silver

# Outline

## 1 Introduction

지금까지 model-free를 배웠.  
이번 강에서는 Model-based

## 2 Model-Based Reinforcement Learning

## 3 Integrated Architectures

## 4 Simulation-Based Search

planning을 해

# Outline

- 1 Introduction
- 2 Model-Based Reinforcement Learning
- 3 Integrated Architectures
- 4 Simulation-Based Search

# Model-Based Reinforcement Learning

- *Last lecture:* learn **policy** directly from experience
- *Previous lectures:* learn **value function** directly from experience
- *This lecture:* learn **model** directly from experience
- and use **planning** to construct a value function or policy
- Integrate learning and planning into a single architecture

# Model-Based and Model-Free RL

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

# Model-Based and Model-Free RL

*RL 101*

- Model-Free RL

- No model

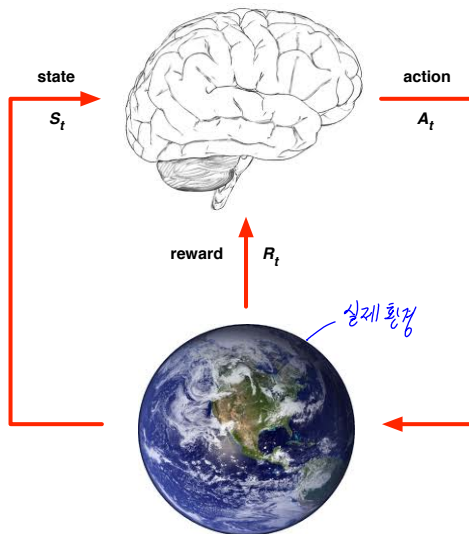
- Learn value function (and/or policy) from experience

- Model-Based RL

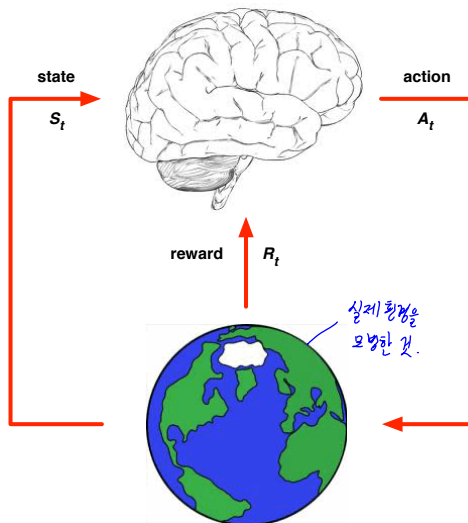
- Learn a model from experience

- Plan value function (and/or policy) from model

# Model-Free RL



# Model-Based RL



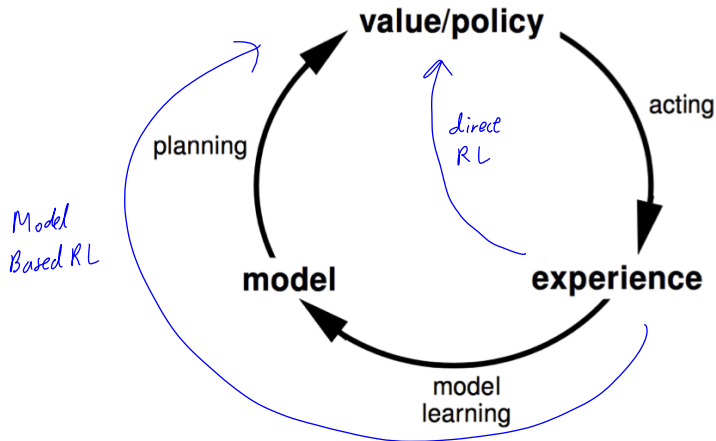


# Outline

- 1 Introduction
- 2 Model-Based Reinforcement Learning**
- 3 Integrated Architectures
- 4 Simulation-Based Search

# Model-Based RL

MDP를 모르는 건 똑같은.



# Advantages of Model-Based RL

Advantages:

*Model이 강한 경우 (ex: 체스)*

- Can efficiently learn model by supervised learning methods
- Can reason about model uncertainty

Disadvantages:

- First learn a model, then construct a value function  
⇒ two sources of approximation error

# What is a Model?

- A *model*  $\mathcal{M}$  is a representation of an MDP  $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ , <sup>Model  $\mathcal{M}$  only</sup> parametrized by  $\eta$ . <sup>state action</sup> <sup>reward</sup> <sup>discounted factor</sup> <sup>transition probability matrix</sup>
- We will assume state space  $\mathcal{S}$  and action space  $\mathcal{A}$  are known
- So a model  $\mathcal{M} = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$  represents state transitions  $\mathcal{P}_\eta \approx \mathcal{P}$  and rewards  $\mathcal{R}_\eta \approx \mathcal{R}$

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1} \mid S_t, A_t)$$

$$R_{t+1} = \mathcal{R}_\eta(R_{t+1} \mid S_t, A_t)$$

- Typically assume conditional independence between state transitions and rewards

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

# Model Learning

- Goal: estimate model  $\mathcal{M}_\eta$  from experience  $\{S_1, A_1, R_2, \dots, S_T\}$
- This is a supervised learning problem

$$S_1, A_1 \rightarrow R_2, S_2$$

$$S_2, A_2 \rightarrow R_3, S_3$$

$$\vdots$$

$$S_{T-1}, A_{T-1} \rightarrow R_T, S_T$$

input과 output들이  
주어진 것은 때, 이를 학습하여  
input과 output 사이의 관계를  
학습. 많이 할수록 supervised 학습 가능.

supervised learning  
으로 볼 수 있다.

두 확률 정보의 차이를  
계산하는 데 쓰이는 함수.

- Learning  $s, a \rightarrow r$  is a regression problem
- Learning  $s, a \rightarrow s'$  is a density estimation problem
- Pick loss function, e.g. mean-squared error, KL divergence, ...
- Find parameters  $\eta$  that minimise empirical loss

gradient descent, Neural network 등  
여러 방법의 조합

# Examples of Models

- Table Lookup Model
- Linear Expectation Model
- Linear Gaussian Model
- Gaussian Process Model
- Deep Belief Network Model
- ...

# Table Lookup Model

- Model is an explicit MDP,  $\hat{\mathcal{P}}, \hat{\mathcal{R}}$
- Count visits  $N(s, a)$  to each state action pair

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

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

평균으로 계산한  
값을 table에  
기록 (lookup)

- Alternatively

- At each time-step  $t$ , record experience tuple  $\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$

- To sample model, randomly pick tuple matching  $\langle s, a, \cdot, \cdot \rangle$

단정: 경험하지 못한 transition은

구할 수 없음

(Neural network 등으로 모르는 값 예측할 수 있겠지?)

# AB Example

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

$A, 0, B, 0$

$B, 1$

$B, 1$

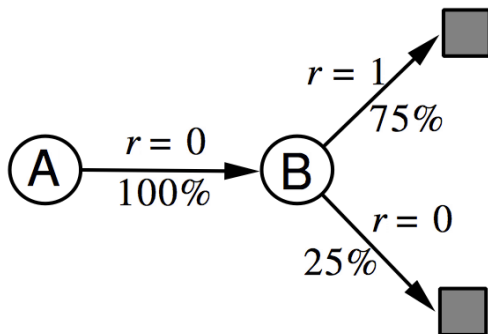
$B, 1$

$B, 1$

$B, 1$

$B, 1$

$B, 0$



We have constructed a **table lookup model** from the experience



# Planning with a Model

$$\mathcal{P} \approx \mathcal{P}_\eta \quad \mathcal{R} \approx \mathcal{R}_\eta$$

- Given a model  $\mathcal{M}_\eta = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- Solve the MDP  $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- Using favourite planning algorithm
  - Value iteration
  - Policy iteration
  - Tree search
  - ...

# Sample-Based Planning

- A simple but powerful approach to planning
- Use the model **only** to generate samples
- **Sample** experience from model

model 기법으로 만드는 sample은 model이

있으면 무한히 생성가능해

Model 없이 sample은 얻음.

이 sample로 model-free RL  
학습 ..

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1} \mid S_t, A_t)$$

$$R_{t+1} = \mathcal{R}_\eta(R_{t+1} \mid S_t, A_t)$$

- Apply **model-free** RL to samples, e.g.:
  - Monte-Carlo control
  - Sarsa
  - Q-learning
- Sample-based planning methods are often more efficient

Dynamic Programming - Naive Full backup

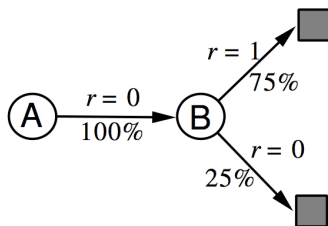
Sample backup - DP에 비해 과묵 일대는 사본으로  
집중해서 학습할 수 있음  
↳ Curse of dimensionality는 해결  
수 있다.

# Back to the AB Example

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

Real experience  $V(A)=0$

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



$V(A)=1$   
 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:  $V(A) = 1, V(B) = 0.75$

# Planning with an Inaccurate Model

- Given an imperfect model  $\langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle \neq \langle \mathcal{P}, \mathcal{R} \rangle$
- Performance of model-based RL is limited to optimal policy for approximate MDP  $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- i.e. Model-based RL is only as good as the estimated model
- When the model is inaccurate, planning process will compute a suboptimal policy
- Solution 1: when model is wrong, use model-free RL
- Solution 2: reason explicitly about model uncertainty

# Outline

- 1 Introduction
- 2 Model-Based Reinforcement Learning
- 3 Integrated Architectures**
- 4 Simulation-Based Search

# Real and Simulated Experience

We consider two sources of experience

**Real experience** Sampled from environment (true MDP)

$$S' \sim \mathcal{P}_{s,s'}^a$$

$$R = \mathcal{R}_s^a$$

**Simulated experience** Sampled from model (approximate MDP)

$$S' \sim \mathcal{P}_\eta(S' \mid S, A)$$

$$R = \mathcal{R}_\eta(R \mid S, A)$$

# Integrating Learning and Planning

- Model-Free RL
  - No model
  - **Learn** value function (and/or policy) from real experience

# Integrating Learning and Planning

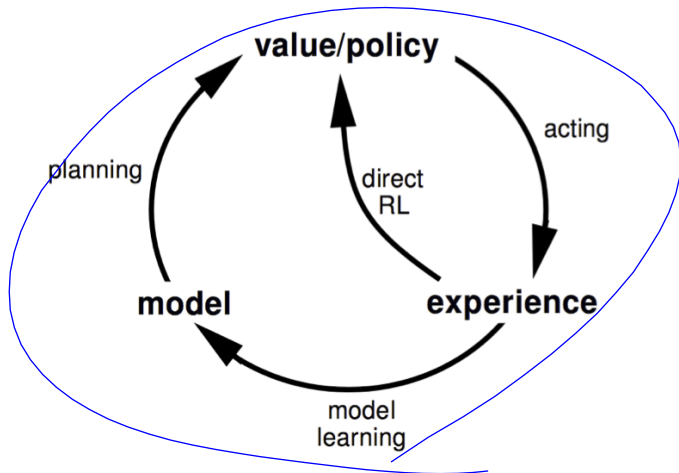
- Model-Free RL
  - No model
  - **Learn** value function (and/or policy) from real experience
- Model-Based RL (using Sample-Based Planning)
  - Learn a model from real experience
  - **Plan** value function (and/or policy) from simulated experience



# 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-Q Algorithm

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 reward,  $R$ , and state,  $S'$

(d)  $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

(e)  $Model(S, A) \leftarrow R, S'$  (assuming deterministic environment)

(f) Repeat  $n$  times: *planning으로 Q는 update 한다.*

$S \leftarrow$  random previously observed state

$A \leftarrow$  random action previously taken in  $S$

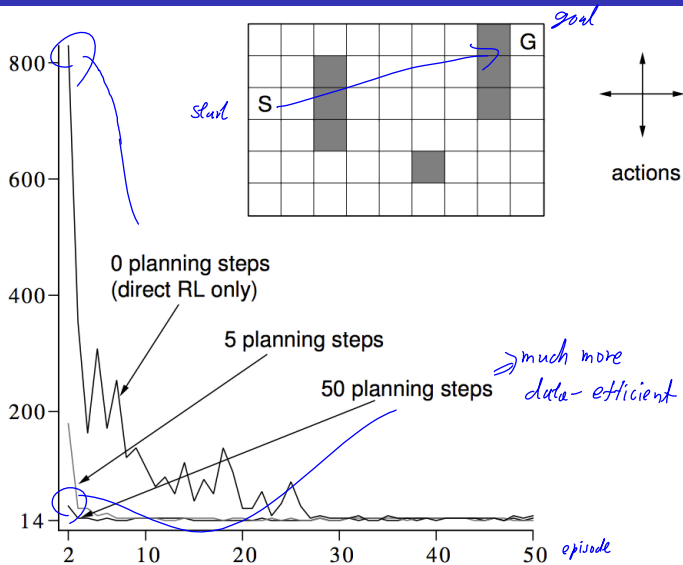
$R, S' \leftarrow Model(S, A)$

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

*model-free*

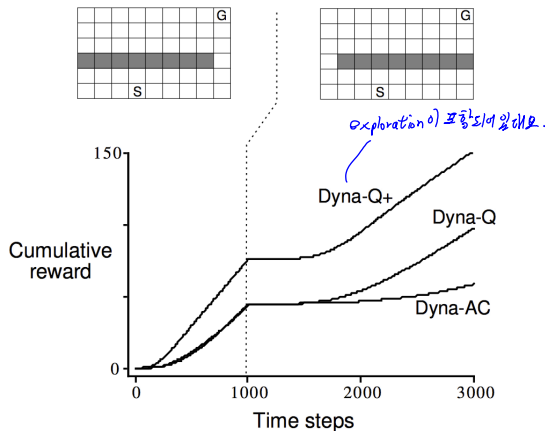
*Model을 활용*

Steps  
per  
episode



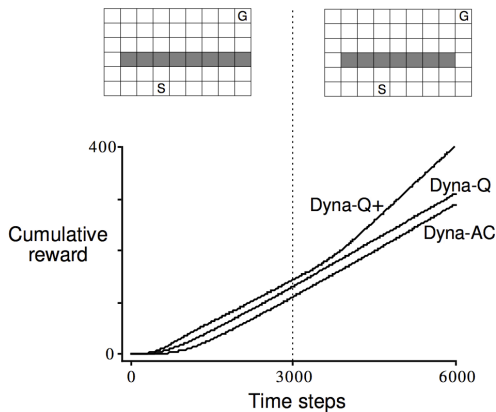
# Dyna-Q with an Inaccurate Model

- The changed environment is **harder**



# Dyna-Q with an Inaccurate Model (2)

- The changed environment is **easier**



# Outline

- 1 Introduction
- 2 Model-Based Reinforcement Learning
- 3 Integrated Architectures
- 4 Simulation-Based Search — *how to plan efficiently*

current state가 중요하다. 너무 먼 미래는 지금 의미없다.

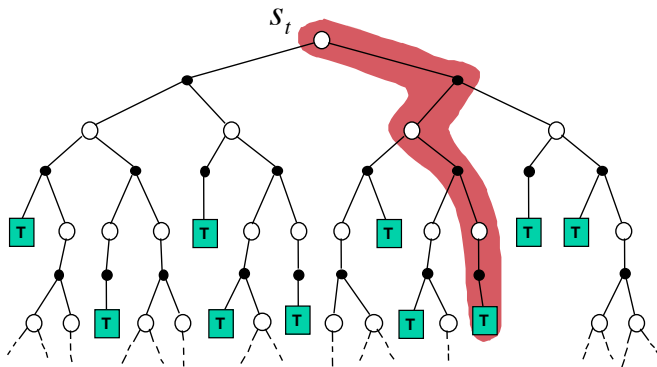
- [illegible]

- Navigation icons: back, forward, search, and other controls.



# 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 (2)

- **Simulate** episodes of experience *forward search unit* from now with the model

$$\{s_t^k, A_t^k, R_{t+1}^k, \dots, S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu$$

- Apply **model-free** RL to simulated episodes
  - Monte-Carlo control  $\rightarrow$  Monte-Carlo search
  - Sarsa  $\rightarrow$  TD search

# Simple Monte-Carlo Search

- Given a model  $\mathcal{M}_\nu$  and a **simulation policy**  $\pi$  (cf) MCTS는  $\pi$ 도 improve.

- For each action  $a \in \mathcal{A}$

- Simulate  $K$  episodes from current (real) state  $s_t$

오직 action simulation k개만  
episode 당

$$\{\overset{\text{real}}{s_t}, \overset{\text{simulation}}{a}, \underbrace{R_{t+1}^k, S_{t+1}^k, A_{t+1}^k, \dots, S_T^k}_{\text{simulation}}\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

- Evaluate actions by mean return (**Monte-Carlo evaluation**)

$$Q(s_t, a) = \frac{1}{K} \sum_{k=1}^K G_t \xrightarrow{P} q_\pi(s_t, a)$$

- Select current (real) action with maximum value

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_t, a)$$

# Monte-Carlo Tree Search (Evaluation)

- MCTS planning forward search simulation based search Monte Carlo search*
- Given a model  $\mathcal{M}_\nu$
  - Simulate  $K$  episodes from current state  $s_t$  using current simulation policy  $\pi$  *policy? algorithm? k times episode? simulate.*

$$\{s_t, A_t^k, R_{t+1}^k, S_{t+1}^k, \dots, S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

- Build a search tree containing visited states and actions
- **Evaluate** states  $Q(s, a)$  by mean return of episodes from  $s, a$

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

- After search is finished, select current (real) action with maximum value in search tree

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_t, a)$$

# Monte-Carlo Tree Search (Simulation)

- In MCTS, the simulation policy  $\pi$  improves
- Each simulation consists of two phases (in-tree, out-of-tree)
  - **Tree policy** (improves): pick actions to maximise  $Q(S, A)$
  - **Default policy** (fixed): pick actions randomly
- Repeat (each simulation)
  - **Evaluate** states  $Q(S, A)$  by Monte-Carlo evaluation
  - **Improve** tree policy, e.g. by  $\epsilon$  – greedy( $Q$ )
- Monte-Carlo control applied to simulated experience
- Converges on the optimal search tree,  $Q(S, A) \rightarrow q_*(S, A)$

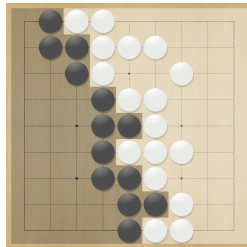
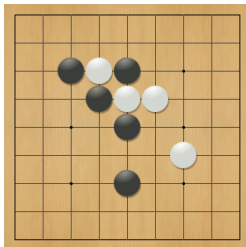
# Case Study: the Game of Go

- The ancient oriental game of Go is 2500 years old
- Considered to be the hardest classic board game
- Considered a grand challenge task for AI (*John McCarthy*)
- Traditional game-tree search has failed in Go



# Rules of Go

- Usually played on 19x19, also 13x13 or 9x9 board
- Simple rules, complex strategy
- Black and white place down stones alternately
- Surrounded stones are captured and removed
- The player with more territory wins the game



# Position Evaluation in Go

- How good is a position  $s$ ?
- Reward function (undiscounted):

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

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

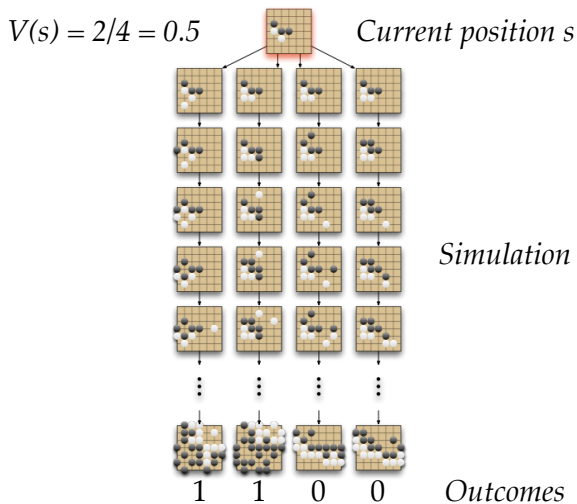
- Policy  $\pi = \langle \pi_B, \pi_W \rangle$  selects moves for both players
- Value function (how good is position  $s$ ):

$$v_\pi(s) = \mathbb{E}_\pi [R_T \mid S = s] = \mathbb{P}[\text{Black wins} \mid S = s]$$

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

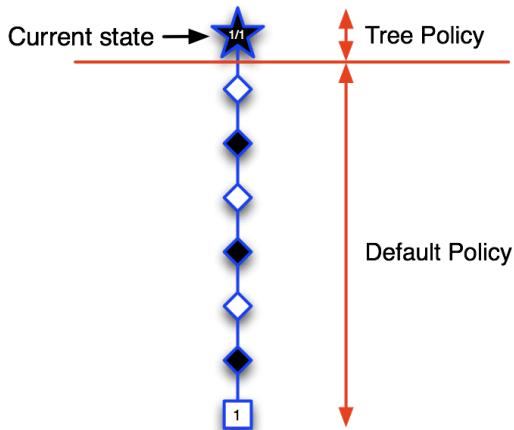


# Monte-Carlo Evaluation in Go

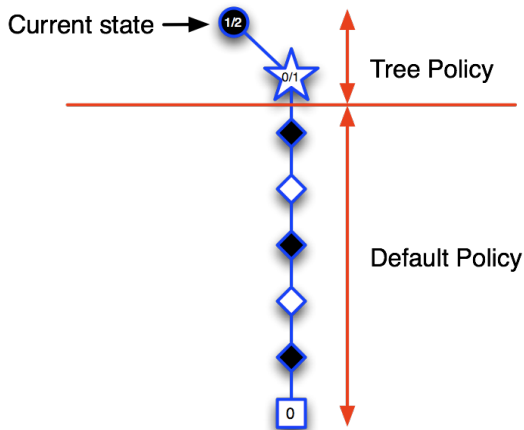


# Applying Monte-Carlo Tree Search (1)

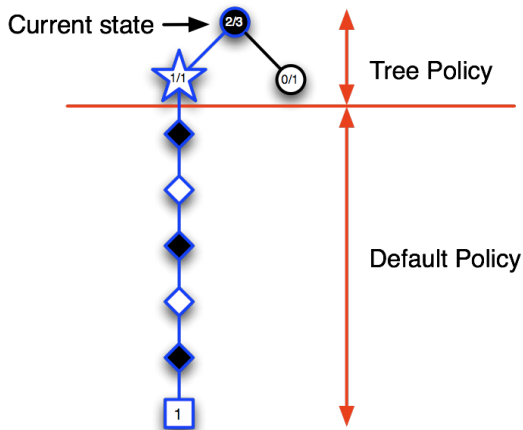
복원된 광미여



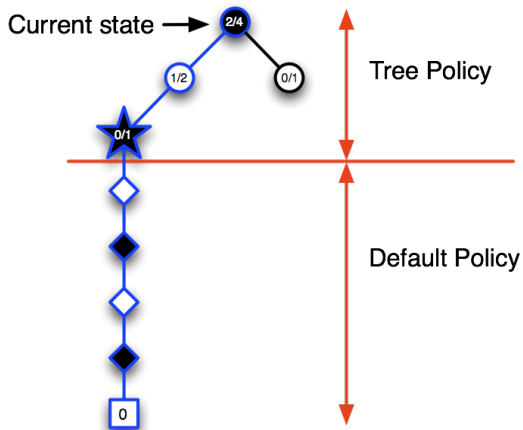
# Applying Monte-Carlo Tree Search (2)



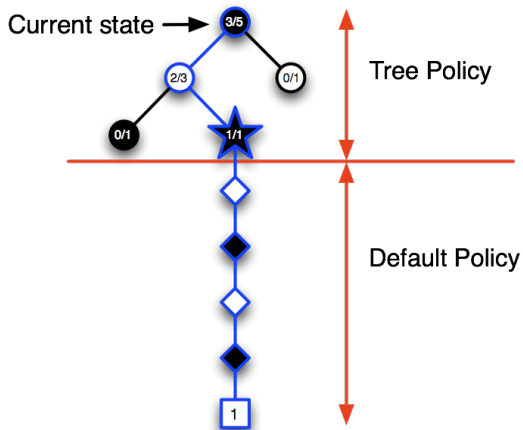
# Applying Monte-Carlo Tree Search (3)



# Applying Monte-Carlo Tree Search (4)



# Applying Monte-Carlo Tree Search (5)

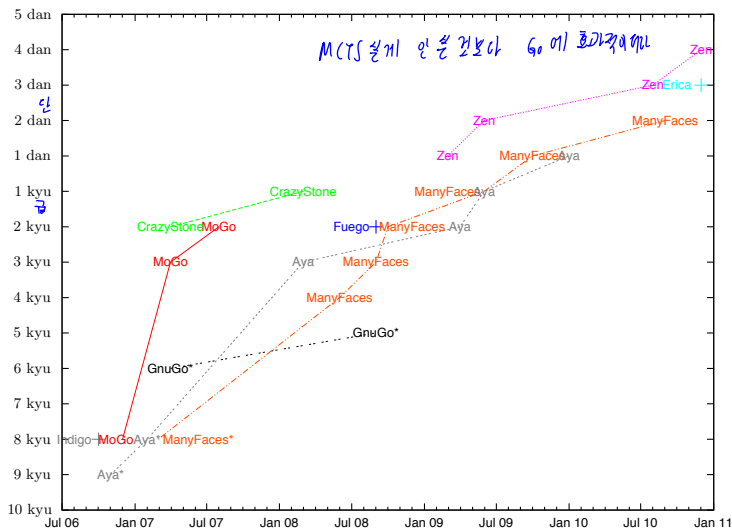


# Advantages of MC Tree Search

- Highly selective best-first search
- Evaluates states *dynamically* (unlike e.g. DP)
- Uses sampling to break curse of dimensionality
- Works for “black-box” models (only requires samples)
- Computationally efficient, anytime, parallelisable

병행성이 좋다.

# Example: MC Tree Search in Computer Go





# Temporal-Difference Search

- Simulation-based search
- Using TD instead of MC (bootstrapping)
- MC tree search applies MC control to sub-MDP from now
- TD search applies Sarsa to sub-MDP from now

# 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
  - $TD(\lambda)$  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
  - $TD(\lambda)$  search can be much more efficient than MC search

# TD Search

- Simulate episodes from the current (real) state  $s_t$
- Estimate action-value function  $Q(s, a)$
- For each step of simulation, update action-values by Sarsa

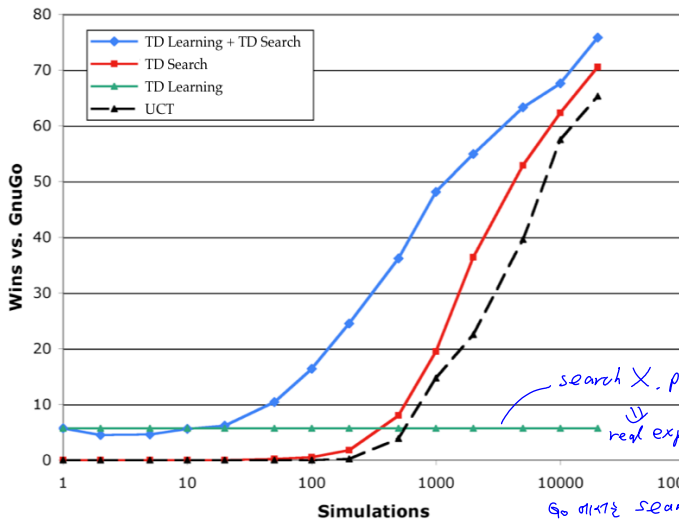
$$\Delta Q(S, A) = \alpha(R + \gamma Q(S', A') - Q(S, A))$$

- Select actions based on action-values  $Q(s, a)$ 
  - e.g.  $\epsilon$ -greedy
- May also use function approximation for  $Q$

# Dyna-2

- In Dyna-2, the agent stores two sets of feature weights
  - Long-term memory
  - Short-term (working) memory
- Long-term memory is updated from real experience using TD learning
  - General domain knowledge that applies to any episode
- Short-term memory is updated from simulated experience using TD search
  - Specific local knowledge about the current situation
- Over value function is sum of long and short-term memories

# Results of TD search in Go



search X, planning X

⇓  
real experience만 쓰면  
효율이 높음.

Go에서는 search가 중요하다.

# Questions?