Lecture 8: Integrating Learning and Planning

David Silver

The final comment is:
He really just wants to emphasize the main ideas that planning simulation-based search and effective method for planning is to use our model, just sample trajectories to measure what (might) happens next and apply your favorite RL method whether is MC control, SARSA, combining real experience and simulated experience, and so forth.
The main idea is that learning from simulation is effective method to search.

Outline

Outline

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

What I mean by planning is something like look ahead trying to figure out by using our model to mention what might happened going into the future.

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

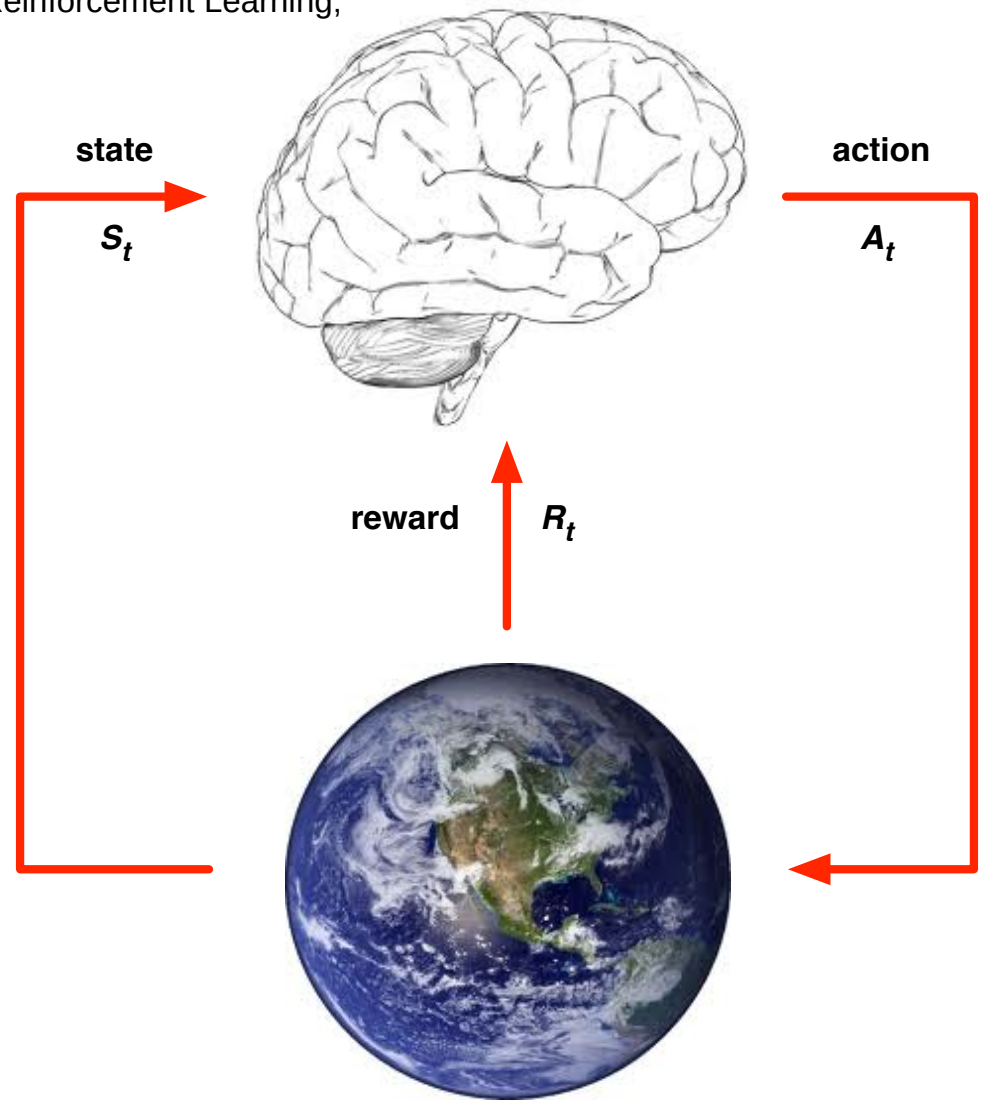
- Model-Free RL
  - No model
  - Learn value function (and/or policy) from experience
- Model-Based RL
  - Learn a model (MDP model) from experience
  - Plan value function (and/or policy) from model

So, the mean by planning, basically using a model, is to look ahead, to think, to compute, to figure out what the right value function and the actions are to select in this environment.

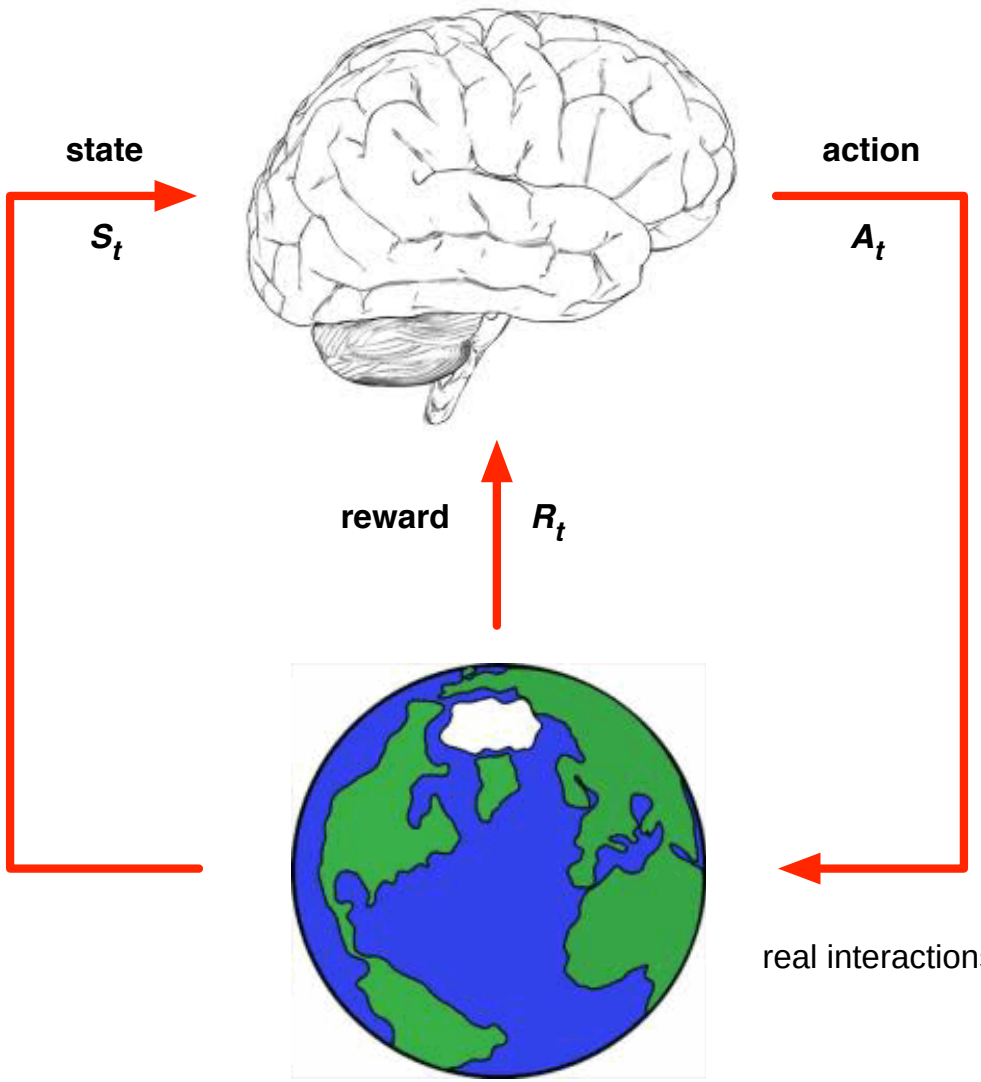
Planning is the process that is learning about policy/value function

Model-Free RL

The fundamental cycle in the Reinforcement Learning;



Model-Based RL

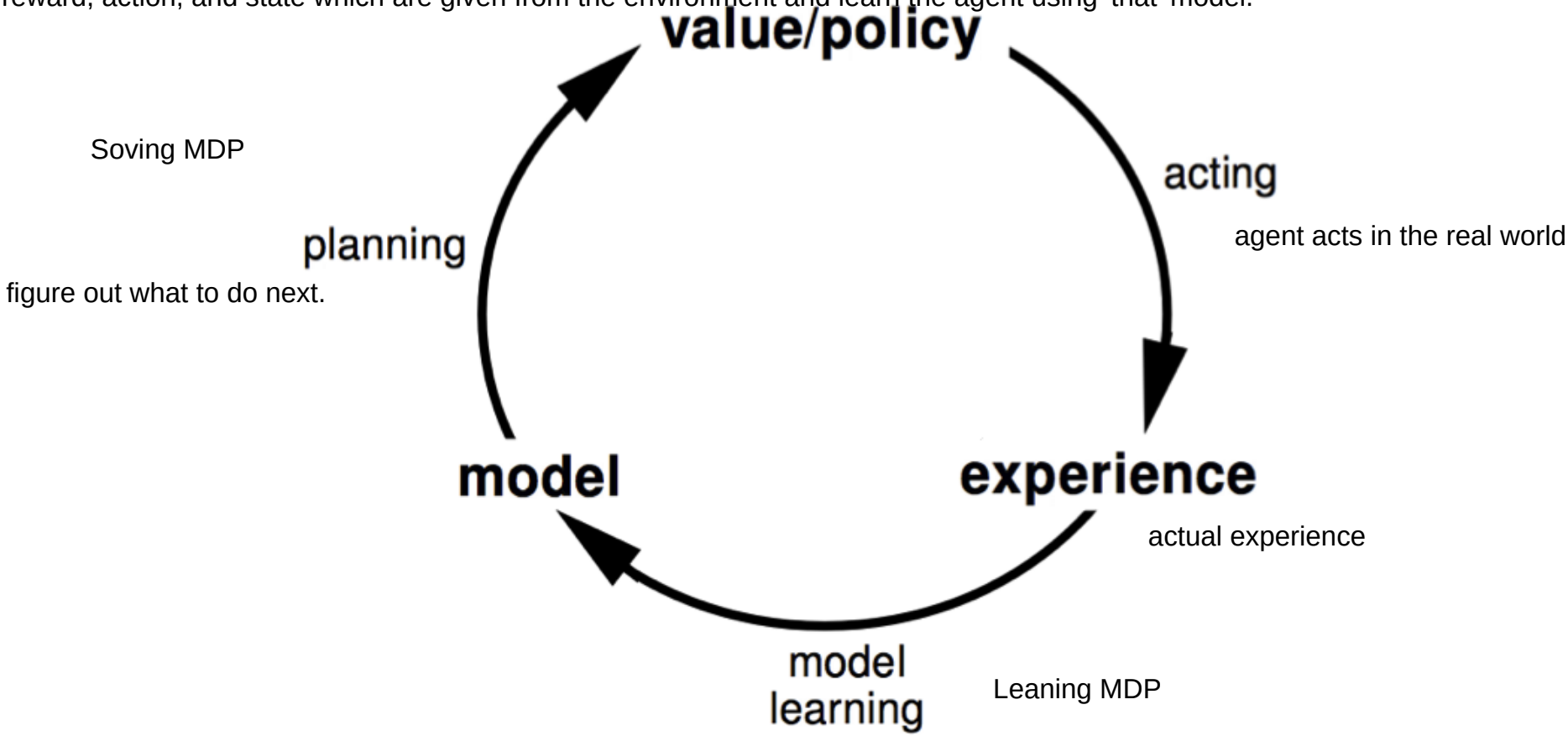


real interactions the agent having

Outline

Model-Based RL

This is a planning (it means learning in model-free) which learns our agent using model which is made by the given environment and rewards, actions, and states from it (since this model is MDP). That is, we make a model from the given environment using reward, action, and state which are given from the environment and learn the agent using 'that' model.



Advantages of Model-Based RL

Advantages:

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

Model gives you a very useful way to reason about what you know and what you don't know in the world.

Disadvantages:

- First learn a model, then construct a value function
  $\Rightarrow$ two sources of approximation error

Model might be approximated (estimated), then use the approximated model to learn a value function, which might also be an approximation. (error estimates error)
If you learn an incorrect model and plan with that, then you will get an incorrect answer.

What is a Model?

More formalliy now

- A *model* $\mathcal{M}$ is a representation of an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$, parametrized by η
- 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 $\mathcal{R}_\eta \approx \mathcal{R}$

one step model (you can do longer time steps as well)

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

- Typically assume conditional independence between state transitions and rewards

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

It is necessary to plan to figure out exactly what situation you're in. Without doing this process of lookahead, it is very hard to say what the next thing happens.

Model Learning

- Goal: estimate model $\mathcal{M}_\eta$ from experience $\{S_1, A_1, R_2, ..., S_T\}$
  until the end of trajectory
- This is a supervised learning problem

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

training sets

how much reward do I expect to get?

- Learning $s, a \rightarrow r$ is a *regression* problem
- Learning $s, a \rightarrow s'$ is a *density estimation* problem

want to learn how distribution of what next state of here

- Pick loss function, e.g. mean-squared error, KL divergence, ...
- Find parameters η that minimise empirical loss

An MDP which has large scale needs to solve with neural network or general approximators.
Everything you've learned about the supervised learning can be applied to here.

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

All I need to do is to count. I just count how many times from here (state and action) I visit each state action pair.

transition dynamic

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

average

parametric

reward dynamic

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

average of rewards

- Alternatively (remember)
  - At each time-step t, record experience tuple you've seen so far $\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$
  - To sample model, randomly pick tuple matching $\langle s, a, \cdot, \cdot \rangle$

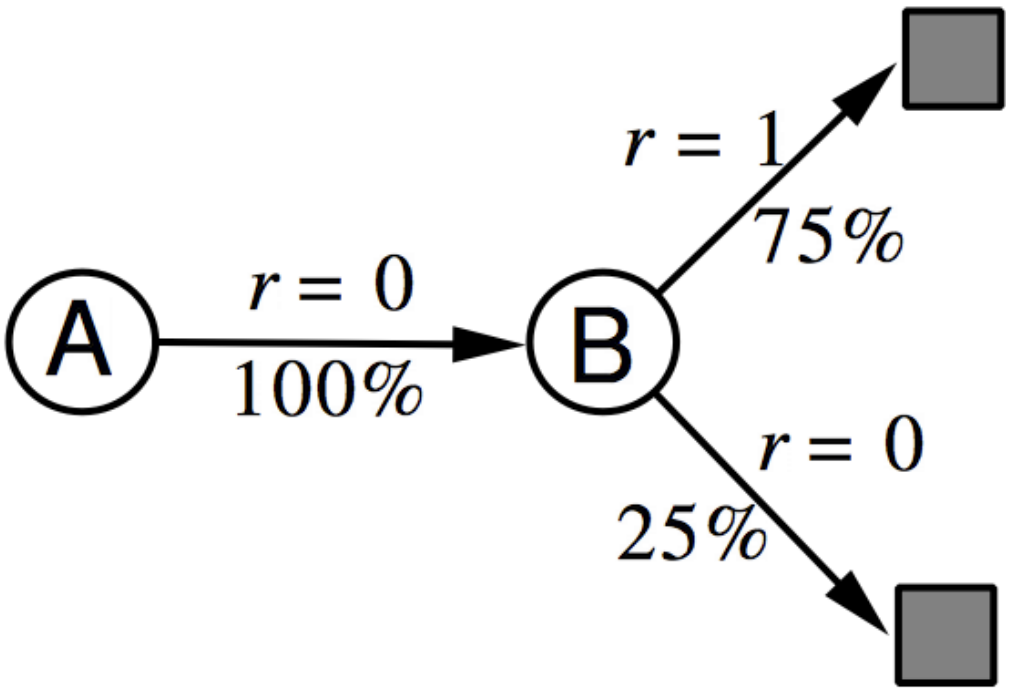
We can do planning using this process.

non-parametric

AB Example

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

$$\begin{aligned}
&\mathrm{A}, 0, \mathrm{B}, 0 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 0
\end{aligned}$$



We have constructed a table lookup model from the experience

Planning with a Model

So far, we talked about learning in previous lecture.

Now, if we model a model, (build a model), we can do planning.
Planning MDP means solving the MDP. Also, it means to find the optimal value function and optimal policy, hence, the best action.

- 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
  - ...

We can apply all learning methods.

Sample-Based Planning

In some sense, this is the simplest possible way to plan, but it also turns out to be the most powerful way t o plan.

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

unlike dynamic programming where you actually look at probability transitions kind of intergrated over those probability.

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

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

We know how to learn sample trajectories.

- Apply model-free RL to samples, e.g.: `Apply model-free RL to samples that we generate using the model above.`
  - Monte-Carlo control
  - Sarsa
  - Q-learning

`Another way to think of that is the agent has some model in his head and imagines what is going to happen next, and he plans by solving for imagined world and simulates experiences and all the rewards ....sample trajectory`

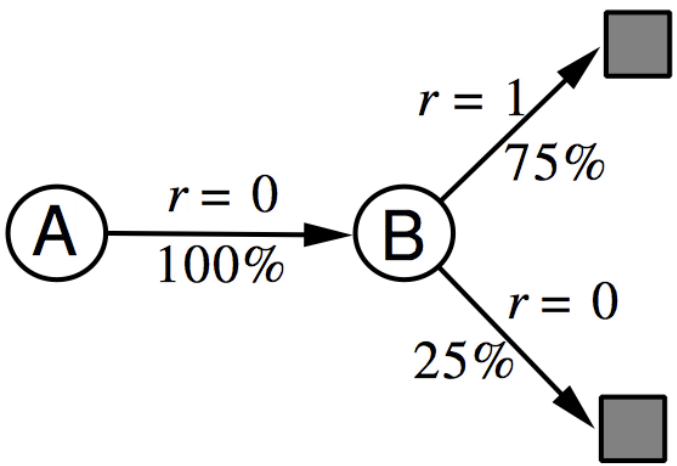
- Sample-based planning methods are often more efficient

Back to the 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, 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: $V(A) = 1, V(B) = 0.75$

If we generate more and more samples or data from model, we will get ultimately the right answer.

Planning with an Inaccurate Model

If we build an imperfect model, we can't learn(plan) the environment properly which we want to learn from.

What happens if our model is not right? (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

and then, we check what is wrong.

Outline

So all of these learning approaches are the normal cases as function approximation.

Real and Simulated Experience

We're really gonna try to bring together now, the best model-free and model-based reinforcement learning, trying to construct something which has advantage of both by combining together.

We consider two sources of experience

Real experience Sampled from environment (true MDP)

real interactions from the environment

We can use the real data from experience if we feel the lack of data (sample data which is simulated experience).

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

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

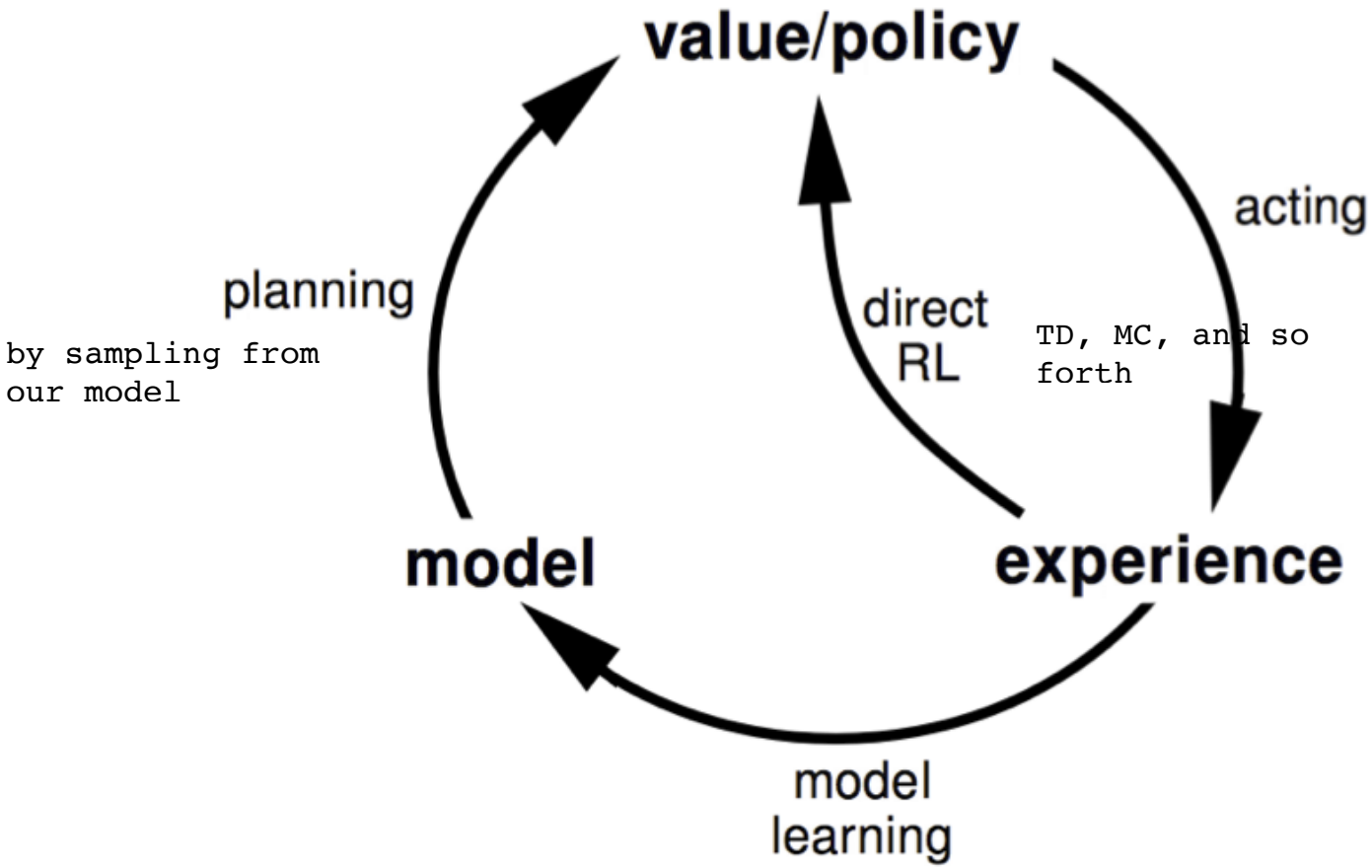
How to integrate learning and planning together.

- 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

simplest version of Dyna

action value function

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) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Execute action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

q-learing update

but also update our model by supervised learning

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

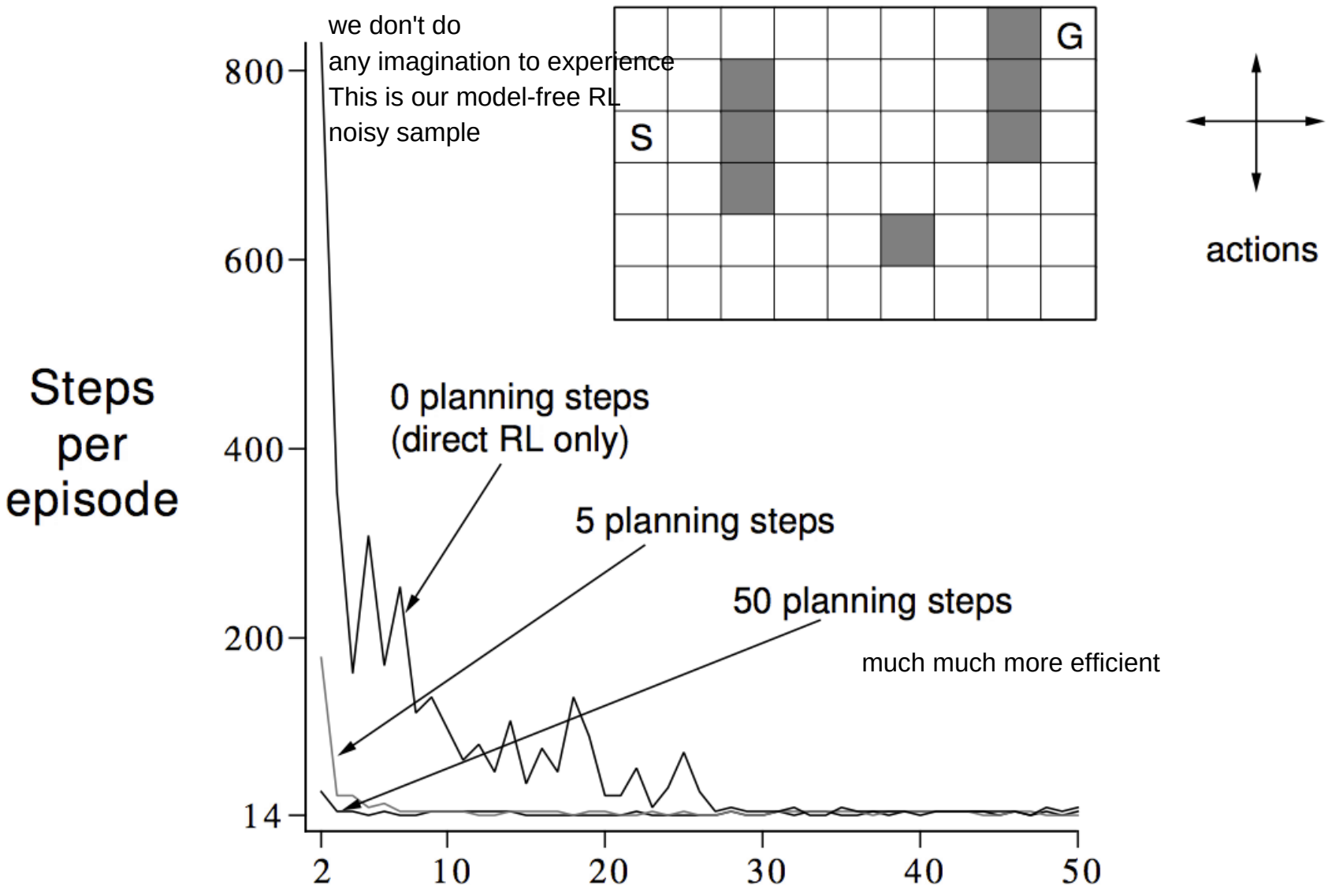
reward and next state we observed after that one step

- (f) Repeat n times:
  - $S \leftarrow$ random previously observed state — we've already seen
  - $A \leftarrow$ random action previously taken in S — we've already taken
  - $R, S' \leftarrow Model(S,A)$ — sample is from memory based model
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

update q value toward the best q value where I ended up (in my imagination)

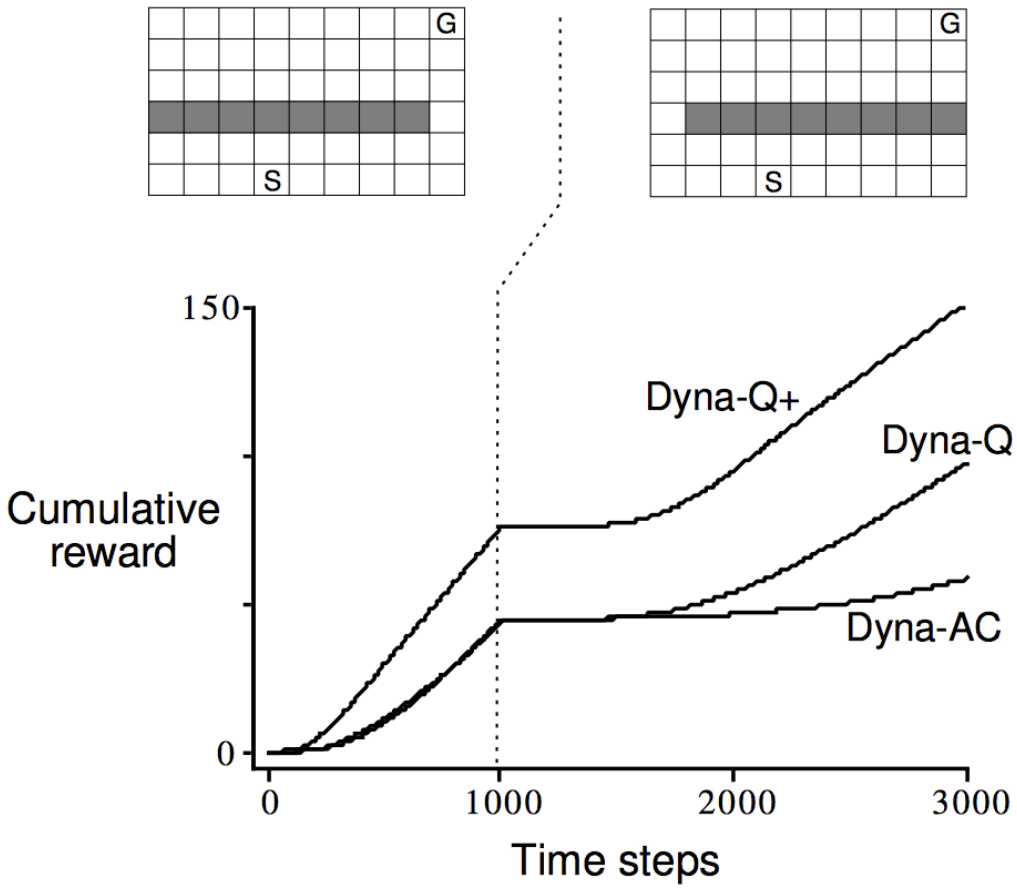
What we're gonna consider is different amount of thinking time. How much do I loop over the sample from our model?

Dyna-Q on a Simple Maze



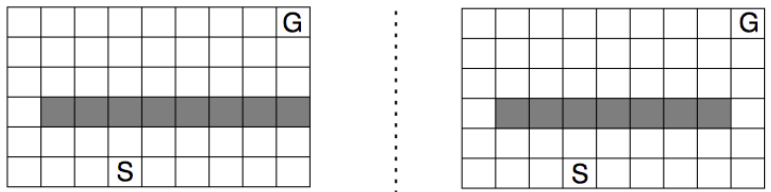
Dyna-Q with an Inaccurate Model

- The changed environment is harder

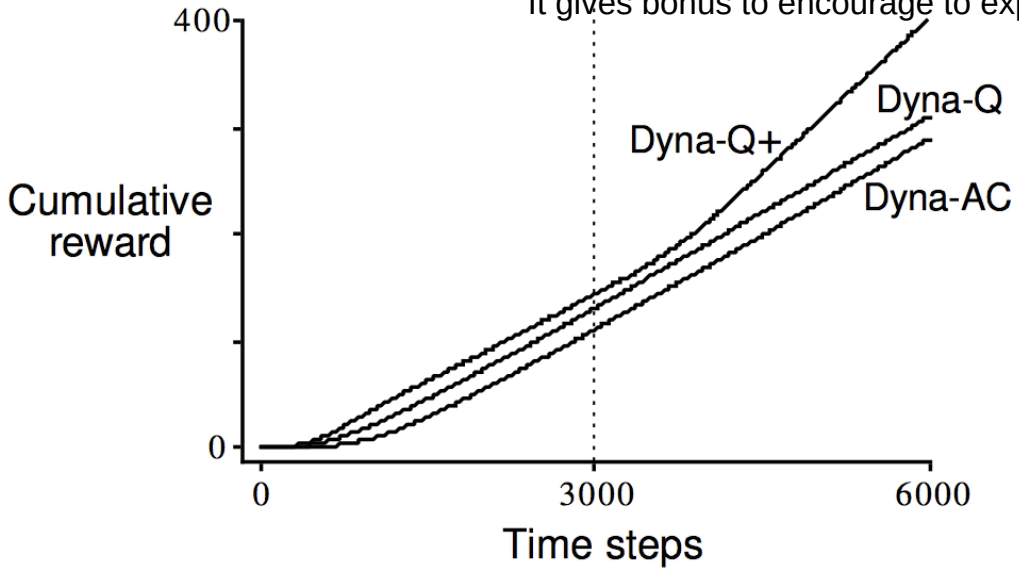


Dyna-Q with an Inaccurate Model (2)

- The changed environment is easier



It is basically the bonus for states you haven't visited before.
It gives bonus to encourage to explore more.



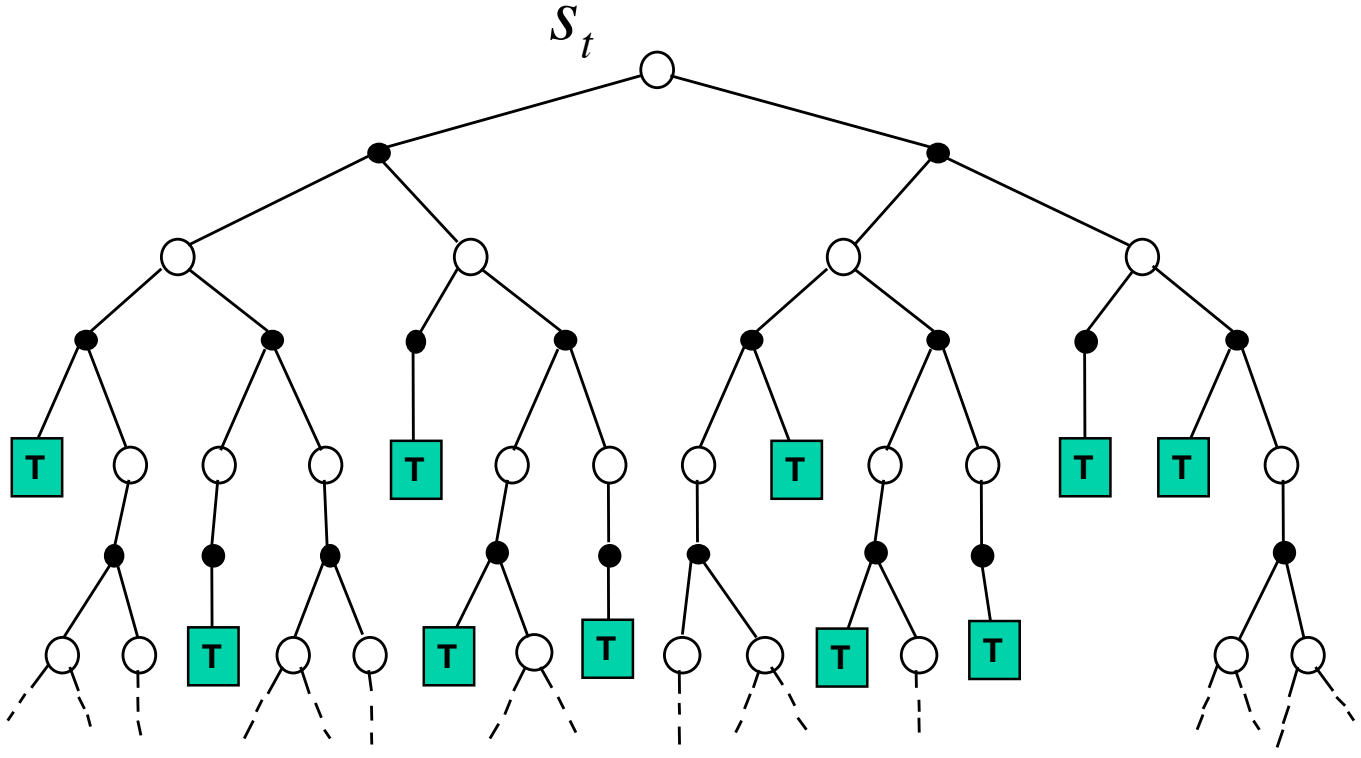
Outline

Last section is a really different view on planning now.

Forward Search

Back of the just one part of planning problem. You can still imagine that we're doing model-based RL and learning a model. We're gonna focus more on the planning part; how to plan effectively. The key idea is sampling and forward search.

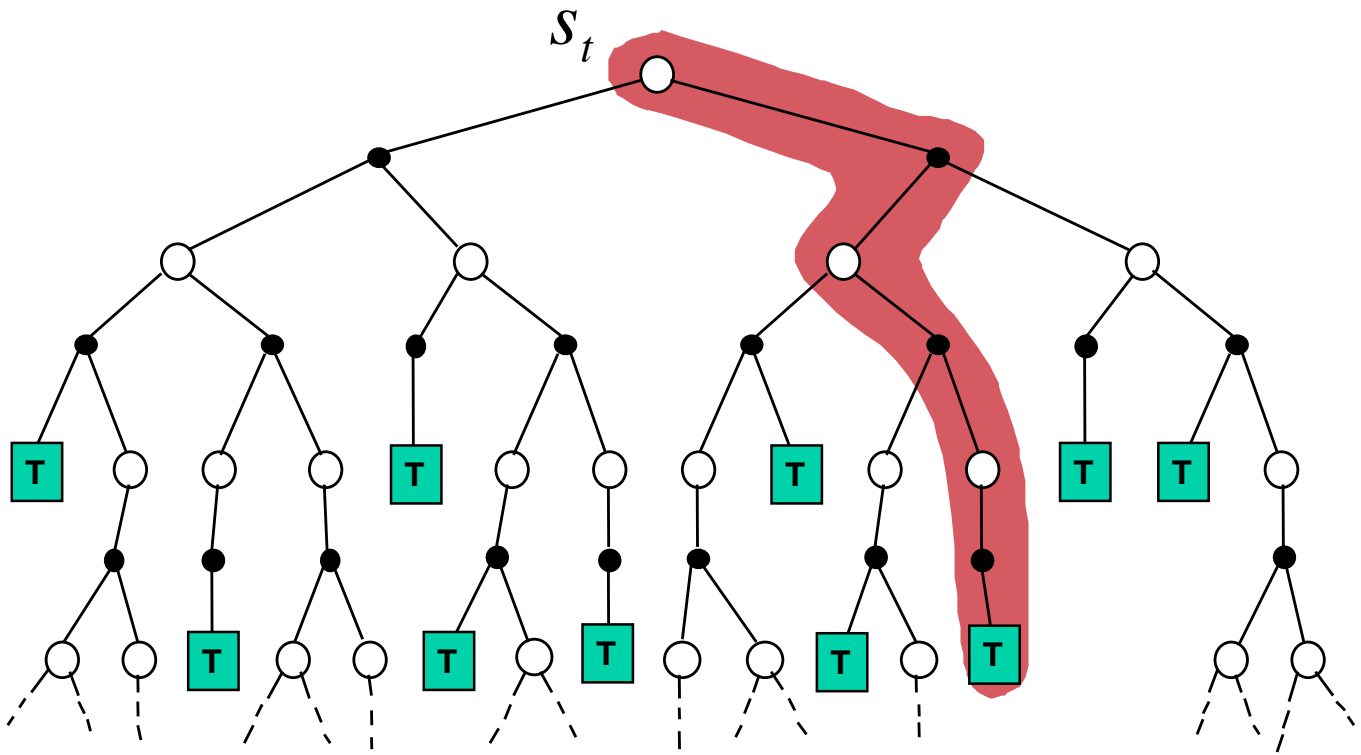
- Forward search algorithms select the best action by lookahead
- They build a search tree with the current state s_t at the root
- Using a model of the MDP to look ahead



- No need to solve whole MDP, just sub-MDP starting from now

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)

root state S_t

- Simulate episodes of experience from now with the model

K times simulations

$$\{s_t^k, A_t^k, R_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu$$ from our model (real world)

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

You can pick and use your favorite method to apply to simulated experience and you have search algorithm. (LSPI, and so forth)

Simple Monte-Carlo Search

Let's start the simplest possible version.

- Given a model $\mathcal{M}_\nu$ and a simulation policy π

some way that we're going to pick actions in our imagination

- For each action $a \in \mathcal{A}$
  - Simulate K episodes from current (real) state s_t

$$\{\overset{\text{root}}{s_t}, a, R^k_{t+1}, S^k_{t+1}, A^k_{t+1}, ..., S^k_T\}^K_{k=1} \sim \mathcal{M}_\nu, \pi$$

We can consider all the actions we can take from that root state.

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

We evaluate each of those in turn. And Q is the action value function for the root.

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

by law of large number

for a simulation policy \pi

- Select current (real) action with maximum value

actions which we can take from current root state

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

Monte-Carlo Tree Search (Evaluation)

This can solve really challenging problems and very effective planning method.

- Given a model $\mathcal{M}_\nu$
- Simulate K episodes from current state s_t using current simulation policy π

The difference is that we use the simulation policy to take action from current state (root state).

root state

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

It precisely means that all of the states we visited so far and all of the actions we tried from those states so far.

- 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

There's one big difference now which is we're going to get rich information in a search tree and we can use the rich information to make our simulation policy better.

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

Monte-Carlo Tree Search (Simulation)

So the way we do that now is that after every simulation, we're going to make a simulation's improve (improvement). We do that in same way that we do policy improvement in previous classes, where we basically look up the q values and we're going to maximize over the q values in a search tree to make them better. The only distinction is that , here, we don't have a complete table of q values everywhere and we've only got the search tree. So, break up our simulation as two phases.

- In MCTS, the simulation policy π 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 $\epsilon - \text{greedy}(Q)$
- Monte-Carlo control applied to simulated experience
- Converges on the optimal search tree, $Q(S, A) \rightarrow q_*(S, A)$

beyond the tree where we just don't have any information

every node about tree

or smarter strategies

already know

This algorithm which turns out is one we've already seen before, which is basically MC control applied to simulated episode experience which start from root state. So we start from this moment now, we're running our experience from now onwards and applying MC control to the experience which we encounter now onwards. (familial method before)

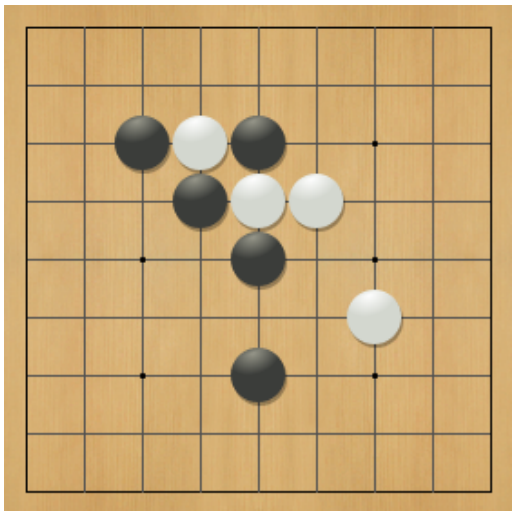
We focus on what's going to happen next and need to understand this (contingency)

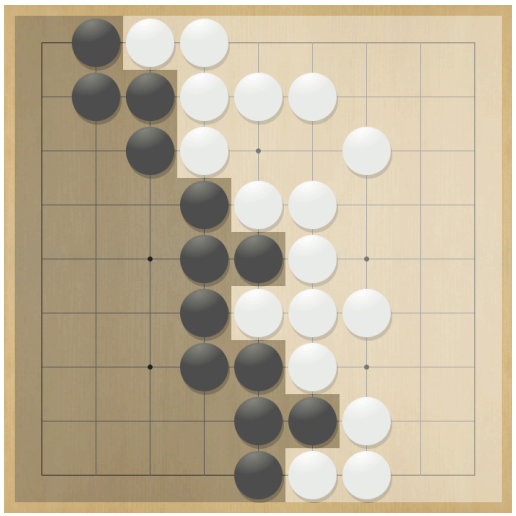
Case Study: the Game of Go

- 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):

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

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

mini-max value

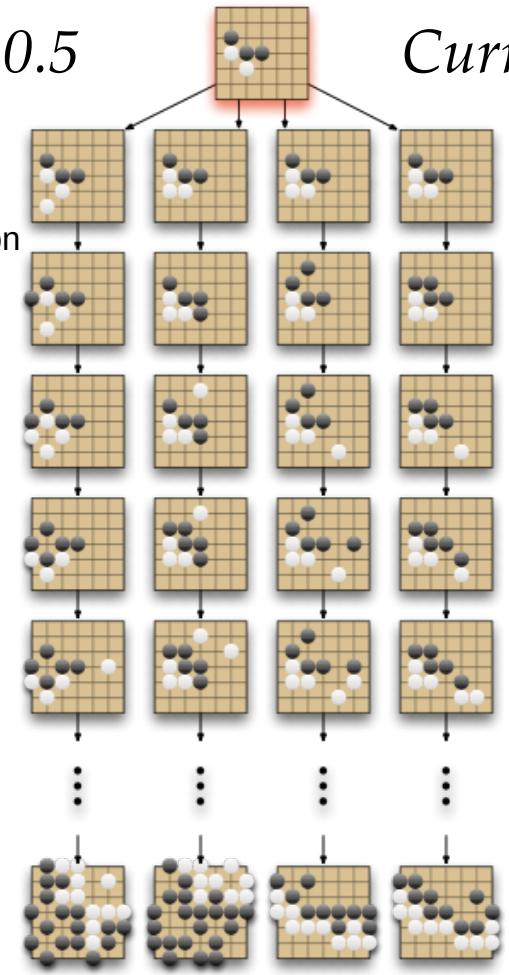
We'll more see in the last lecture (lecture 10).

Monte-Carlo Evaluation in Go

$V(s) = 2/4 = 0.5$

Current position s

forward search
(sort of really focusing on this start situation
ignoring all the many other games we
might be relevant right now)

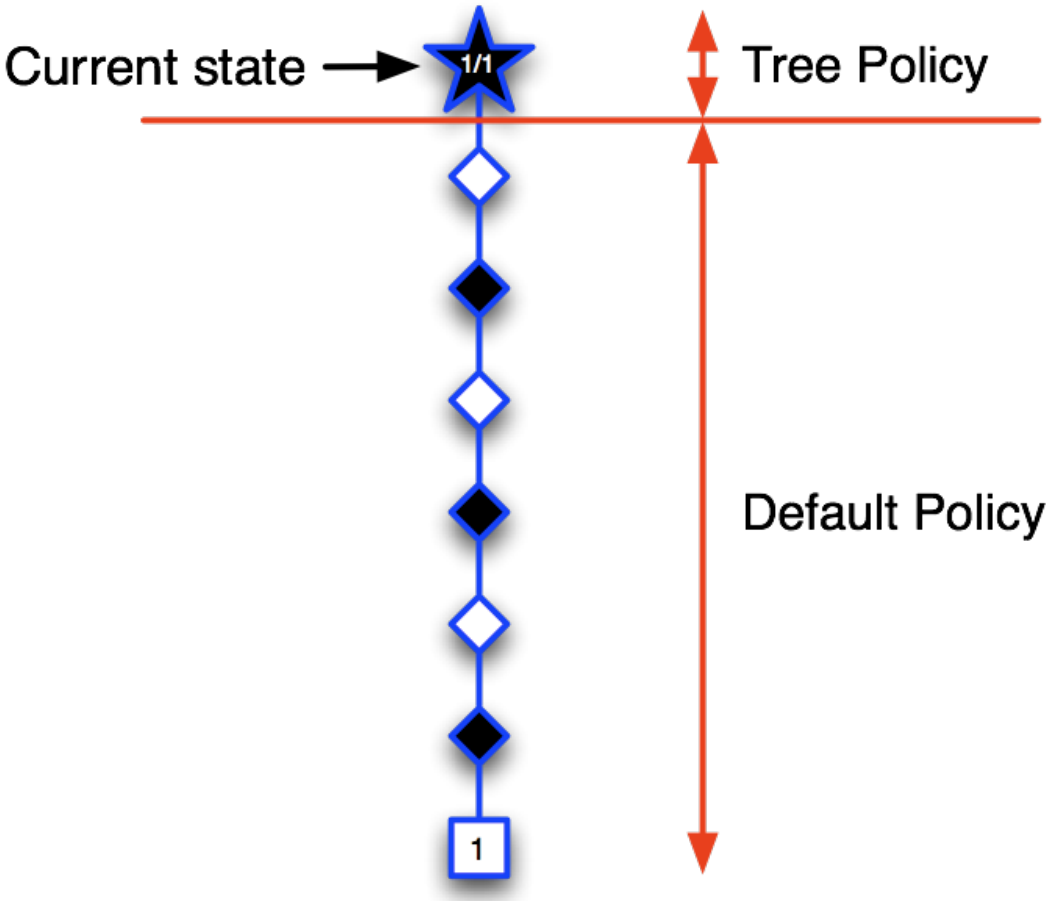


Simulation

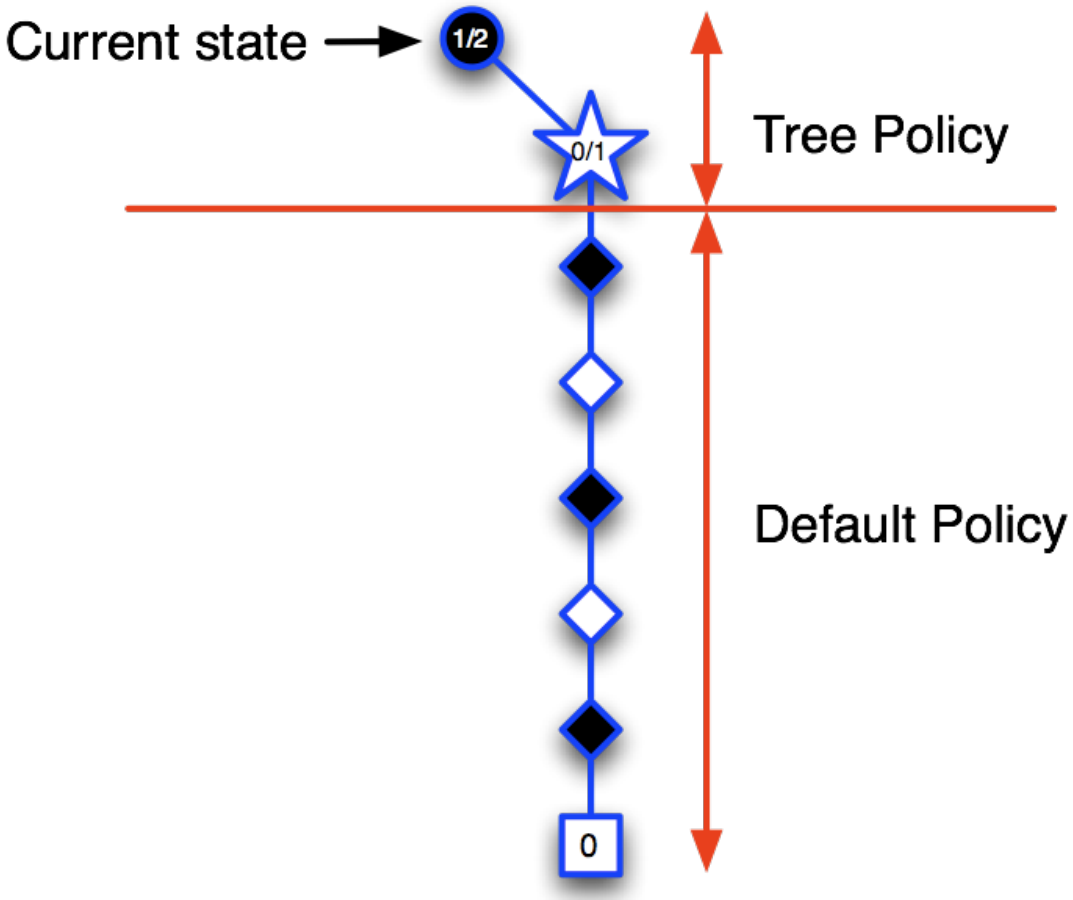
1 1 0 0

Outcomes

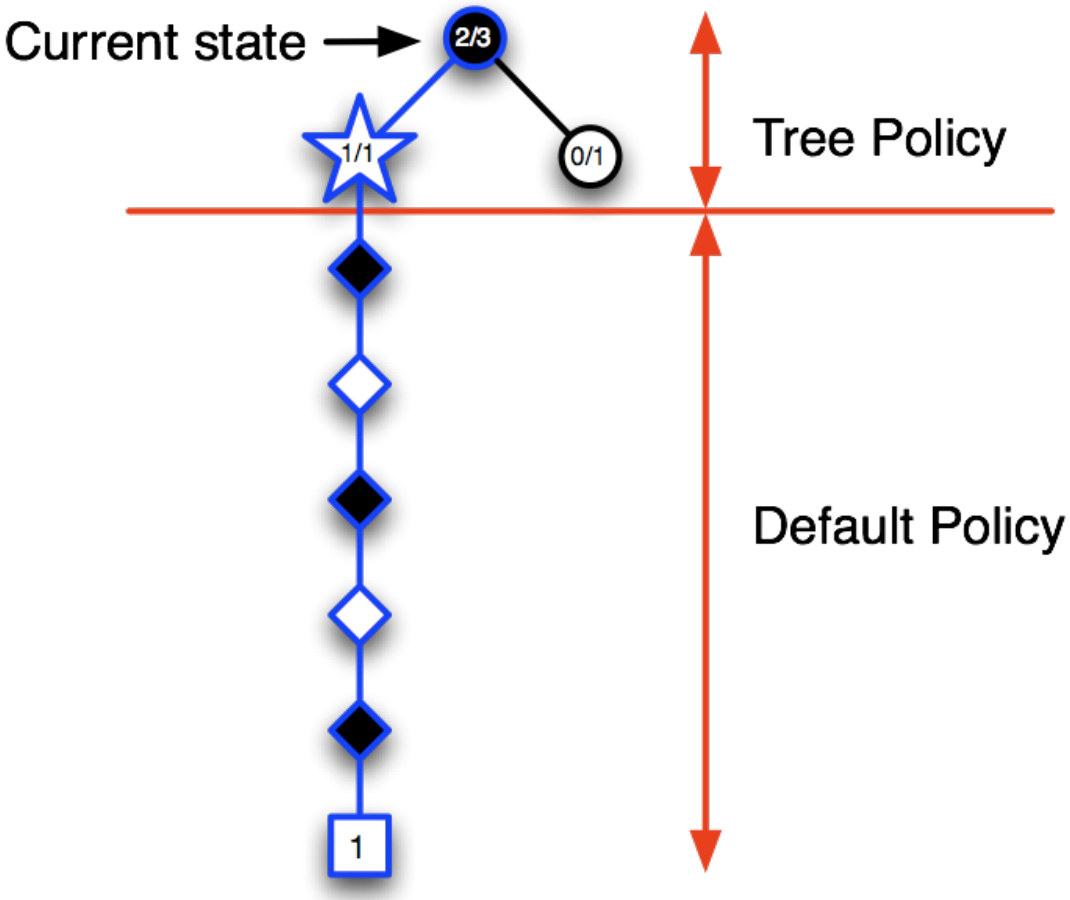
Applying Monte-Carlo Tree Search (1)



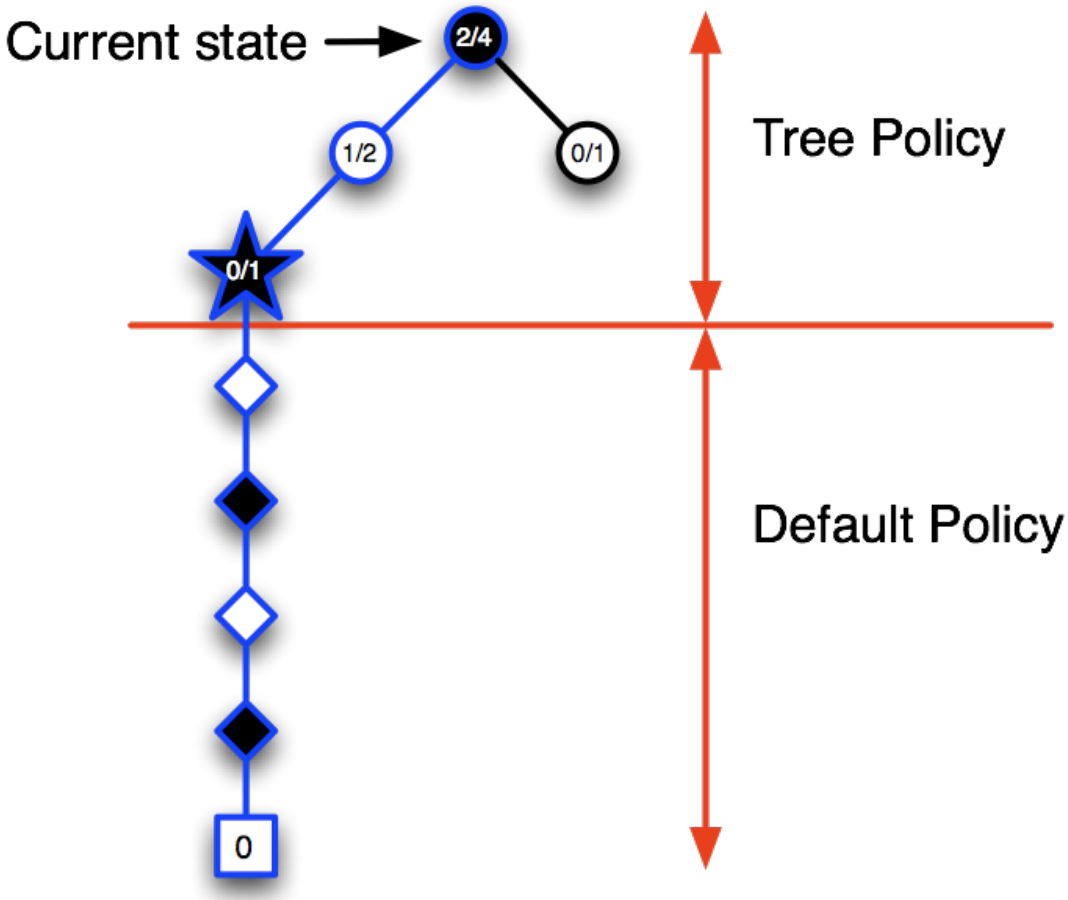
Applying Monte-Carlo Tree Search (2)



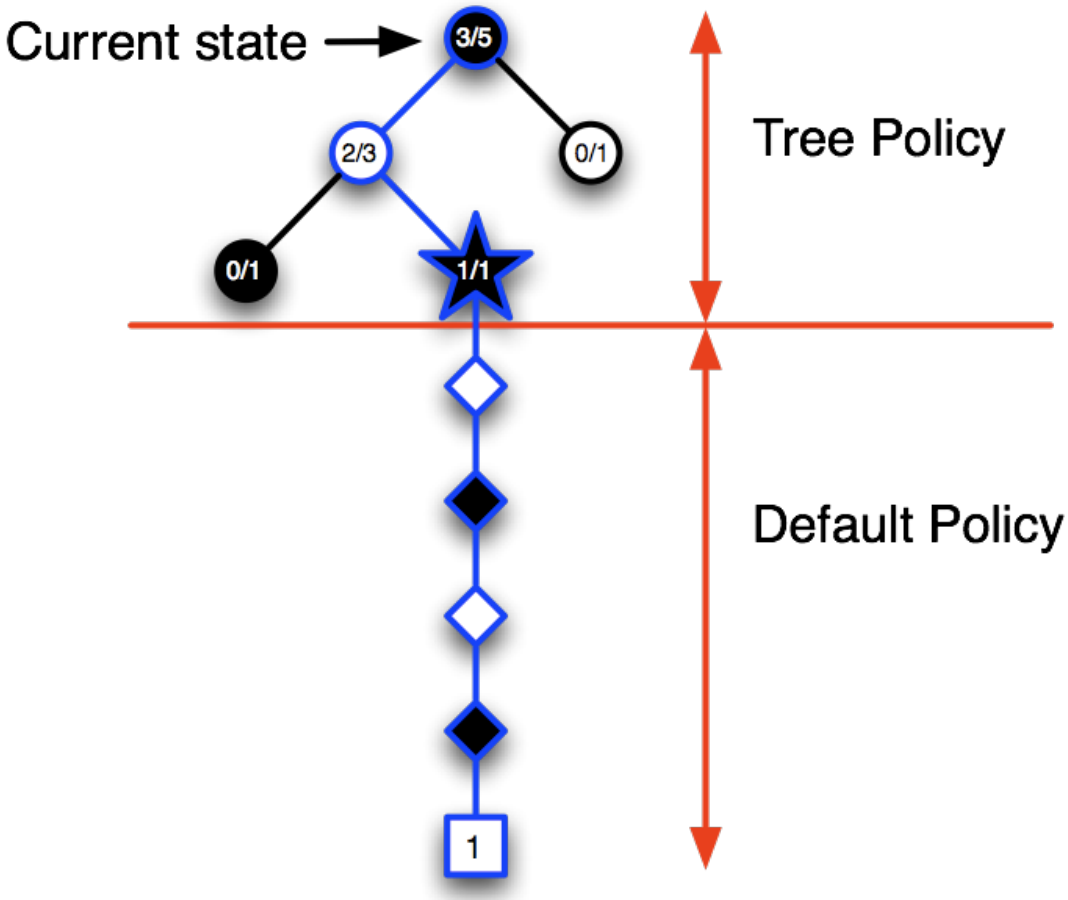
Applying Monte-Carlo Tree Search (3)



Applying Monte-Carlo Tree Search (4)



Applying Monte-Carlo Tree Search (5)



Advantages of MC Tree Search

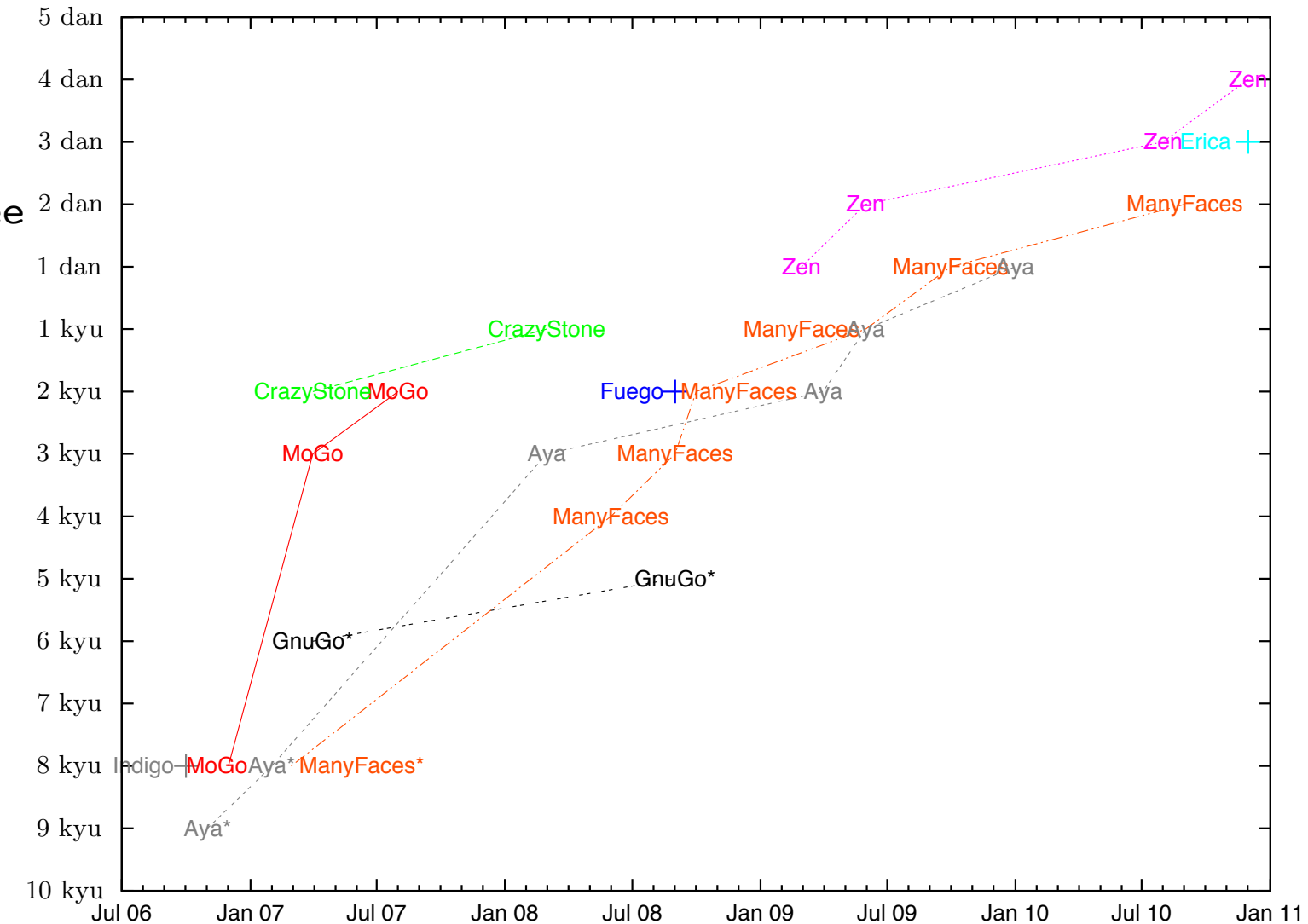
So, why is this good idea?

- Highly selective best-first search 경우의 수가 아주 많은 경우 사용하기 좋다.
- 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

lots of good property 복잡한 문제를 효과적으로 접근할 수 있다.

Example: MC Tree Search in Computer Go

one tree move



Temporal-Difference Search

MC tree search approach is very effective method for planning. Now, we want just bringing us back.
Also, there are many good property of TD in case of model-free RL.
We can also apply to simulation-based search.

```
We're gonna get episode using simulation from model and update Q in each
step using bootstrapping
```

- 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

Why should we do this?

- 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

How to?

- 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. ϵ-greedy
- May also use function approximation for Q

There's no reason we have to represent our Q values (action-value functions) (in both cases TD and MC) using search trees or using table lookup

Dyna-2

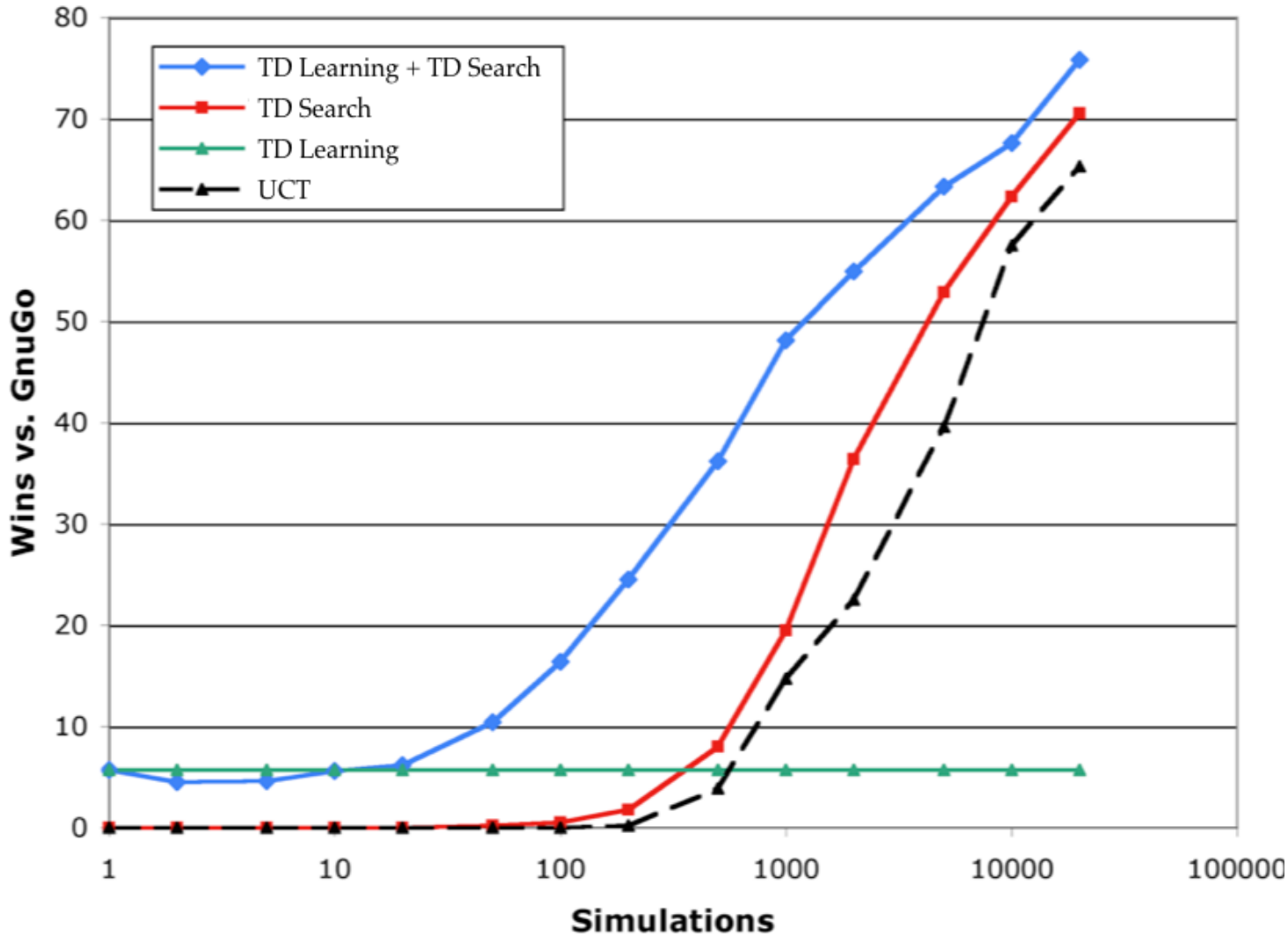
Dyna idea is that we can combine both real experience and simulated experience.
Let's do the same thing with our forward search algorithm now.

- In Dyna-2, the agent stores two sets of feature weights
  - Long-term memory
  - Short-term (working) memory

Dyna-2 basically maintains two value functions.

- 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



Questions?