Integrating Learning and Planning

I-Chen Wu

- Sutton, R.S. and Barto, A.G., Reinforcement Learning: An Introduction, MIT Press, Cambridge, MA, 1998. (Bible for RL)
  – http://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html
  – Chapters 2&9
- David Silver, Online Course for Deep Reinforcement Learning.
  – http://www.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html
  – Chapters 8-9

Outline

- Introduction
- Model-Based Reinforcement Learning
- Integrated Architectures
- Simulation-Based Search

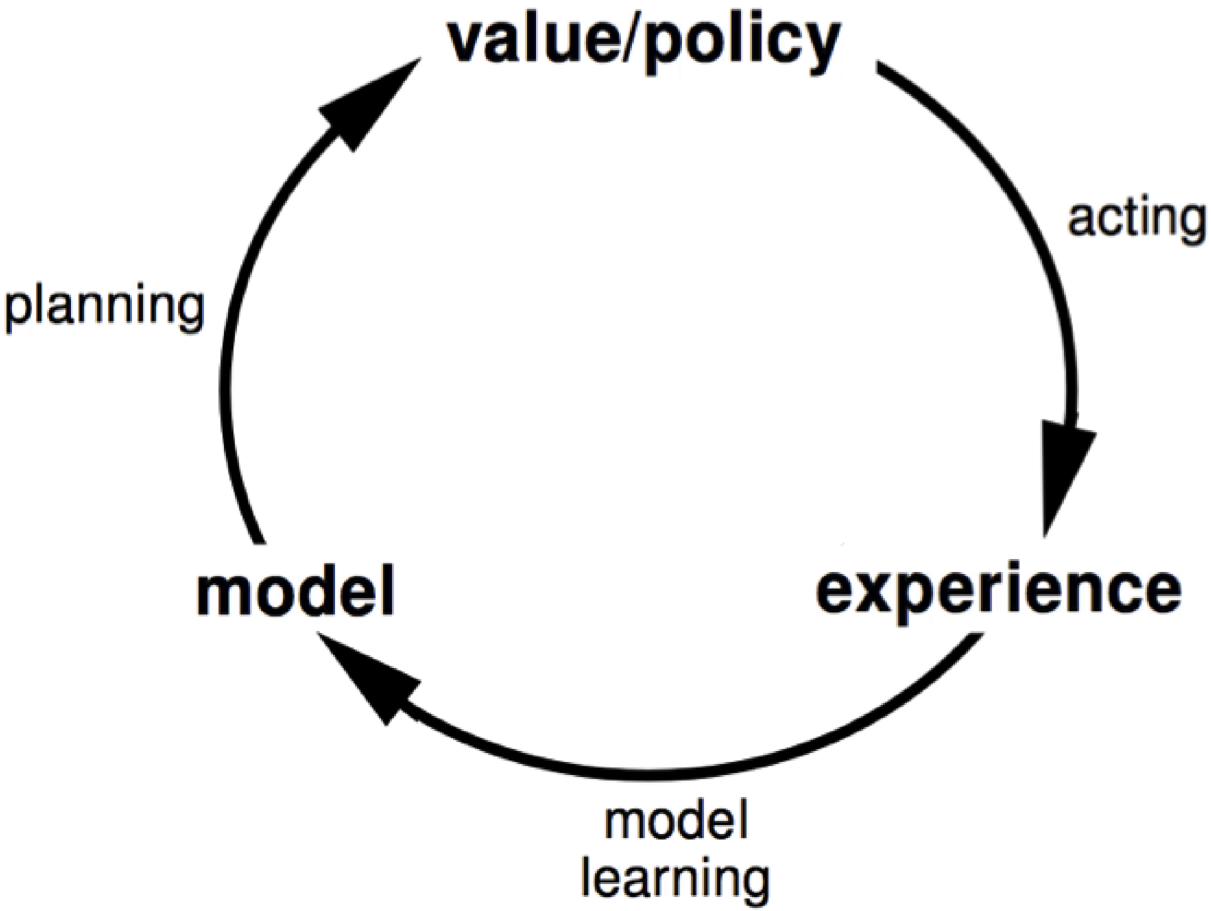
Model-Based Reinforcement Learning

- Previous lectures:
  - learn policy directly from experience
  - learn value function directly from experience
- This lecture:
  - learn model directly from experience
  - 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 RL
  - Learn a model from experience
  - Plan value function (and/or policy) from model

Model-Based RL



Advantages of Model-Based RL

- Advantages:
  - 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 $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$, 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 $\mathcal{R}_\eta \approx \mathcal{R}$

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

- 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

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

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

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

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

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

- 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 $\langle s, a, \cdot, \cdot \rangle$

AB Example

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

  A, 0, B, 0

  B, 1

  B, 1

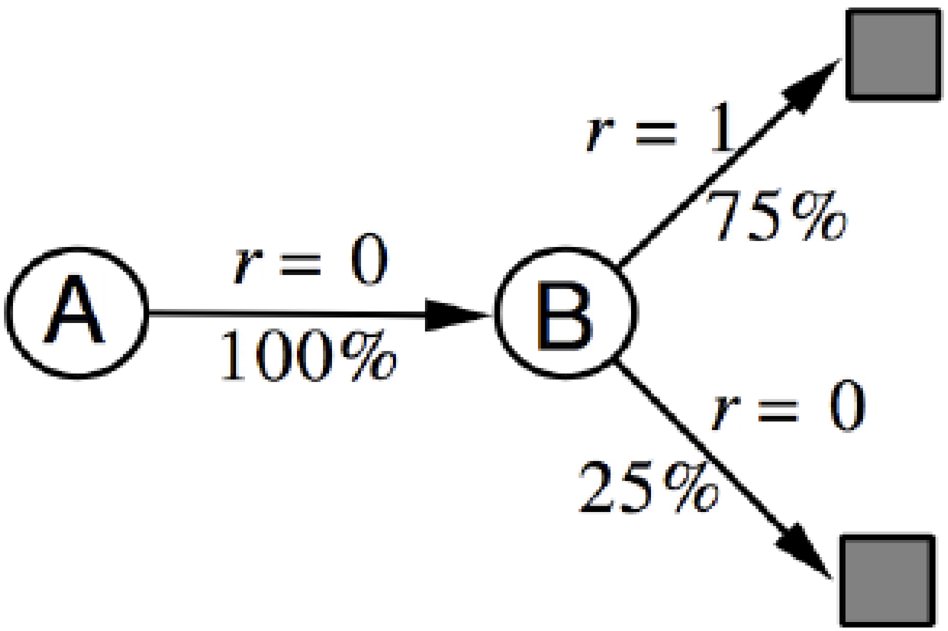
  B, 1

  B, 1

  B, 1

  B, 1

  B, 0



- We have constructed a table lookup model from the experience

Planning with a Model

- 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 favorite planning algorithm
  - Value iteration (previous lectures)
  - Policy iteration (previous lectures)
  - Tree search
  - ...

Sample-Based Planning

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

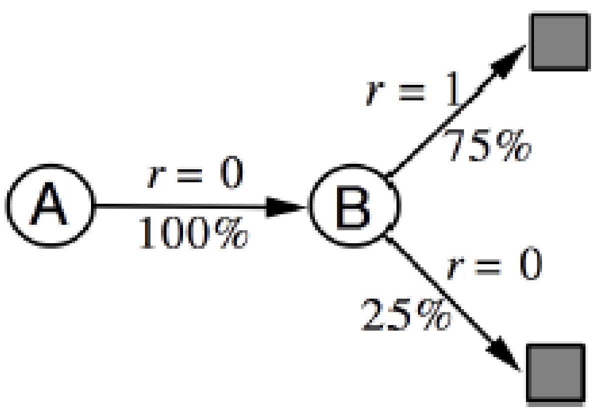
$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1}|S_t, A_t)$$
$$R_{t+1} \sim \mathcal{R}_\eta(R_{t+1}|S_t, A_t)$$

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

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, 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

Planning with an Inaccurate Model

- Given an imperfect model $\langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle \neq \langle P, R \rangle$
- 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
- Solutions
  - when model is wrong, use model-free RL
  - reason explicitly about model uncertainty

Real and Simulated Experience

- We consider two sources of experience
- Real experience
  - Sampled from environment (true MDP)

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

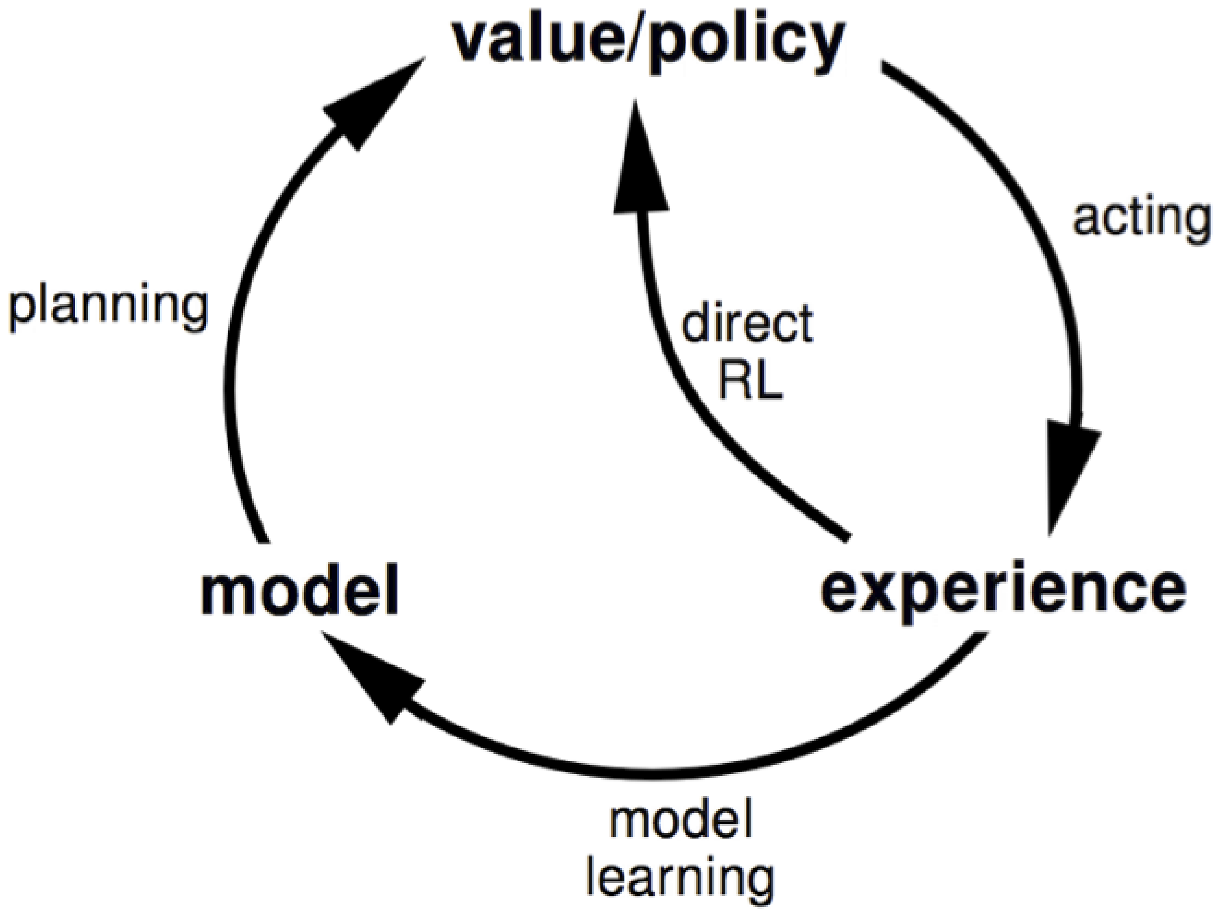
- Simulated experience
  - Sampled from model (approximate MDP)

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

Integrating Learning and Planning

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

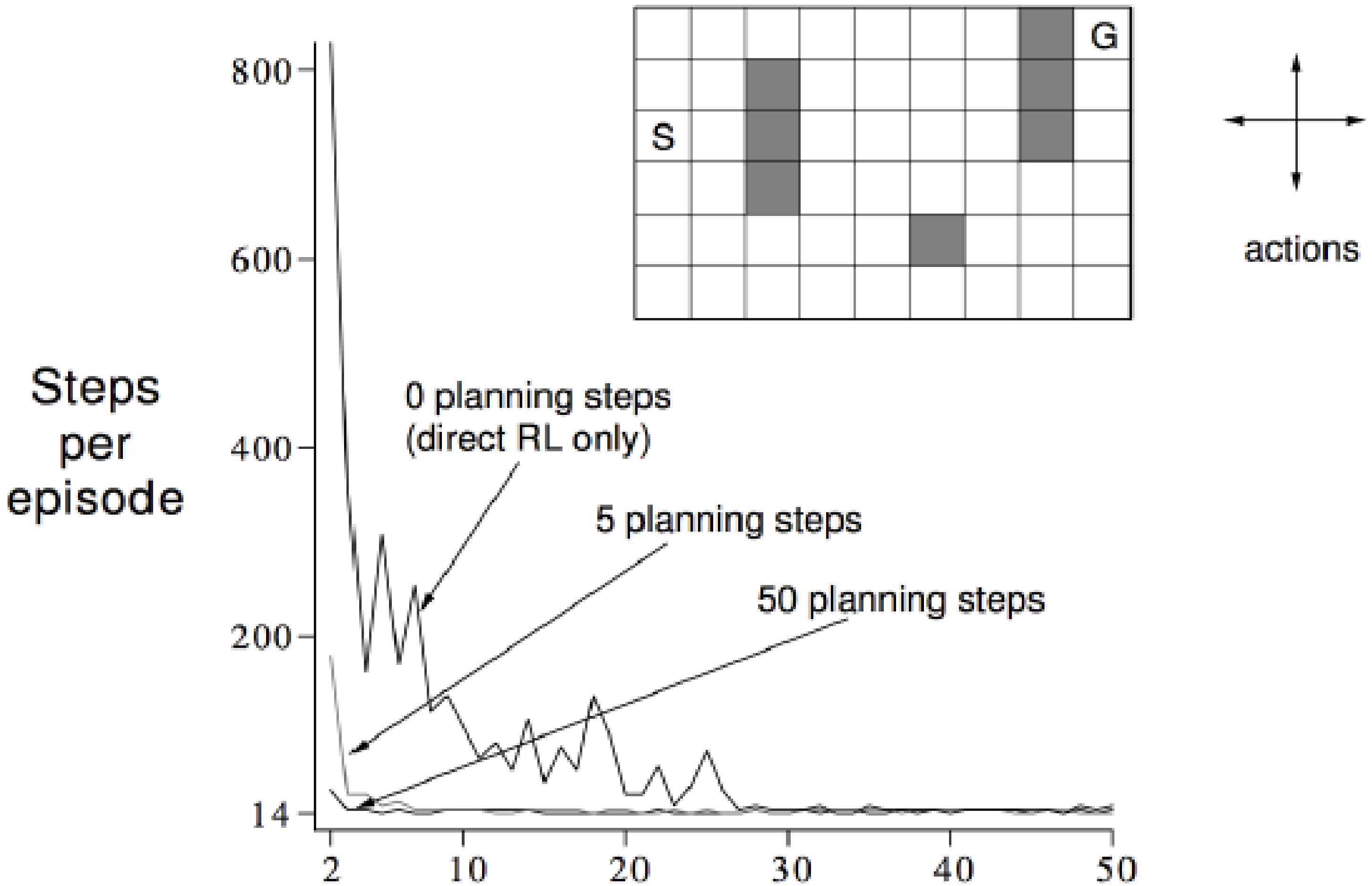


Dyna-Q Algorithm

- Repeat n times for learning Q with planning

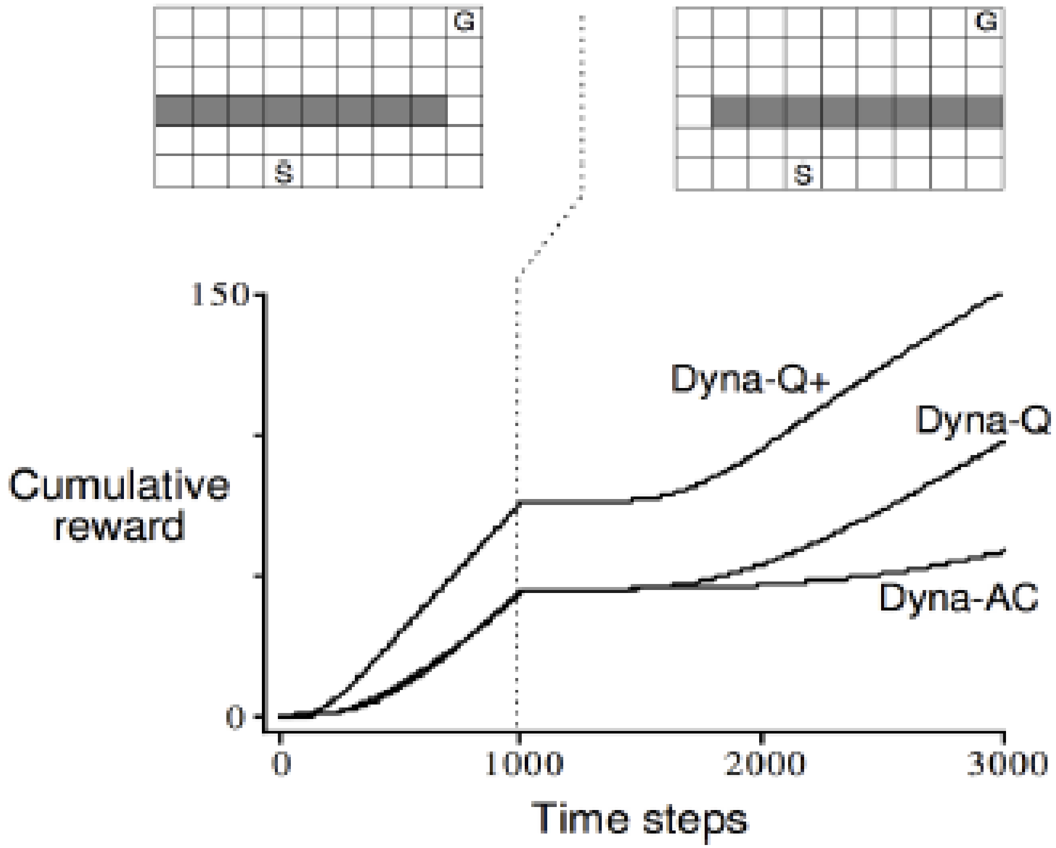
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon\text{-greedy}(S,Q)$
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Dyna-Q on a Simple Maze



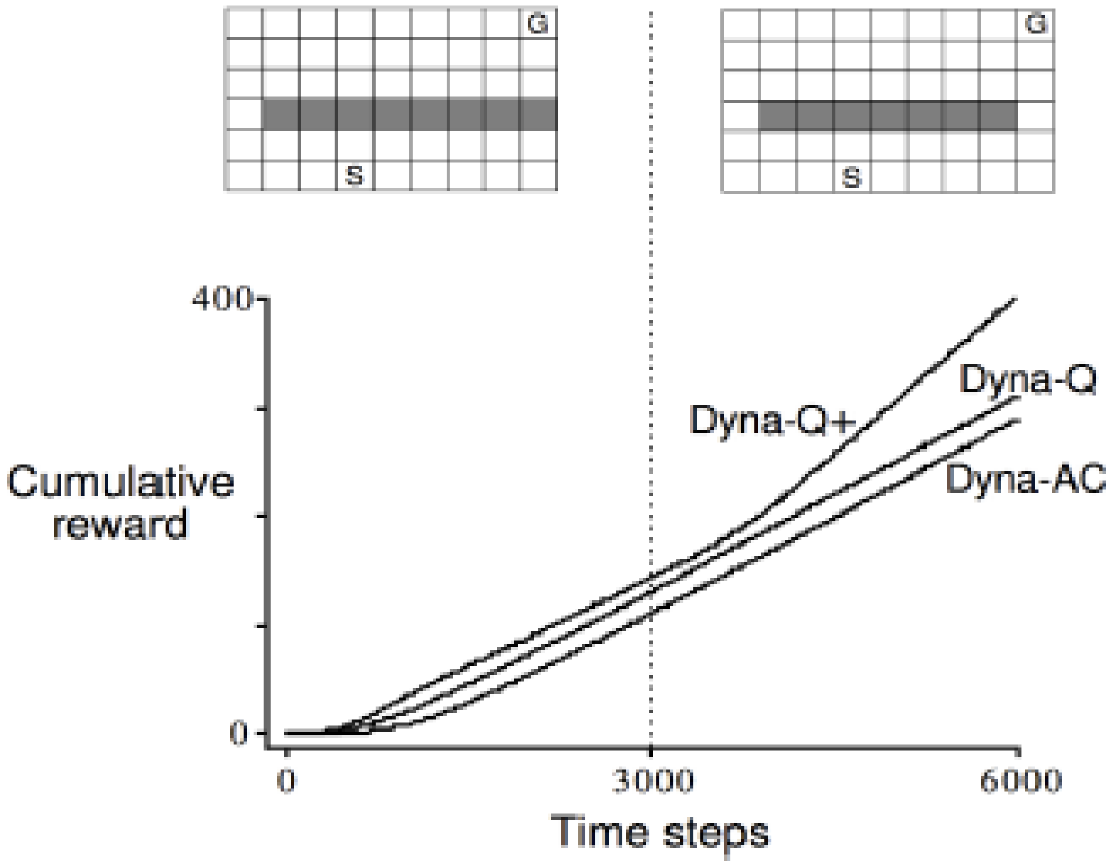
Dyna-Q with an Inaccurate Model

- When the changed environment is harder



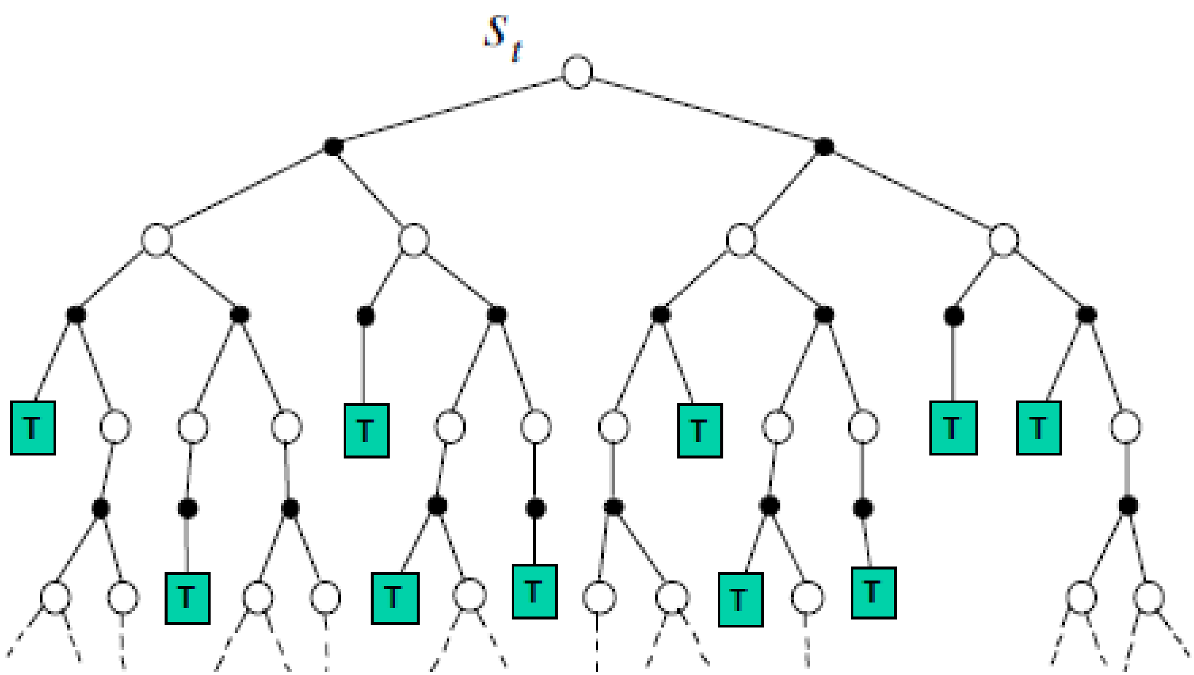
Dyna-Q with an Inaccurate Model (2)

- When the changed environment is easier



Forward Search

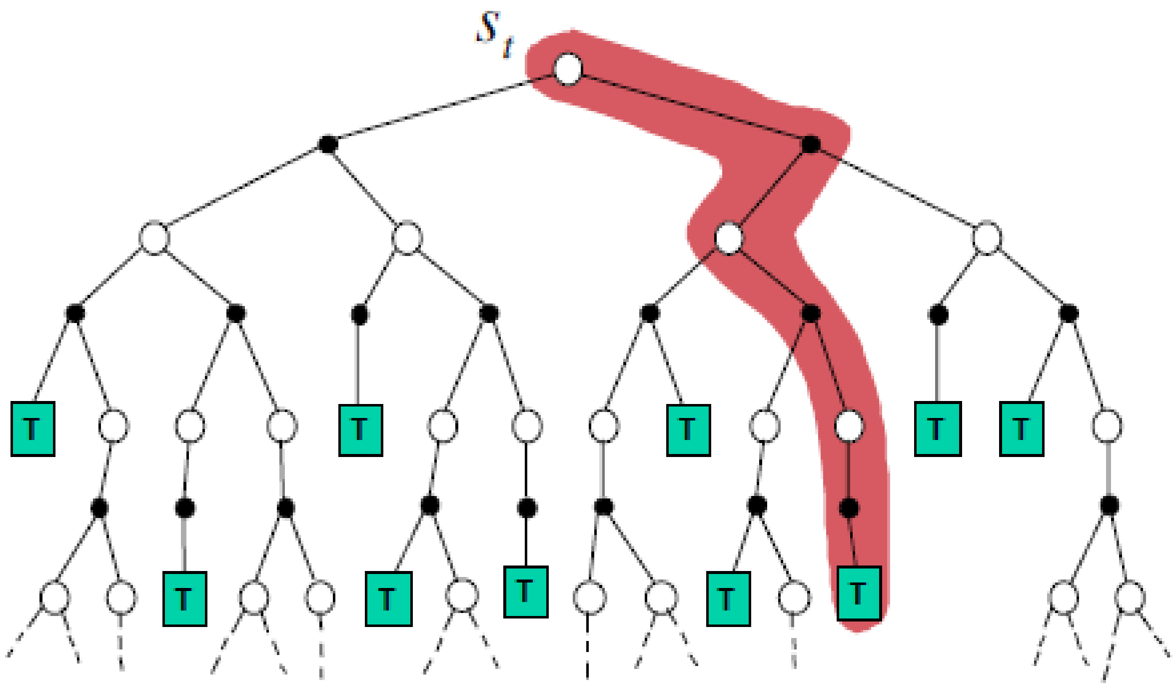
- Forward search algorithms select the best action by lookahead
  - build a search tree with the current state S_t at the root
  - use a model of the MDP to look ahead



- 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)

- Simulate episodes of experience from now with the model

$$\{s_t^k, A_t^k, R_{t+1}^k, \ldots, S_t^k\}_{k=1}^K \sim \mathcal{M}_v$$

- Apply model-free RL to simulated episodes
  - Monte-Carlo control → Monte-Carlo search
  - Sarsa → TD search

Simple Monte-Carlo Search

- Given a model $\mathcal{M}_v$ and a simulation policy π
- For each action $a \in A$ from current (real) state s_t
  - Simulate K episodes

$$\{s_t, a, R_{t+1}^k, s_{t+1}^k, A_{t+1}^k, \ldots, S_T^k\}_{k=1}^K \sim \mathcal{M}_v, \pi$$

  - 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

$$a_t = \underset{a \epsilon A}{\text{armax}}\ Q(s_t, a)$$

Monte-Carlo Tree Search (Evaluation)

- Given a model M_v
- Simulate K episodes from current state s_t using current simulation policy π

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

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

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

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

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

Monte-Carlo Tree Search (Simulation)

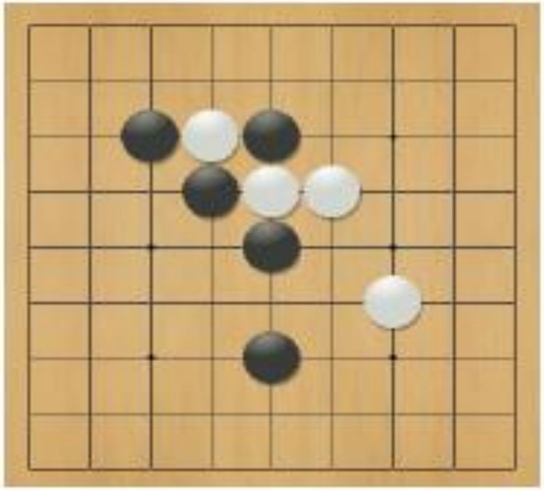
- In MCTS, the simulation policy π improves
- Each simulation consists of two phases (in-tree, out-of-tree)
  - Tree policy (improves): pick actions to maximize $Q(S, A)$
  - Default policy (fixed): pick actions randomly
- Repeat (each simulation)
  - Evaluate states $Q(S, A)$ by Monte-Carlo evaluation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

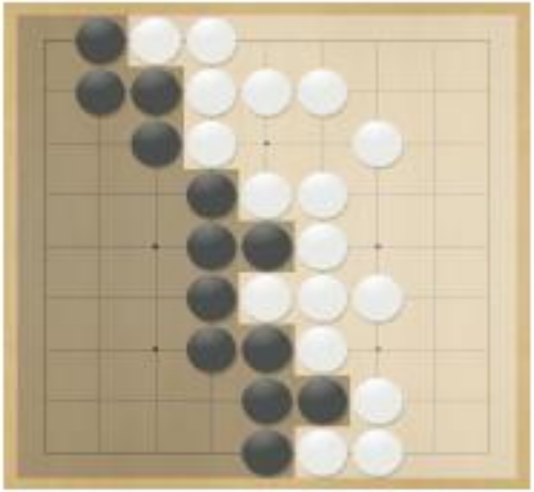
Case Study: the Game of Go

- 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):

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

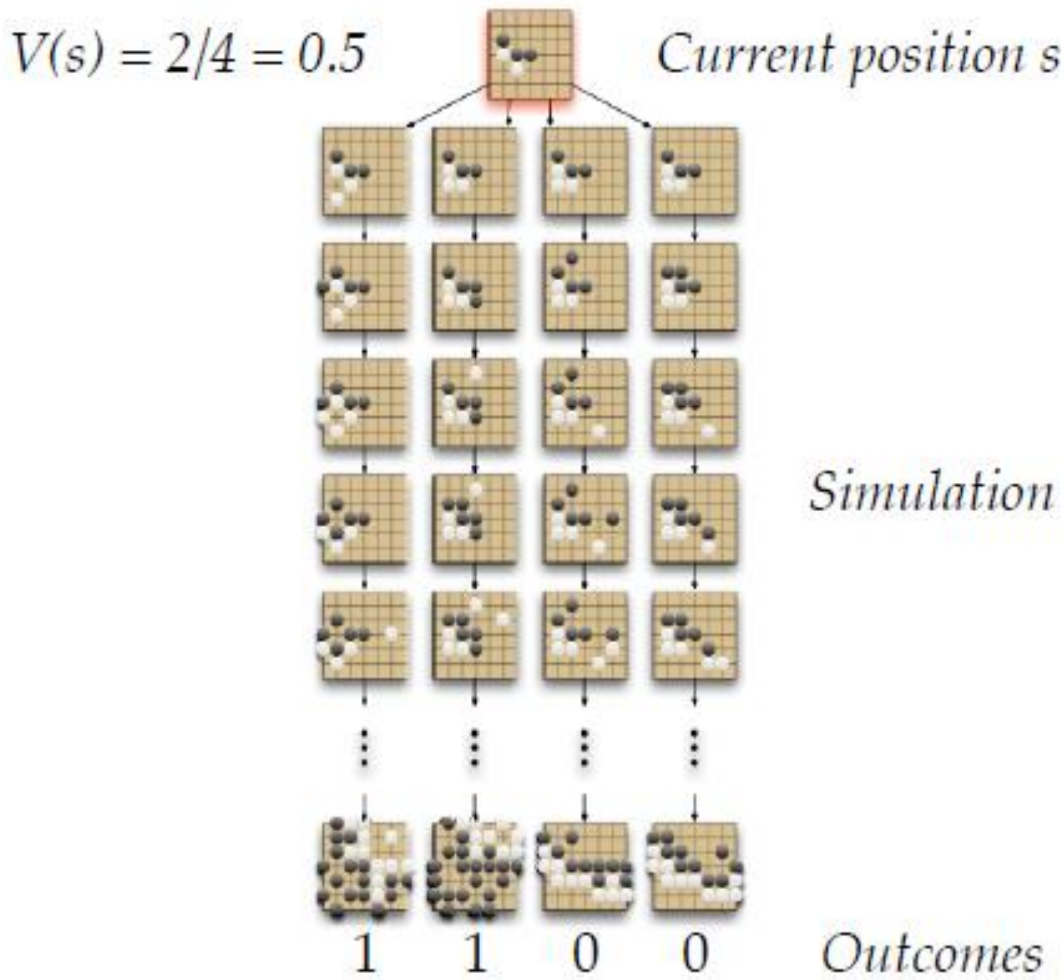
$$R_T = \begin{Bmatrix} 1 & \text{if Black wins} \\ 0 & \text{if White wins} \end{Bmatrix}$$

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

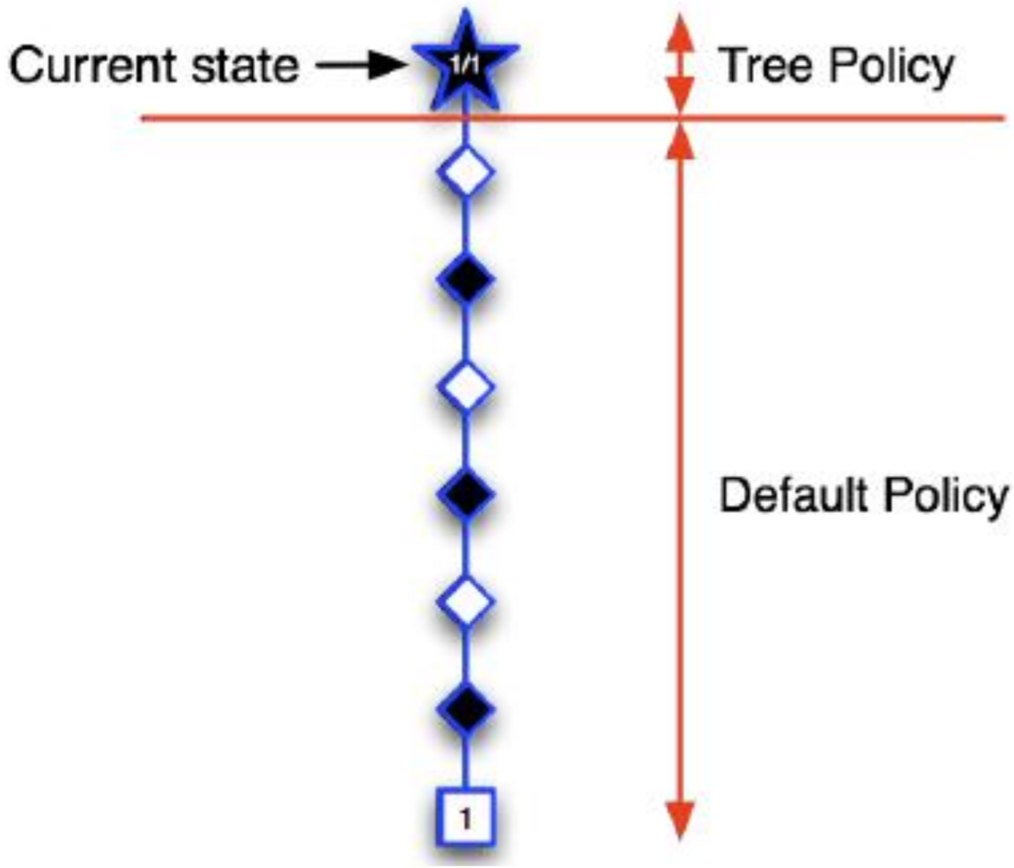
$$v_\pi(s) = \mathbb{E}_\pi[R_T \mid S = s] = \mathbb{P}[\text{Black wins } \mid S = s]$$

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

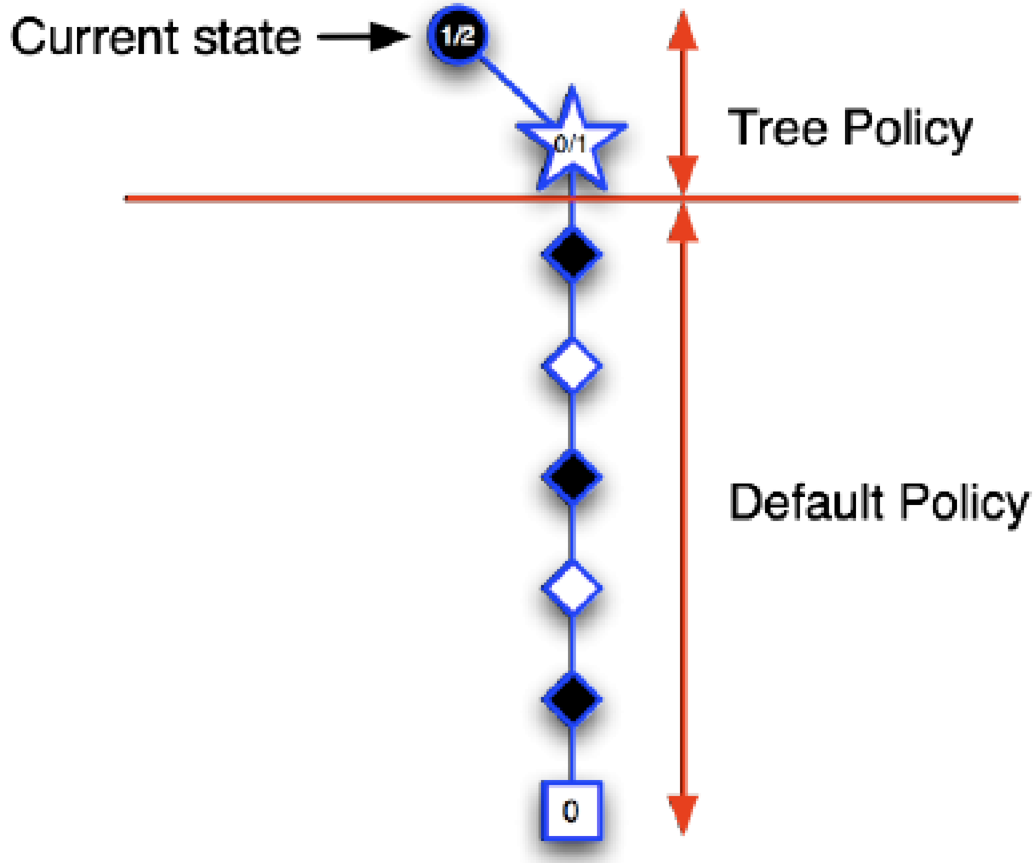
Monte-Carlo Evaluation in Go



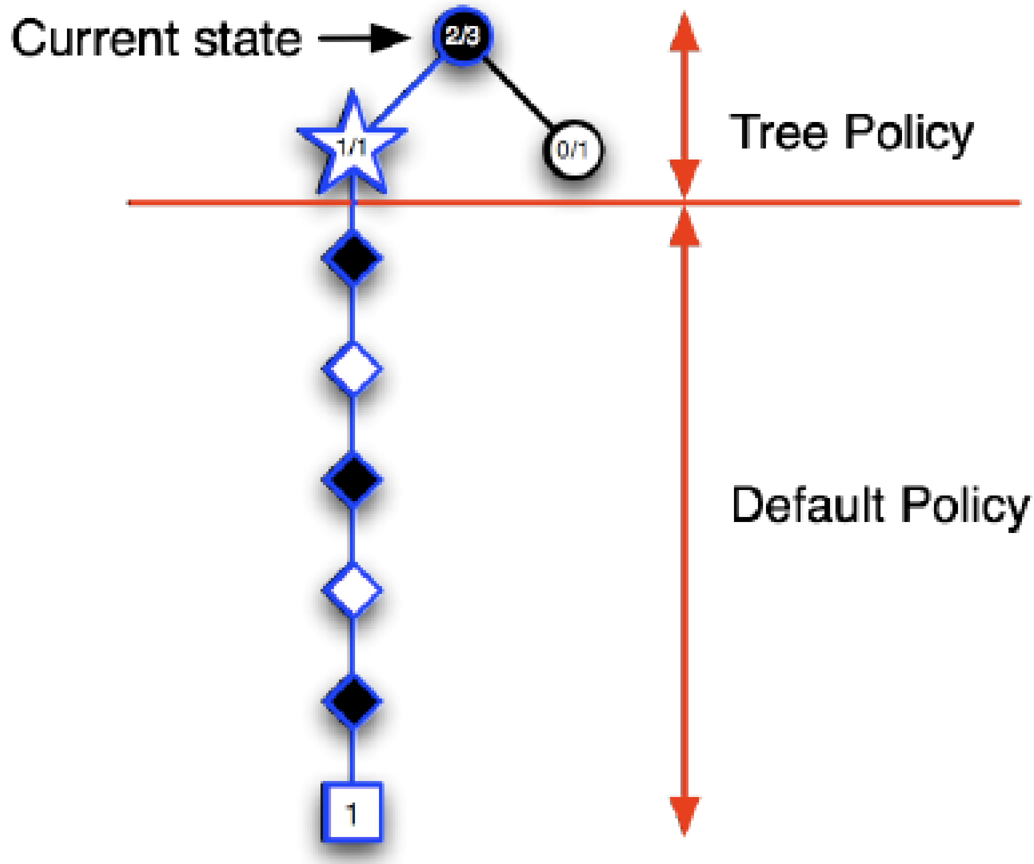
Applying Monte-Carlo Tree Search (1)



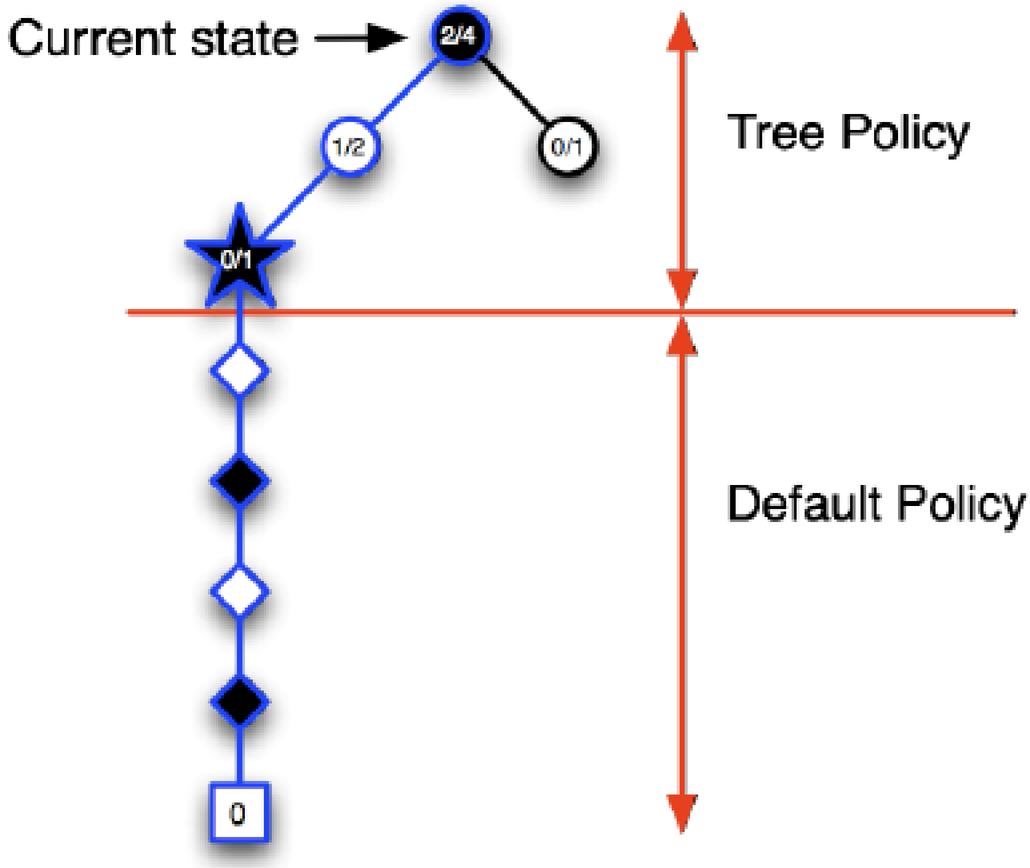
Applying Monte-Carlo Tree Search (2)



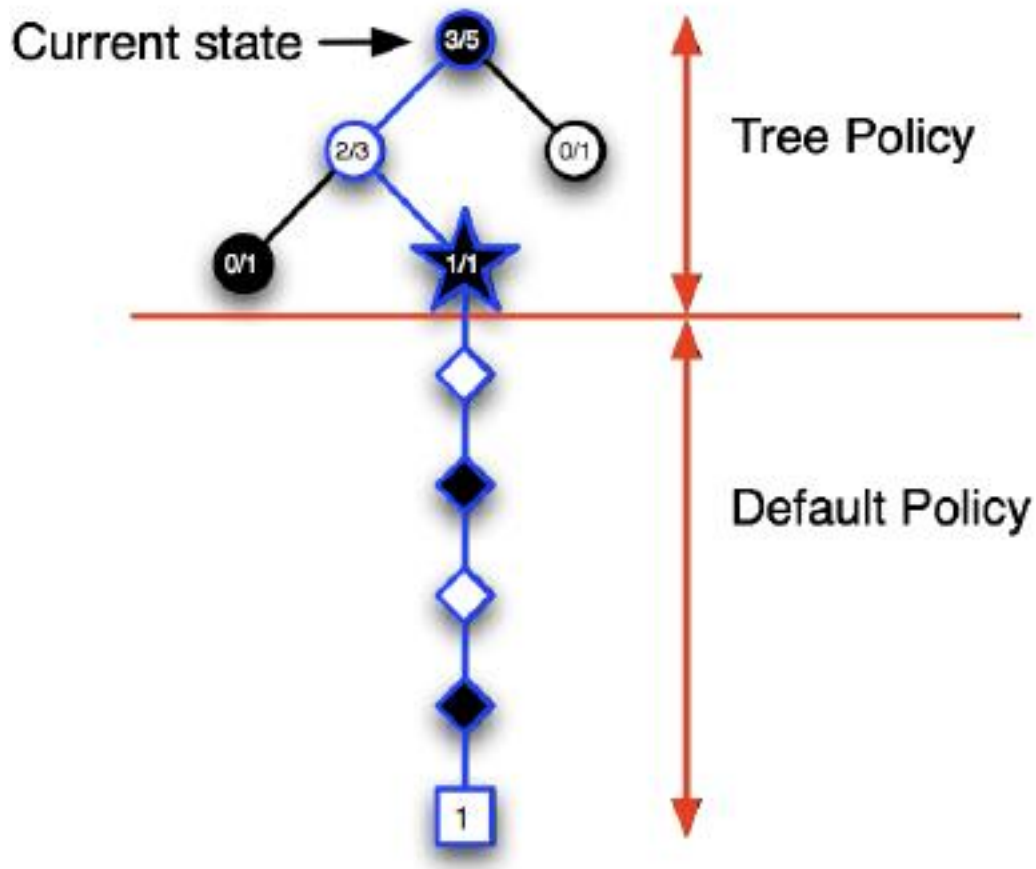
Applying Monte-Carlo Tree Search (3)



Applying Monte-Carlo Tree Search (4)



Applying Monte-Carlo Tree Search (5)

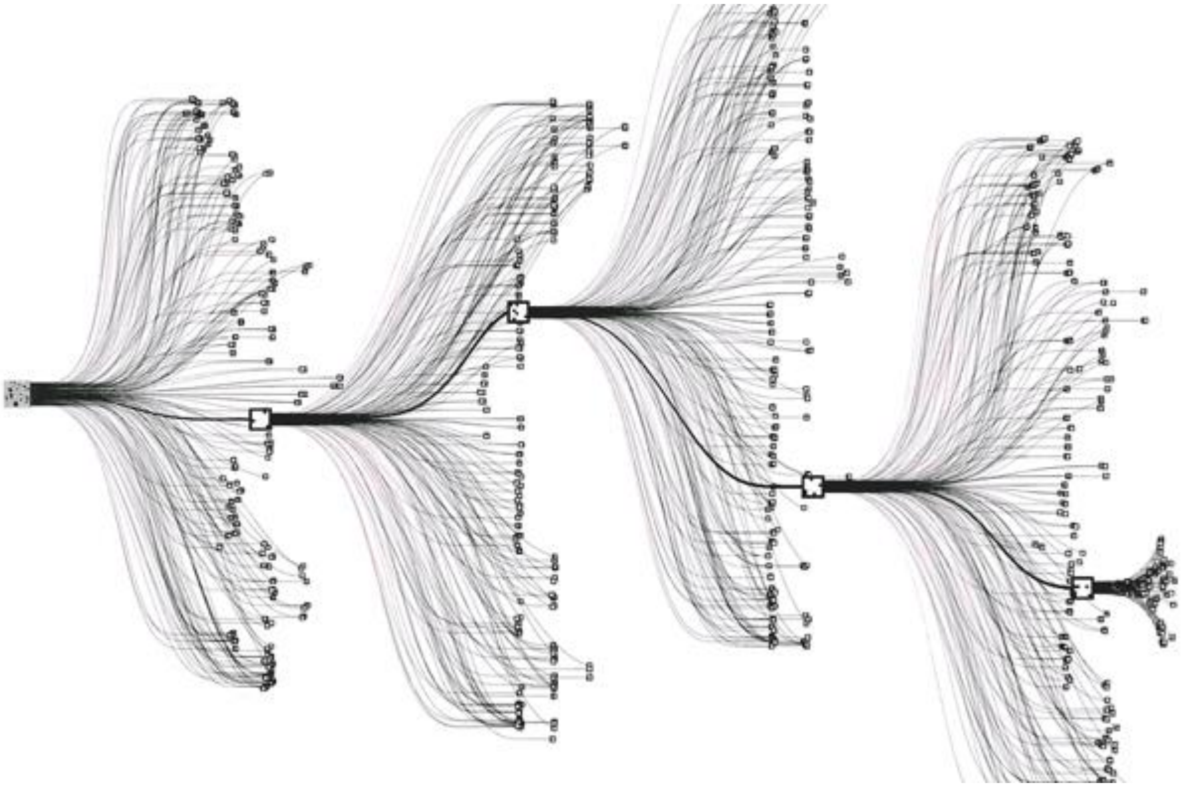


Advantages of MC Tree Search

- 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, parallelizable

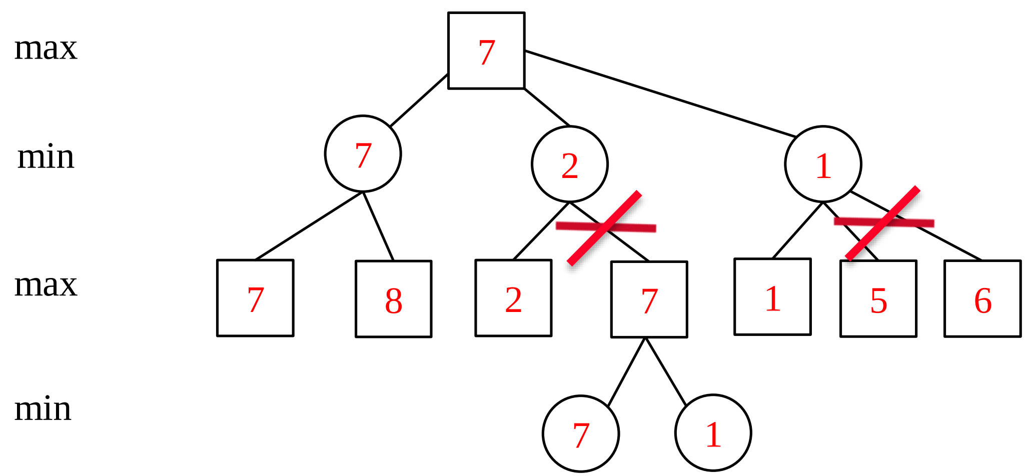
Go – One of the Most Popular Games

- Game tree complexity: about 10^{360}
  - It is just impossible to try all moves.



Can Alpha-Beta Search Work for Go?

- Alpha-Beta Search
  - Very successful for many games such as chess.
    - Almost dominate all computer games before 2006.
    - This is what Deep Blue used.
- The key for chess: evaluate position accurately and efficiently. E.g., features:
  - King: 1000
  - Queen: 200
  - Rook: 100
  - Knight: 80
  - Bishop: 70
  - Pawn: 30
  - Guarded Pawns: 30
  - Guarded Knights: 40
  - …



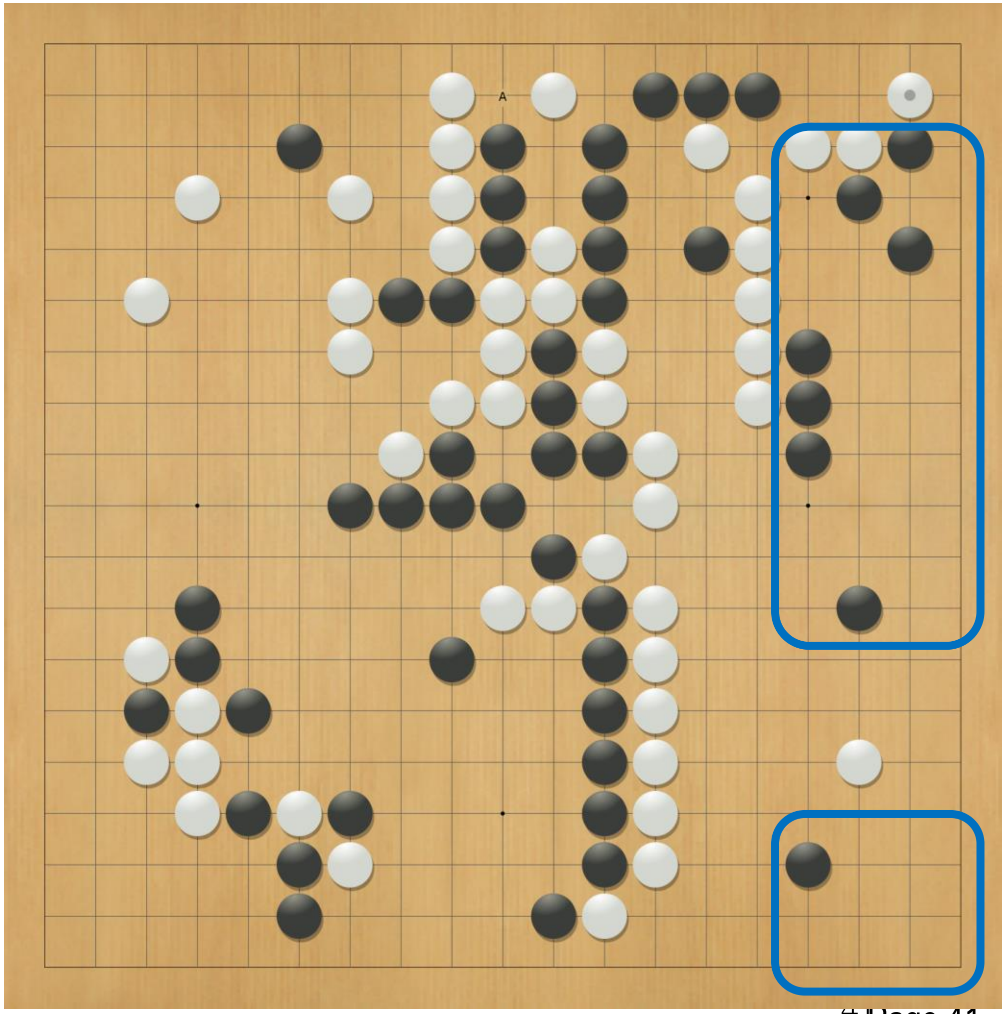
- Problem for chess:
  - need to consult with experts for feature values.

Why not alpha-beta search for Go?

- No simple heuristics like chess.
  - Only black/white pieces (no difference)
- Must know life-and-death
  - But, all are correlated.
    - Like the lower-right one.
  - But, this is very complex.

Since no simply heuristics to evaluate,

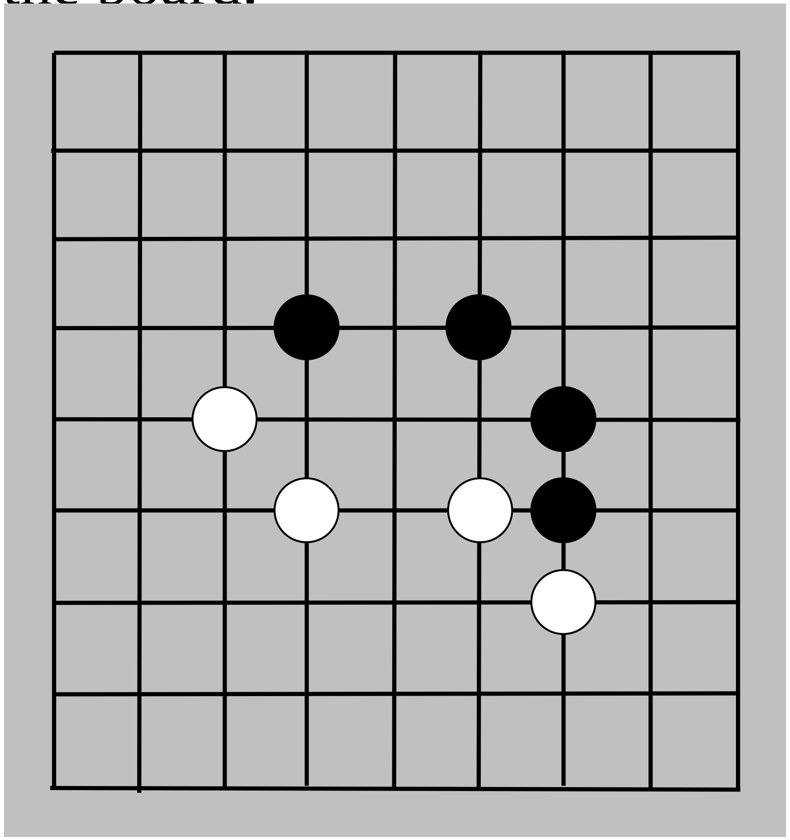
- Why not use Monte-Carlo?
- Calculate it based on stochastics.



Game 1: AlphaGo vs. 李世石

Rules Overview Through a Game (opening 1)

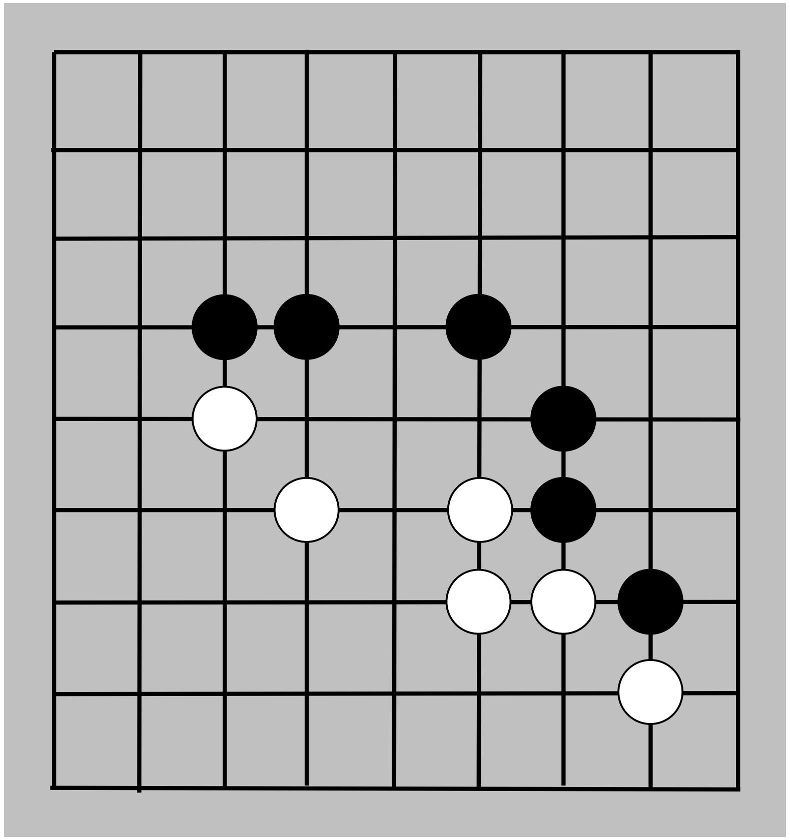
- Black/White move alternately by putting one stone on an intersection of the board.



The example was given by B. Bouzy at CIG'07.

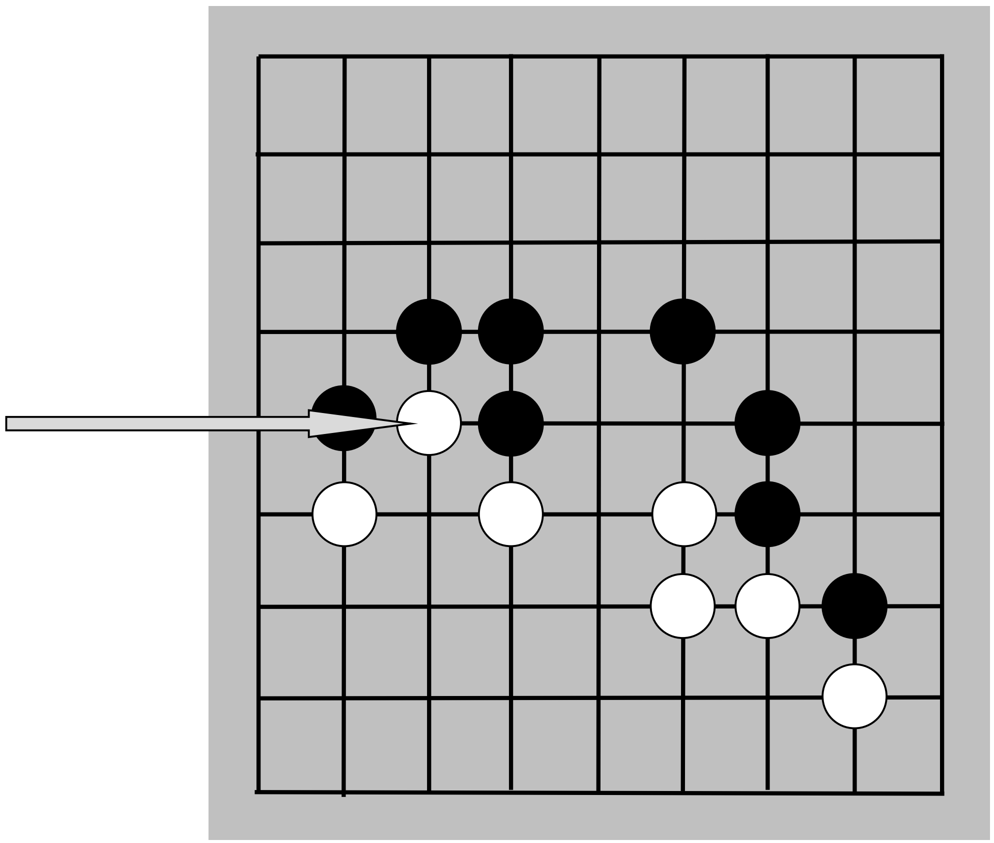
Rules Overview Through a Game (opening 2)

- Black and White aims at surrounding large « zones »



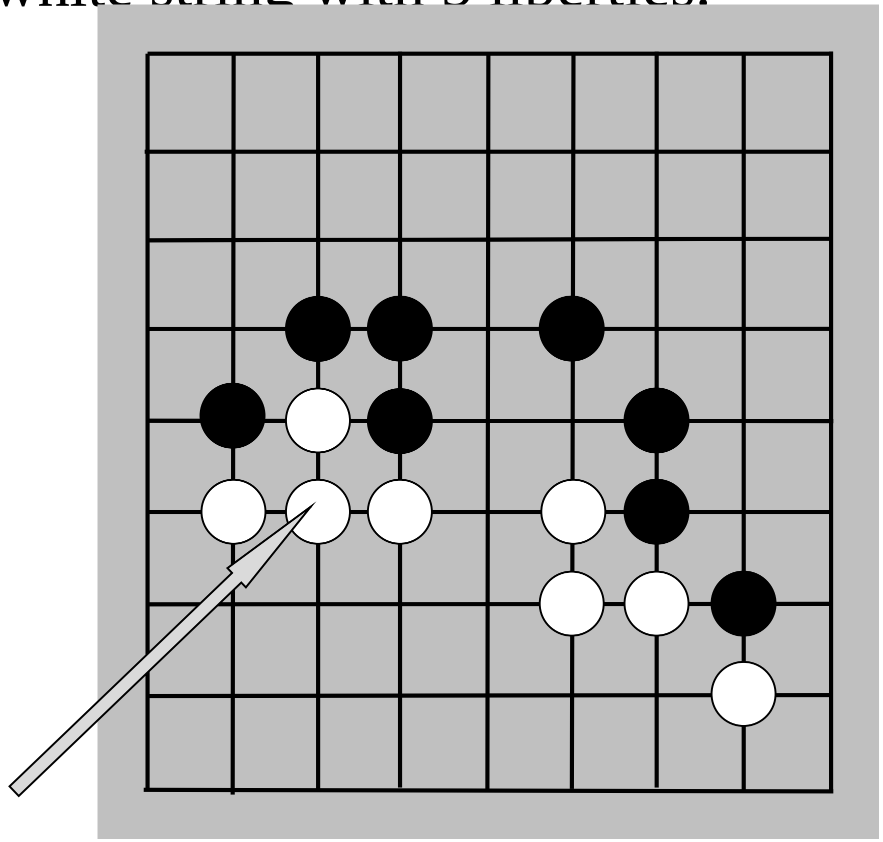
Rules Overview Through a Game (atari 1)

- A white stone is put into « atari » : it has only one liberty left.



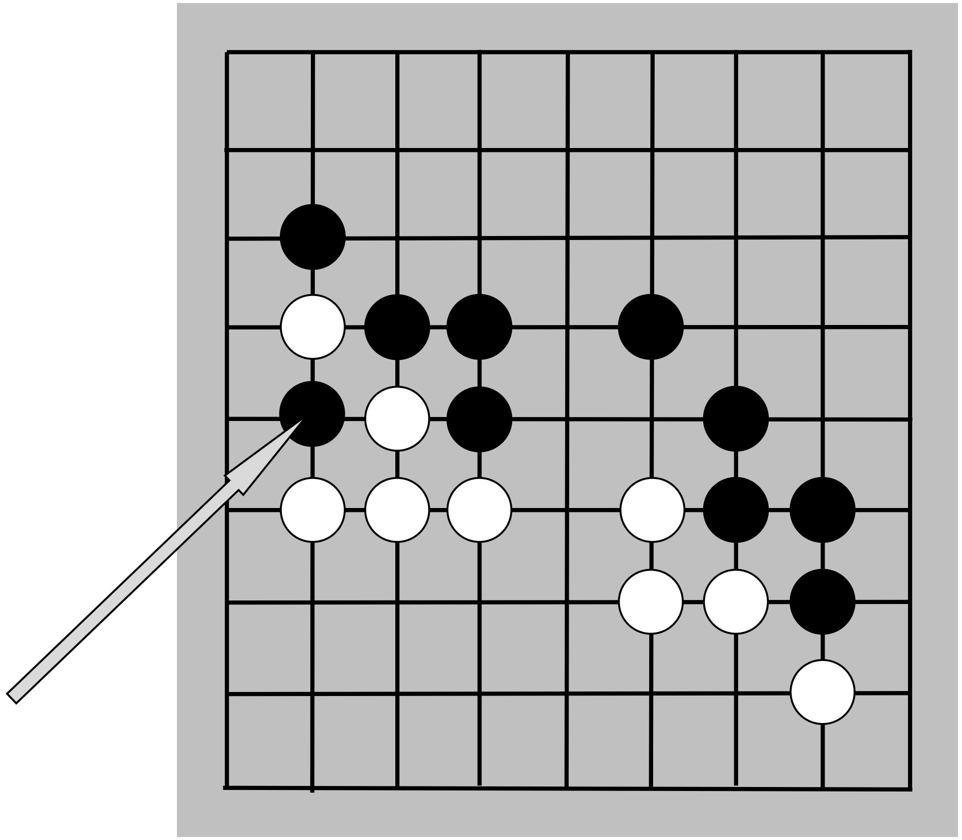
Rules Overview Through a Game (defense)

- White plays to connect the one-liberty stone yielding a four-stone white string with 5 liberties.



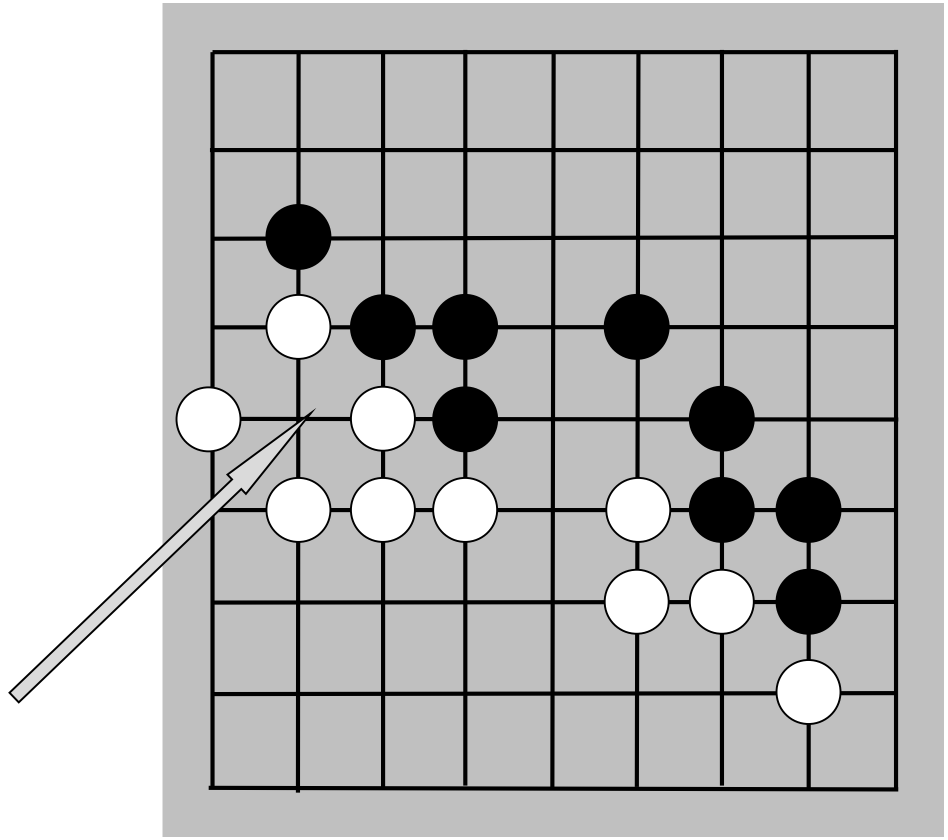
Rules Overview Through a Game (atari 2)

- It is White’s turn. One black stone is atari.



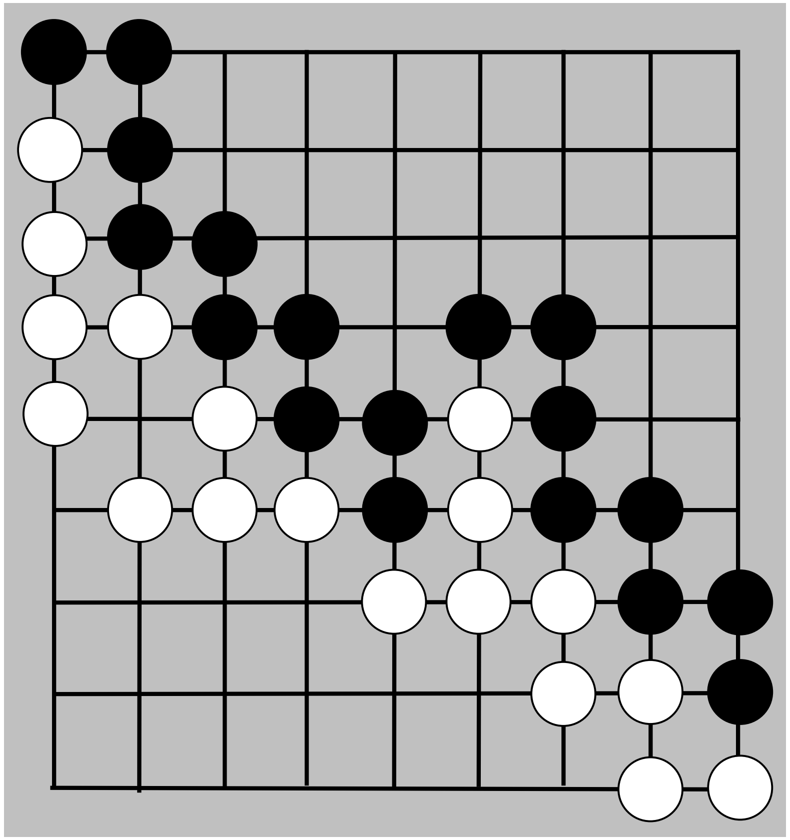
Rules Overview Through a Game (capture 1)

- White plays on the last liberty of the black stone which is removed



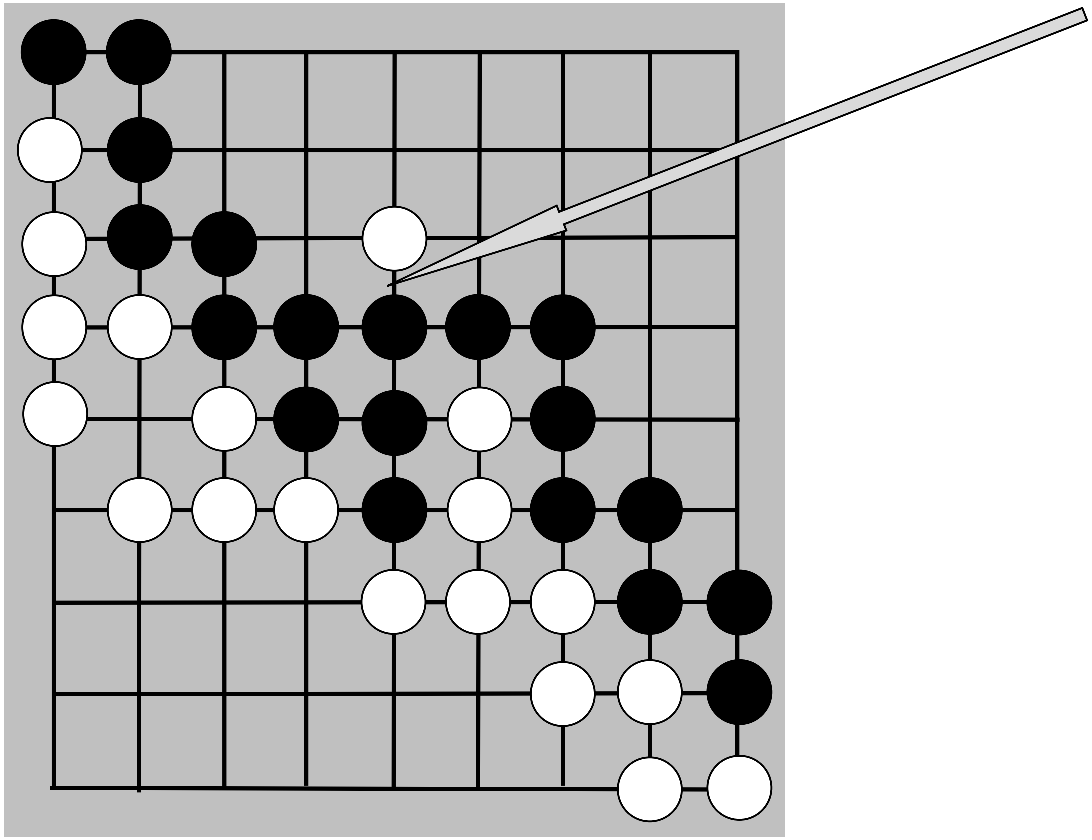
Rules Overview Through a Game (human end of game)

- The game ends when the two players pass. (Experts would stop here)



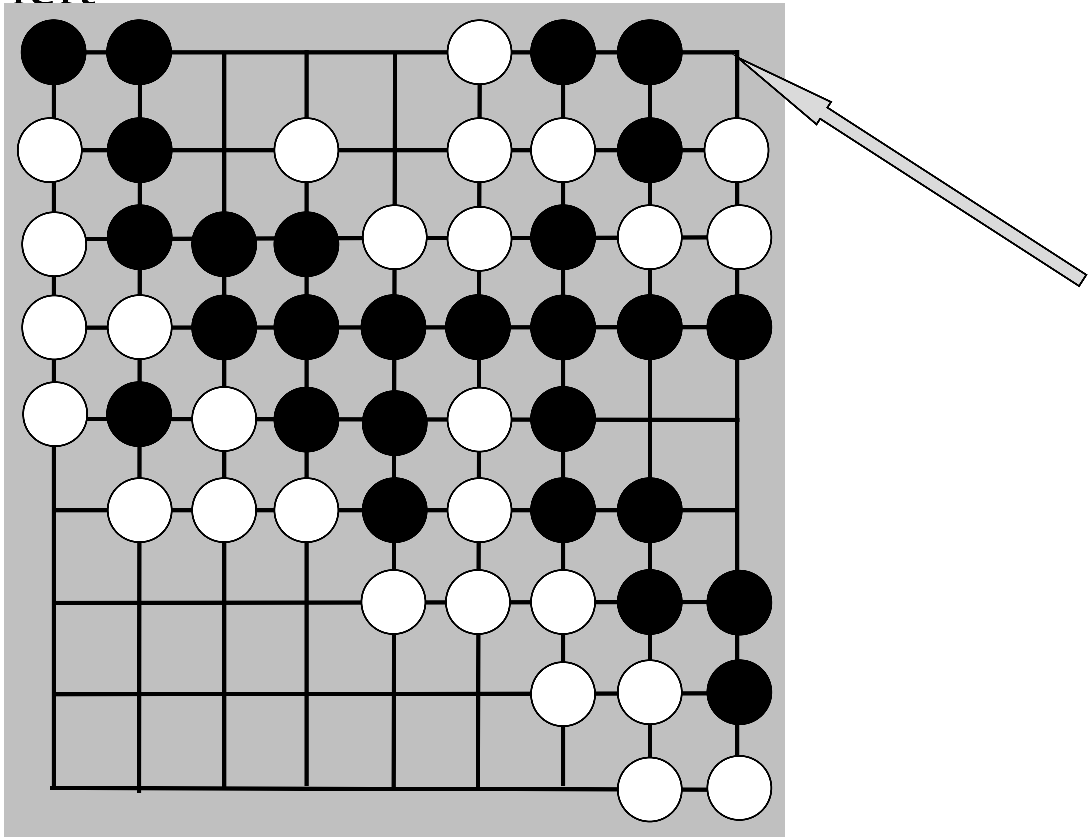
Rules Overview Through a Game (contestation 1)

- White contests the black « territory » by playing inside.



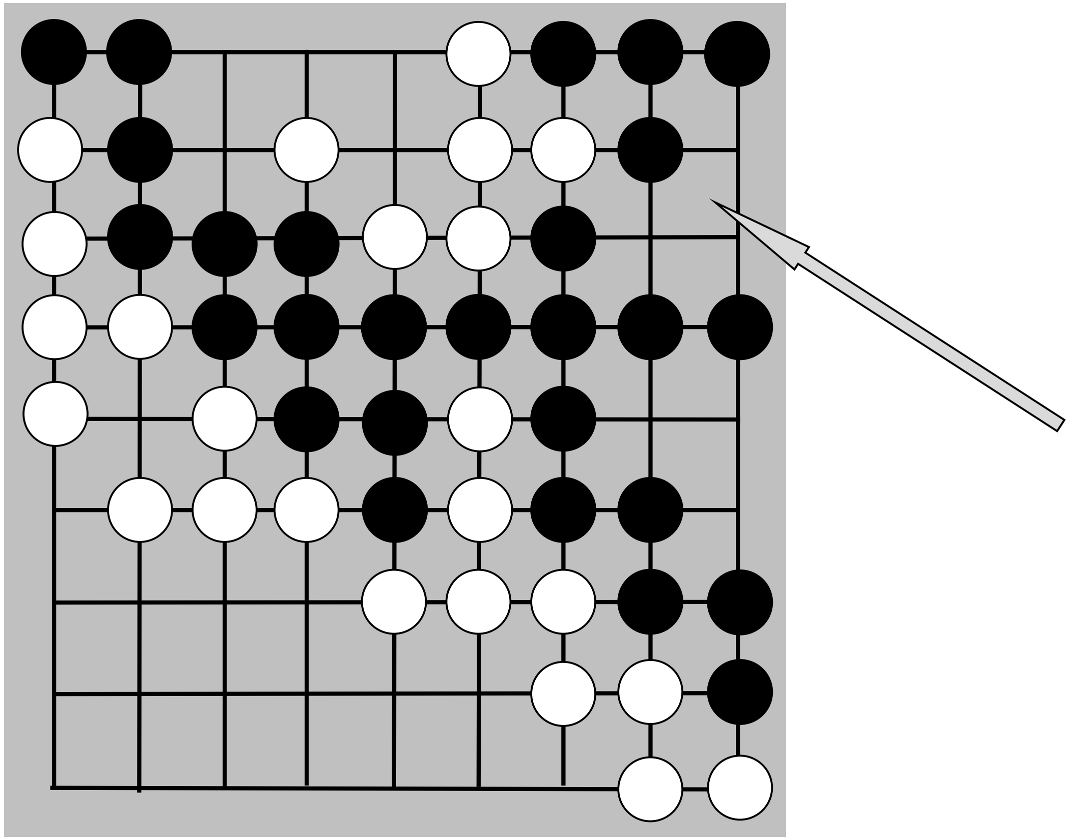
Rules Overview Through a Game (contestation 2)

- White contests black territory, but the 3-stone white string has one liberty left



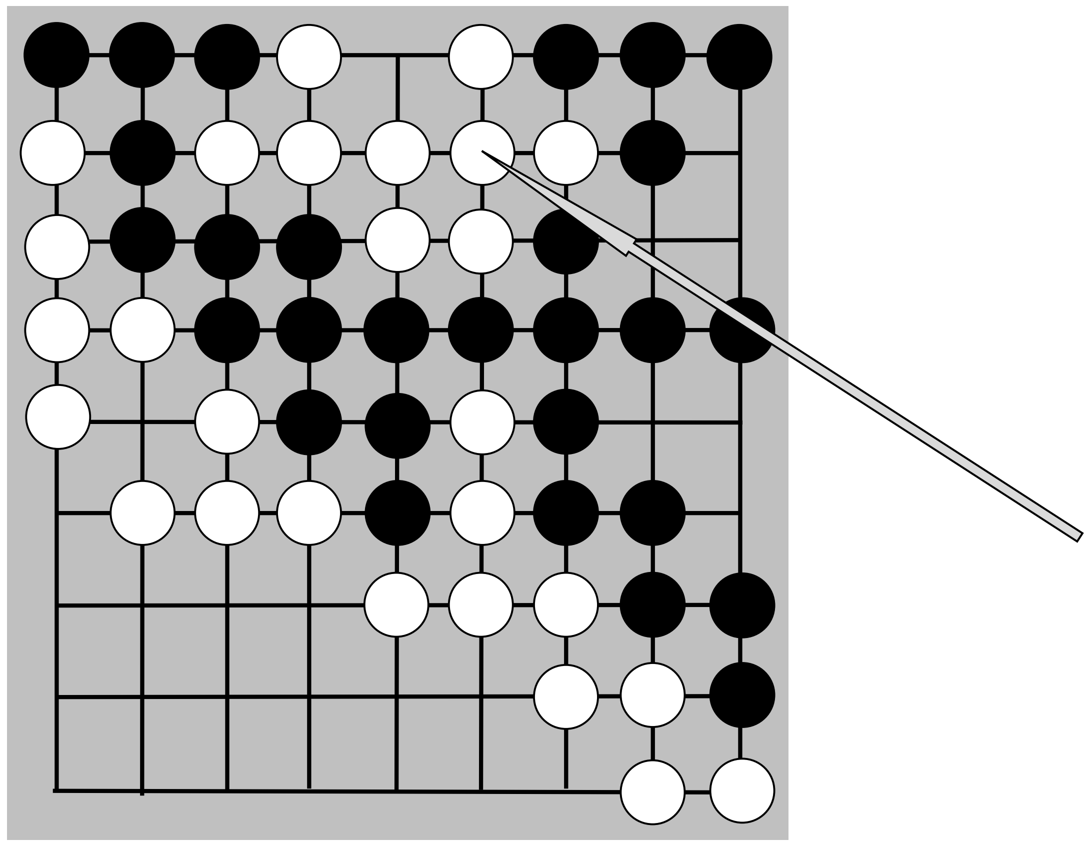
Rules Overview Through a Game (follow up 1)

- Black has captured the 3-stone white string



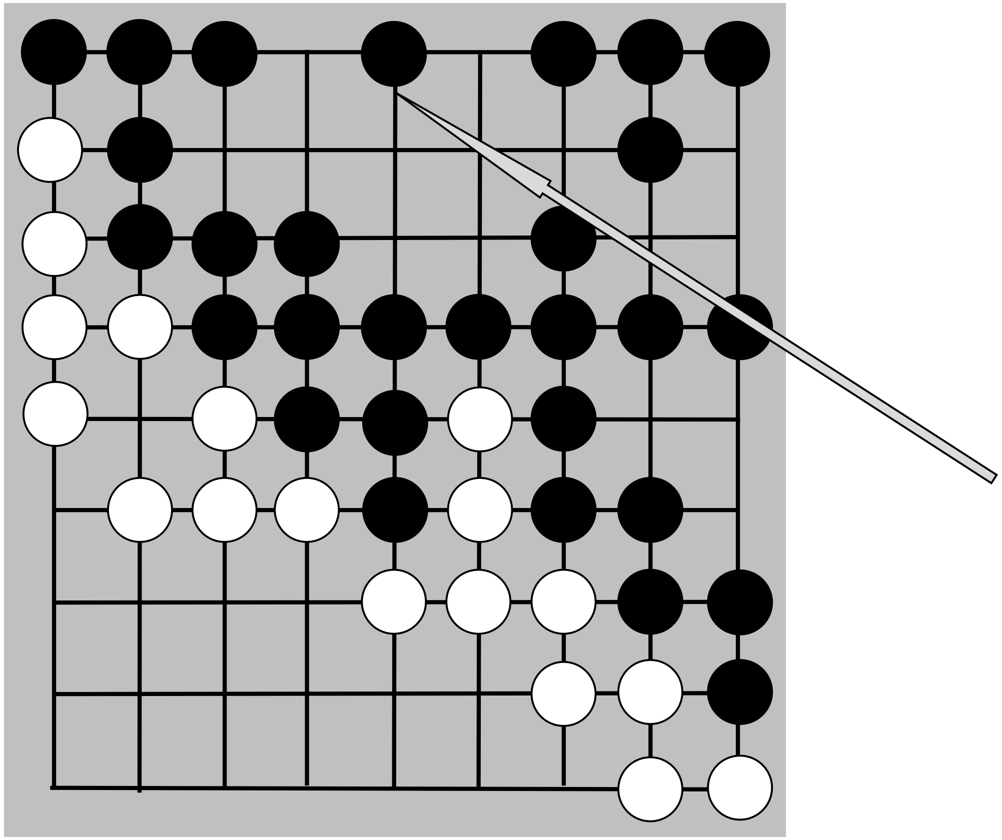
Rules Overview Through a Game (follow up 2)

- White lacks liberties…



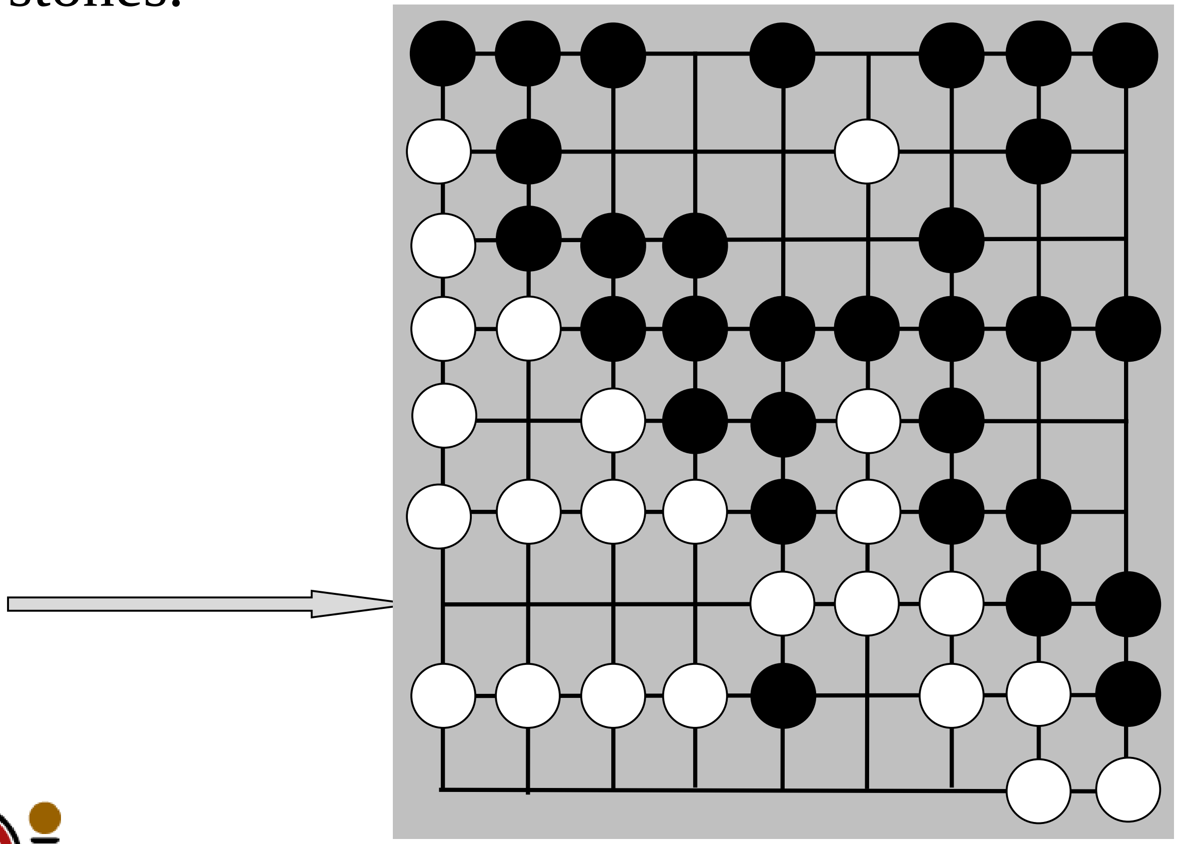
Rules Overview Through a Game (follow up 3)

- Black suppresses the last liberty of the 9-stone string
- Consequently, the white string is removed



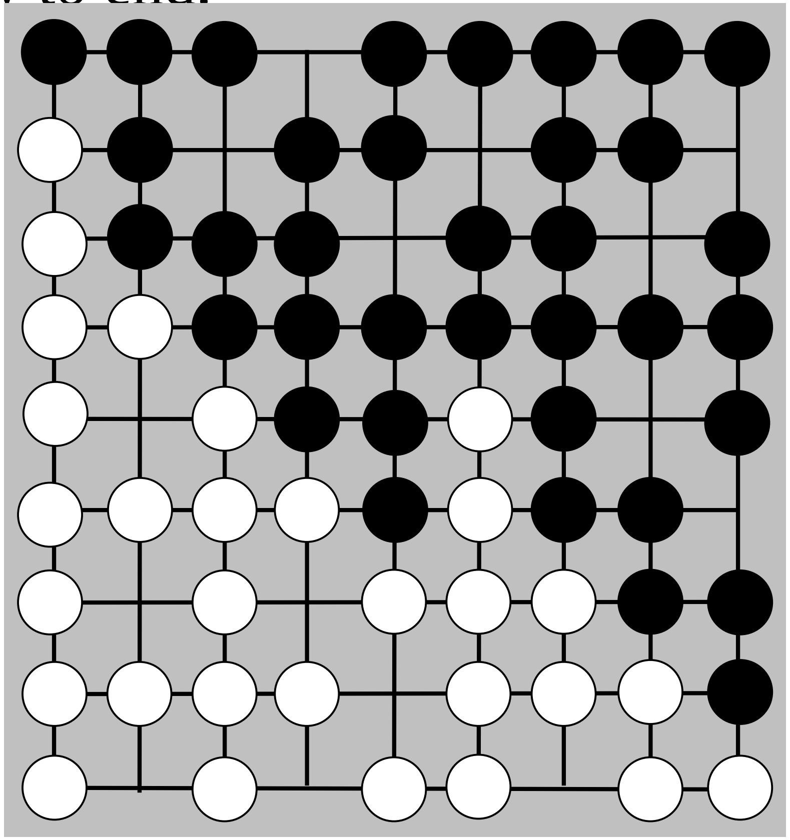
Rules Overview Through a Game (follow up 4)

- Contestation is going on. White has captured four black stones.



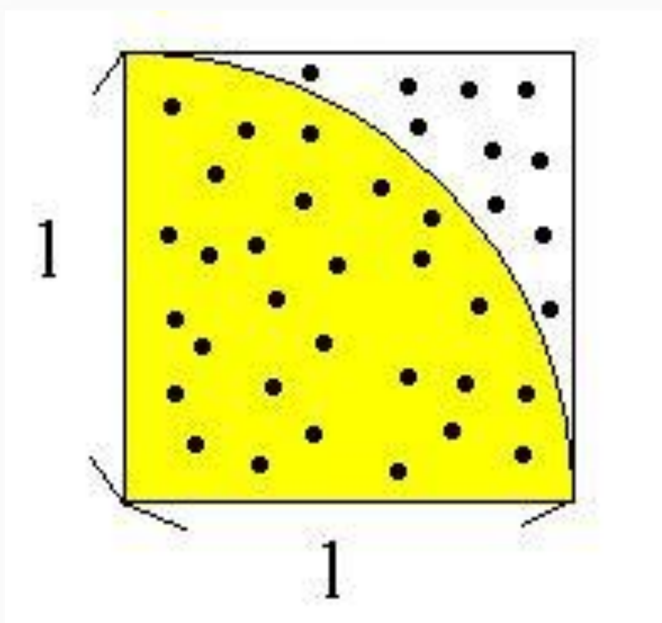
Rules Overview Through a Game (concrete end of game)

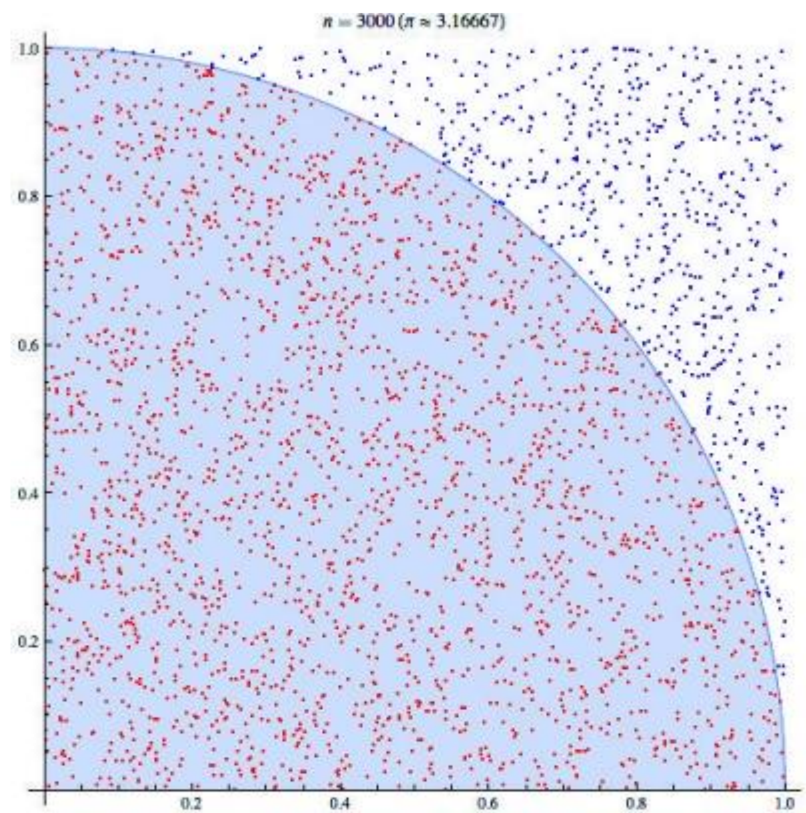
- The board is covered with either stones or « eyes ». Programs know to end.



Stochastics

- Calculate values based on stochastics.
  - Good example: calculate π.





Multi-Armed Bandit Problem (吃角子老虎問題)

- Assume that you have infinite plays
  - How to choose the one with the maximal average return?

Exploration vs. Exploitation

- Example for the exploration vs exploitation dilemma
  - **Exploration**: is a long-term process, with a risky, uncertain outcome.
  - **Exploitation**: by contrast is short-term, with immediate, relatively certain benefits

Deterministic Policy: UCB1

- UCB: Upper Confidence Bounds. [Auer *et al.*, 2002]
- Observed rewards when playing machine i: $X_{i,1}$, $X_{i,2}$, ...
- Initialization: Play each machine once.
- Loop:
  - Play machine j that maximizes,

$$\bar{X}_j + \sqrt{\frac{2 \log n}{T_j(n)}}$$

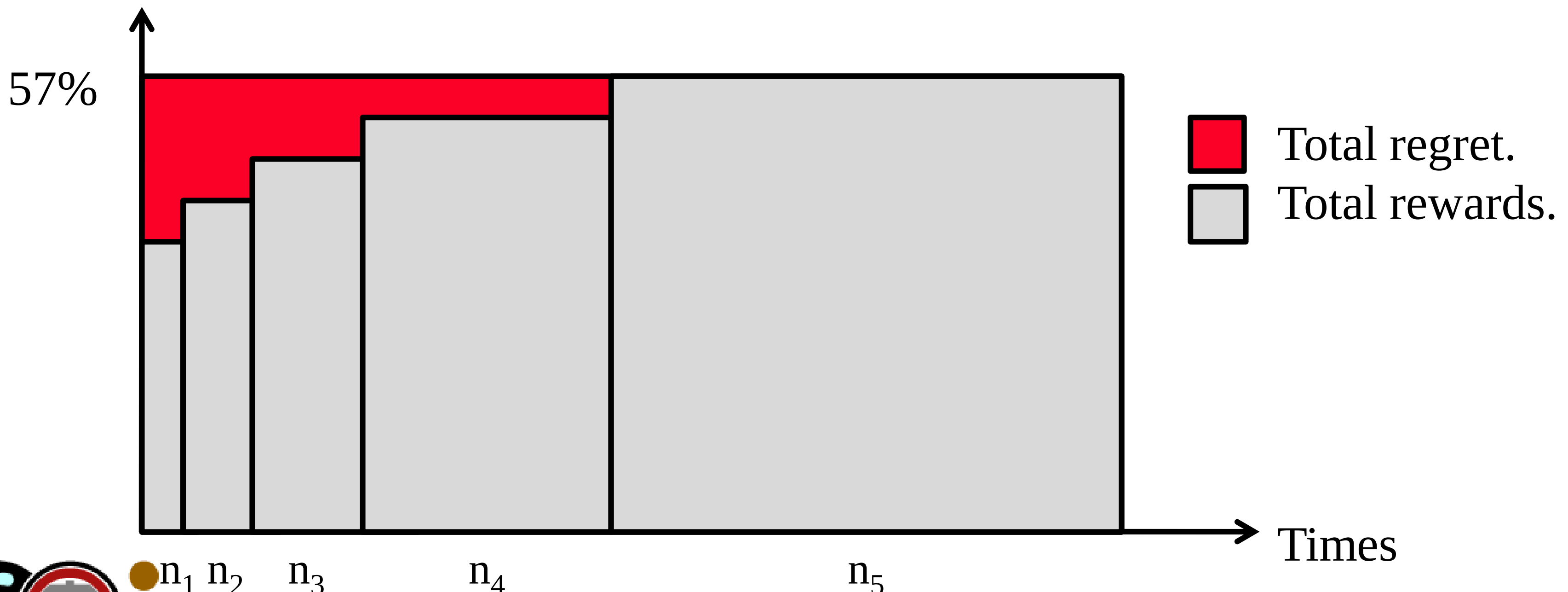
    where n is the overall number of plays done so far,

$$\bar{X}_{i,s} = \frac{1}{s} \sum_{j=1}^{s} X_{i,j} \quad , \quad \bar{X}_i = \bar{X}_{i,T_i(n)} \, ,$$

- Key:
  - Ensure optimal machine is played exponentially more often than any other machine.

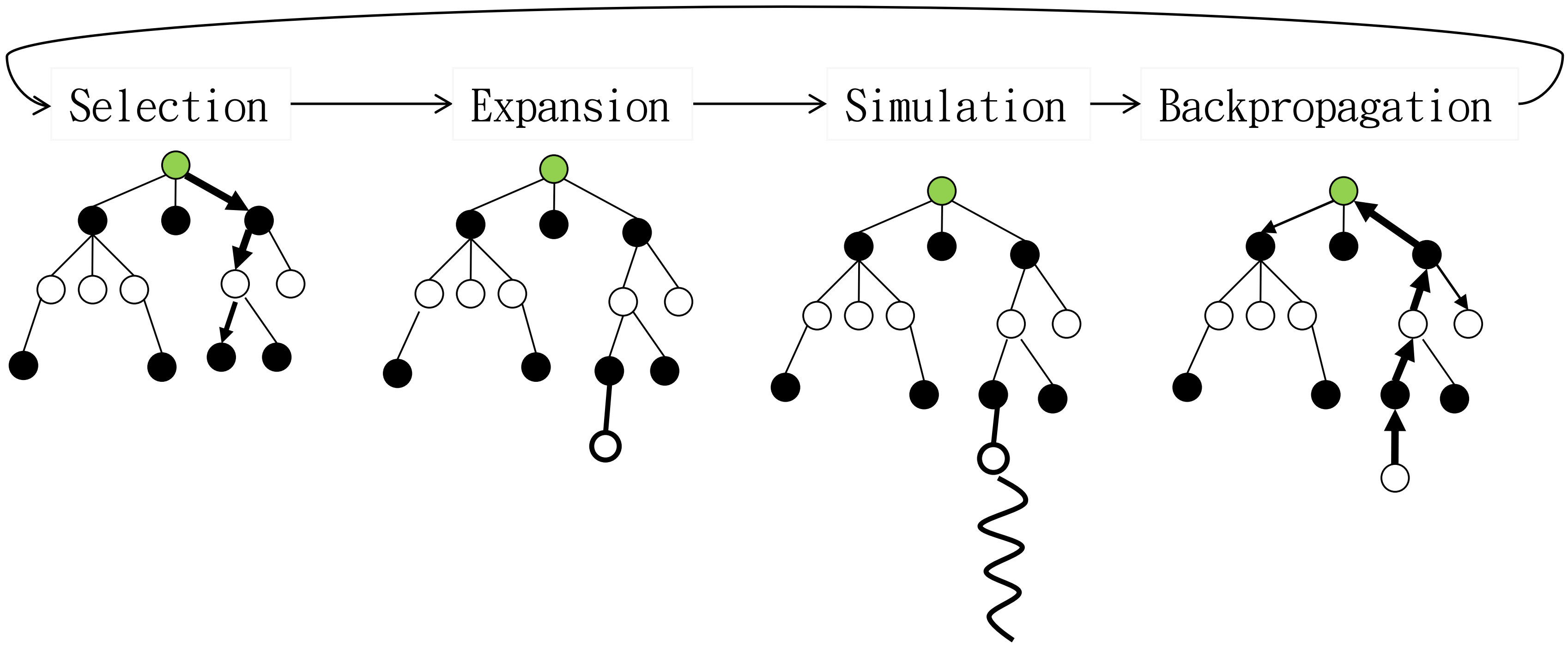
Cumulative Regret

- Assume Machines M_1, M_2, M_3, M_4, M_5
  - Win rates: 37%, 42%, 47%, 52%, 57%
  - Trial numbers: n_1, n_2, n_3, n_4, n_5.



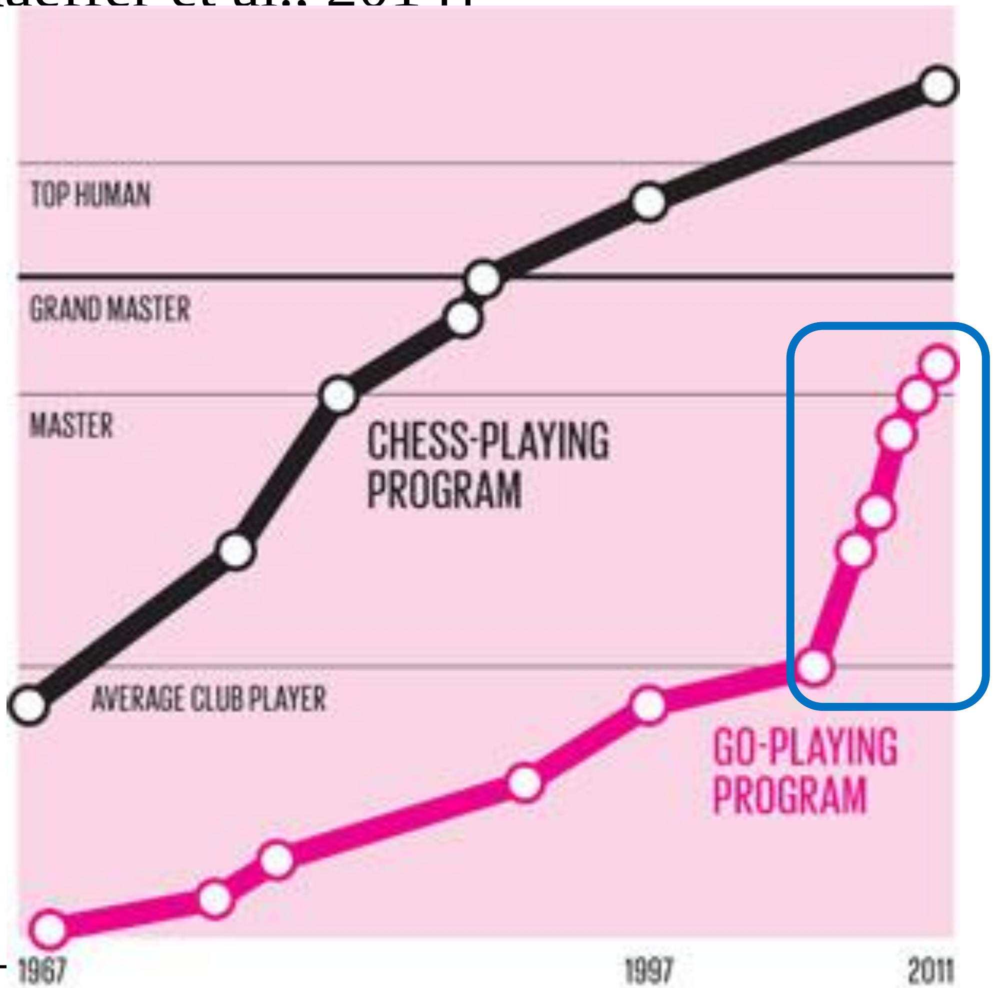
Monte-Carlo Tree Search

- A kind of planning
- A kind of Reinforcement learning



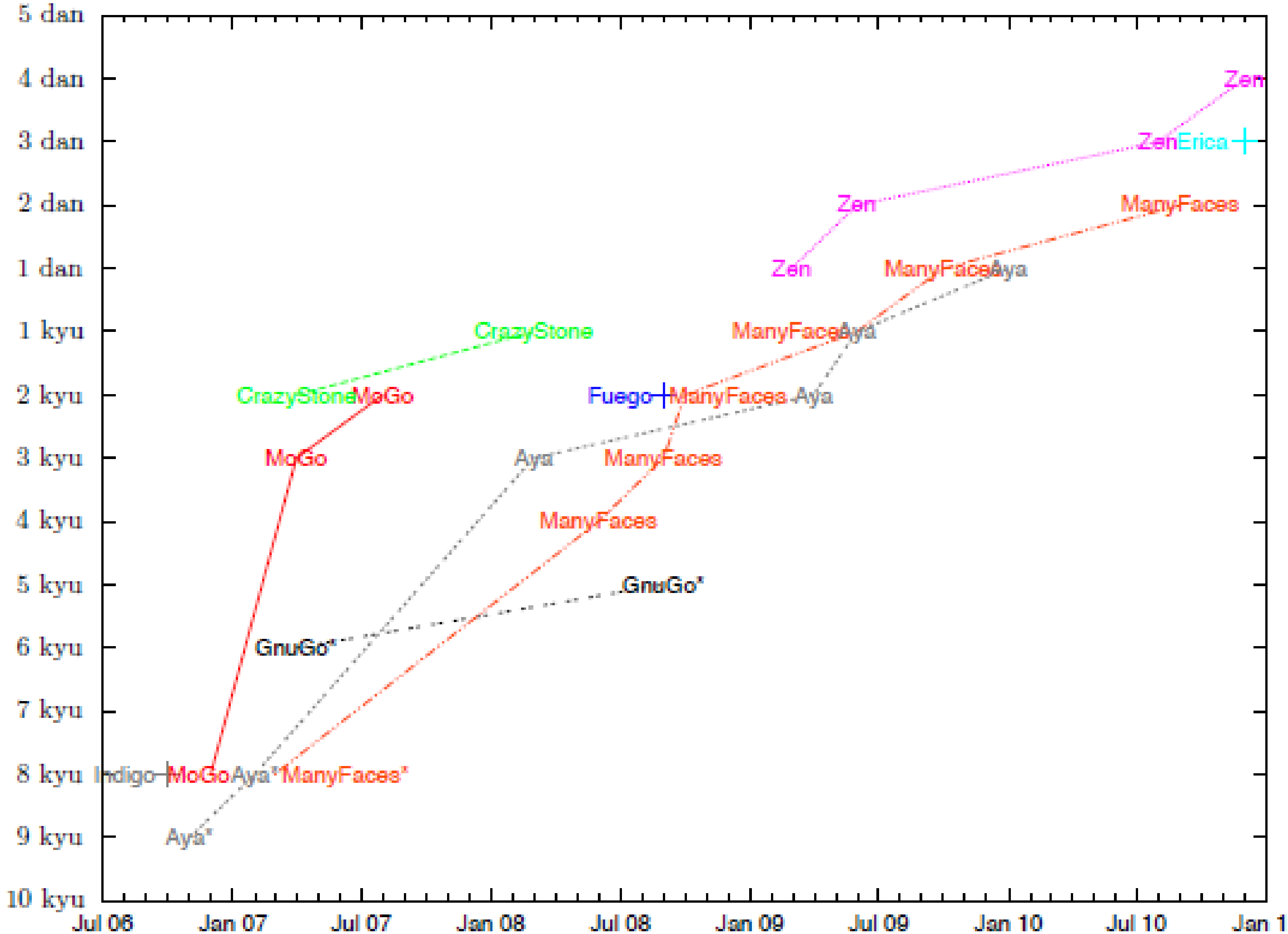
Strength of Go Program after MCTS

- [Schaeffer et al.. 2014]



Strength grew fast, after MCTS.

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(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
- Question: can we try TD search for 2048?

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. ϵ-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