Outline of This Course

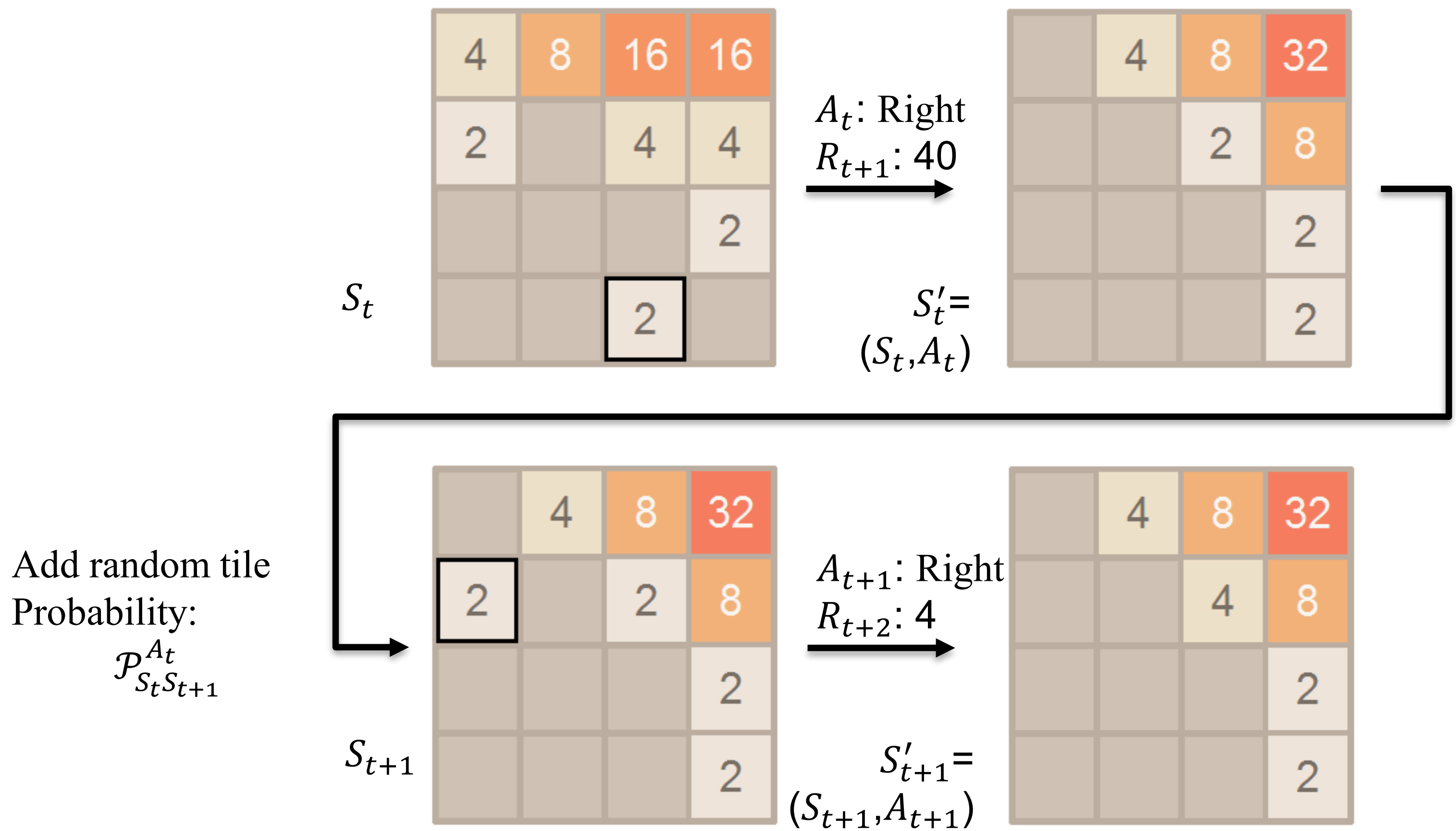
- RL1: Introduction to Reinforcement Learning
- RL2: Reinforcement Learning for Lightweight Model
  - Applications
  - Fundamentals of RL
- RL3: Value Based Reinforcement Learning
  - Fundamentals of Value Based RL
  - Algorithms
- RL4: Policy-based Reinforcement Learning
  - Fundamentals of Policy Based RL
  - Algorithms

Reinforcement Learning for Lightweight Model

- Applications
  - 2048 (Temporal Difference Learning)
  - Go Programs (with Monte-Carlo Tree Search)
- Fundamentals of Reinforcement Learning
  - Markov Decision Process (MDP)
  - Dynamic Programming (Tabular RL)

Case Study: 2048

- [Szubert et al., 2014; Yeh et al., 2016]



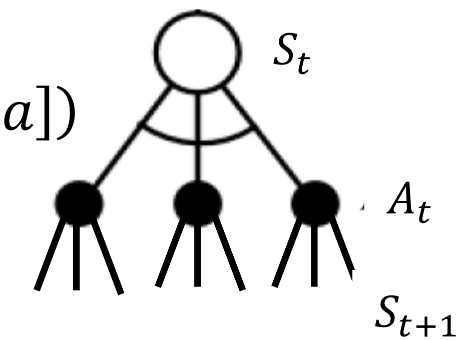
2048 RL Agent

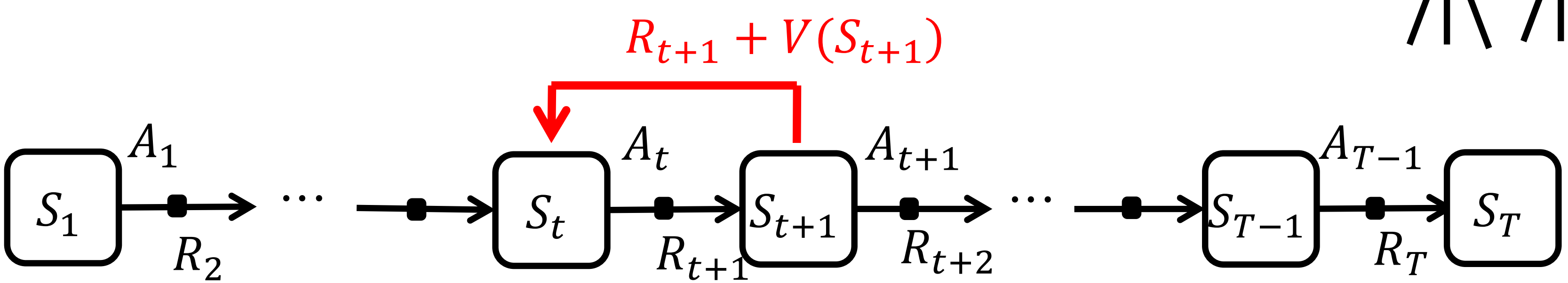
17 different numbers on each cell
And 4x4 (=16) cells in total.

- Value function:
  - The expected score/return G_t from a board S
  - But, #states is huge
    - About 17^{16} ($\cong 10^{20}$).
      - Empty (→ 0), 2 (=2^1 → 1), 4 (=2^2 → 2), 8 (=2^3 → 3), …, 65536 (=2^{16} → 16).
  - Need to use value function approximator.
- Policy:
  - Simply choose the action (move) with the maximal value based on the approximator.
- Model: agent's representation of the environment
  - After a move, randomly generate a tile:
    - 2-tile: with probability of 9/10
    - 4-tile: with probability of 1/10
  - Reward: simply follow the rule of 2048.

TD Learning in 2048

- Value function: (Normally $\gamma = 1$)
  - Update value $V(S_t)$ toward TD target $R_{t+1} + \gamma V(S_{t+1})$
    $V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
    - TD error: $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
- Making a decision (based on value).
  $$\pi(s) = argmax_a(R_{t+1} + \mathbb{E}[V(S_{t+1}) \mid S_t = s, A_t = a])$$
  - Problem: Less efficient upon making decision.





Q-Learning in 2048

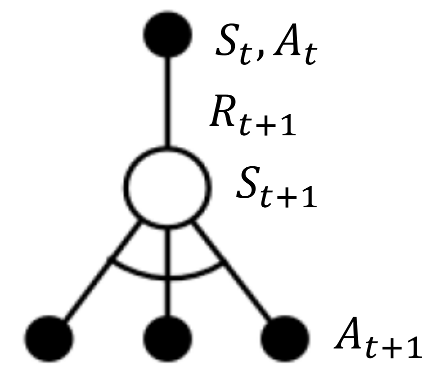
- Q-value function: (Normally $\gamma = 1$)
  - Update value $Q(S_t, A_t)$ toward TD target $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$

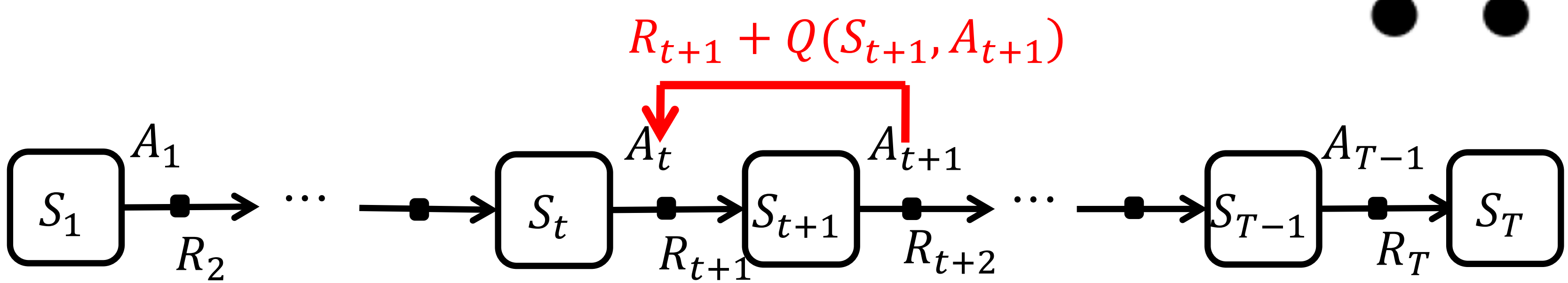
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

- Making decision (based on value).

$$\pi(s) = argmax_a(Q(S_t, a))$$

  - more efficient.
  - A minor problem: Four times more memory

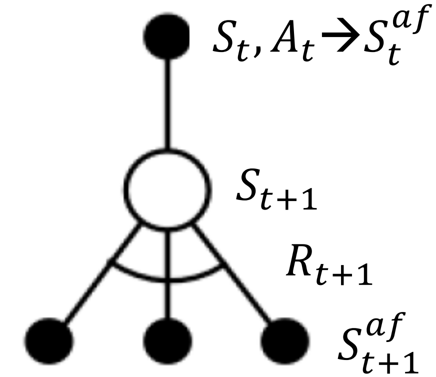




Afterstates in 2048

- Afterstate S_t^{af} is a state after action A_t at S_t.
  - Why not use S_t^{af} instead of (S_t, A_t)?
  - Note: in 2048, the reward R_{t+1} is not included in S_t^{af}.
- Afterstate value function: (Normally $\gamma = 1$)
  - Update value $V^{af}\left(S_t^{af}\right)$ toward TD target $R_{t+1} + \gamma \max_a (V^{af}\left(S_{t+1}^{af}\right))$

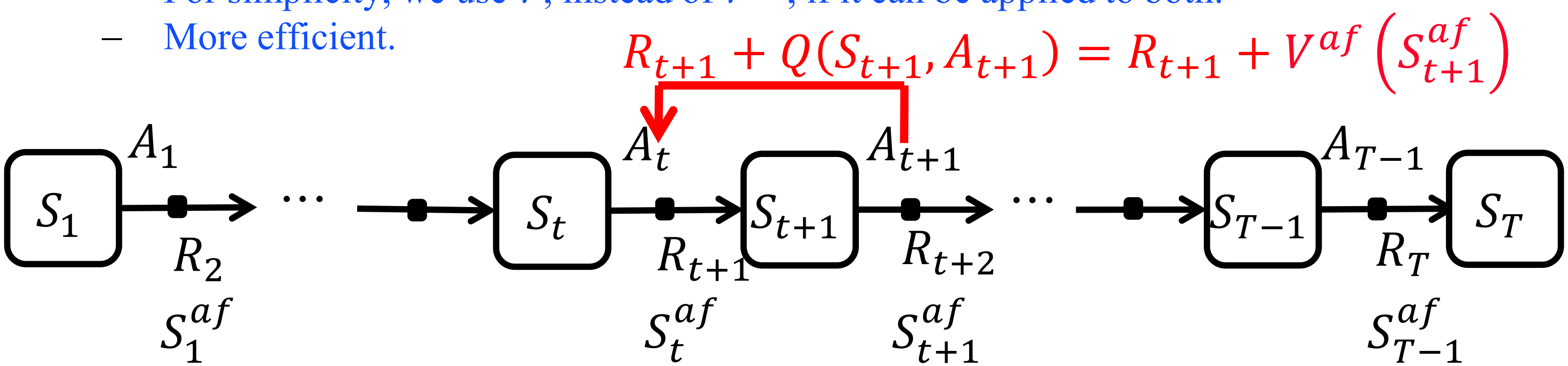
$$V^{af}\left(S_t^{af}\right) \leftarrow V^{af}\left(S_t^{af}\right) + \alpha(R_{t+1} + \gamma \max_a (V^{af}\left(S_{t+1}^{af}\right)) - V^{af}\left(S_t^{af}\right))$$



- Making decision (based on value).

$$\pi(s) = argmax_a\left(V^{af}\left(S_t^{af}\right)\right)$$

  - For simplicity, we use V, instead of V^{af}, if it can be applied to both.
  - More efficient.



Value Function Approximation

- As mentioned above, #states is huge, so we need to use value function approximation.
  - Use a value function approximator, $\hat{v}(S, \theta) \approx V(S)$.
  - Simply use deterministic policy: $\pi(S) = argmax_a(\hat{v}(S, \theta))$
- But, what kind of value function approximator can we use?
  - What features can we choose?
    - Traditionally, # of empty cells, # of continuous cells, big tiles, etc.
  - Linear (like n-tuple network) vs. non-linear (like NN)
- n-tuple network is a powerful network for 2048.
  - Explore a large set of features.
  - Simplify operations by linear value function approximation.
  - Features in each network is one-hot vector.

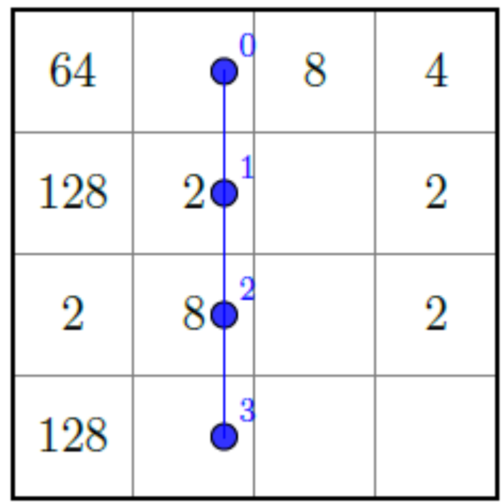
Gradient Descent

Now, how to do the update: $V(S_t) \leftarrow V(S_t) + \alpha\Delta V$

- Update value $V(S_t)$ towards TD target $y_t = R_{t+1} + V(S_{t+1})$
  $$\Delta V = \left(R_{t+1} + V(S_{t+1}) - V(S_t)\right) = \left(y_t - V(S_t)\right)$$
  α: learning rate, or called step size.
  - Note: $\gamma = 1$ here.
- Objective function is to minimize the following loss in parameter θ. (note: $\hat{v}(S,\theta) = x(S)^{\mathrm{T}}\theta$)
  $$\mathcal{L}(\theta) = \mathbb{E}\left[\left(y_t - \hat{v}(S,\theta)\right)^2\right]$$
  $$\nabla_\theta \mathcal{L}(\theta) = \left(y_t - \hat{v}(S,\theta)\right) \cdot \nabla_\theta \hat{v}(S,\theta) = \Delta V \cdot x(S)$$
- Update features w: step-size * prediction error * feature value
  $$\theta \leftarrow \theta + \alpha\Delta V \cdot \frac{x(S)}{\|x(S)\|} \Rightarrow$$
  $$V(S_t) \leftarrow V(S_t) + \alpha\Delta V$$

N-Tuple Network

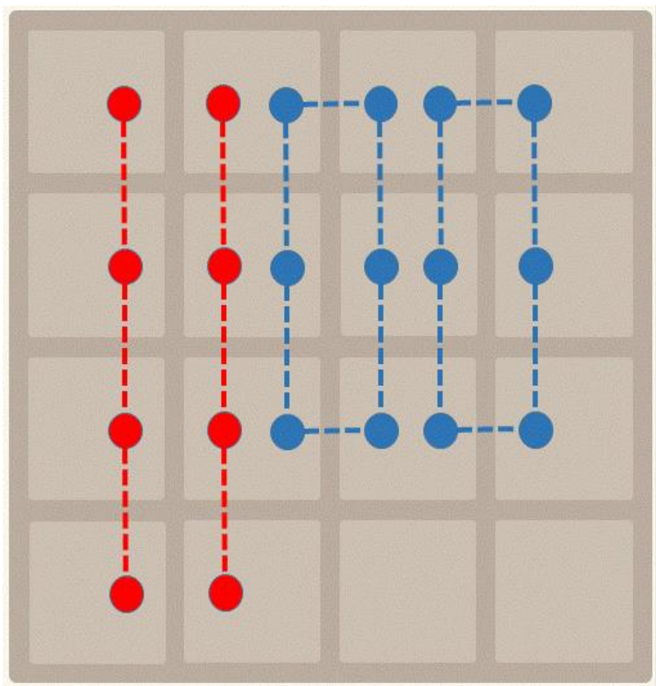
- Characteristics:
  - Provide with a large number of features.
  - Easily update.
- Example: 4-tuple networks as shown.
  - Each cell has 16 different tiles
  - 16^4 features for this network.
    - But only one is on, others are 0.
      - […, 0, 0, 1, 0, 0, …]
      - So-called one-hot vector.
    - So, we can use table lookup to find the feature weight.
  - Note: tabular RL is just like 16-tuple network in the case of 2048.

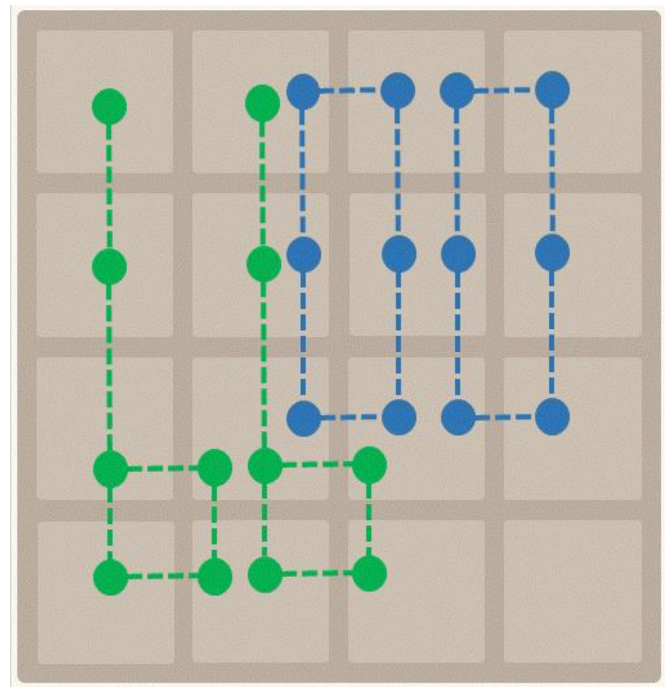


0123	weight
0000	3.04
0001	−3.90
0002	−2.14
⋮	⋮
0010	5.89
⋮	⋮
0130	-2.01
⋮	⋮

Other N-Tuple Networks

- Left: [Szubert et al., 2014]; Right: [Yeh et al., 2016]
- Some researchers even used 7-tuple network.





Update Features in N-Tuple Networks

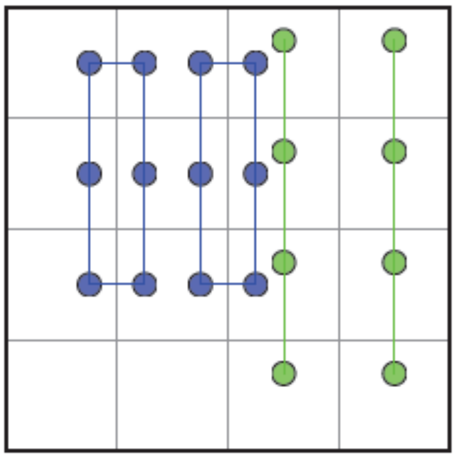
- For each n-tuple networks, simply update one weights.
- Features:
  - 8 x 16^4 features, $x(S) = [0, 1, 0, \ldots, 0, 0, 1, \ldots, \ldots, 1, 0, 0, \ldots]$
    - All 0s, except for 8 ones.
      - One 1 every 16^4 features.
      - Let their indices be $\boldsymbol{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8}$.
  - Only need to update $\alpha\Delta V$ at the features indexed by these indices.
  - Very efficient and fast.
- For k n-tuple networks,

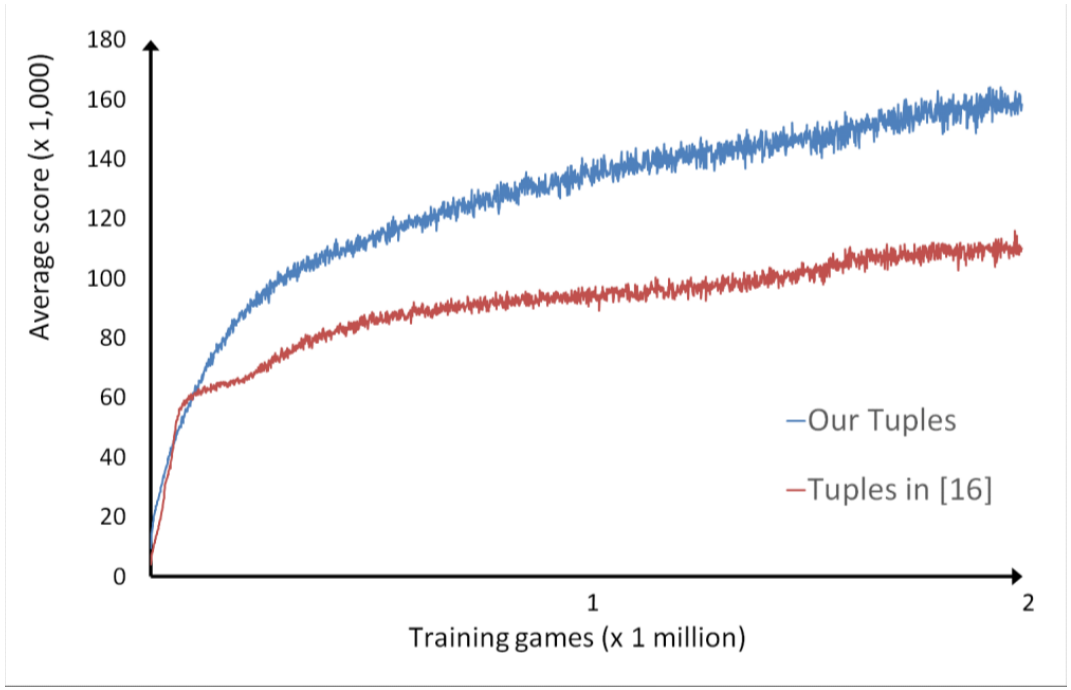
$$\hat{v}(S,\theta) = x(S)^{\mathrm{T}}\theta = \sum_{j=1}^{n} x_j(S)\theta_j = \sum_{i=1}^{k} LUT_i[index(s_i)]$$

  - LUT_i: the i-th n-tuple network lookup table.
  - $index(s_i)$: The index in the i-th n-tuple network of state S.
- Update features w: step-size * prediction error * feature value
  - $\theta \leftarrow \theta + \alpha\Delta V \cdot x(S)$
  - Only need to update values θ_j with $\alpha\Delta V$ at $LUT_i[index(s_i)]$.

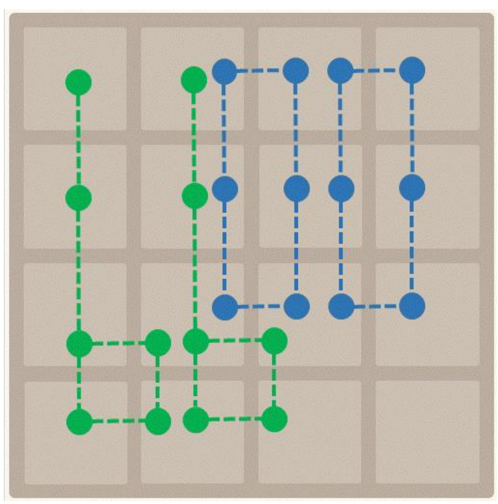
The N-Tuple Networks Used

- Use the following [Szubert and Jaskowaski 2014]





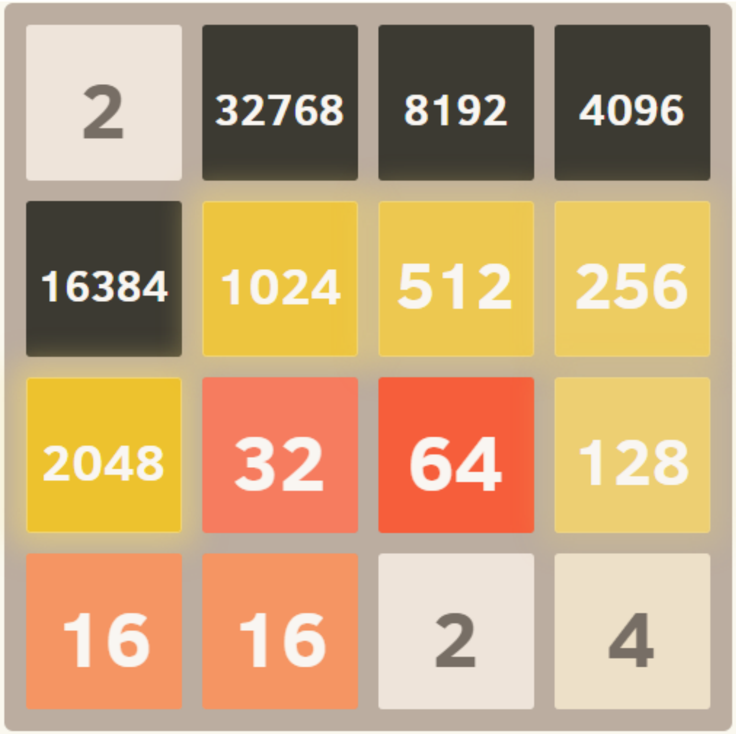
- Ours:

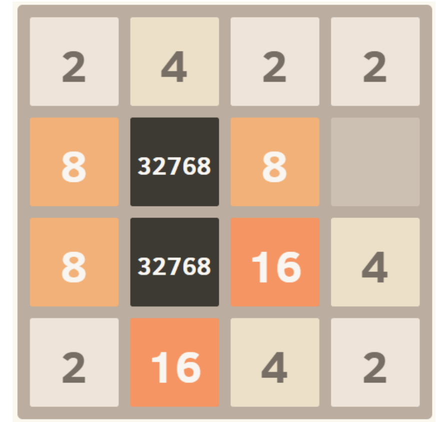


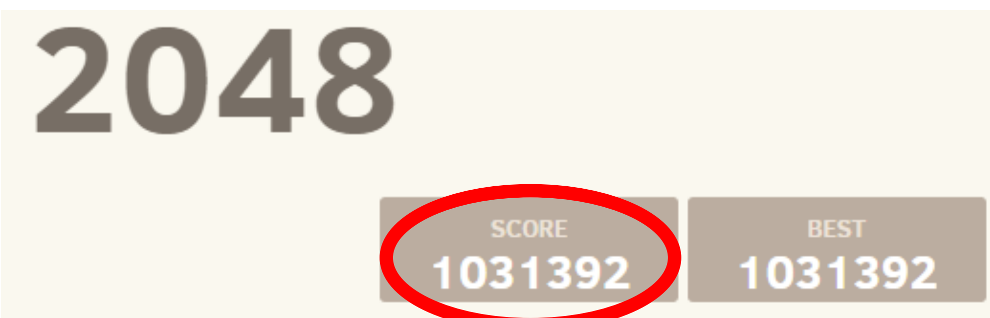
Our Results (2021)

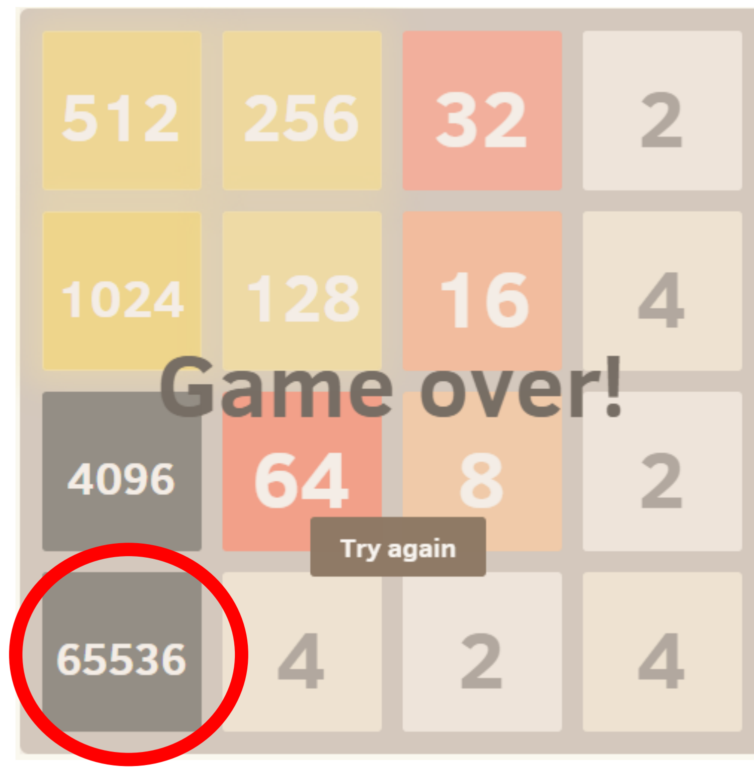
100 tested games	**CGI-2048 (2nd in contest, 2014)**	**Kcwu (1st in contest, 2014)**	**Jaśkowski (2018, Previous SOTA)**	**Current CGI-2048 (2021, Current SOTA)**
2048	100%	100%	**100%**	**100.0%**
4096	100%	100%	**100%**	**100.0%**
8192	94%	96%	**98%**	**99.8%**
16384	59%	67%	**97%**	**98.8%**
32768	0%	2%	**70%**	**72.0%**
Max score	367956	625260	**N/A**	**840384**
Avg score	251794	277965	**609104**	**625377**
Speed	500 moves/sec	>100 moves/sec	**1 move/sec**	**2.5 moves/sec**

The First 65536









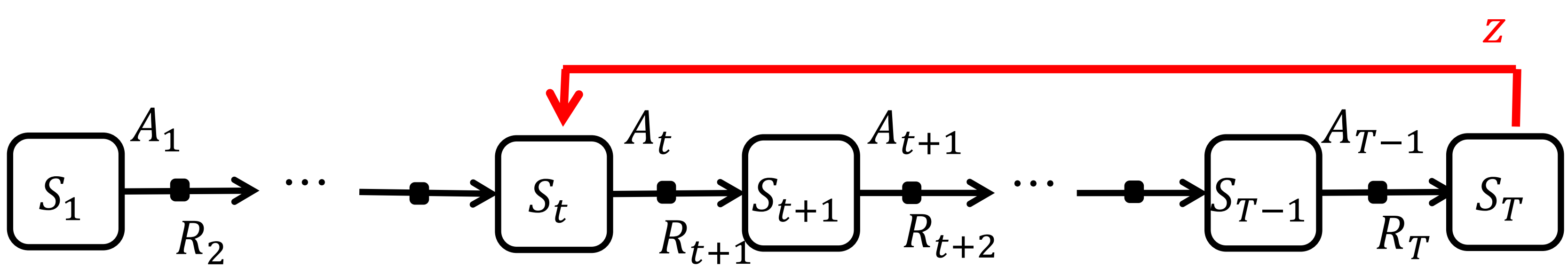
I-Chen Wu

Reinforcement Learning for Lightweight Model

- Applications
  - 2048 (Temporal Difference Learning)
  - Go Programs (with Monte-Carlo Tree Search)
- Fundamentals of Reinforcement Learning
  - Markov Decision Process (MDP)
  - Dynamic Programming (Tabular RL)

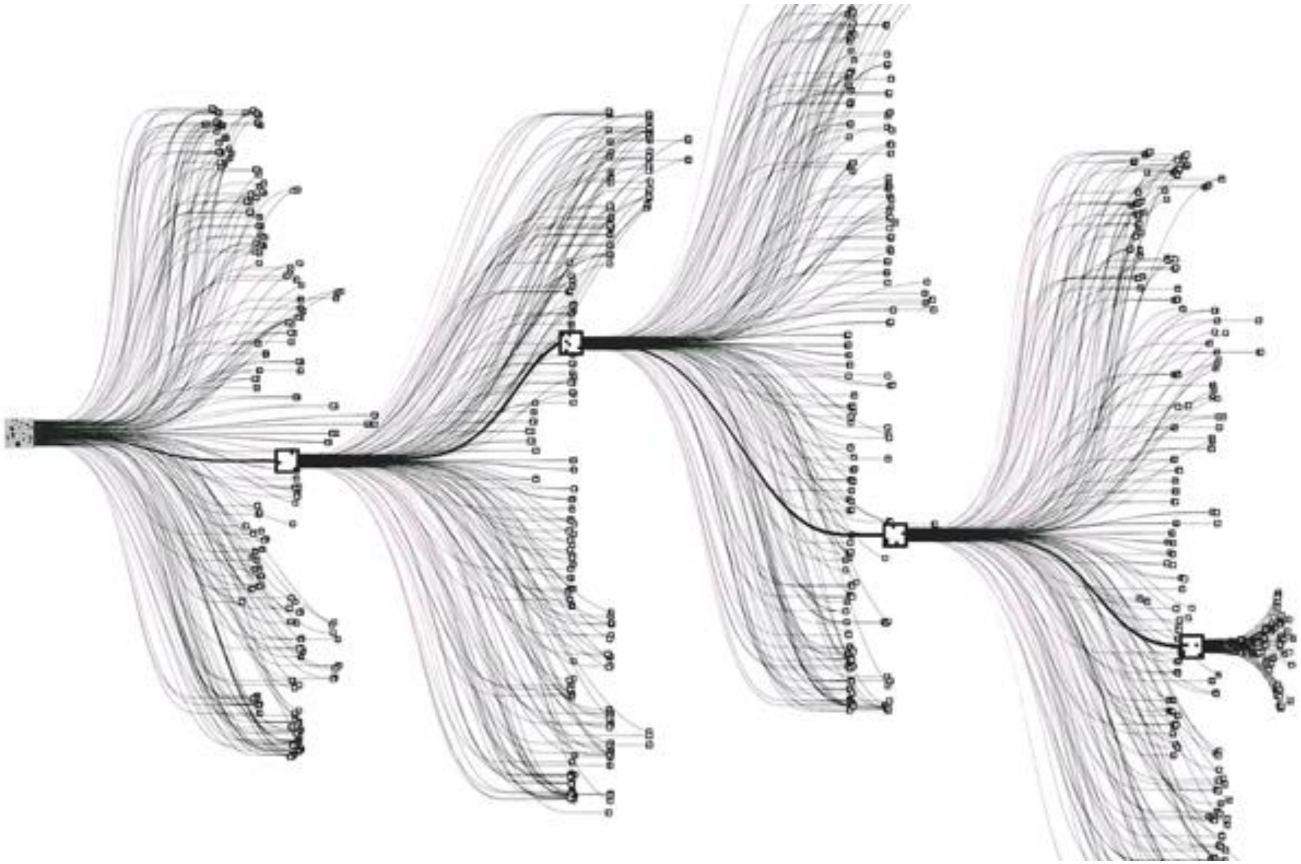
Case Study: Go

- Monte-Carlo Tree Search:
  - Monte-Carlo (MC) Learning (z: 1 for win, 0 for loss)
  - Multi-Armed Bandits
  - **Planning**
- Very successful for Go in the past two decades.
- And also applied to others successfully.
  - Other games like Havannah, Hex, GGP
  - Other applications, like mathematical optimization problems (scheduling, UCP, camera coverage).



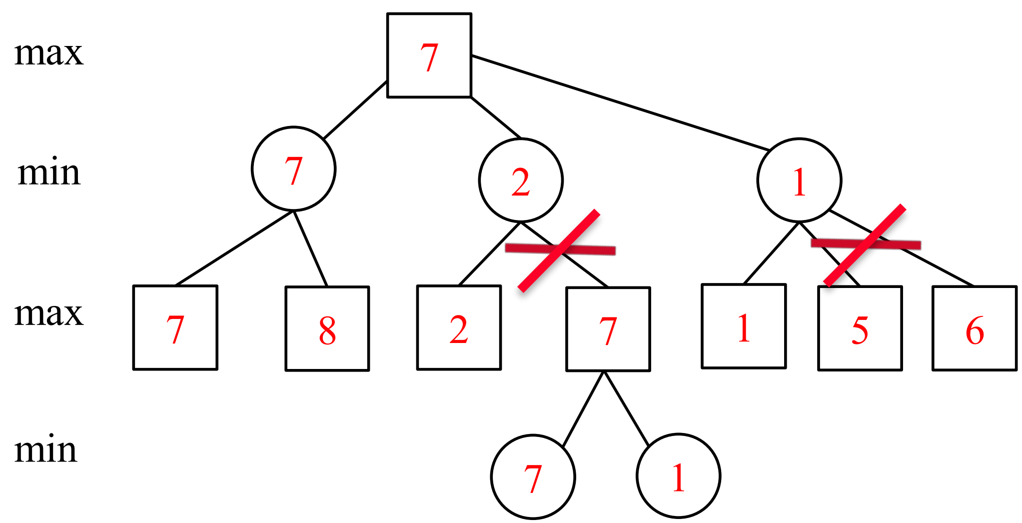
Go – One of the Most Popular Games

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



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
  - …



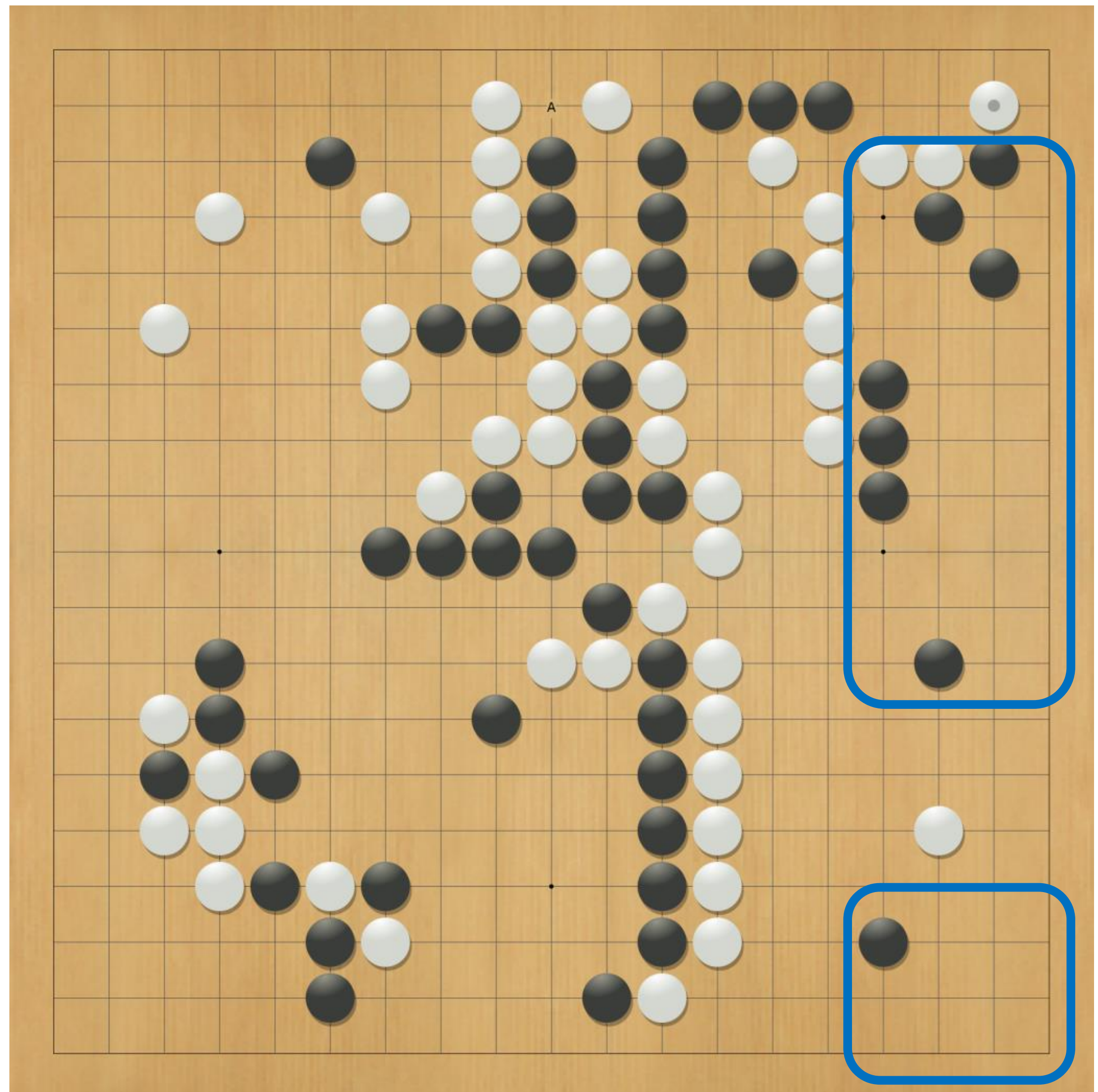
- Problem for chess:
  - need to consult with experts for feature values.

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,

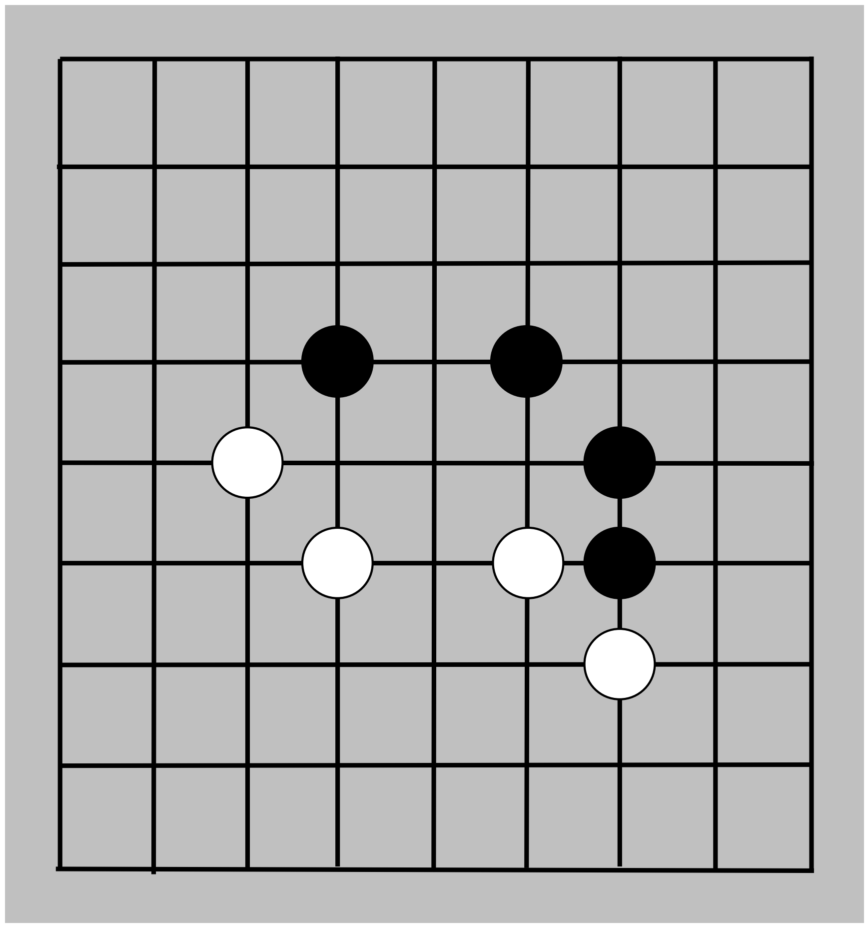
- Why not use Monte-Carlo?
- Calculate it based on stochastics.



Game 1: AlphaGo vs. 李世石

Rules Overview Through a Game (opening 1)

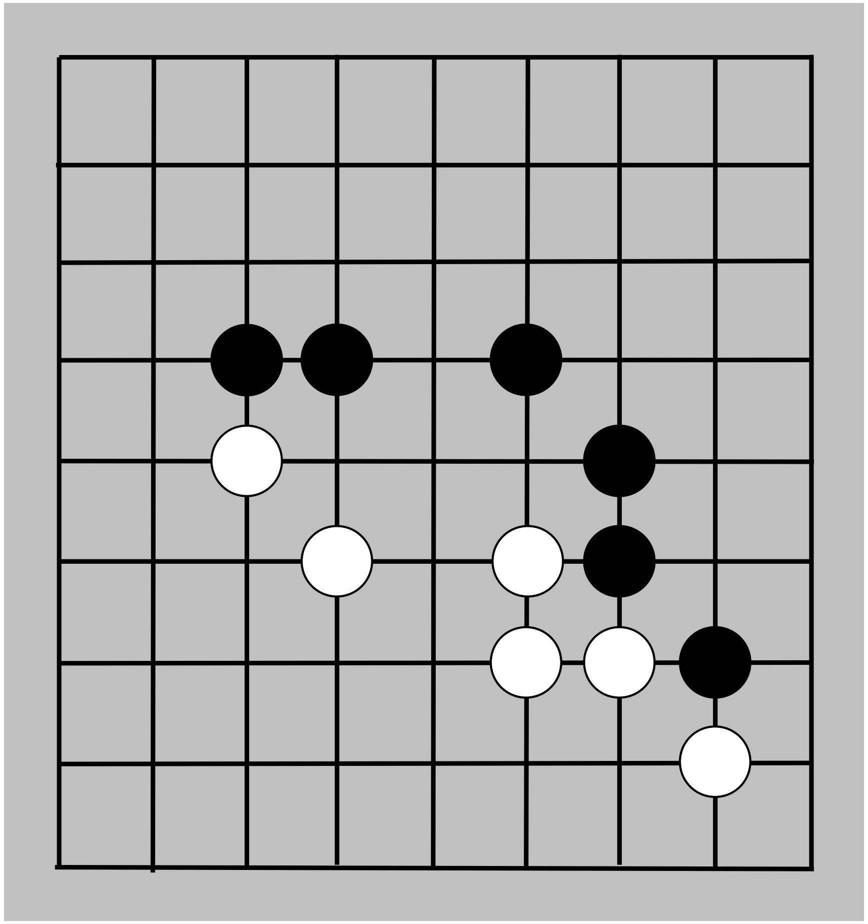
- Black/White move alternately by putting one stone on an intersection of the board.



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

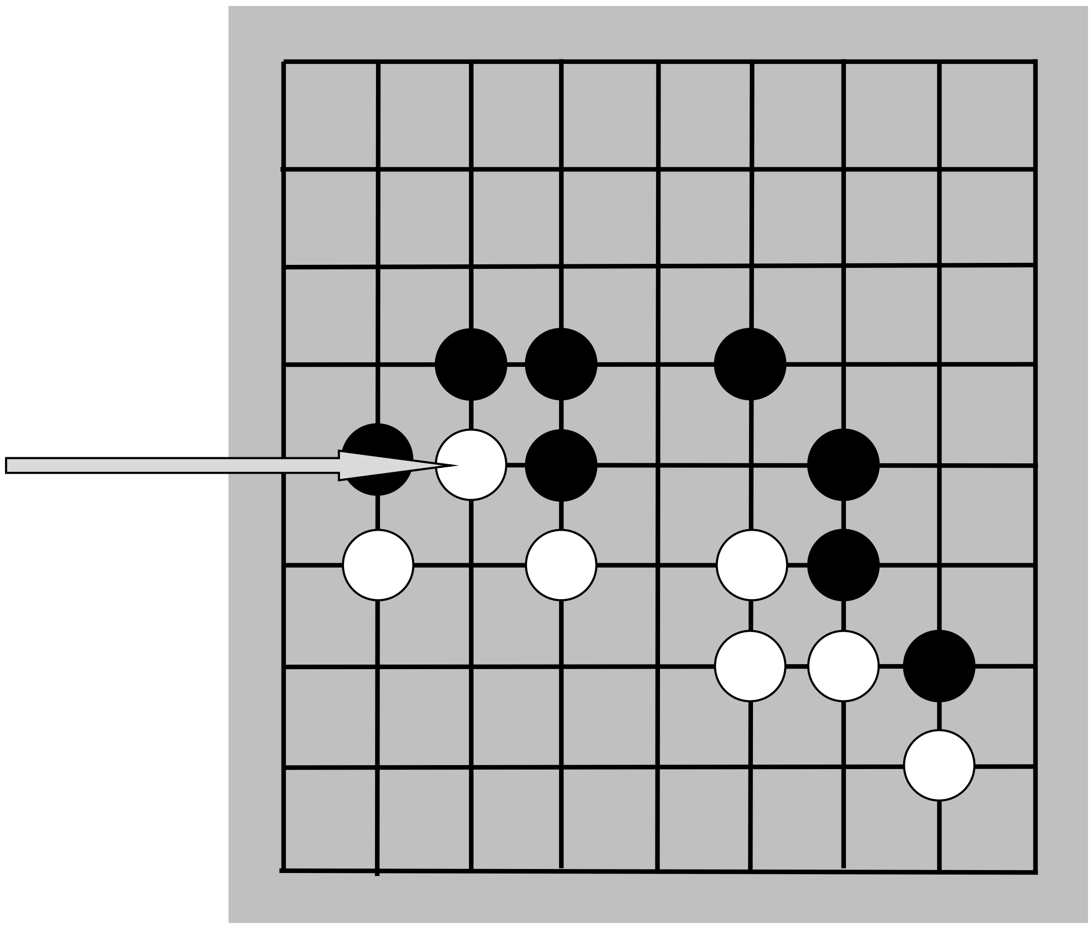
Rules Overview Through a Game (opening 2)

- Black and White aims at surrounding large « zones »



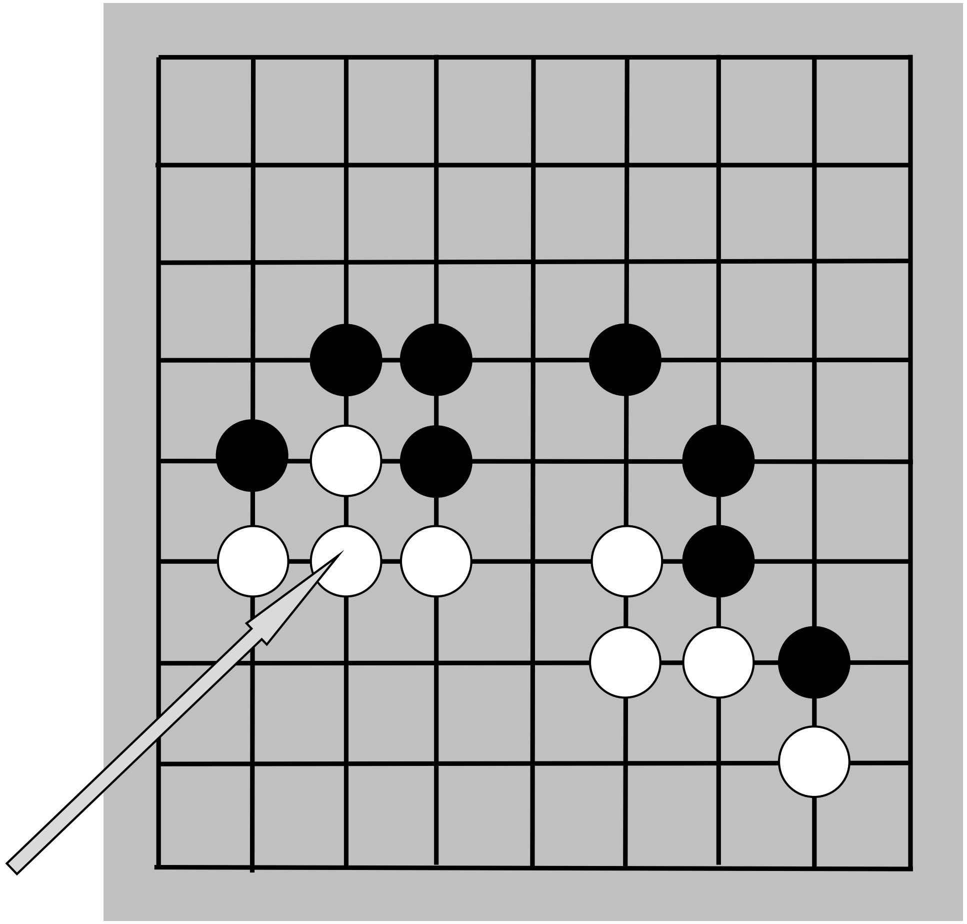
Rules Overview Through a Game (atari 1)

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



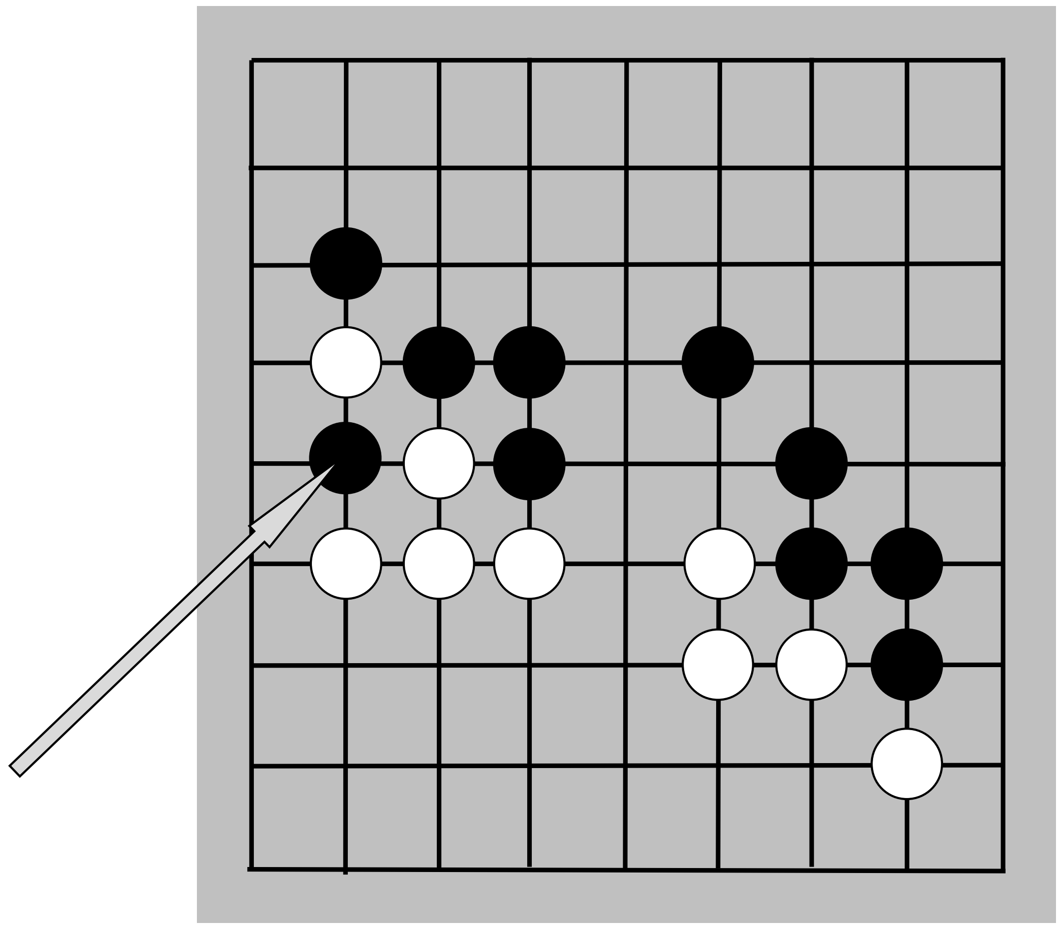
Rules Overview Through a Game (defense)

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



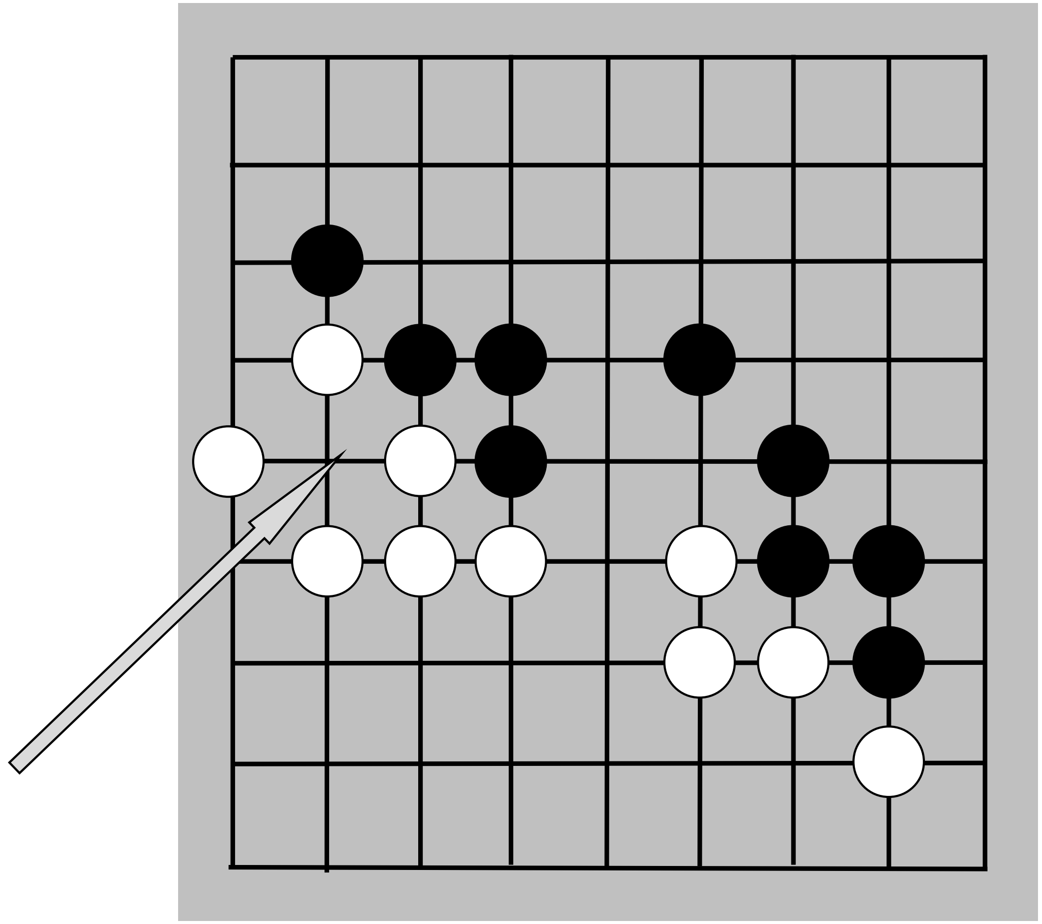
Rules Overview Through a Game (atari 2)

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



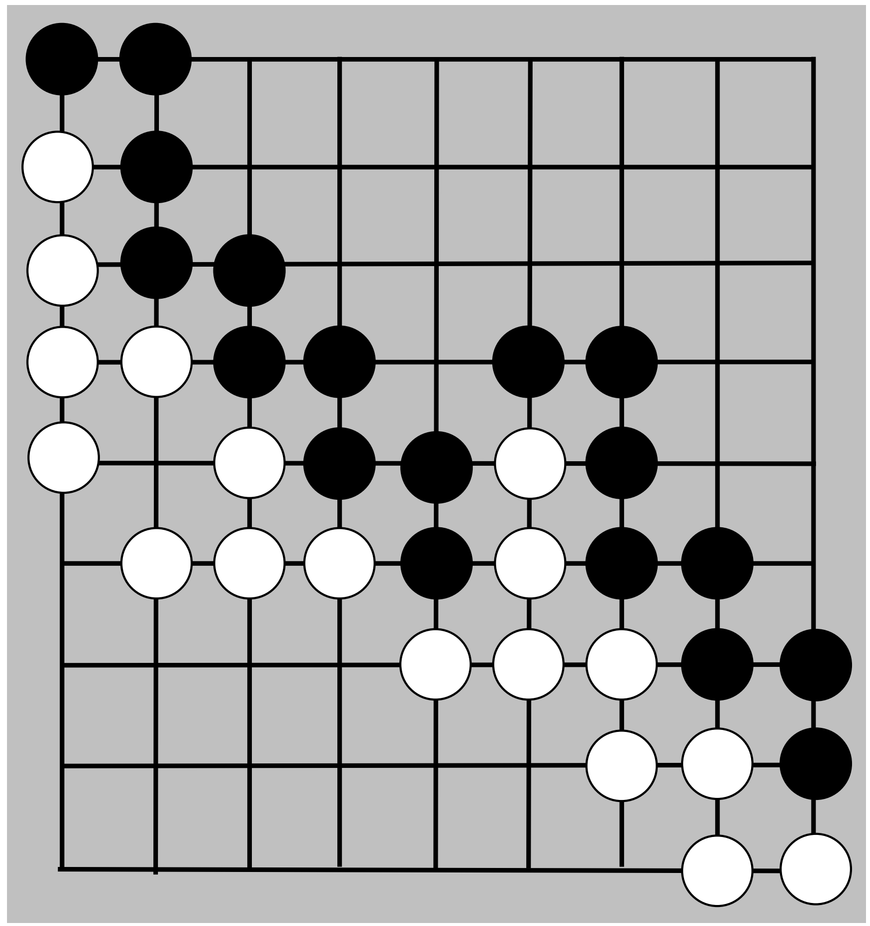
Rules Overview Through a Game (capture 1)

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



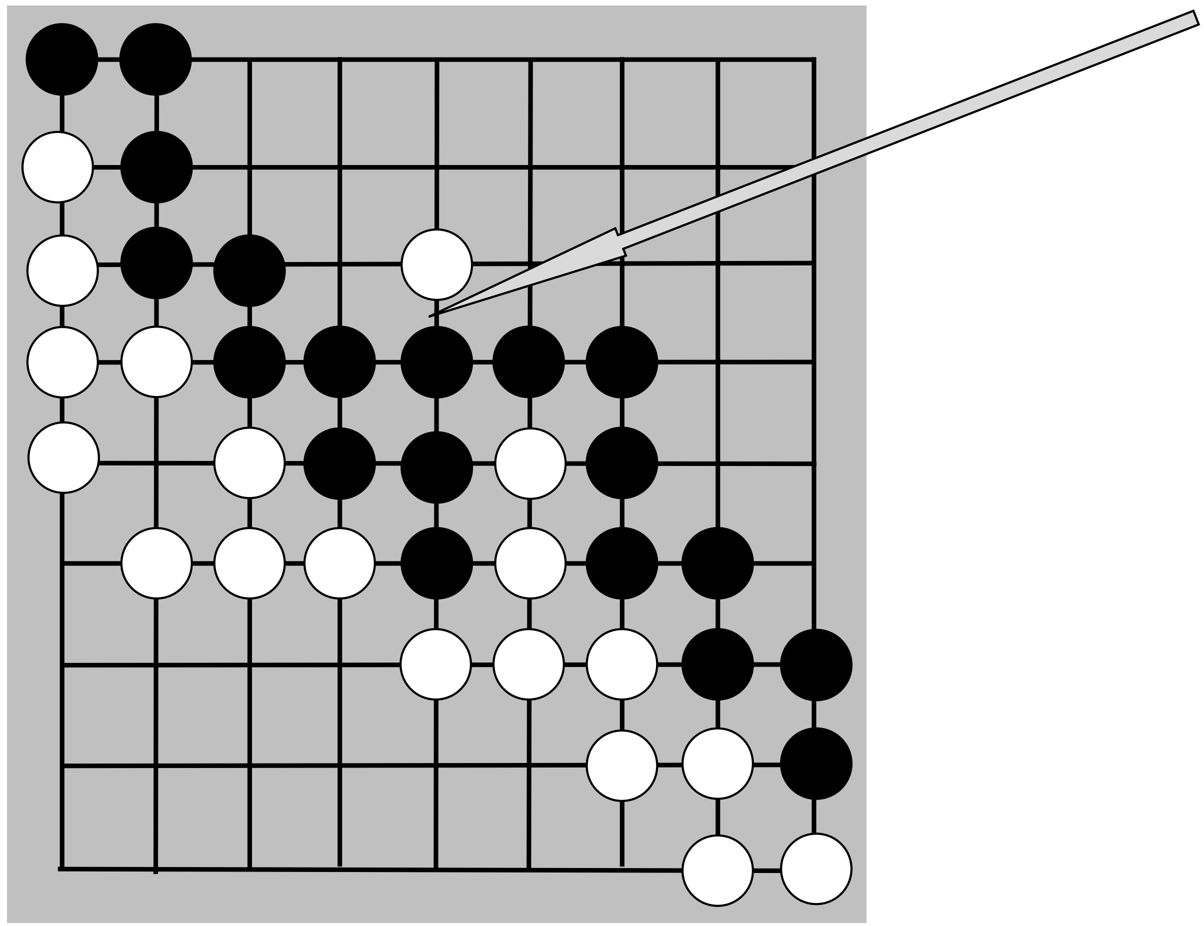
Rules Overview Through a Game (human end of game)

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



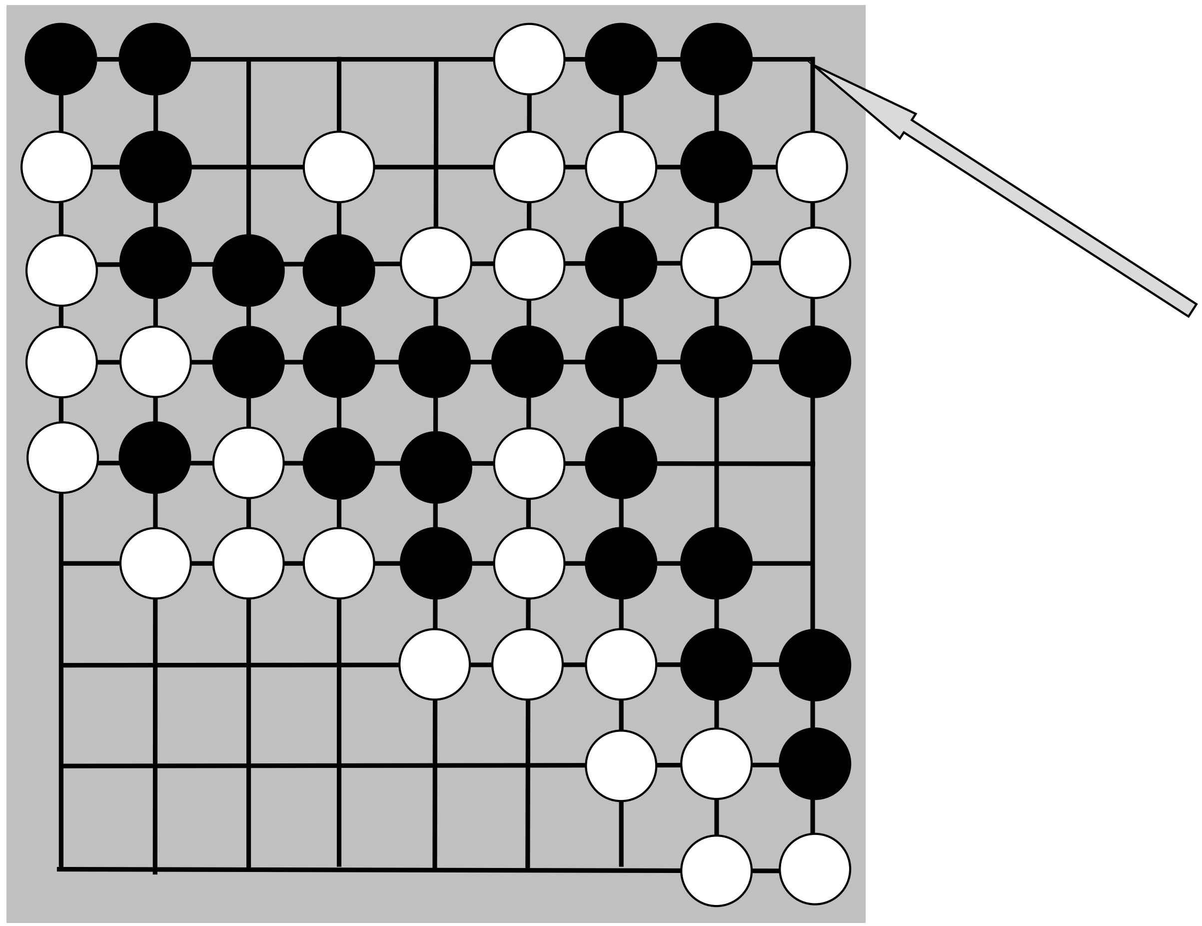
Rules Overview Through a Game (contestation 1)

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



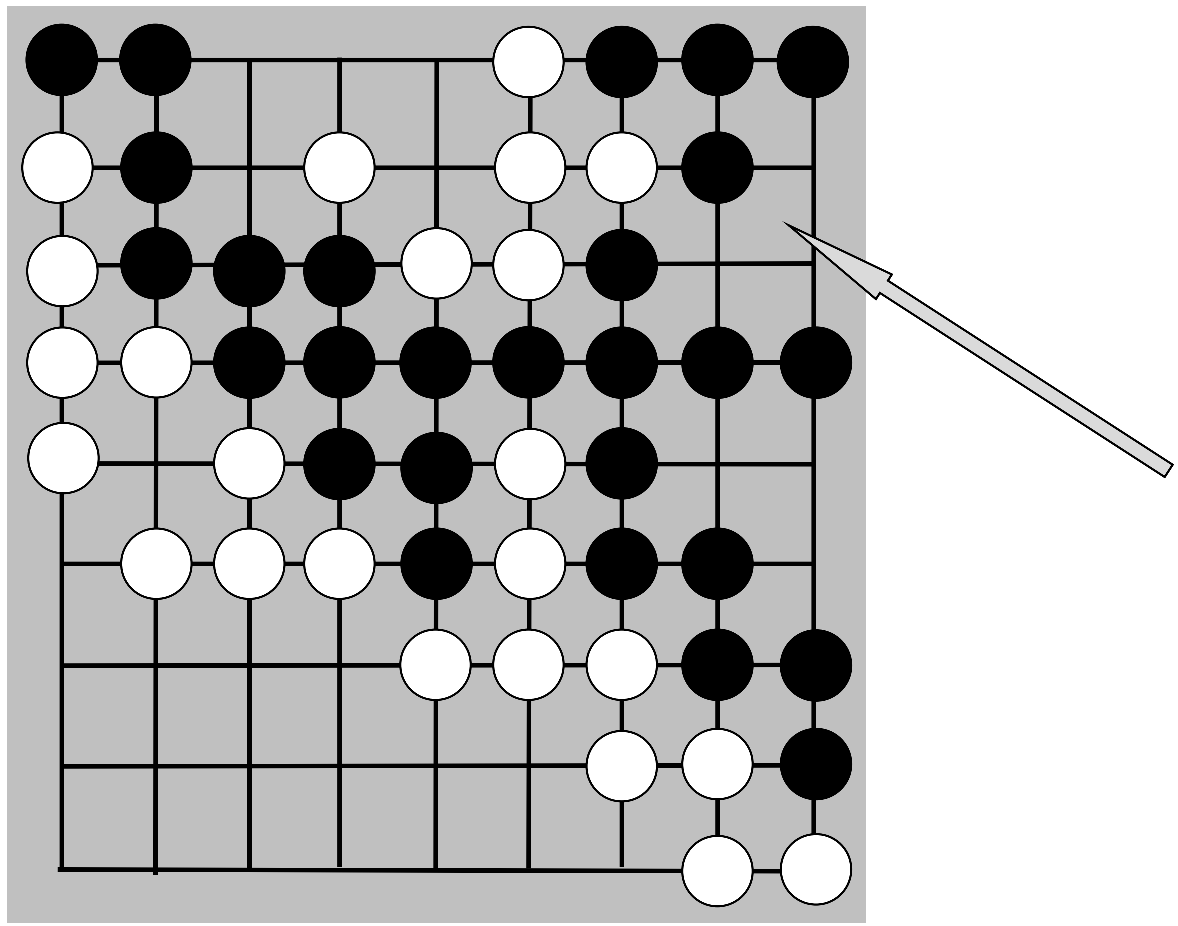
Rules Overview Through a Game (contestation 2)

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



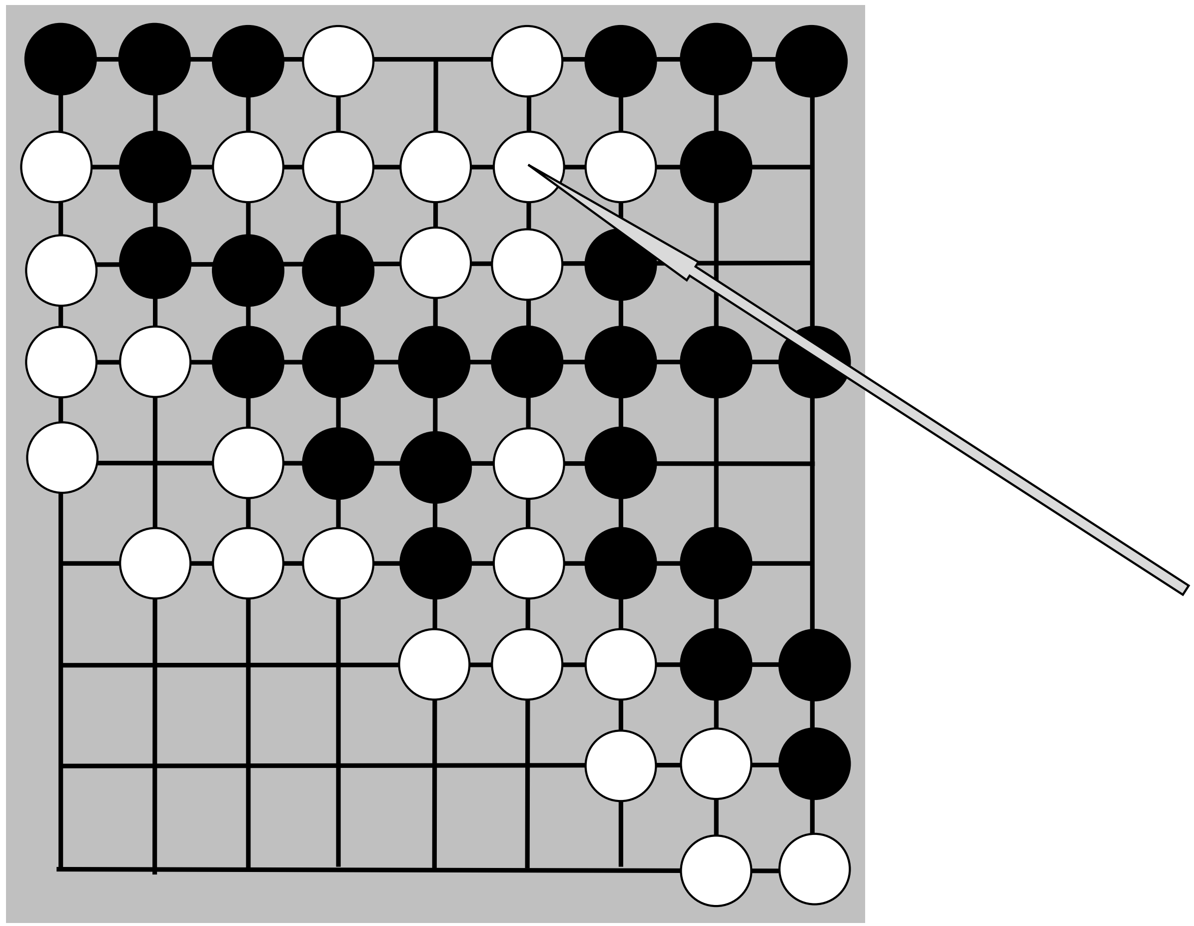
Rules Overview Through a Game (follow up 1)

- Black has captured the 3-stone white string



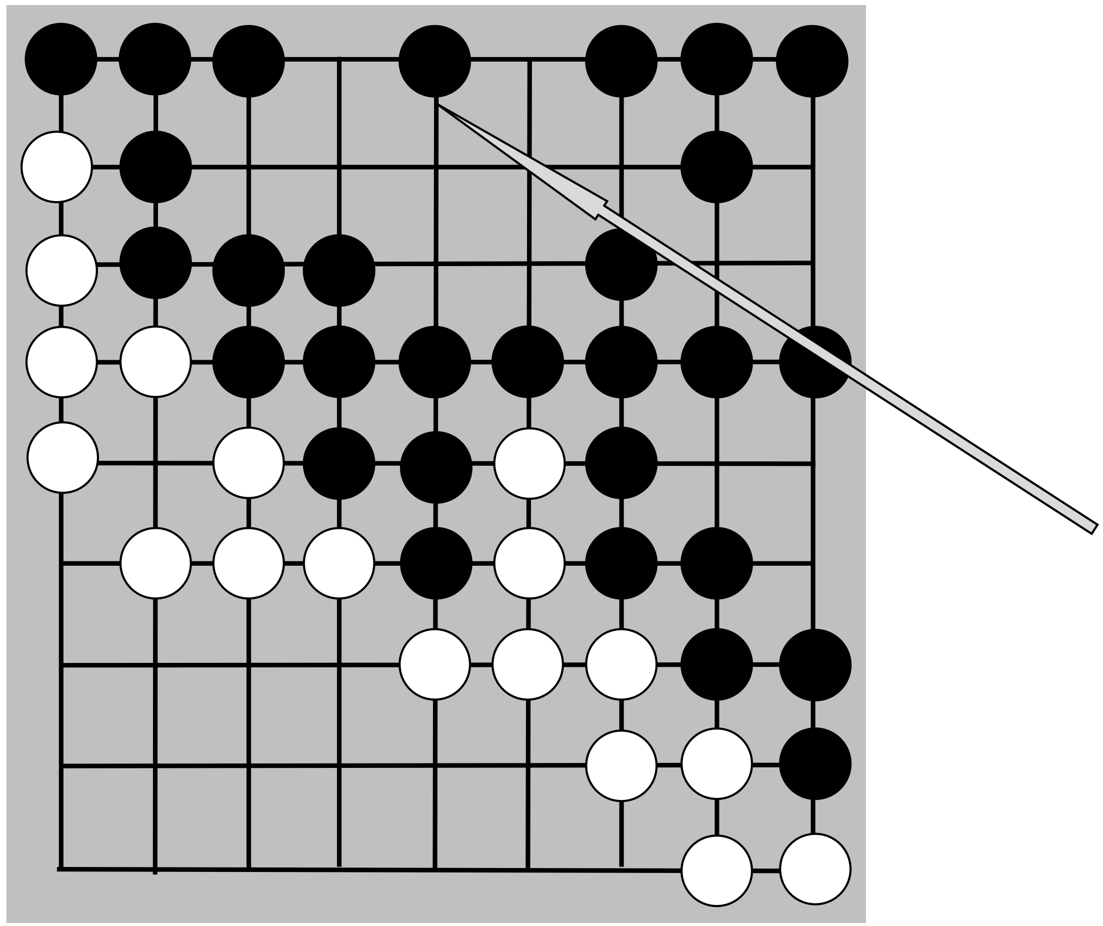
Rules Overview Through a Game (follow up 2)

- White lacks liberties…



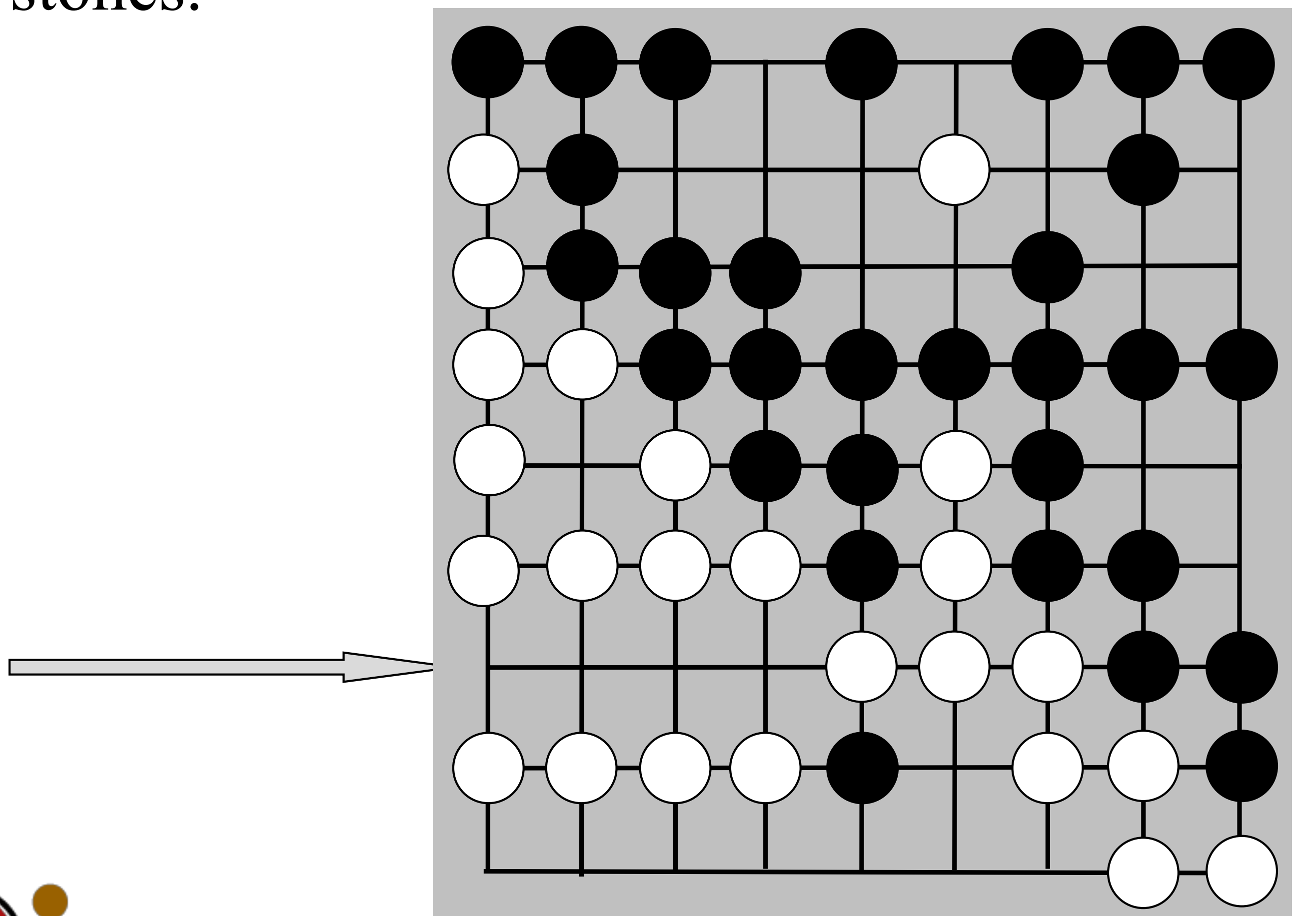
Rules Overview Through a Game (follow up 3)

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



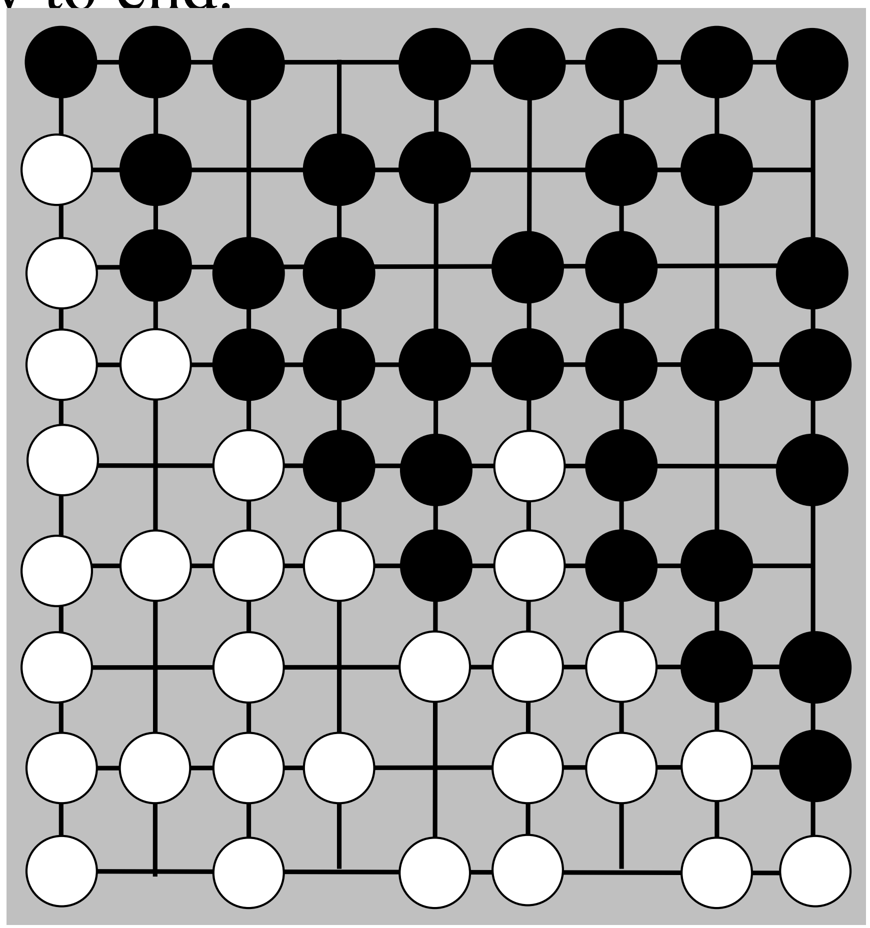
Rules Overview Through a Game (follow up 4)

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



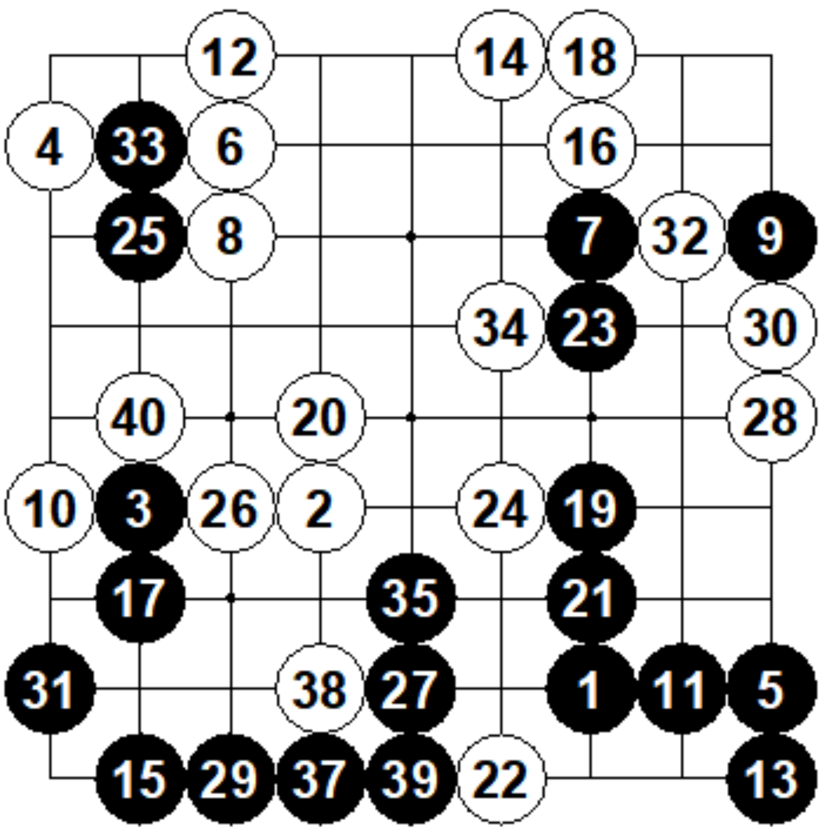
Rules Overview Through a Game (concrete end of game)

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

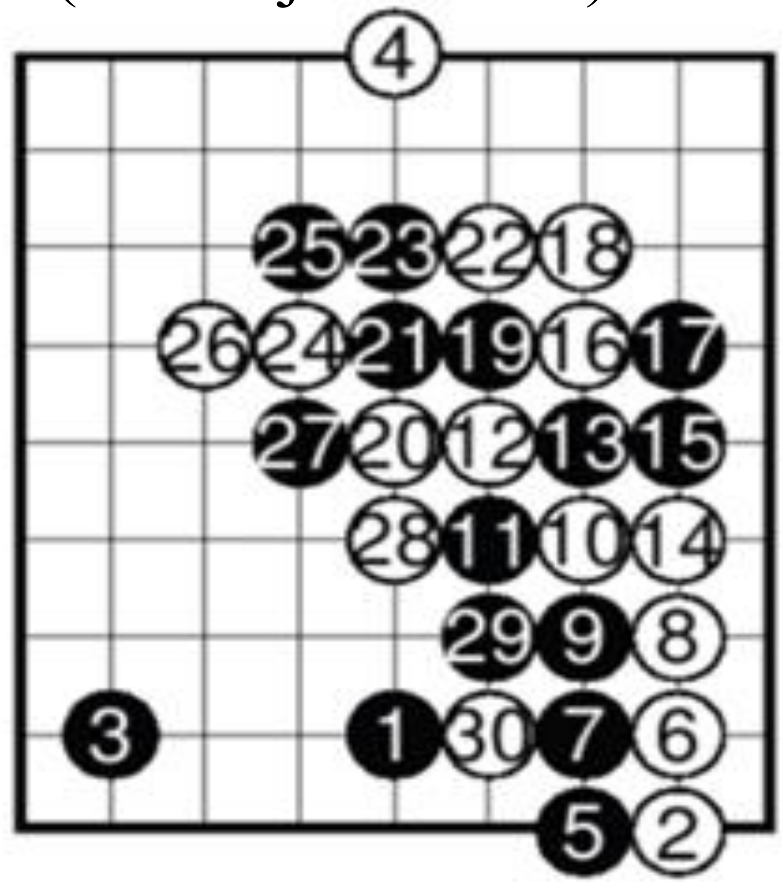


Performed OK Even for Moves (Nearly) at Random

Purely at random

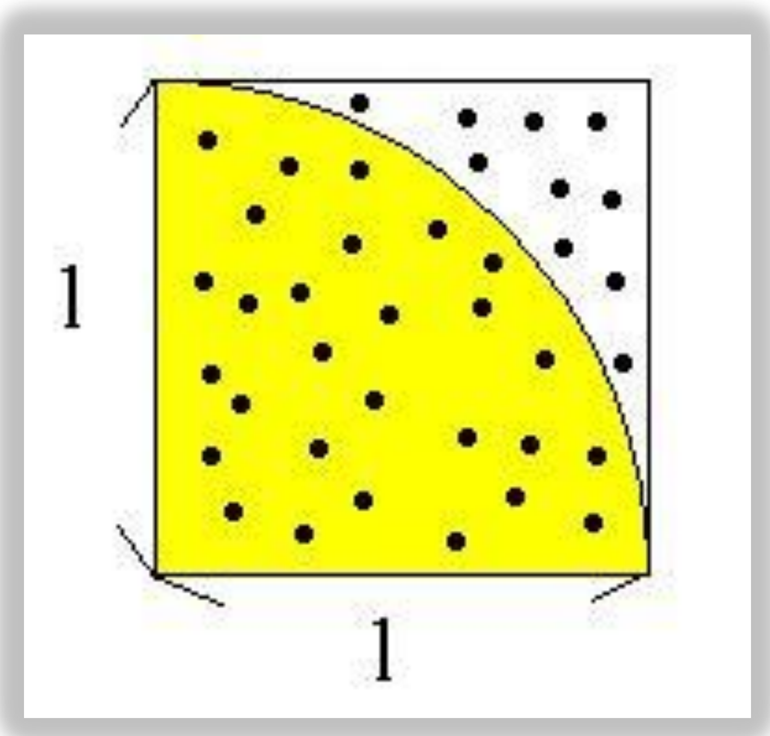


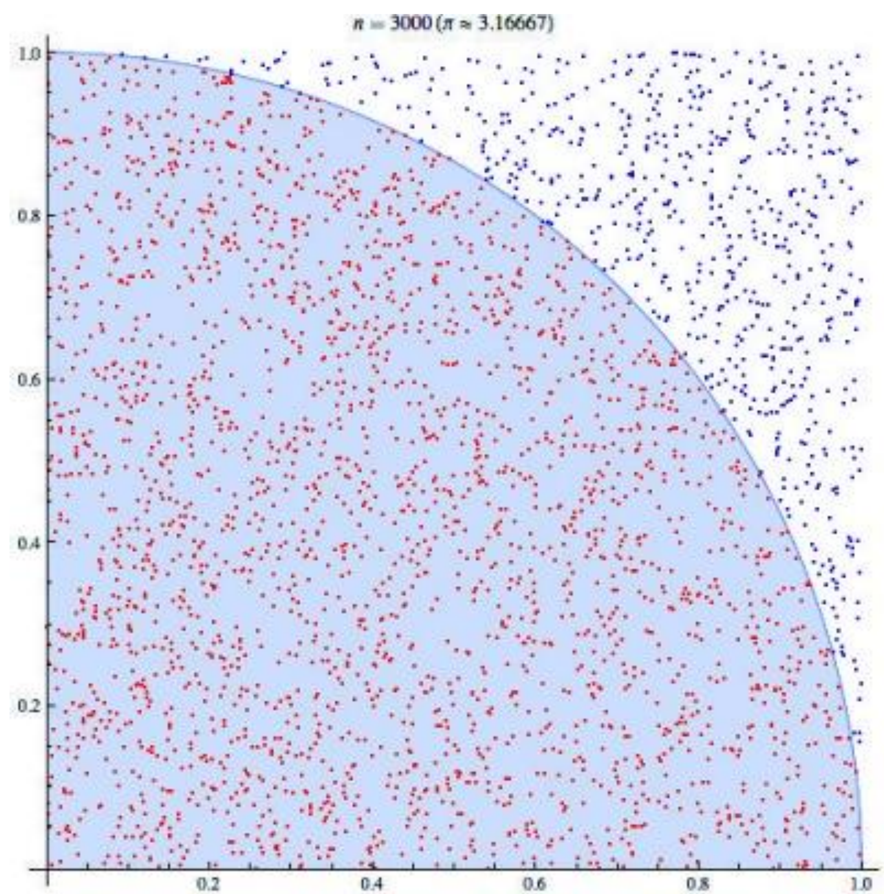
Have some heuristic (from Aja's Thesis)



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

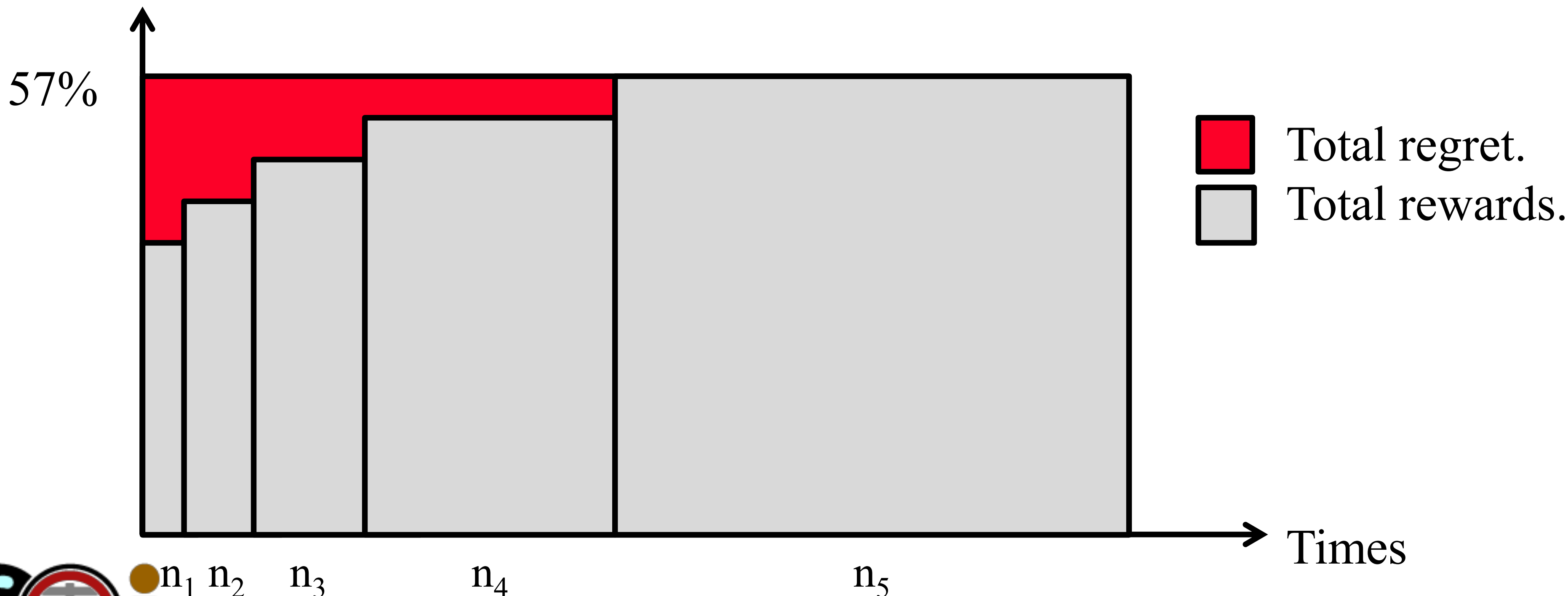
- UCB: Upper Confidence Bounds. [Auer *et al.*, 2002]
- Initialization: Play each machine once.
- Loop:
  - Play machine i that maximizes,

$$X_i + \sqrt{\frac{2 \log n}{n_i}}$$

  - where
    - $n = \sum_{i=1}^{k} n_i$ is the total number of playing trials.
    - n_i is the number of playing trials on machine i.
    - X_i is the (average) win rate on machine i.
- Key:
  - Ensure optimal machine is played exponentially more often than any other machine.

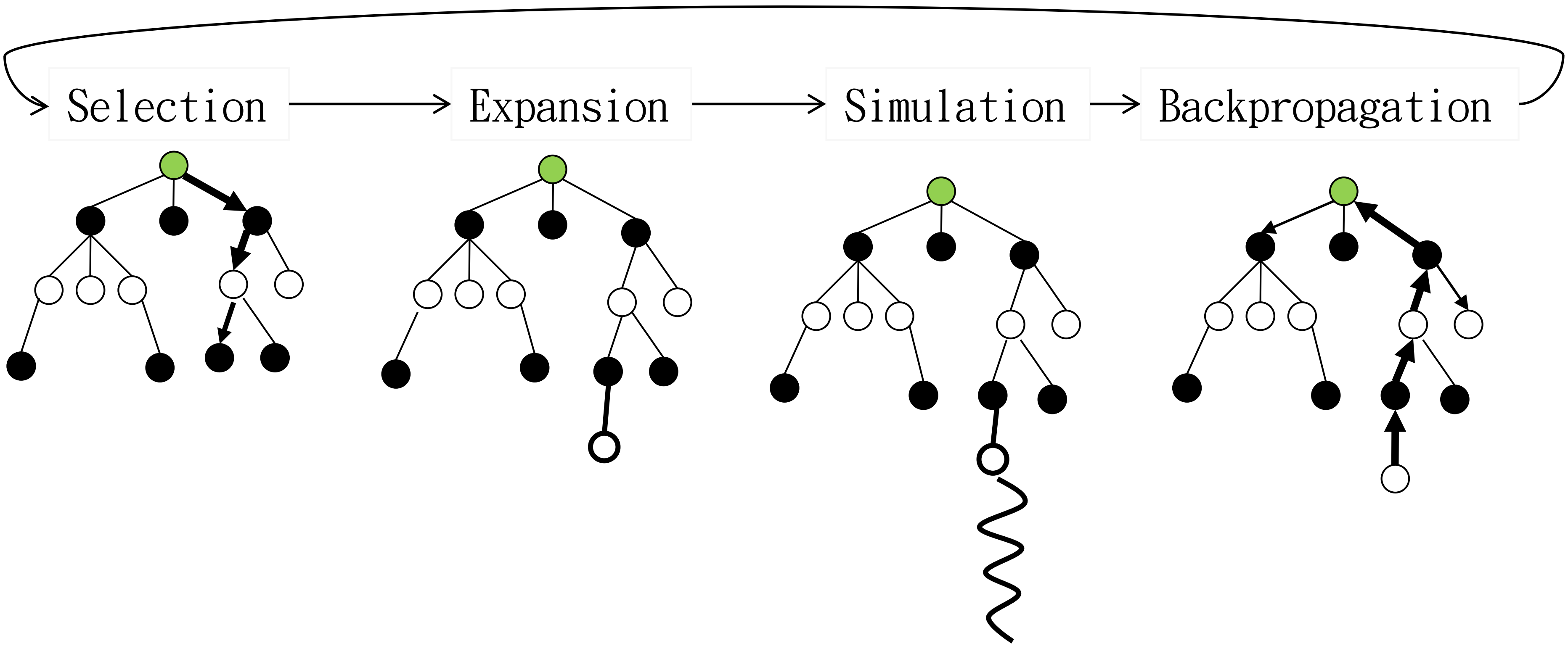
Cumulative Regret

- 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.



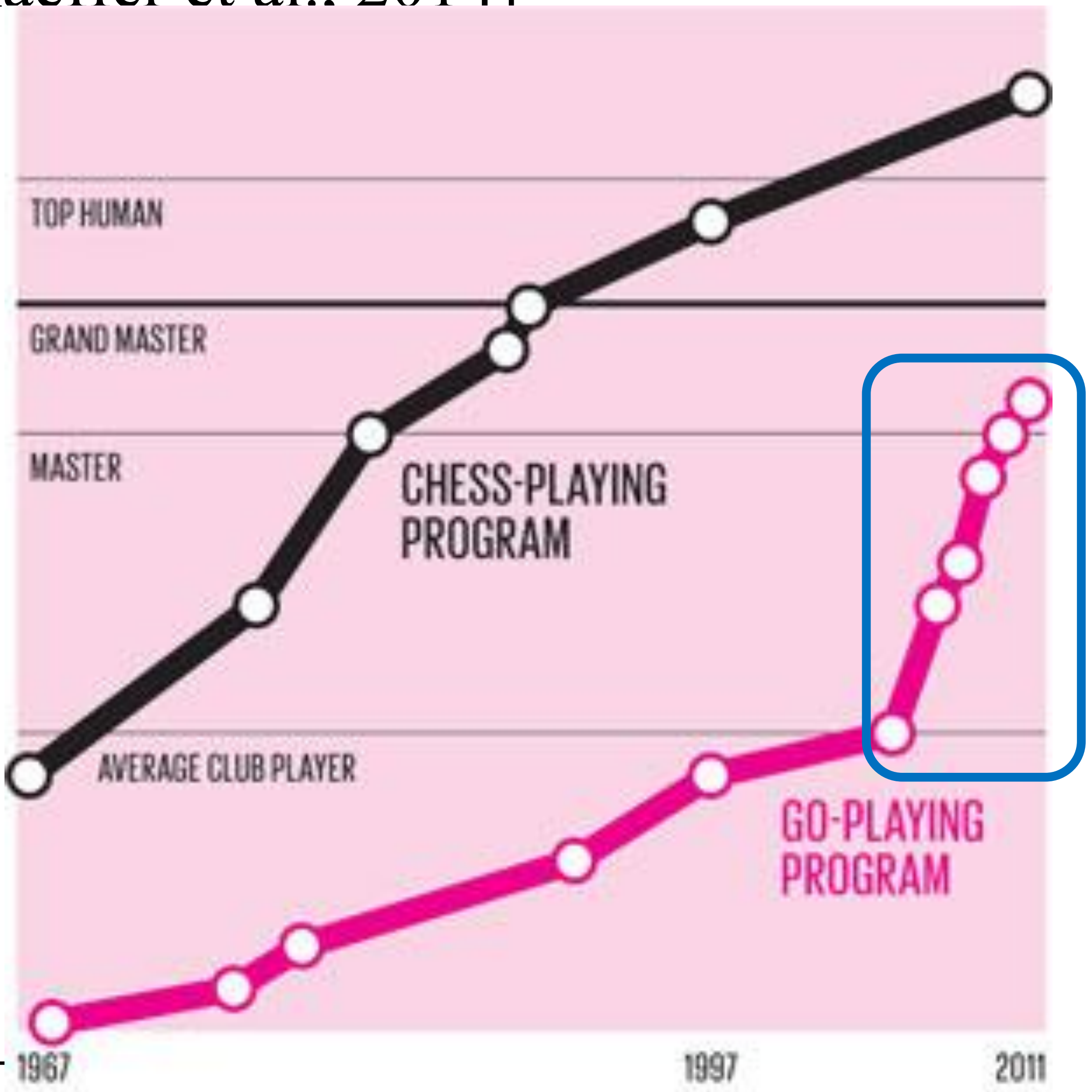
Monte-Carlo Tree Search

- A kind of planning
- A kind of Reinforcement learning



Strength of Go Program after MCTS

- [Schaeffer et al., 2014]



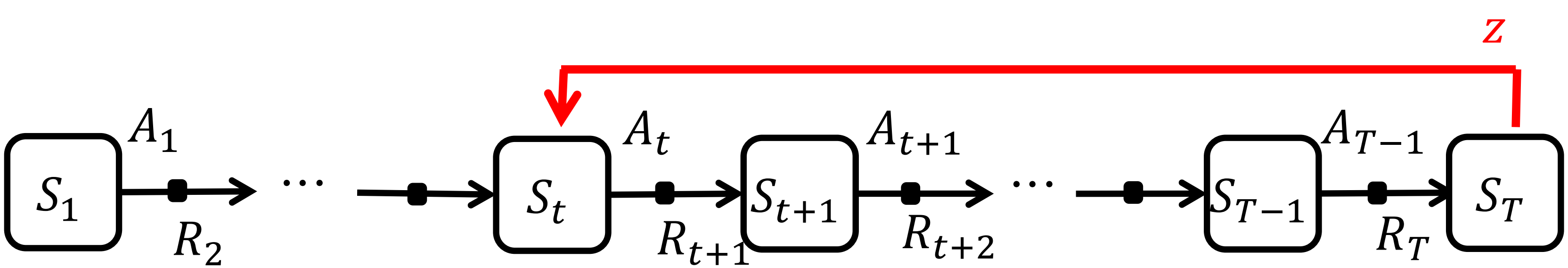
Strength grew fast, after MCTS.

Case Study: AlphaGo

- Use stochastic policy gradient ascent to maximize the likelihood of the human move a selected in state s

$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) \cdot z$$

  - θ: network parameter.
  - α: learning rate
  - z: the value of the episode
    - win/loss (1/-1) of the game



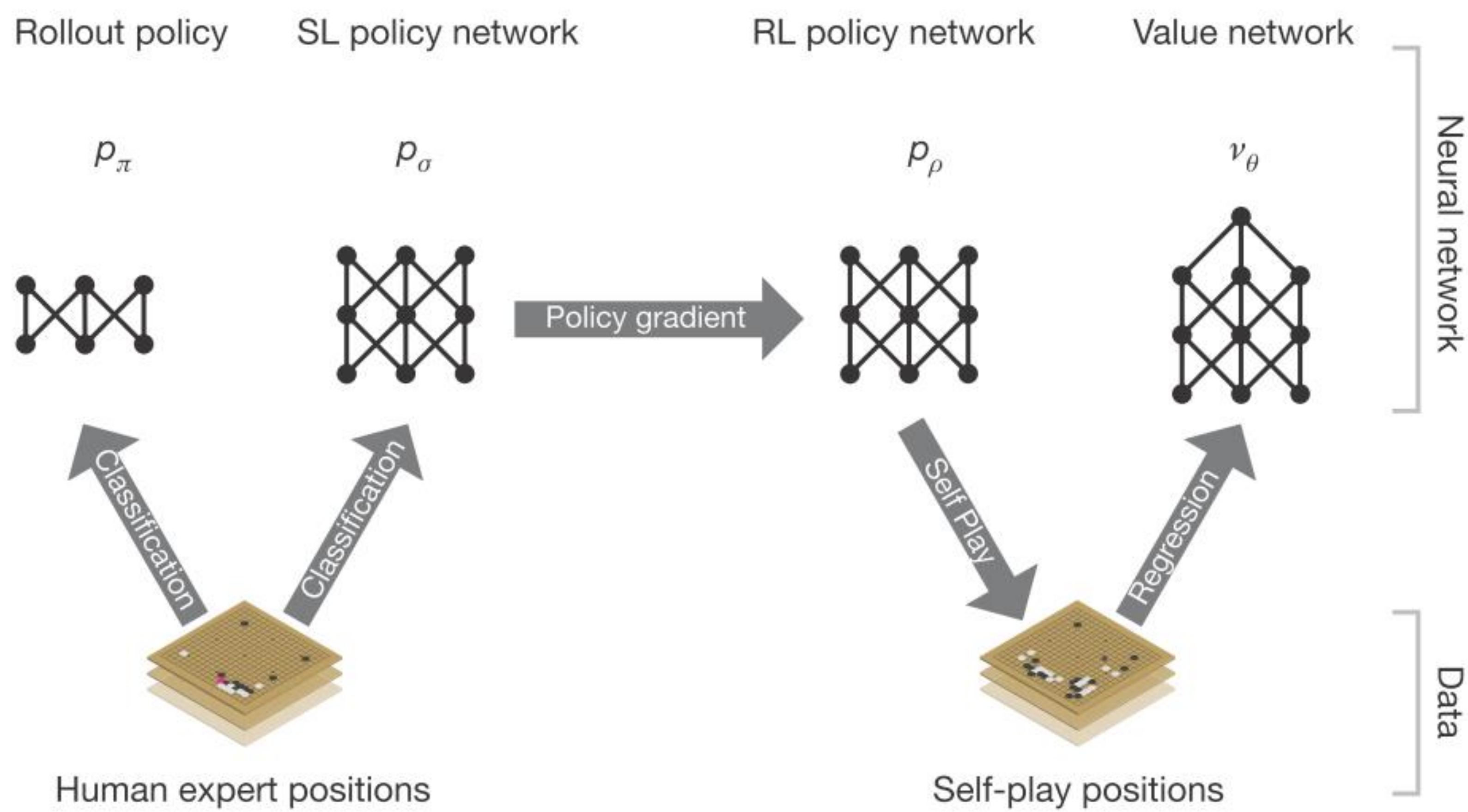
AlphaGo's Algorithm

- Use DCNN to learn experts' moves
  - (學習高手的著手策略)
- Use Monte-Carlo Tree Search (MCTS) for search to avoid pitfalls (避開陷阱)
  - MCTS is a kind of RL that do planning.
- Use DCNN to train "reinforcement learning (RL) network"
- Use DCNN to train "value network" (價值網路)
  - Learn the values of game positions (學習盤勢之優劣)

AlphaGo's Algorithm

- Use DCNN to learn experts' moves → DL
  - (學習高手的著手策略)
- Use Monte-Carlo Tree Search (MCTS) for search to avoid pitfalls (避開陷阱) → RL
  - MCTS is a kind of RL that do planning.
- Use DCNN to train "reinforcement learning (RL) network" → DRL (Policy Gradient)
- Use DCNN to train "value network" (價值網路)
  - Learn the values of game positions (學習盤勢之優劣) →DL

Policy Network and Value Network

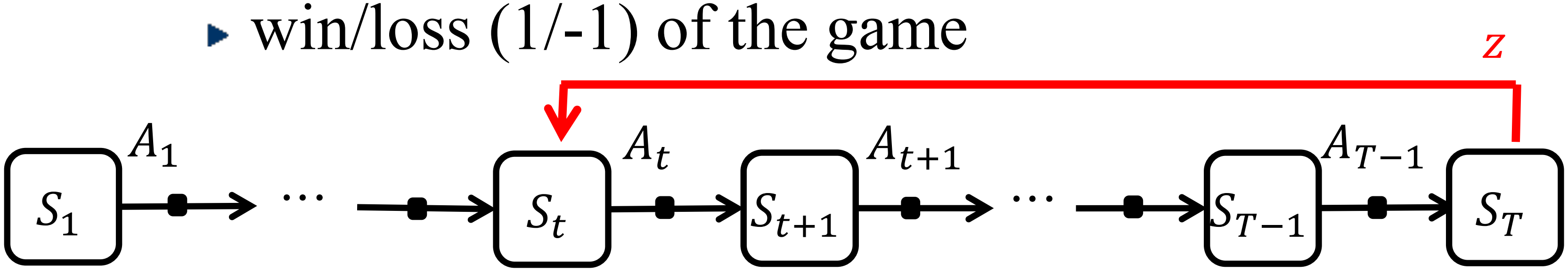


RL Policy Network: AlphaGo

- Use stochastic policy gradient ascent to maximize the likelihood of the human move a selected in state s

$$\Delta\theta = \alpha\nabla_\theta \log \pi_\theta(s_t, a_t) \cdot z$$

  - θ: network parameter.
  - α: learning rate
  - z: the value of the episode
    - win/loss (1/-1) of the game



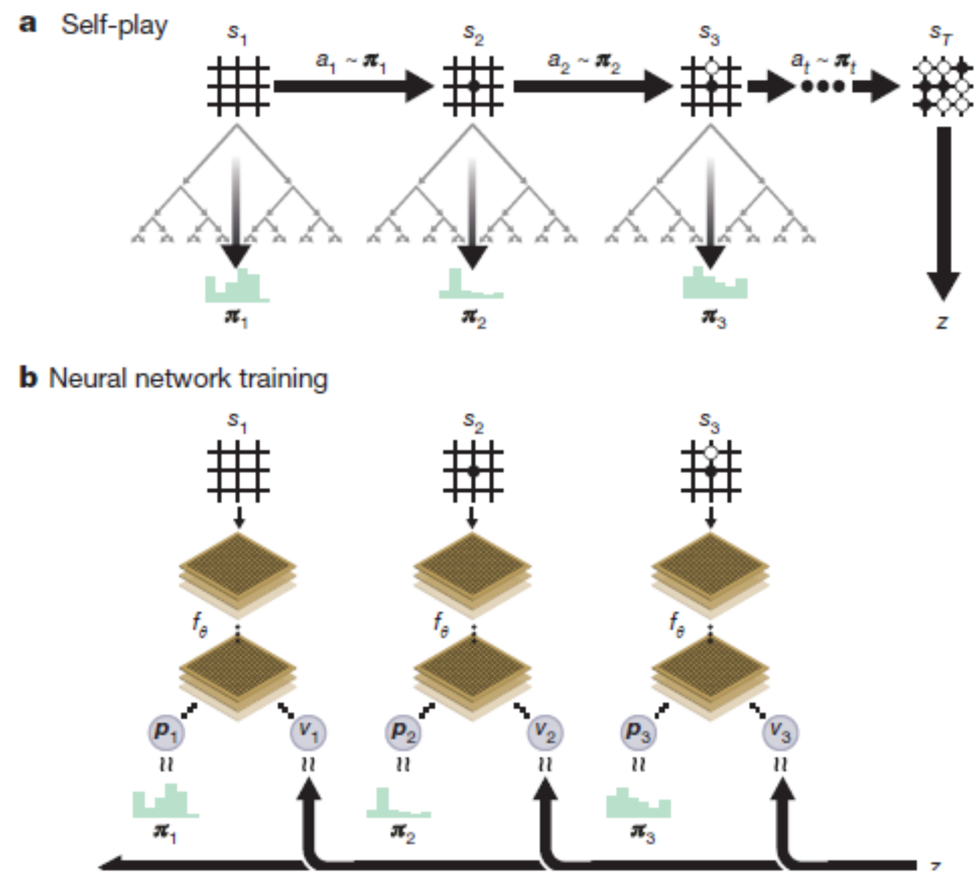
AlphaGo Zero

- Use Monte-Carlo Tree Search (MCTS) → RL
  - Learn to find the best move (avoid pitfalls)
- Combine “value/policy network” → DRL

Like a tutor

Learn from Zero Knowledge!!!

Like a student



Reinforcement Learning for Lightweight Model

- Applications
  - 2048 (Temporal Difference Learning)
  - Go Programs (with Monte-Carlo Tree Search)
- Fundamentals of Reinforcement Learning
  - Markov Decision Process (MDP)
  - Dynamic Programming (Tabular RL)

Outline

- Introduction
- Markov Property
- Markov Process
- Markov Reward Process (MRP)
- Markov Decision Process (MDP)
- Partially Observable Markov Decision Process (POMDP)

The purpose of this chapter:

- Introduce all kinds of Markov processes

Introduction

- Markov decision processes formally describe an environment for reinforcement learning
  – where the environment is fully observable.
  – i.e. The current state completely characterizes the process
  – E.g., 2048.
- Almost all RL problems can be formalized as MDPs, e.g.
  – Optimal control primarily deals with continuous MDPs
  – Partially observable problems can be converted into MDPs
  – Bandits are MDPs with one state

Markov Property

- Markov Property:
  - "The future is independent of the past given the present"
  - Definition: A state S_t is Markov if and only if
    $$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, \ldots, S_t]$$
- Comments:
  - The state captures all relevant information from the history
  - Once the state is known, the history may be thrown away
  - i.e. The state is a sufficient statistic of the future
- But, what if the history does matter?
  - Simply let S_t carry all information of history, $H_t = (S_1, \ldots, S_{t-1})$.
    - E.g., the castling rule for chess.
  - Then, it satisfies Markov Property.

Markov Process

- A Markov process is a memoryless random process,
  - i.e. a sequence of random states S_1, S_2, ... with the Markov property.

Definition:

- A Markov Process (or Markov Chain) is a tuple $<\mathcal{S}, \mathcal{P}>$
  - $\mathcal{S}$ is a (finite) set of states
  - $\mathcal{P}$ is a state transition probability matrix (part of the environment), $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s]$

State Transition Matrix

- For a Markov state s and successor state s', the state transition probability is defined by

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$$

- State transition matrix $\mathcal{P}$: (assume n states)

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \cdots & & \cdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix}$$

  - Each row of matrix sums to 1.
- Stationary distribution:
  - Let π be the stationary distribution of states.
  - Then, $\pi\mathcal{P} = \pi$.
  - Use eigenvectors to derive it. (But not the scope of this course)

Markov Reward Process (MRP)

- A Markov reward process is a Markov chain with values.

Definition:

- A Markov Reward Process is a tuple $<\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma>$
  - $\mathcal{S}$ is a (finite) set of states
  - $\mathcal{P}$ is a state transition probability matrix (part of the environment),
    $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$
  - $\mathcal{R}$ is a reward function,
    $\mathcal{R}_s = \mathbb{E}[R_{t+1} \mid S_t = s]$
  - γ is a discount factor $\gamma \in [0, 1]$.
-

Return

Definition

- The return G_t is the total discounted reward from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Notes:

- The discount $\gamma \in [0, 1]$ is the present value of future rewards
- The value of receiving reward R is diminishing
  - $\gamma^k R$, after $k + 1$ time-steps.
- This values immediate reward above delayed reward.
- Discount:
  - γ close to 0 leads to ”myopic” evaluation
  - γ close to 1 leads to ”far-sighted” evaluation

Value Function

- The value function $v(s)$ gives the long-term value of s
- Definition
  - The state value function $v(s)$ of an MRP is the expected return starting from state s
  - $v(s) = \mathbb{E}[G_t \mid S_t = s]$

Bellman Equation for MRPs

- The value function can be decomposed into two parts:
  - immediate reward R_{t+1}
  - discounted value of successor state $\gamma\, v(S_{t+1})$
- $$\begin{aligned} v(s) &= \mathbb{E}[G_t \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \cdots) \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma\, G_{t+1} \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma\, v(S_{t+1}) \mid S_t = s] \end{aligned}$$
- For a transition (s, r, s'), we have

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}\, v(s')$$

Bellman Equation in Matrix Form

- The Bellman equation can be expressed concisely using matrices, (closed form)

$$v = \mathcal{R} + \gamma \mathcal{P} v$$

  - where v is a column vector with one entry per state.

$$\begin{bmatrix} v(1) \\ \dots \\ v(n) \end{bmatrix} = \begin{bmatrix} R_1 \\ \dots \\ R_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \dots & & \dots \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \dots \\ v(n) \end{bmatrix}$$

Solving the Bellman Equation

- The Bellman equation is a linear equation
- It can be solved directly:

$$v = \mathcal{R} + \gamma \mathcal{P} v$$
$$v = (1 - \gamma \mathcal{P})^{-1} \mathcal{R}$$

- Computational complexity is $O(n^3)$ for n states
- Direct solution only possible for small MRPs
- There are many iterative methods for large MRPs, e.g.
  - Dynamic programming
  - Monte-Carlo evaluation
  - Temporal-Difference learning

Markov Decision Processes (MDP)

- A (Finite) Markov Decision Process is a tuple $<\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma>$
  - $\mathcal{S}$ is a (finite) set of states
  - $\mathcal{A}$ is a (finite) set of actions
  - $\mathcal{P}$ is a state transition probability matrix (part of the environment), $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$
    - Let $\mathcal{P}^a$ denote the matrix $\mathcal{P}^a_{::}$.
  - $\mathcal{R}$ is a reward function, $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$
  - γ is a discount factor $\gamma \in [0, 1]$.

Example: Recycling Robot

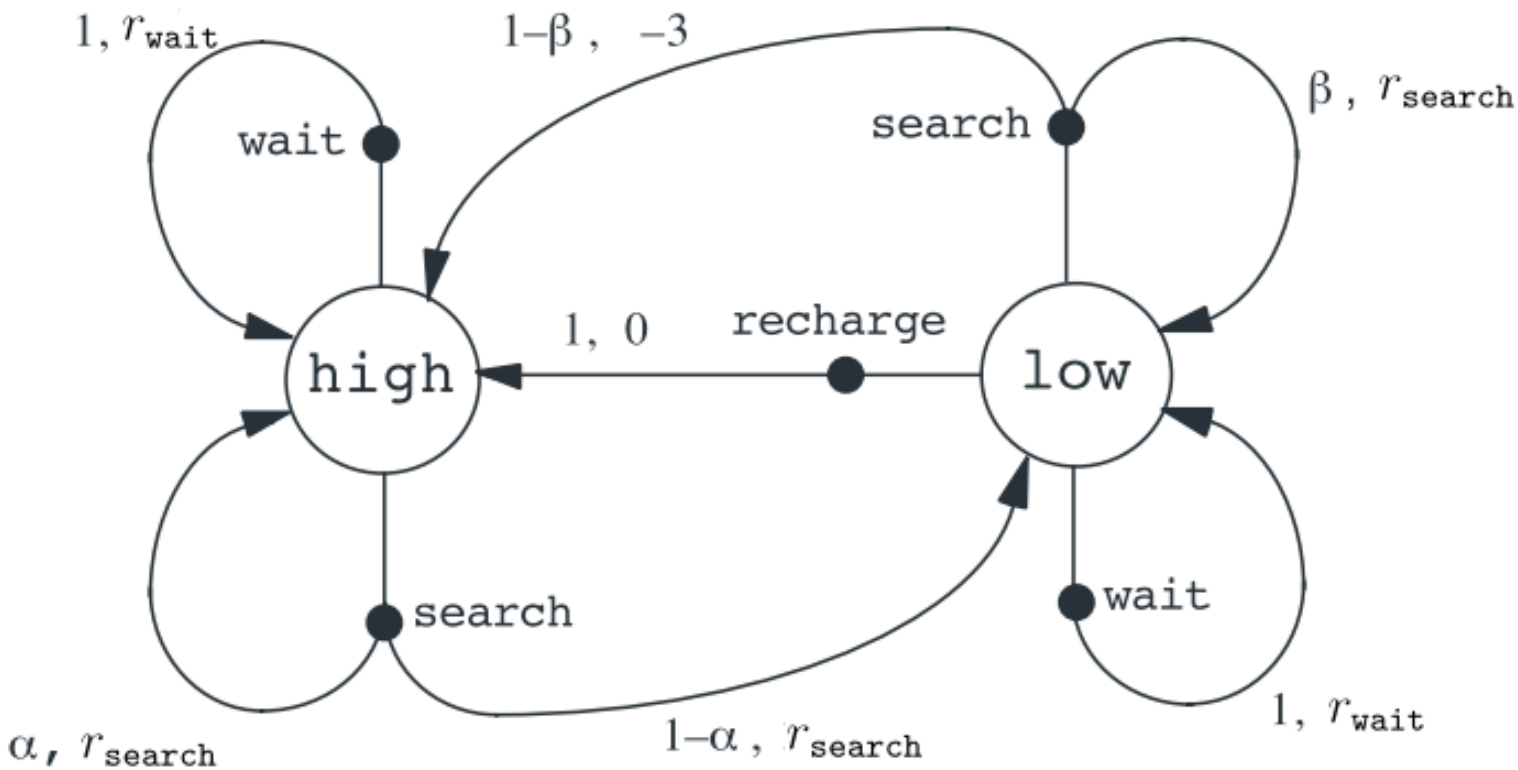


Figure 3.3: Transition graph for the recycling robot example.

Example: Recycling Robot

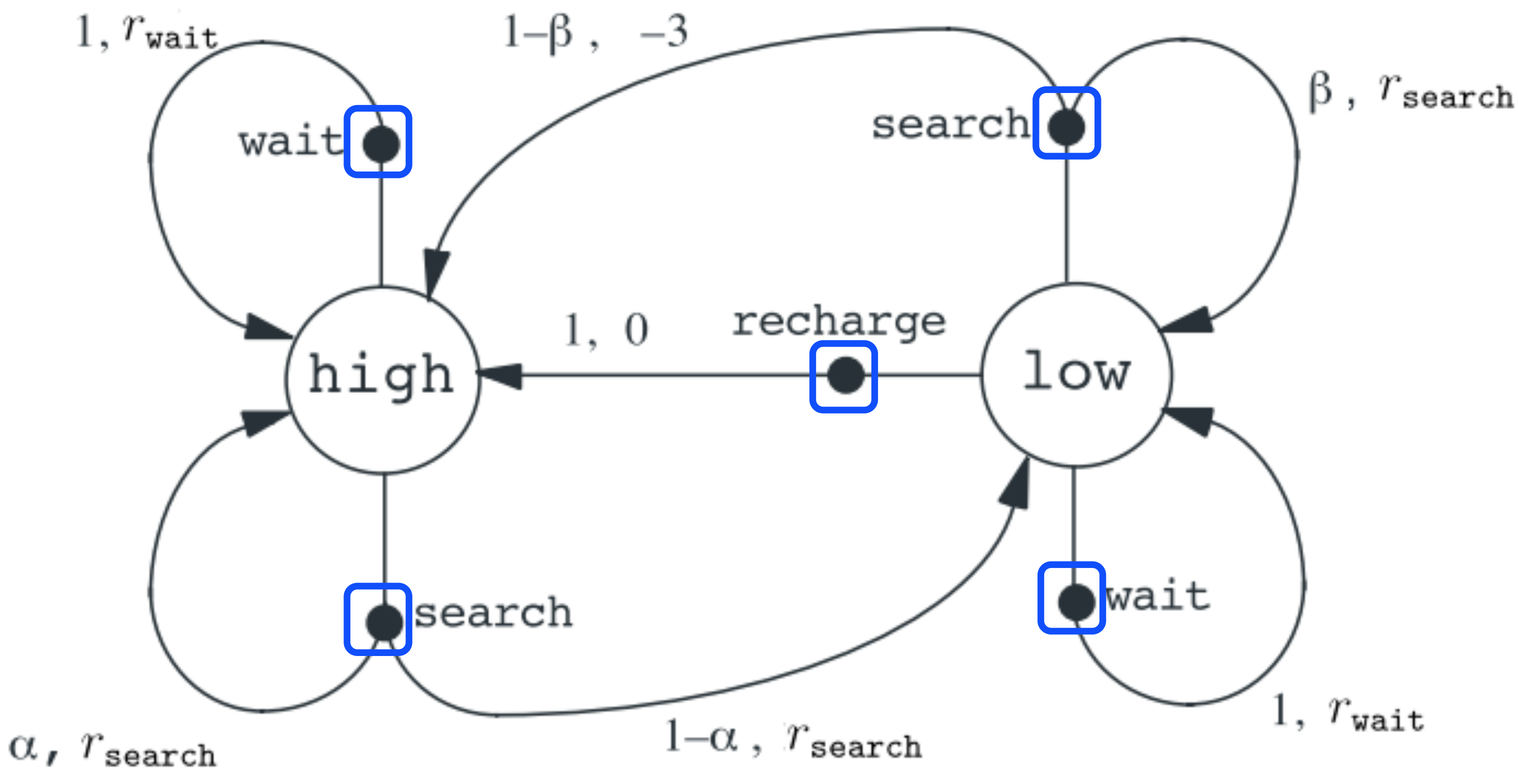


Figure 3.3: Transition graph for the recycling robot example.

Example: Recycling Robot

- Transition and Rewards:

s	s'	a	$p(s' \mid s, a)$	$r(s, a, s')$
high	high	search	α	$r_{\texttt{search}}$
high	low	search	$1-\alpha$	$r_{\texttt{search}}$
low	high	search	$1-\beta$	-3
low	low	search	β	$r_{\texttt{search}}$
high	high	wait	1	$r_{\texttt{wait}}$
high	low	wait	0	$r_{\texttt{wait}}$
low	high	wait	0	$r_{\texttt{wait}}$
low	low	wait	1	$r_{\texttt{wait}}$
low	high	recharge	1	0
low	low	recharge	0	0.

Policies

- A policy is the agent's behavior
  - It is a map from state to action
  - A policy fully defines the behavior of an agent
  - MDP policies depend on the current state (not the history)
    - i.e. Policies are stationary (time-independent),
      $A_t \sim \pi(\cdot\,|S_t), \forall t > 0$
- Policy types:
  - Deterministic policy: $a = \pi(s_i)$
  - Stochastic policy: $\pi(a|s) = \mathbb{P}[A_t = a|\ S_t = s]$
    - Sometimes, written in $\pi(s, a)$.
    - Note: for deterministic policy,
      - if $a = \pi(s_i)$, $\pi(a|s) = 1$. otherwise, $\pi(a|s) = 0$.
- Examples:
  - In 2048: Up/down/left/right
  - In robotics: angle/force/…

Policy and MRP

- Given an MDP $<\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma>$ and a policy π
- The state sequence $S_1, S_2, \ldots$ is a Markov process $<\mathcal{S}, \mathcal{P}^{\pi}>$
- The state and reward sequence $S_1, R_2, S_2, R_3, \ldots$ becomes a Markov reward process (MRP) $<\mathcal{S}, \mathcal{P}^{\pi}, \mathcal{R}^{\pi}, \gamma>$
  - $\mathcal{P}^{\pi}$ is a state transition probability matrix (part of the environment),

$$\mathcal{P}^{\pi}_{ss'} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}^{a}_{ss'}$$

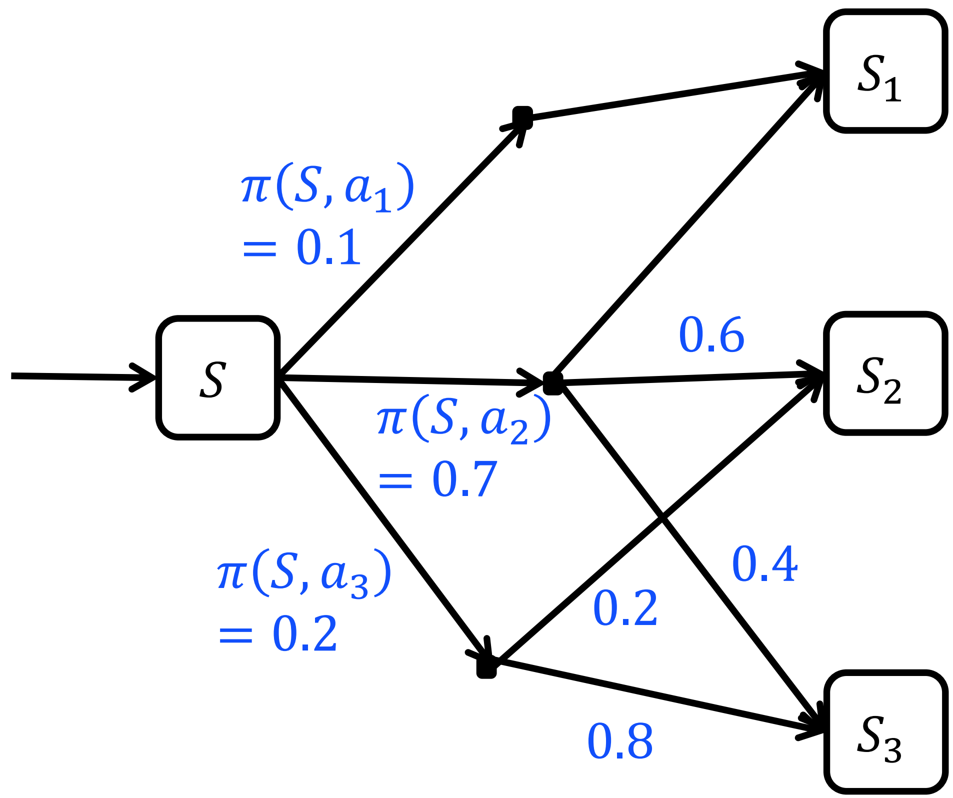
  - $\mathcal{R}^{\pi}$ is a reward function,

$$\mathcal{R}^{\pi}_{s} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}^{a}_{s}$$

- So, the property of MRP can be applied.

Example

- We have $\mathcal{P}_{ss_3}^{\pi} = 0.7 * 0.4 + 0.2 * 0.8 = 0.44$



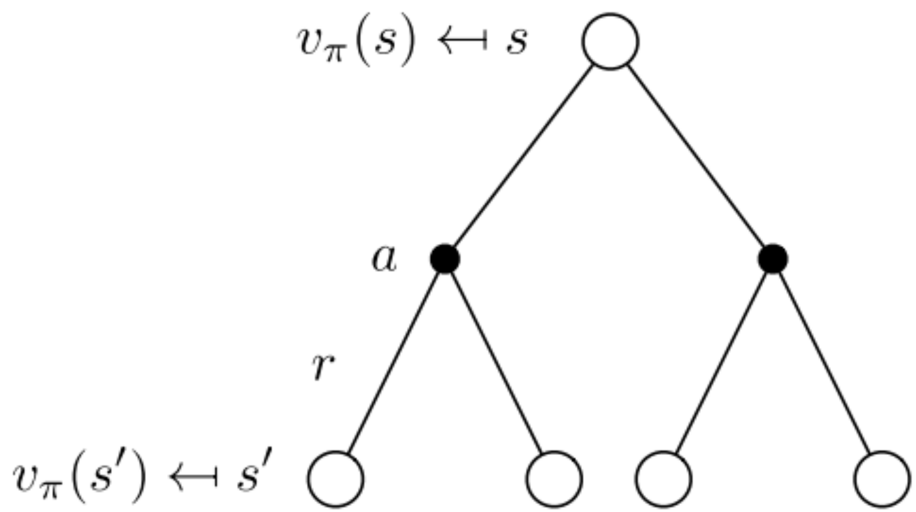
Value Function

- A value function is a prediction of future reward
  - Used to evaluate the goodness/badness of states
    - therefore to select between actions.
  - Return $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$
- Types of value functions under policy π:
  - State value function: the expected return from s.
    $$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s] \\ &= \mathbb{E}_\pi[G_t \mid S_t = s] \end{aligned}$$
  - Q-Value function: the expected return from s taking action a.
    $$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$
- Examples:
  - In 2048, the expected score from a board S_t.

Bellman Expectation Equation for π

- State value function:

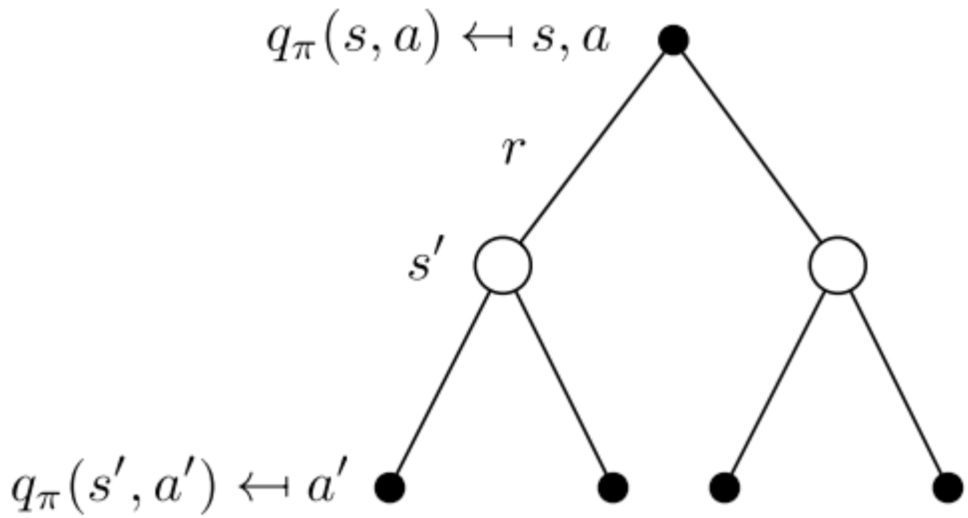
$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$



Bellman Expectation Equation for π

- Q value

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$$



Bellman Expectation Equation in Matrix

- The Bellman expectation equation can be expressed concisely using the induced MRP.
- So, it can be solved directly:

$$v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi$$
$$v_\pi = (1 - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi$$

Optimal Value Function

- The optimal state-value function $v_*(s)$ is the maximum value function over all policies
$$v_*(s) = \max_\pi v_\pi(s)$$
- The optimal action-value function $q_*(s,a)$ is the maximum action-value function over all policies
$$q_*(s,a) = \max_\pi q_\pi(s,a)$$
- Notes:
  - The optimal value function specifies the best possible performance in the MDP.
  - An MDP is "solved" when we know the optimal value function.

Optimal Policy

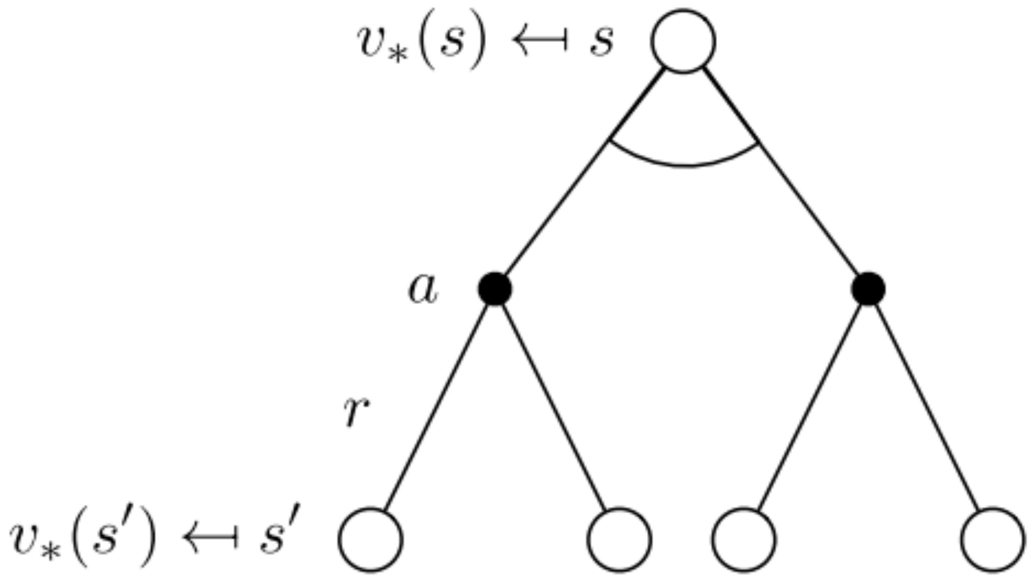
- Define a partial ordering over policies
  $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s), \forall s$

- Theorem: For any Markov Decision Process,
  - There exists an optimal policy π_* that is better than or equal to all other policies, $\pi_* \geq \pi, \forall \pi$.
  - All optimal policies achieve the optimal value function,
    $$v_{\pi_*}(s) = v_*(s)$$
  - All optimal policies achieve the optimal action-value function,
    $$q_{\pi_*}(s, a) = q_*(s, a)$$

Finding an Optimal Policy

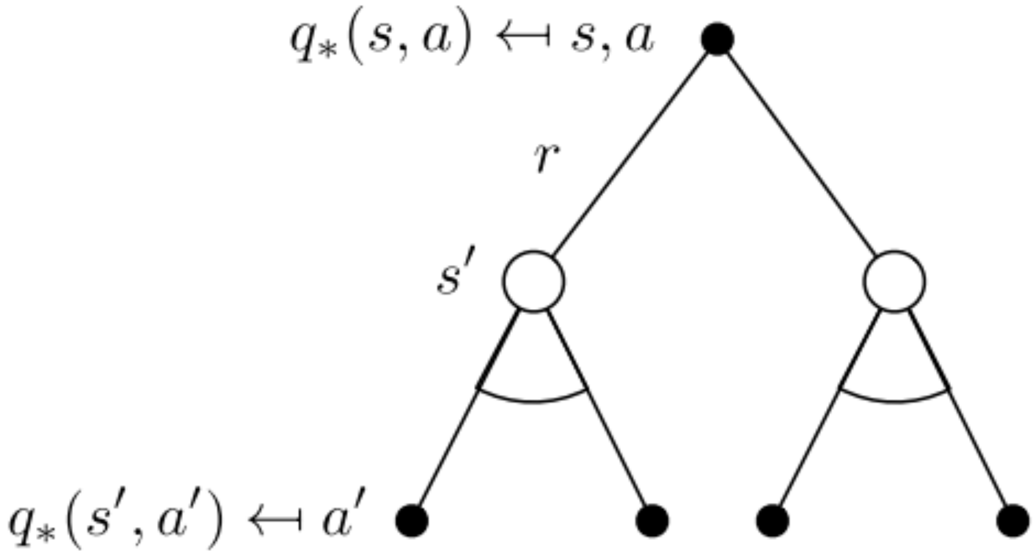
- An optimal policy can be found by maximizing over $q_*(s,a)$,
  - $\pi(a|s) = 1$, if $a = \underset{a\in\mathcal{A}}{\operatorname{argmax}}\, q_*(s,a)$
  - $\pi(a|s) = 0$, otherwise.
- There is always a deterministic optimal policy for any MDP
- If we know $q_*(s,a)$, we immediately have the optimal policy
- What about state value function $v_*(s)$?
  - Similar, but we need to know model, $\mathcal{P}_{ss'}^{a}$. → not model free.

Bellman Optimality Equation for V*



$$v_*(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_*(s') \right)$$

Bellman Optimality Equation for Q*



$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a \max_{a' \in \mathcal{A}} q_\pi(s,a')$$

Solving the Bellman Optimality Equation

- Bellman Optimality Equation is non-linear
- No closed form solution (in general)
- Many iterative solution methods
  - Value Iteration
  - Policy Iteration
  - Q-learning
  - Sarsa

Extensions to MDPs

- Infinite and continuous MDPs
  - Countably infinite state and/or action spaces
    - Straightforward
  - Continuous state and/or action spaces
    - Closed form for linear quadratic model (LQR)
  - Continuous time
    - Requires partial differential equations
    - Hamilton-Jacobi-Bellman (HJB) equation
    - Limiting case of Bellman equation as time-step
- Partially observable MDPs
  - E.g., Mahjong (as we mentioned)
- Undiscounted, average reward MDPs (ignored)

Prediction vs. Control

- For prediction: evaluate values
  - Input: MDP <$\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$, γ> and policy π
    or: MRP <$\mathcal{S}$, $\mathcal{P}^\pi$, $\mathcal{R}^\pi$, γ>
  - Output: value function v_π or q_π
- For control: find the optimal policy.
  - Input: MDP <$\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$, γ>
  - Output: optimal value function v_* or q_*
    and: optimal policy, π_*

	state values	action values
prediction	v_π	q_π
control	v_*	q_*

Reinforcement Learning for Lightweight Model

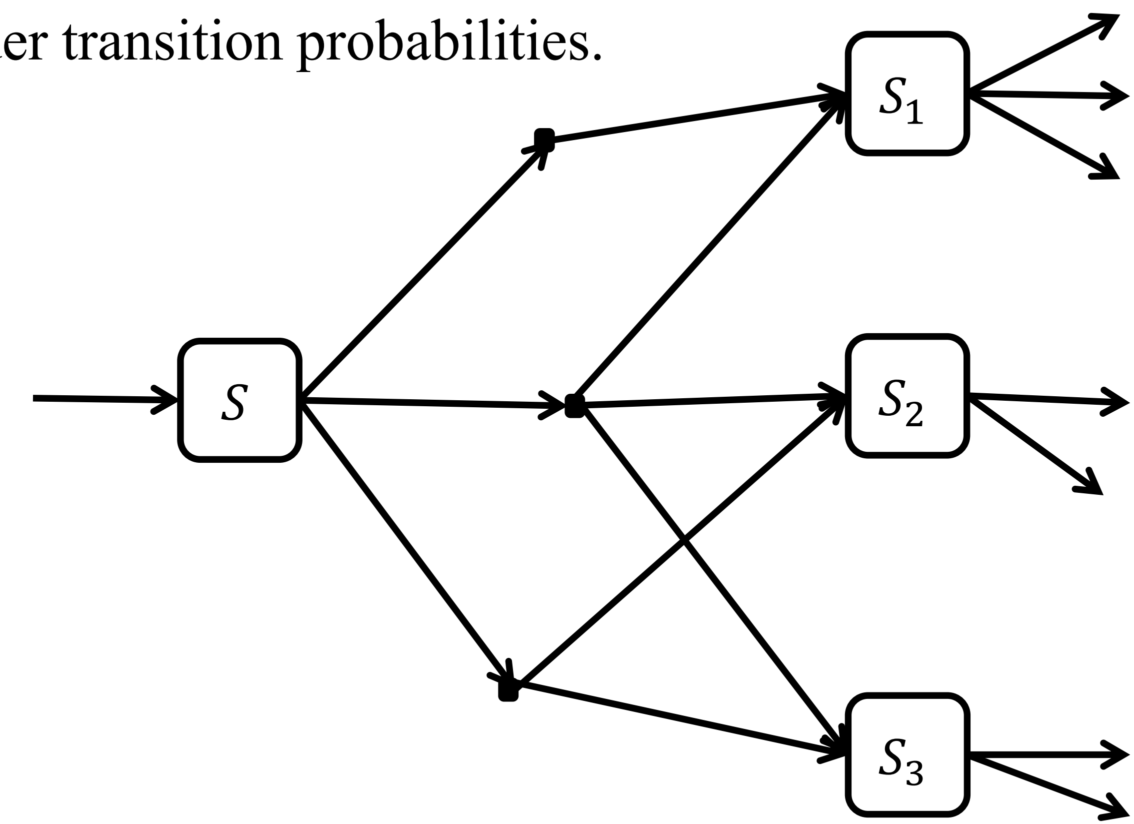
- Applications
  - 2048 (Temporal Difference Learning)
  - Go Programs (with Monte-Carlo Tree Search)
- Fundamentals of Reinforcement Learning
  - Markov Decision Process (MDP)
  - Dynamic Programming (Tabular RL)

Dynamic Programming (Chapter 3)

- (Sutton) The term dynamic programming (DP) refers to a collection of algorithms that
  - compute optimal policies given a perfect model of the environment as a Markov decision process (MDP).
- (Silver) A method for solving complex problems by breaking them down into subproblems
  - Solve the subproblems,
  - Combine solutions to subproblems
- (Algorithm textbook by Cormen et al.) says
  - DP, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
  - DP is typically applied to optimization problems.
  - Applications:
    - String algorithms (e.g. sequence alignment)
    - Graph algorithms (e.g. shortest path algorithms)
    - Bioinformatics (e.g. lattice models)

Example

- By dynamic programming, we don't have to repeat calculate the state values, such as S_1, S_2, S_3.
- In most algorithms given in Algorithms course
  - Rarely consider transition probabilities.



Why is DP related?

- Sequential or temporal component to the problem optimizing
  - a “program”, i.e. a policy,
  - values, i.e., state values and state action values
- Like solving LCS (longest common sequence) problem.
  - The optimal actions.
  - The optimal values.
  - $\mathcal{P}$ and π are deterministic.
  - Exercise: shortest path problem.

j		0	1	2	3	4	5	6
i		y_j	*B*	*D*	*C*	*A*	*B*	*A*
0	x_i	0	0	0	0	0	0	0
1	*A*	0	↑0	↑0	↑0	↖1	←1	↖1
2	*B*	0	↖1	←1	←1	↑1	↖2	←2
3	*C*	0	↑1	↑1	↖2	←2	↑2	↑2
4	*B*	0	↖1	↑1	↑2	↑2	↖3	←3
5	*D*	0	↑1	↖2	↑2	↑2	↑3	↑3
6	*A*	0	↑1	↑2	↑2	↖3	↑3	↖4
7	*B*	0	↖1	↑2	↑2	↑3	↖4	↑4

Requirements for Dynamic Programming

- Dynamic Programming is a very general solution method for problems which have two properties:
  - Optimal substructure
    - Principle of optimality applies
    - Optimal solution can be decomposed into subproblems
  - Overlapping subproblems
    - Subproblems recur many times
    - Solutions can be cached and reused
- Markov decision processes satisfy both properties
  - Bellman equation gives recursive decomposition
  - Value function stores and reuses solutions

Planning by Dynamic Programming

- Dynamic programming assumes full knowledge of the MDP
  - It is used for planning in an MDP
- For prediction: evaluate values
  - Input: MDP <$\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$, γ> and policy π
    or: MRP <$\mathcal{S}$, $\mathcal{P}^{\pi}$, $\mathcal{R}^{\pi}$, γ>
  - Output: value function v_{π}
- For control: find the optimal policy.
  - Input: MDP <$\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$, γ>
  - Output: optimal value function v_*
    and: optimal policy, π_*

Three Approaches

- Policy Evaluation
  - Directly solve Bellman Equation in matrix form (see above)
    - Given an MDP $<\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma>$ and a policy π, it becomes a MRP problem $<\mathcal{S}, \mathcal{P}^{\pi}, \mathcal{R}^{\pi}, \gamma>$.
  - Use Iterative Policy Evaluation
- Policy Iteration
- Value Iteration

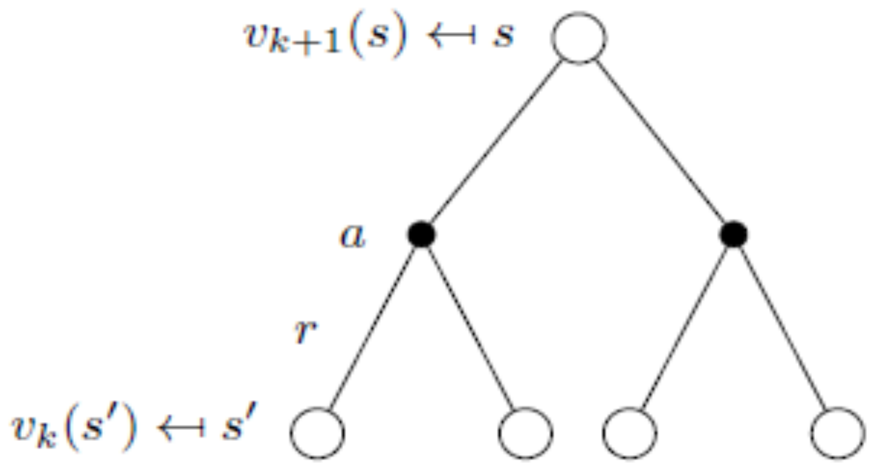
Iterative Policy Evaluation

- Problem: evaluate a given policy π
- Solution: iterative application of Bellman expectation backup

  $$v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_*$$

- Using synchronous backups,
  - At each iteration $k+1$,
    - for all states $s \in S$, update $v_{k+1}(s)$ from $v_k(s')$ where s' is a successor state of s
- Notes:
  - We will discuss asynchronous backups later
  - Convergence to v_π will be proven at the end of the lecture
  - Review the Bellman-Ford algorithm for the shortest path problem.

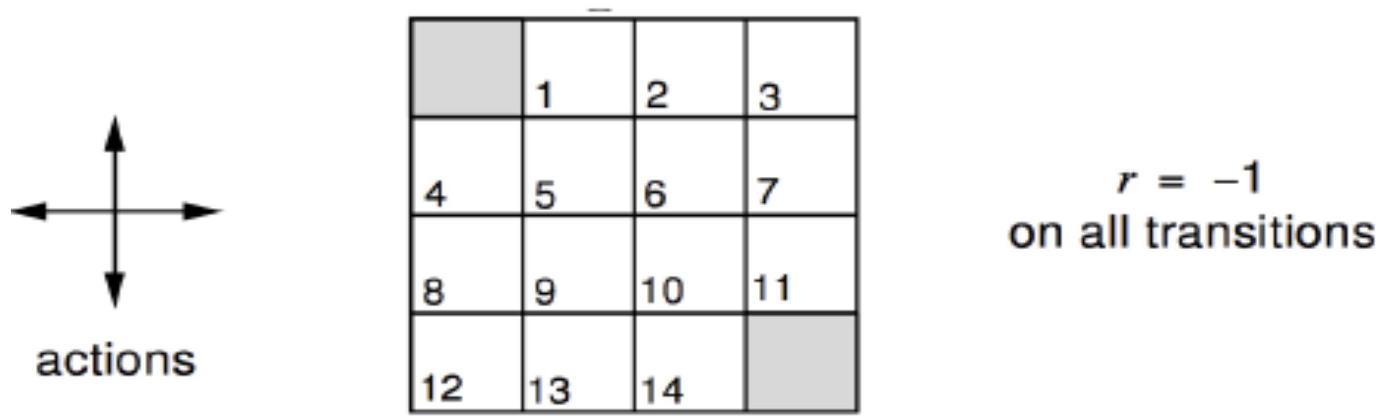
Iterative Policy Evaluation



$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\boldsymbol{v}^{k+1} = \boldsymbol{\mathcal{R}}^{\pi} + \gamma \boldsymbol{\mathcal{P}}^{\pi} \boldsymbol{v}^{k}$$

Example: Evaluating a Random Policy in the Small Gridworld



- States:
  - Nonterminal states 1, …, 14
  - One terminal state (shown twice as shaded squares)
- Actions
  - Four directional moves
  - leading out of the grid leave state unchanged
- Reward
  - -1 until the terminal state is reached
- Undiscounted: episodic MDP ($\gamma = 1$)
- Agent follows uniform random policy

$$\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 0.25$$

Iterative Policy Evaluation in Small Gridworld (I)

v_k for the Random Policy | Greedy Policy w.r.t. v_k

$k = 0$

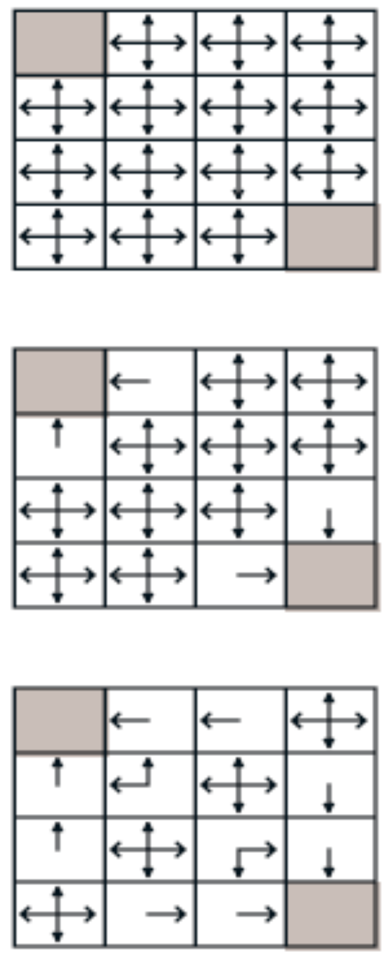
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0



random policy

Iterative Policy Evaluation in Small Gridworld (2)

$k = 3$

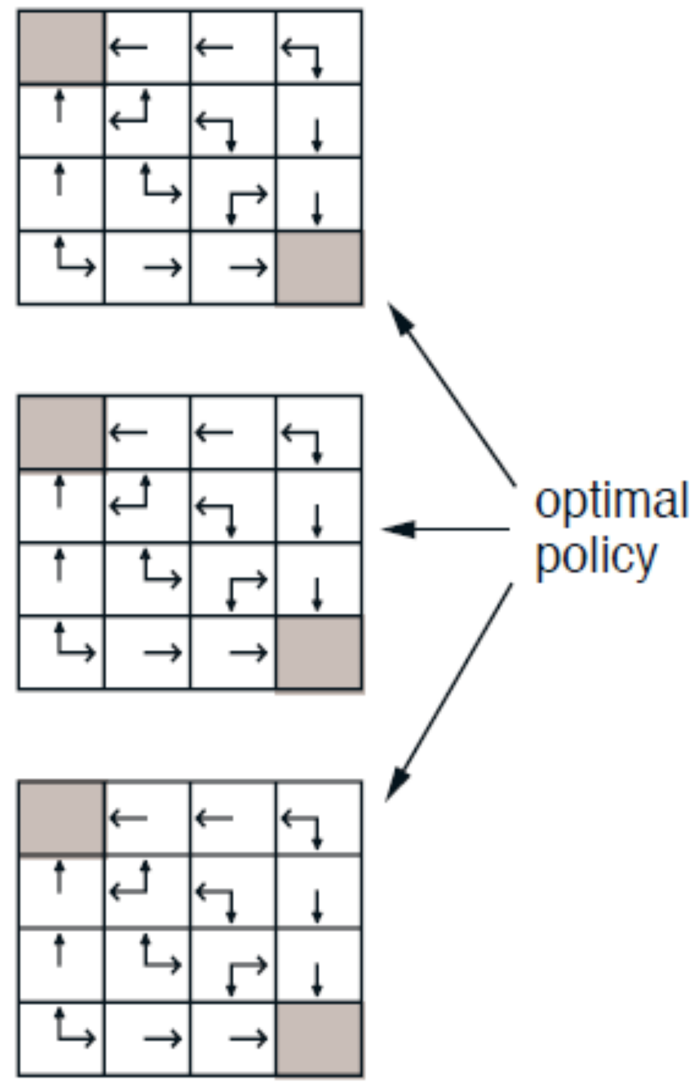
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

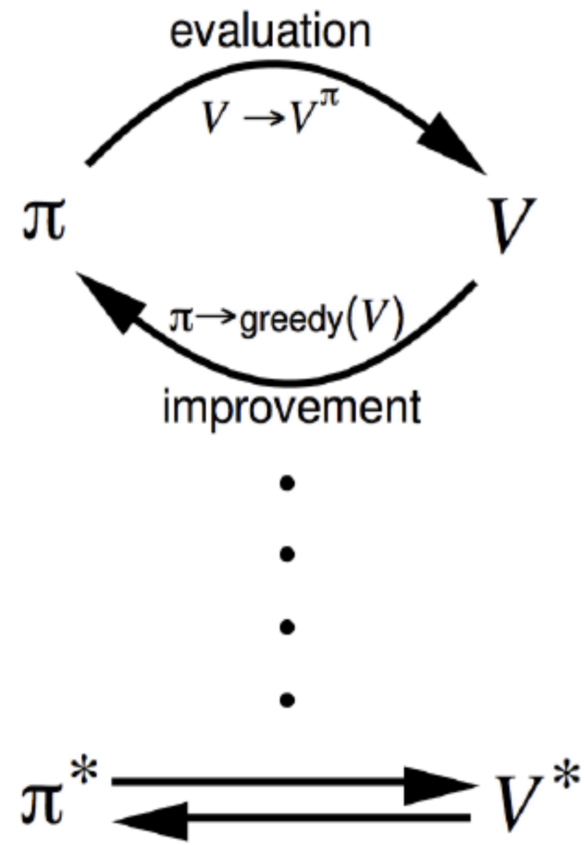


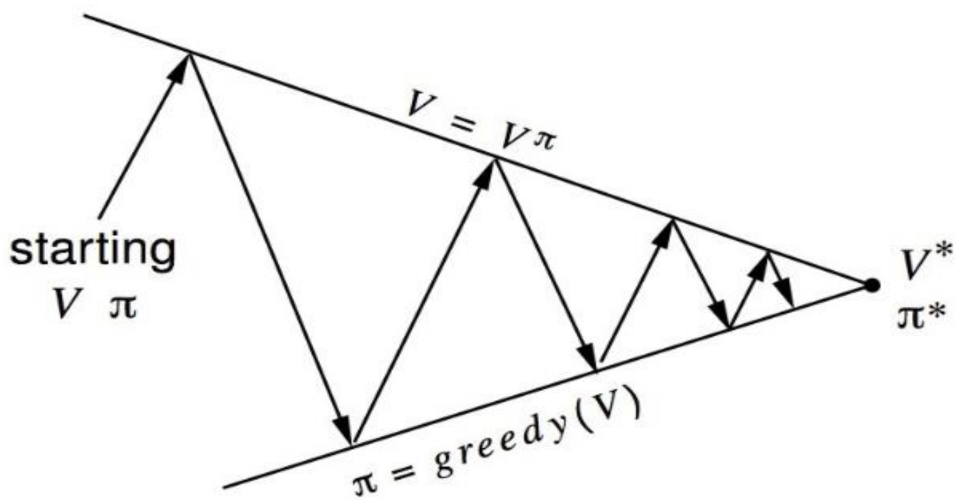
How to Improve a Policy

- Definition of policy improvement
  - Let π and π' be any pair of deterministic policies
    - For all $s \in S$, "$\pi(s)$ performs better than $\pi'(s)$". (We will see example)
- Given a policy π
  - Evaluate the policy π
$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \cdots | S_t = s]$$
  - Improve the policy by acting greedily with respect to v_π
$$\pi' = \text{greedy}(v_\pi)$$
- Notes:
  - In Small Gridworld improved policy was optimal, $\pi' = \pi^*$
  - In general, need more iterations of improvement / evaluation
  - But this process of policy iteration always converges to π^*

Policy Iteration

- Policy evaluation → Estimate v_π
  - Iterative policy evaluation
- Policy improvement → Generate $\pi' \geq \pi$
  - Greedy policy improvement





Proof of Policy Improvement

- Consider a deterministic policy, $a = \pi(s)$
- We can improve the policy by acting greedily

$$\pi'(s) = \underset{a \in A}{\operatorname{argmax}}\, q_\pi(s, a)$$

- This improves the value from any state s over one step,

$$q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$

- It therefore improves the value function, $v_{\pi'}(s) \geq v_\pi(s)$.

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) = \mathbb{E}_{\pi'}[R_{t+1} + \gamma\, v_\pi(S_{t+1}) \mid S_t = s] \\
&\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) \mid S_t = s\right] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \cdots \mid S_t = s] = v_{\pi'}(s)
\end{aligned}$$

Converge of Policy Improvement

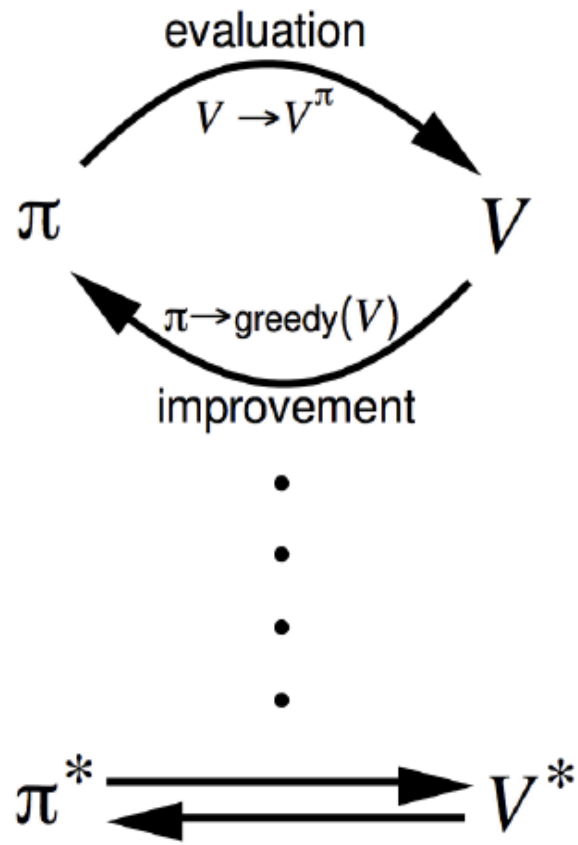
- If improvements stop,
  - That is, for $q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$
    - "≥" becomes "=" when stopping.
- Then the Bellman optimality equation has been satisfied
$$v_\pi(s) = \max_{a \in A} q_\pi(s, a)$$
- This implies $v_\pi(s) = v_*(s)$ for all $s \in S$
- The above proves that π will converge to an optimal policy.

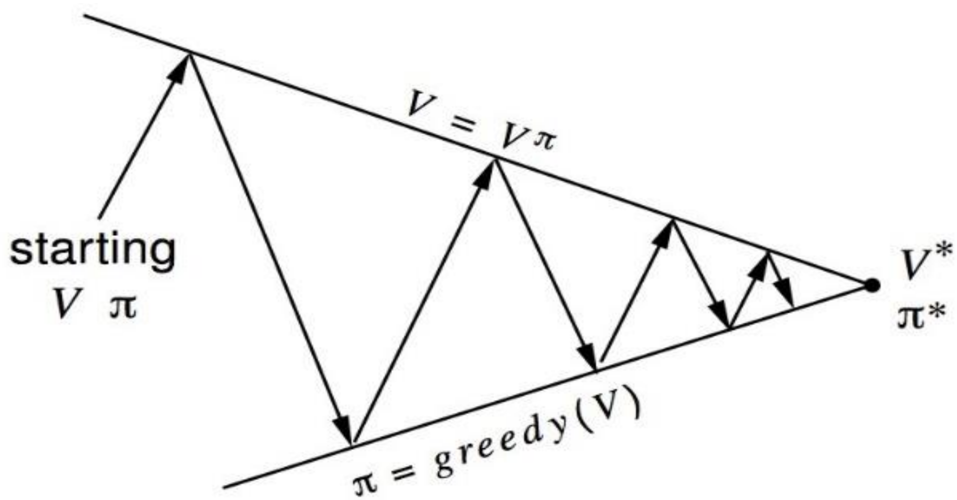
Variations of Policy Iteration

- Questions:
  - Does policy evaluation need to converge to v_π?
  - Should we introduce a stopping condition, e.g. ∈-convergence of value function?
  - Simply stop after k iterations of iterative policy evaluation?
    - For example, in the small gridworld $k = 3$ was sucient to achieve optimal policy
    - Why not update policy every iteration? i.e. stop after $k = 1$

Generalized Policy Iteration

- Policy evaluation → Estimate v_π
  - Any policy evaluation algorithm
- Policy improvement → Generate $\pi' \geq \pi$
  - Any policy improvement algorithm





Principle of Optimality

- Theorem (Principle of Optimality)
  - A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$, if and only if
  - For any state s' reachable from s, π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$

Deterministic Value Iteration

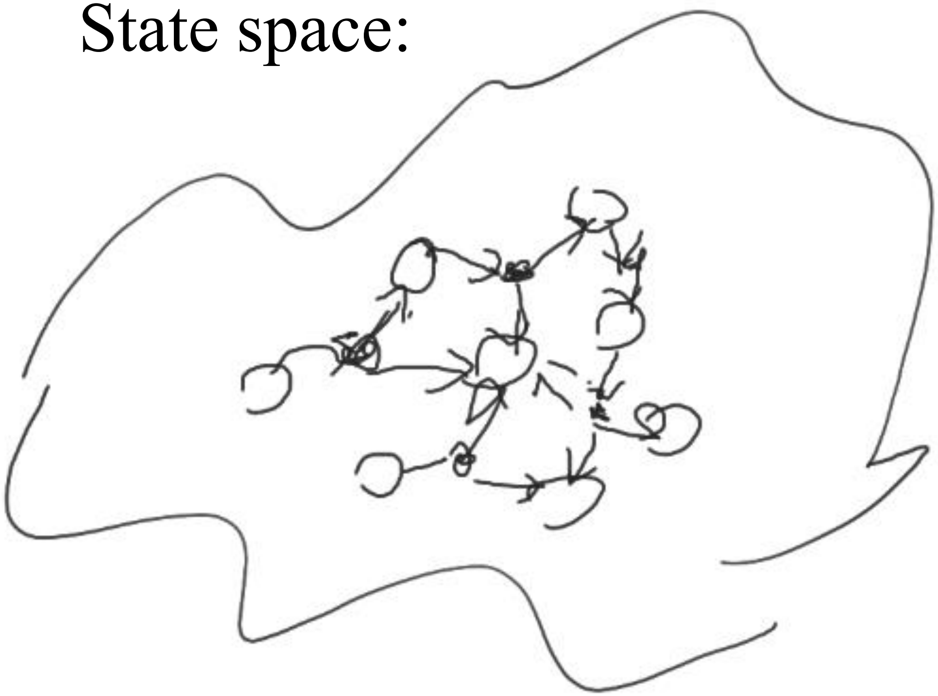
- If we know the (optimal) solution to subproblems $v_*(s')$
- Then solution $v_*(s)$ can be found by one-step lookahead

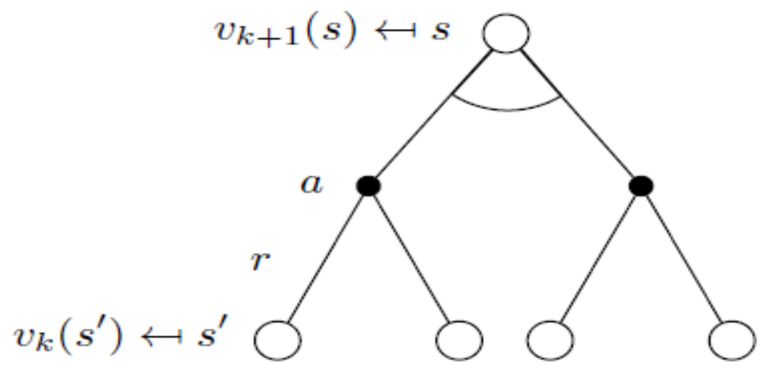
$$v_*(s) \leftarrow \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_*(s') \right)$$

- Intuition:
  - Start with final rewards and work backwards
  - apply these updates iteratively
- Notes:
  - Still works with loopy, stochastic MDPs
  - Like most DP problems. (e.g., shortest path problem)

Bellman Optimality

State space:





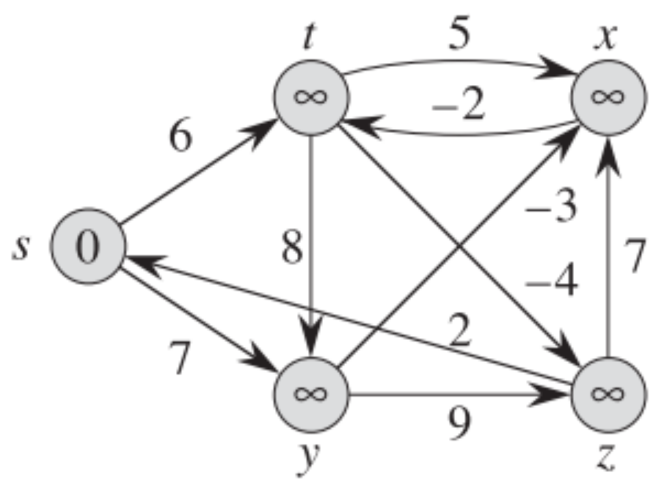
$$v_{n+1}(s) = \max_{a \in A}\left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a\, v_n(s')\right)$$

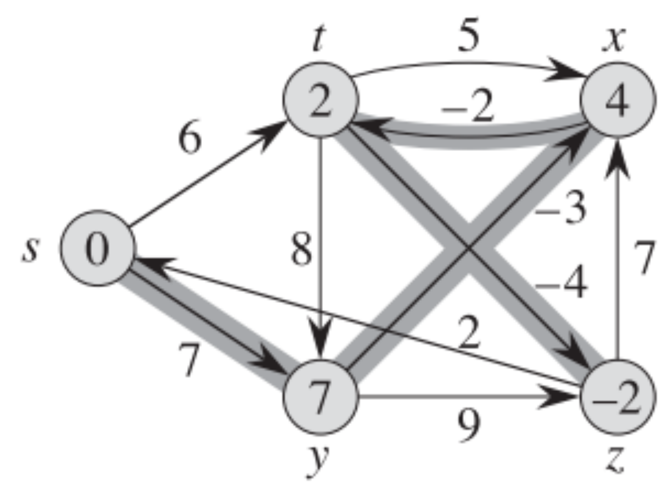
or:

$$V^{(n+1)}(s) = \max_{a \in \mathcal{A}}\left(\mathbb{E}_{s'|s,a}\left[r + \gamma V^{(n)}(s')\right]\right)$$

The Shortest Path Problem

- A very simple MDP problem with
  - deterministic state transition $\mathcal{P}$.
    (Just consider the case without state-action or black dots)
- A good example to get a quick idea about why it works.
  (see Cormen's Algorithm textbook)





Algorithms for the Shortest Path Problem

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

- Bellman-Ford Algorithm:
  - Simple, but it works.
    - All are based on Relexation
    - Complexity for all pairs: $O(n^2 e)$, n: vertex count, e: edge count.
- Dijkstra Algorithm:
  - Faster, but complex and no negative values
    - Complexity for all pairs: $O(ne + n^2 \log n)$
- Note:
  - The concept of Value Iterative is based on Bellman-Ford.

Value Iteration

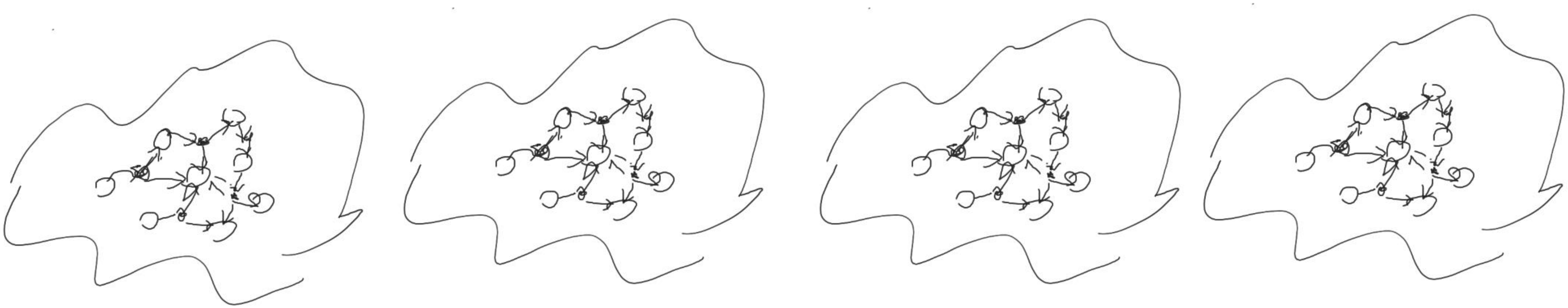
- Problem:
  - find optimal policy π
- Solution: directly find the optimal v_* without π.
  - iterative application of Bellman optimality backup

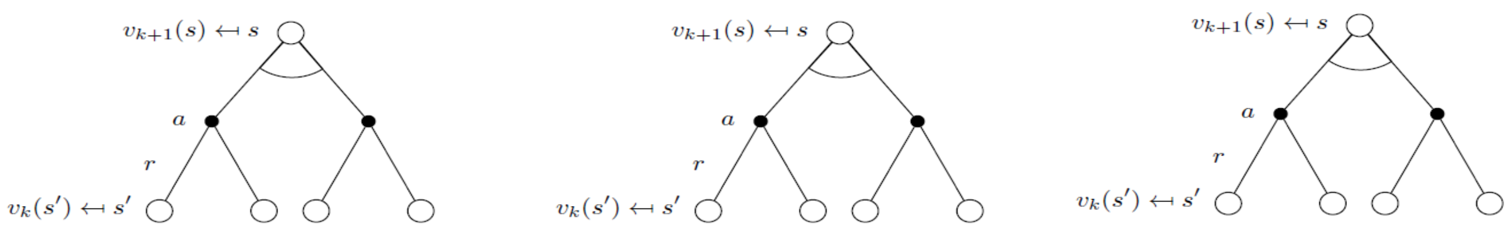
$$v_1 \to v_2 \to \dots \to v_*$$

- Using synchronous backups (like Bellman-Ford)
  - At each iteration $k+1$
    - For all states $s \in S$
      - Update $v_{k+1}(s)$ from $v_k(s')$
- Convergence to v_* will be proven later
- Unlike policy iteration, there is no explicit policy

Value Iteration

$v_1 \rightarrow$ $v_2 \rightarrow$ … $\rightarrow v_*$

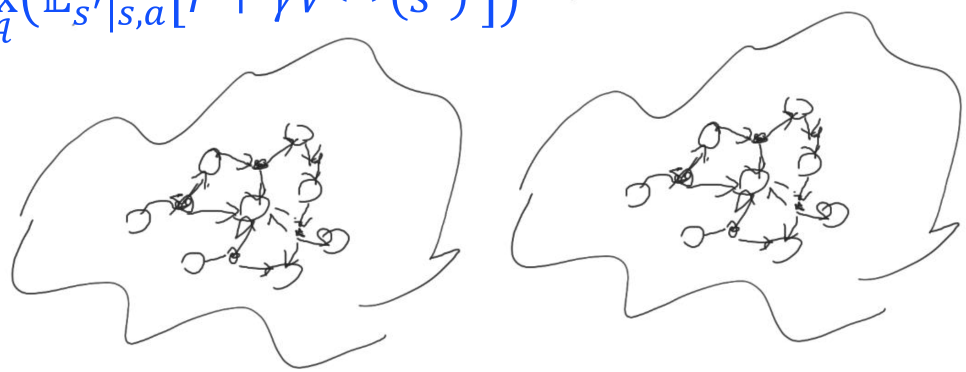




Operator View

- Value iteration update

$$V^{(n+1)}(s) = \max_{a\in\mathcal{A}}\left(\mathbb{E}_{s'|s,a}\left[r + \gamma V^{(n)}(s')\right]\right)$$



- It can be viewed as:
  - A function $\mathcal{T}: \mathcal{S} \to \mathcal{S}$.
  - Called backup operator.

$$[\mathcal{T}V](s) = \max_{a\in\mathcal{A}}\left(\mathbb{E}_{s'|s,a}\left[r + \gamma V(s')\right]\right)$$

$$V^{(n+1)} = \mathcal{T}V^{(n)}$$

(Let V be an array of $v(s)$)

Algorithm Value Iteration

Initialize $V^{(0)}$ arbitrarily.

for n = 0, 1, 2, . . . until termination condition do

$V^{(n+1)} = \mathcal{T}V^{(n)}$

end

Value Function Space

- Consider the vector space V over value functions
  - There are $|S|$ dimensions
  - Each point in this space fully species a value function $v(s)$
- What does a Bellman backup do to points in this space?
  - It brings value functions closer
  - Therefore the backups must converge on a unique solution

Value Function ∞-Norm

- We will measure distance between state-value functions u and v by the ∞-norm
  - i.e. the largest difference between state values,

$$||U - V||_\infty = \max_s |u(s) - v(s)|$$

- Let $\delta = ||(U - V)||_\infty$
  - $u(s) - v(s) \leq \delta$ for all s

Contraction for Bellman Optimality Backup

- Bellman optimality backup operator $\mathcal{T}$ is a γ-contraction.
- Proof: Since
  $$\max_{a\in\mathcal{A}}\big(x(a)\big) - \max_{a\in\mathcal{A}}\big(y(a)\big) \le \max_{a\in\mathcal{A}}(x(a) - y(a))$$
- we have $||\mathcal{T}U - \mathcal{T}V||_\infty$
  $= ||\max_{a\in\mathcal{A}}(\mathcal{R}^a + \gamma\,\mathcal{P}^a U) - \max_{a\in\mathcal{A}}(\mathcal{R}^a + \gamma\,\mathcal{P}^a V)||_\infty$
  $\le ||\max_{a\in\mathcal{A}}[(\mathcal{R}^a + \gamma\,\mathcal{P}^a U) - (\mathcal{R}^a + \gamma\,\mathcal{P}^a V)]||_\infty$
  $= ||\max_{a\in\mathcal{A}}[\gamma\,\mathcal{P}^a(U - V)]||_\infty = \gamma||\max_{a\in\mathcal{A}}[\,\mathcal{P}^a(U - V)]||_\infty$
  $\le \gamma\delta = \gamma||(U - V)||_\infty$
  - Note: $(\mathcal{P}^a_{s,:}(U - V)) \le \delta$ for all s
    $\rightarrow ||\mathcal{P}^a(U - V)||_\infty \le \delta$
    - For $\mathcal{P}^a$, each row of matrix sums to 1.

Contraction Mapping Theorem

- Backup operator $\mathcal{T}$ is a γ-contraction with modulus $\gamma(< 1)$ under ∞-norm
$$||\mathcal{T}U - \mathcal{T}V||_\infty \leq \gamma\ ||U - V||_\infty$$
- By contraction-mapping principle, it has a fixed point V^*
  - by iterating
$$V, \mathcal{T}V, \mathcal{T}^2V, \ldots \rightarrow V^*$$
- Proof:
$$||\mathcal{T}V - \mathcal{T}V^*||_\infty \leq \gamma\ ||V - V^*||_\infty$$
  - Since $\mathcal{T}V^* = V^*$,
$$||\mathcal{T}V - V^*||_\infty \leq \gamma\ ||V - V^*||_\infty$$
  - By recurrence,
$||\mathcal{T}^nV - V^*||_\infty \leq \gamma\ ||\mathcal{T}^{n-1}V - V^*||_\infty \leq \cdots \leq \gamma^n\ ||V - V^*||_\infty$
  - Since $\gamma^n \rightarrow 0$, $||\mathcal{T}^nV - V^*||_\infty \rightarrow 0$.
  - That is, $\mathcal{T}^nV \rightarrow V^*$

Policy Evaluation

- Problem: how to evaluate fixed policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t \mid S_t = s] = \mathbb{E}_{\pi}[R_{t+1} + \gamma V^{\pi}(S_{t+1}) \mid S_t = s]$$

- Backwards recursion involves a backup operation

$$V^{(k+1)} = \mathcal{T}^{\pi} V^{(k)}$$

  – $\mathcal{T}^{\pi}$ is defined as:

$$[\mathcal{T}^{\pi} V](s) = \mathbb{E}_{s'|s,a=\pi(s)}[r + \gamma V(s')]$$

- $\mathcal{T}^{\pi}$ is also a contraction with modulus γ, sequence

$$V, \mathcal{T}^{\pi} V, (\mathcal{T}^{\pi})^2 V, (\mathcal{T}^{\pi})^3 V, \ldots \rightarrow V^{\pi}$$

- $V = \mathcal{T}^{\pi} V$ is a linear equation that we can solve directly.

Contraction for Bellman Expectation Backup

- Bellman Expectation Backup operator $\mathcal{T}^{\pi}$ is a γ-contraction,
- Proof:

$$\left|\left|\mathcal{T}^{\pi}U - \mathcal{T}^{\pi}V\right|\right|_{\infty} = ||(\mathcal{R}^{\pi} + \gamma\,\mathcal{P}^{\pi}U) - (\mathcal{R}^{\pi} + \gamma\,\mathcal{P}^{\pi}V)||_{\infty}$$
$$= ||\gamma\,\mathcal{P}^{\pi}(U - V)||_{\infty}$$
$$\leq \gamma\delta = \gamma||(U - V)||_{\infty}$$

  - Note:
    - $(\mathcal{P}^{\pi}_{s:.}(U - V)) \leq \delta$ for all s
      $\rightarrow ||\mathcal{P}^{\pi}(U - V)||_{\infty} \leq \delta$
      - For $\mathcal{P}^{\pi}$, each row of matrix sums to 1.

Policy Iteration: Overview

- Alternate between
  - Evaluate policy $\pi \Rightarrow V^{\pi}$
  - Set new policy to be greedy policy for V^{π}

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, \mathbb{E}_{s'|s,a}[R_{t+1} + \gamma V^{\pi}(s')\,]$$

- Guaranteed to converge to optimal policy and value function in a finite number of iterations, when $\gamma < 1$
- Value function converges faster than in value iteration

Algorithm Policy Iteration

Initialize $\pi^{(0)}$ arbitrarily.

for $n = 1, 2, \ldots$ until termination condition do

$V^{(n+1)} = \text{Solve}\ [V = \mathcal{T}^{\pi^{(n-1)}} V]$

$\pi^{(n)} = \mathcal{G}V^{(n)}$ $\qquad$ $\mathcal{G}$: a greedy mapping function.

end

Modified Policy Iteration

- Update π to be the greedy policy, then value function with k backups (k-step lookahead)

Algorithm Modified Policy Iteration

Initialize $V^{(0)}$ arbitrarily.

for $n = 1, 2, \ldots$ until termination condition do

$$\pi^{(n+1)} = \mathcal{G}V^{(n)}$$

$$V^{(n+1)} = \left(\mathcal{T}^{\pi^{(n+1)}}\right)^k V^{(n)}, \text{ for integer } k \geq 1.$$

end

- $k = 1$: value iteration
- $k = \infty$: policy iteration

Exercise

- What if $\gamma = 1$?
  - – Hint: Like The Shortest Path Problem
    - The shortest path to node 0.