Search (III)

Mingsheng Long

Tsinghua University

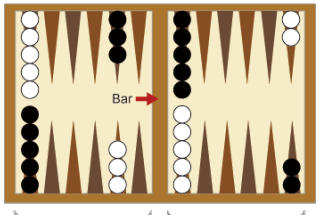
Spring 2023

Outline

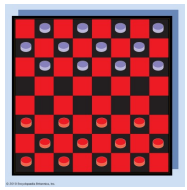
- **Games**
- Adversarial Search
  - Minimax Algorithm
  - Alpha-Beta Pruning
  - Real-Time Decisions
  - Games with Uncertain Outcomes
- Monte Carlo Tree Search

History of AI in Games

Computer masters **Tic-Tac-Toe** 1952



2015-Wikipedia: "Thus, it is very unlikely that it will be possible to program a reasonably fast algorithm for playing the Go endgame flawlessly, let alone the whole Go game."



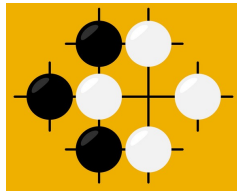
Checkers **CHINOOK** defeated the human champion 1990

Backgammon **TD-GAMMON** competitive with top human players 1995

Chess IBM's **DEEP BLUE** defeated world champion Garry **Kasparov** 1997

Othello(Reversi) **LOGISTELLO** defeated the human world champion 1997

Atari Google's **DQN** learn to play Atari games 2014



Go **AlphaGo** defeats **Fan Hui** (2-dan player) 2015

Go **AlphaGo** defeats **Lee Sedol** 4-1 (9-dan player) 2016

Birth of AI 1956

Deep Learning Era 2012

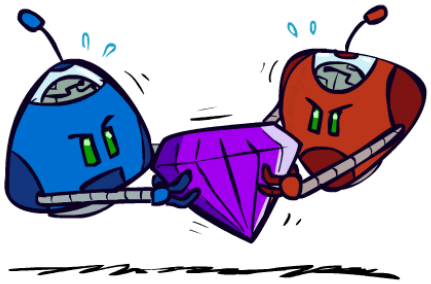
Types of Games

- Game = task environment with more than one agents
- Axes:
  - Deterministic or stochastic?
  - Perfect information
    - Fully observable?
  - Two, or more players?
  - Turn-taking or simultaneous?
  - Zero sum?

	Deterministic	Nondeterministic
Fully Observable	Chess Checkers Go	Backgammon Monopoly
Partially Observable	Kriegspiel Battleship	Card Games

- Want algorithms for calculating a contingent plan (strategy or policy) which recommends a move for every possible eventuality (state)

Zero-Sum Games



- Zero-Sum Games
  - A better term: "Constant-sum"
  - Agents have **opposite** utilities
  - Adversarial, pure competition
    - One maximizes
    - The other minimizes

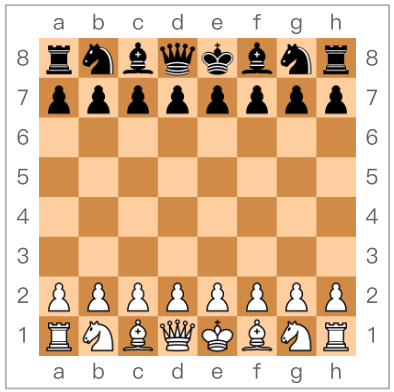
- General Games
  - Agents have **independent** utilities
  - Cooperation, indifference, competition, shifting alliances, and more are all possible

Standard Games

- Standard games are deterministic, turn-taking, two-player, zero-sum games of perfect information (such as checkers, chess, go).

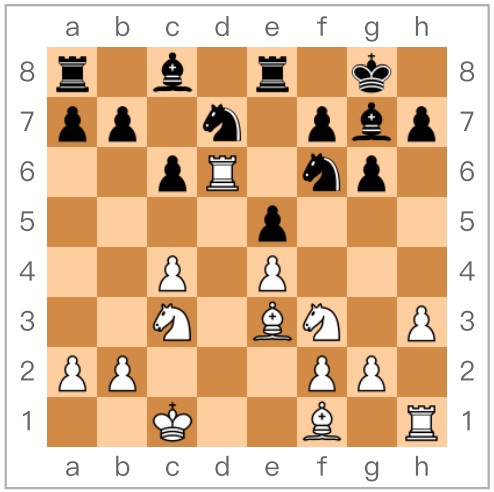
Standard Game Formulation:

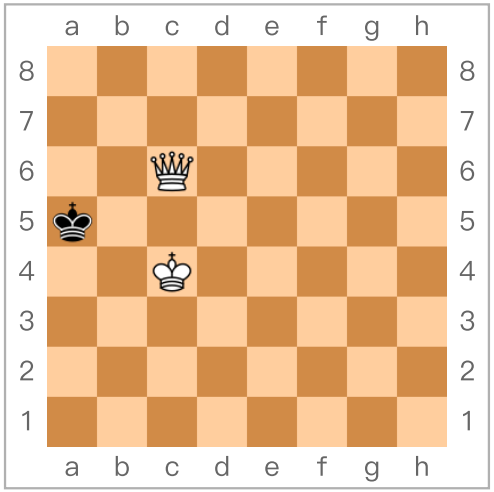
- Initial state: s_0
- Players: $\textbf{Player}(s)$ indicates whose move it is
- Actions: $\textbf{Actions}(s)$ for player on move
- Transition model: $\textbf{Result}(s, a)$
- Terminal test: $\textbf{Terminal-Test}(s)$
- Utility function: $\textbf{Utility}(s, p)$ for player p
  - Or just $\textbf{Utility}(s)$ for player making the decision at root



Example: Chess

- Define the search problem of chess:
- **State** s: the tuple (position of all pieces, whose turn it is)
- $\textbf{Player}(s) \in \{\textbf{white}, \textbf{black}\}$
- $\textbf{Actions}(s)$: legal chess moves that $\textbf{Player}(s)$ can make
- $\textbf{Terminal-Test}(s)$: whether s is checkmate or draw
- $\textbf{Utility}(s)$:
  - $+\infty$ if white wins
  - 0 if draw
  - $-\infty$ if black wins



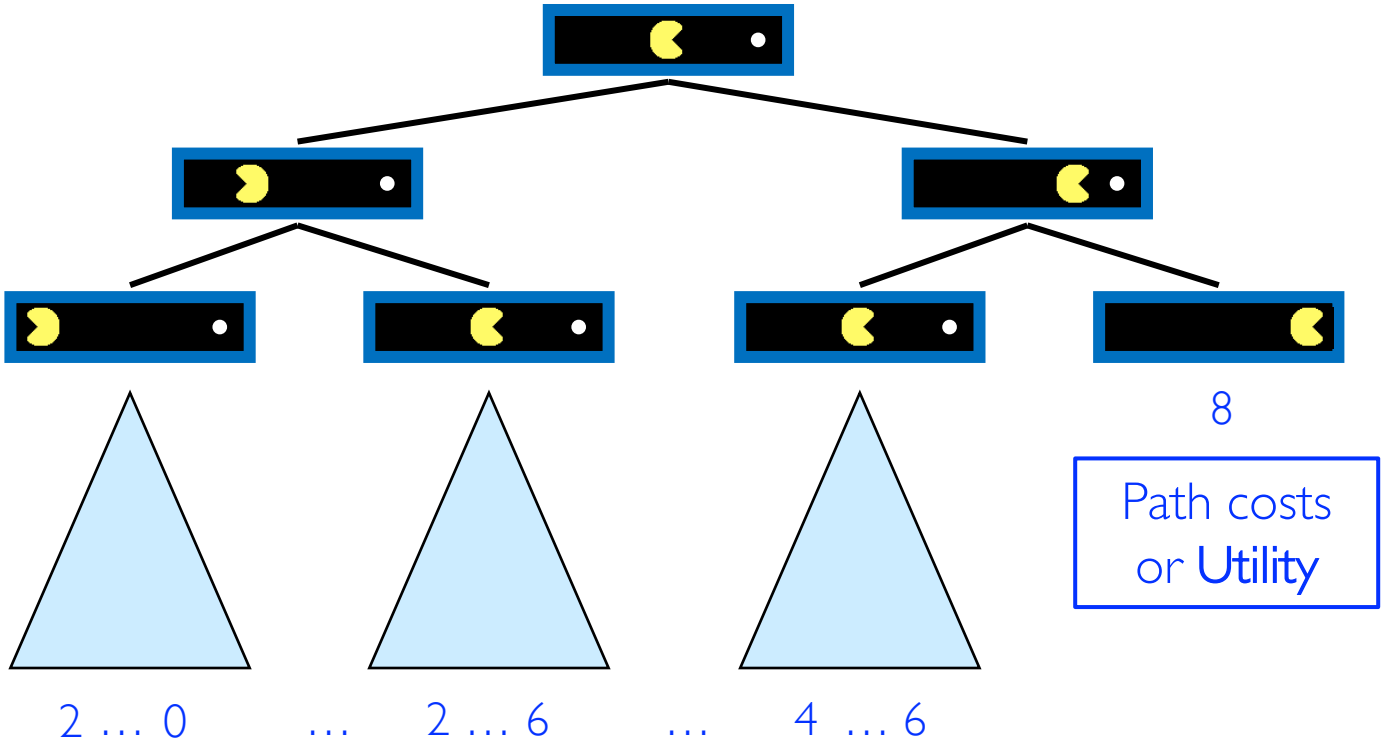


Outline

- Games
- **Adversarial Search**
  - Minimax Algorithm
  - Alpha-Beta Pruning
  - Real-Time Decisions
  - Games with Uncertain Outcomes
- Monte Carlo Tree Search

Single-Agent Trees

- Pacman: starts with 10 points and loses 1 point per move until he eats the pellet

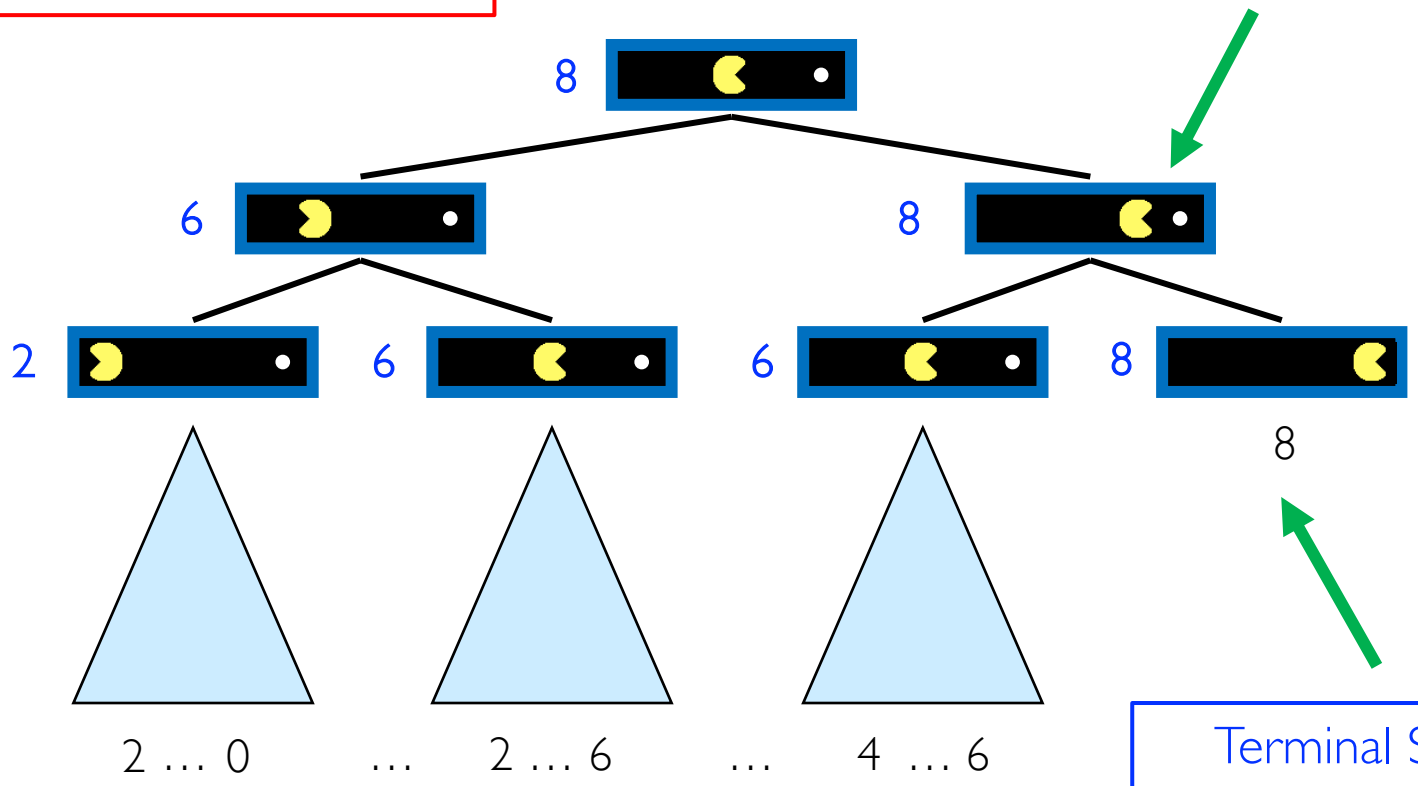


Value of State

Value of state: The best achievable outcome (utility) from that state

Non-Terminal States:

$$V(s) = \max_{a \in \text{Actions}(s)} V(\text{Result}(s, a))$$

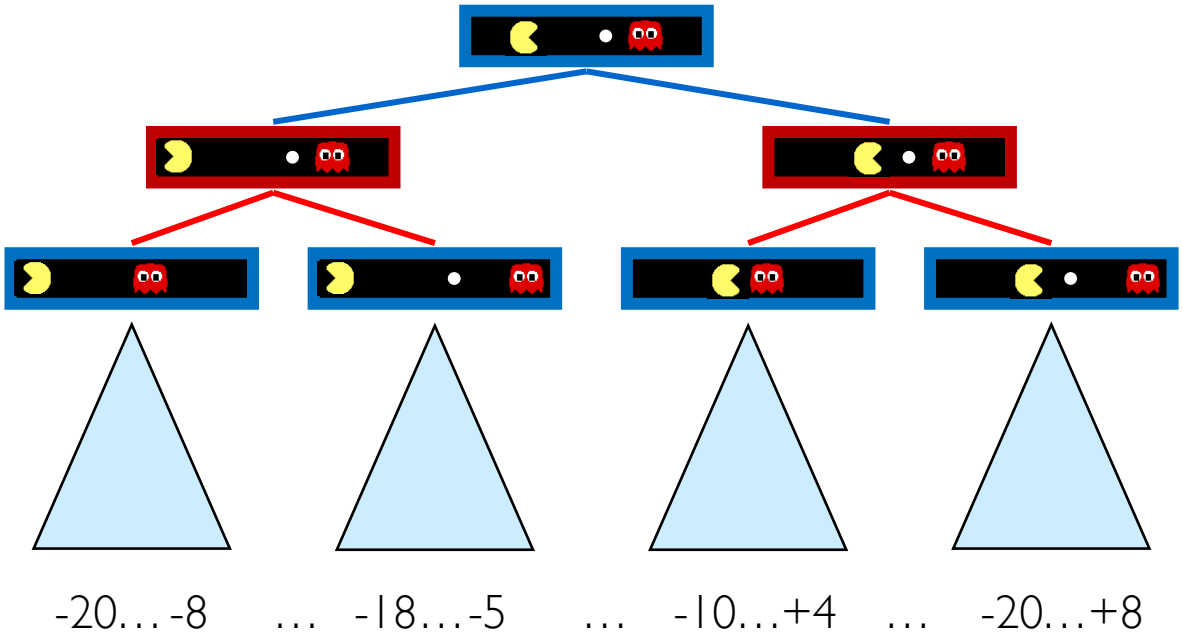


Terminal States:

$$V(s) = \text{Utility}(s)$$

Adversarial Game Trees

- An adversarial ghost that wants to keep Pacman from eating the pellet (minimizing Pacman's points)
  - Zero-sum game: The best action for one is the worst for the other
  - How do we define best and worst—Alternate max and min



Outline

- Games
- **Adversarial Search**
  - **Minimax Algorithm**
  - Alpha-Beta Pruning
  - Real-Time Decisions
  - Games with Uncertain Outcomes
- Monte Carlo Tree Search

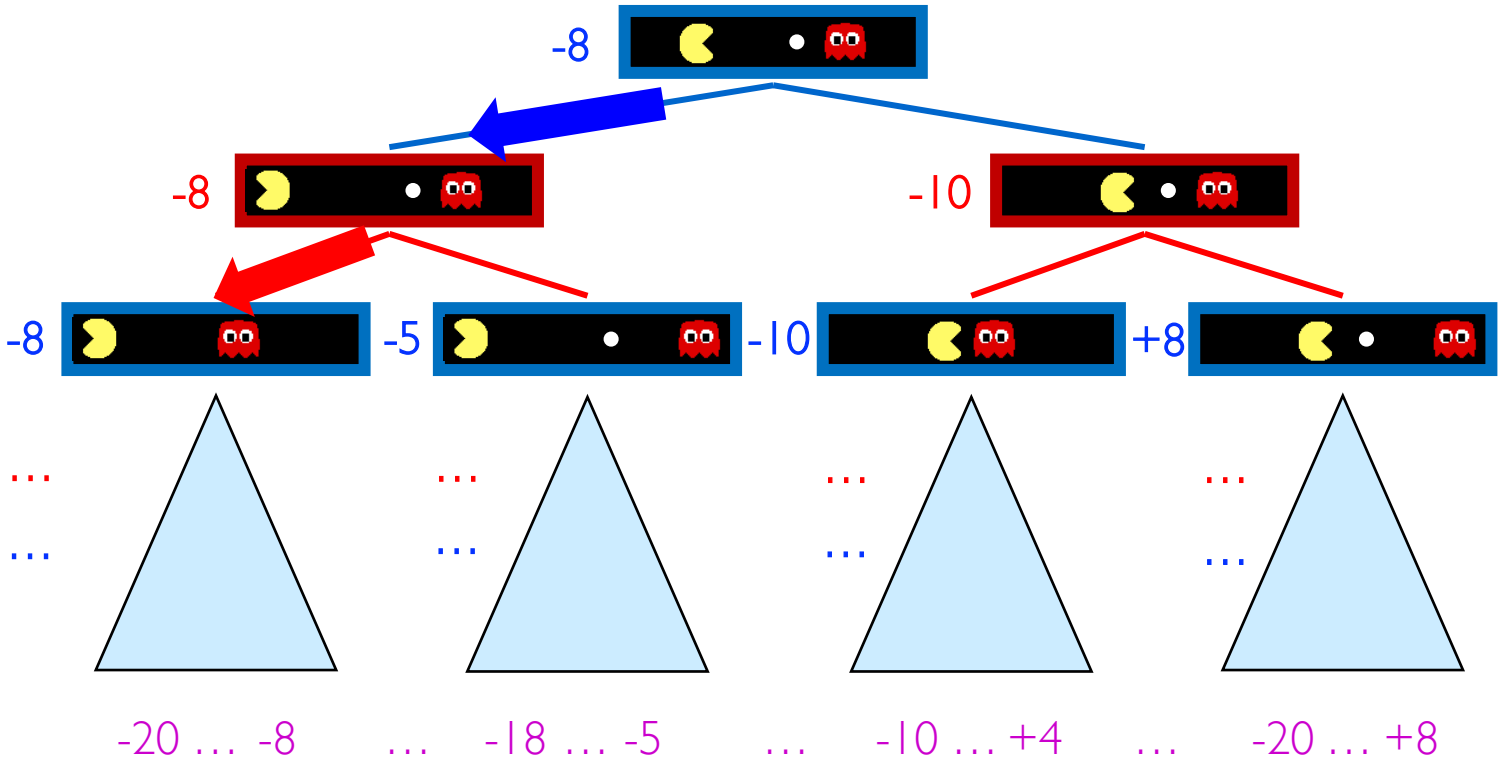
Minimax Values

MAX nodes: Under Agent's Control

$$V(s) = \max_{a \in \text{Actions}(s)} V(\text{Result}(s, a))$$

MIN nodes: Under Opponent's Control

$$V(s) = \min_{a \in \text{Actions}(s)} V(\text{Result}(s, a))$$



Terminal States: $V(s) = \text{Utility}(s)$

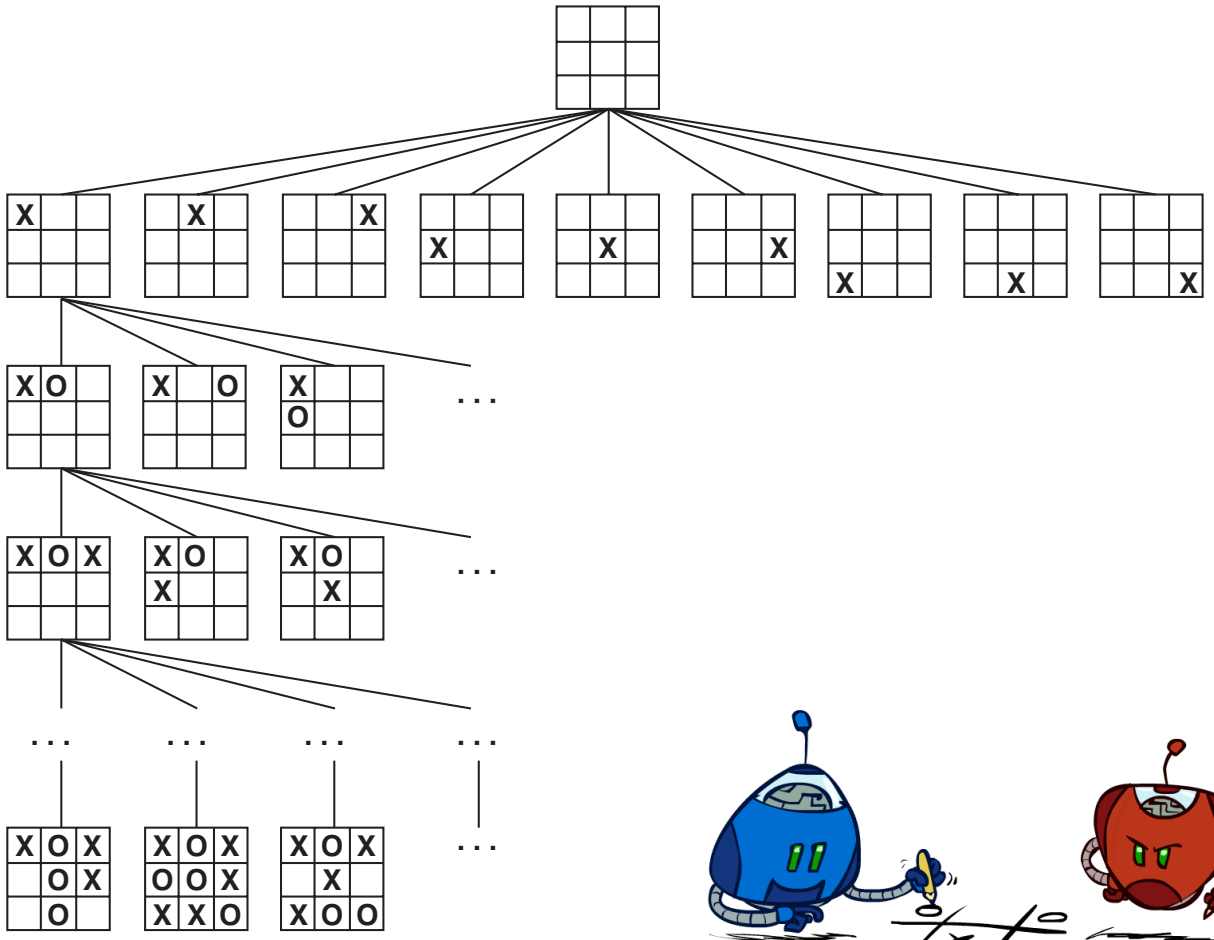
Example: Tic-Tac-Toe Game Tree

MAX(x)

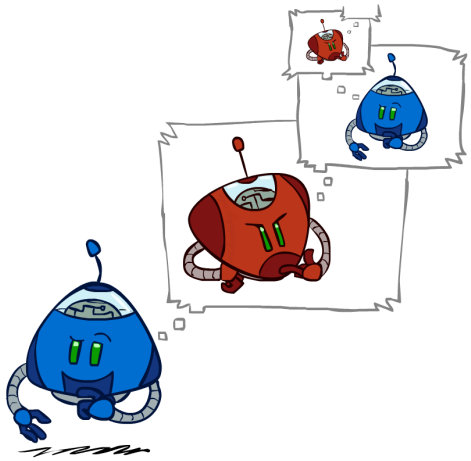
MIN(o)

MAX(x)

MIN(o)



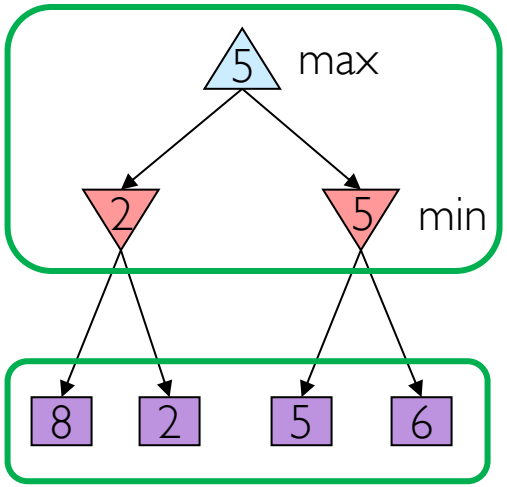
Minimax Algorithm



- Minimax values: the best achievable utility against a rational (optimal) adversary

- Minimax search:
  - A state-space tree search (DFS)
  - Players alternate turns
  - Compute each node's minimax value
- Deterministic policy:
  - Choose action leading to the state with best minimax value

Minimax values: computed recursively



Terminal values: part of the game

Minimax Algorithm

function MINIMAX-DECISION(s) **returns** *action*

return the action a in Actions(s) with the highest MIN-VALUE(Result(s, a))

Deterministic policy:
Choose an action leading to state with best minimax value

function MAX-VALUE(s) **returns** *value*
- **if** Terminal-Test(s) **then return** Utility(s)
- initialize $v = -\infty$
- **for each** a **in** Actions(s):
  - v = max(v, MIN-VALUE(Result(s, a)))
- **return** v

function MIN-VALUE(s) **returns** *value*
- **if** Terminal-Test(s) **then return** Utility(s)
- initialize $v = +\infty$
- **for each** a **in** Actions(s):
  - v = min(v, MAX-VALUE(Result(s, a)))
- **return** v

MAX nodes: Under Agent's Control

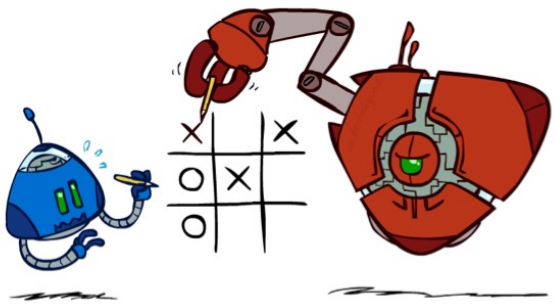
$$V(s) = \max_{a \in \text{Actions}(s)} V(\text{Result}(s, a))$$

MIN nodes: Under Opponent's Control

$$V(s) = \min_{a \in \text{Actions}(s)} V(\text{Result}(s, a))$$

Minimax Properties

- Assume: All future moves will be optimal
  - Rational against a rational player
- Otherwise
  - What if MIN does not play optimally?



Definition: Minimax values produce policies

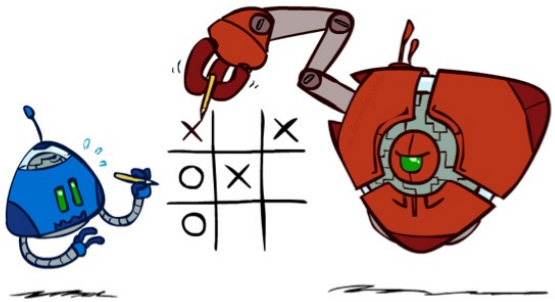
$$V_{\text{minimax}} \Rightarrow \pi_{\text{max}}, \pi_{\text{min}}$$

Definition: To play $\pi_{\text{agent}}, \pi_{\text{opp}}$ against each other, which produces utility

$$V(\pi_{\text{agent}}, \pi_{\text{opp}})$$

Minimax Properties

- What if MIN does not play optimally?
  - MAX will do even better.
- Other strategies against suboptimal opponents may do better than the minimax strategy, but necessarily worse against optimal opponents.



Proposition: lower bound against any opponent

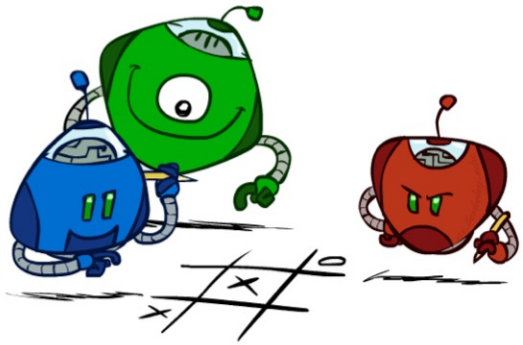
$$V(\pi_{\max}, \pi_{\min}) \leq V(\pi_{\max}, \pi_{\text{opp}}) \text{ for all } \pi_{\text{opp}}$$

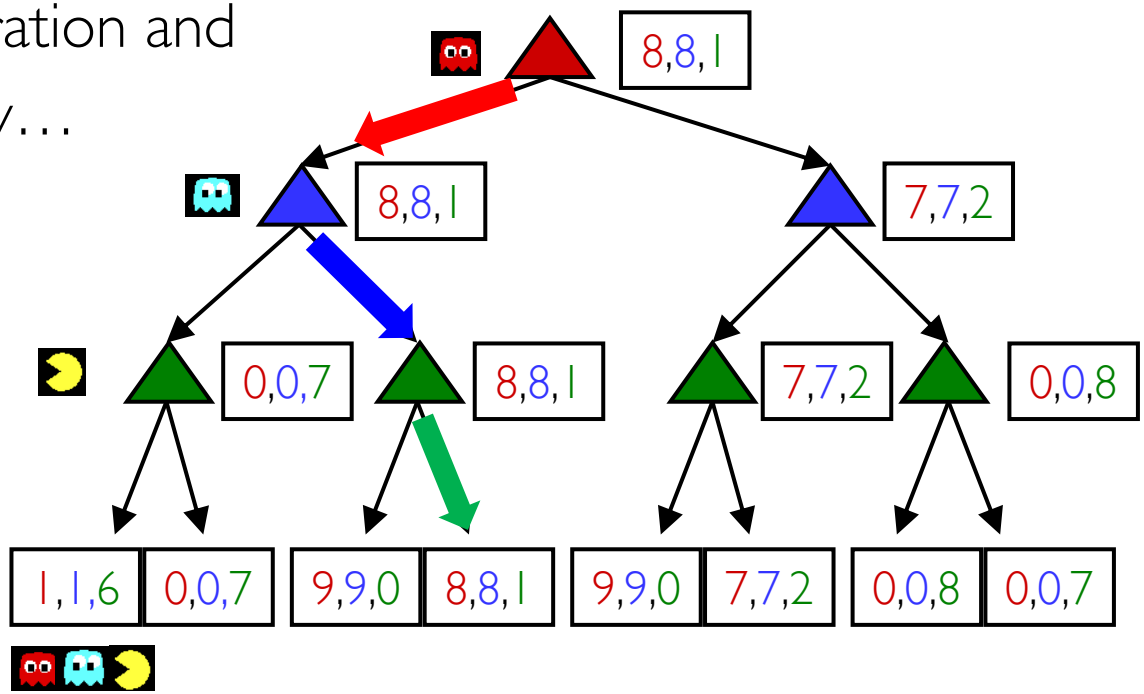
Proposition: best against minimax opponent

$$V(\pi_{\max}, \pi_{\min}) \geq V(\pi_{\text{agent}}, \pi_{\min}) \text{ for all } \pi_{\text{agent}}$$

Generalized Minimax

- What if the game is not zero-sum, or has multiple players?
- Generalization of minimax:
  - Terminals have utility tuples: node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically…

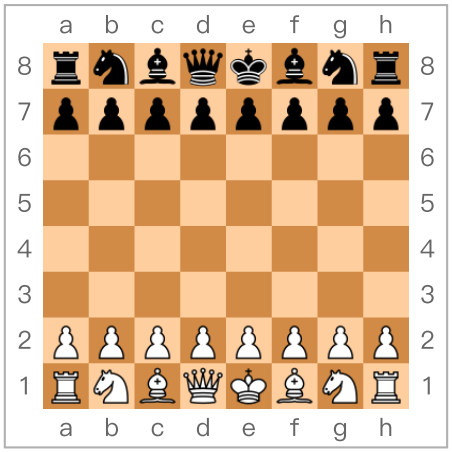




Minimax Efficiency

- How efficient is minimax?
  - Similar to (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$
- Unfortunately, we cannot eliminate the exponent

- Chess: $b \approx 35, m \approx 100$
  - Exact solution is completely infeasible
  - Humans cannot do this either
  - But, do we need to explore the whole tree?

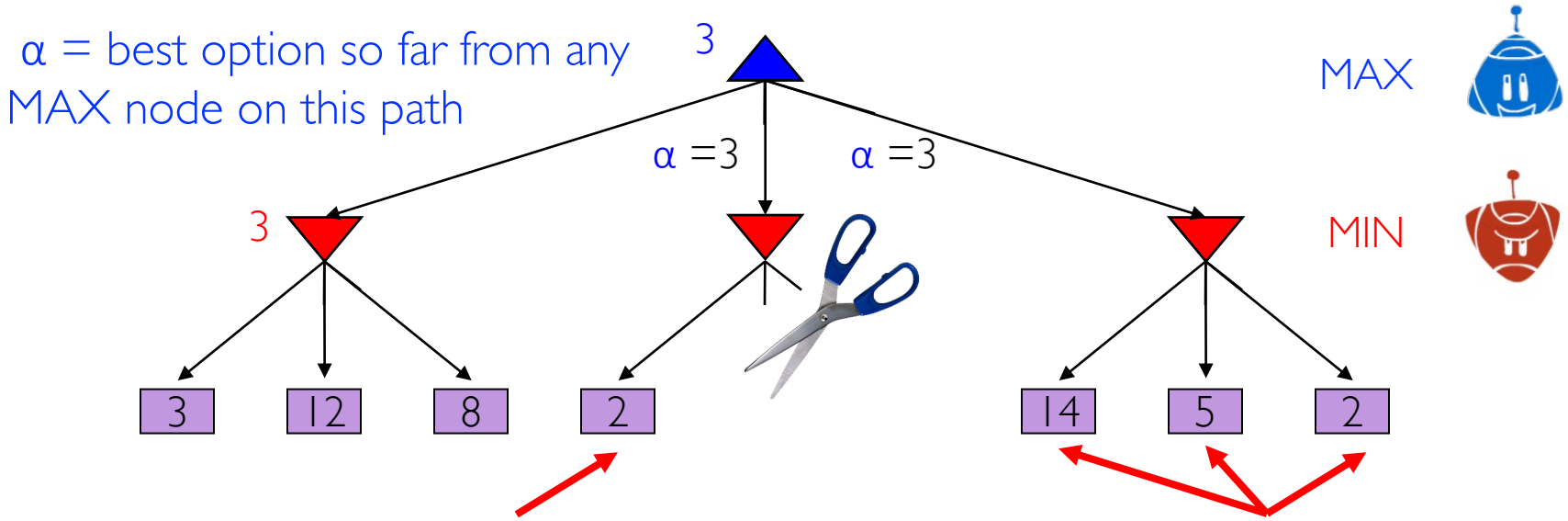


Outline

- Games
- **Adversarial Search**
  - Minimax Algorithm
  - **Alpha-Beta Pruning**
  - Real-Time Decisions
  - Games with Uncertain Outcomes
- Monte Carlo Tree Search

Alpha-Beta Pruning

- **Intuition**: prune the branches that cannot influence the final decision



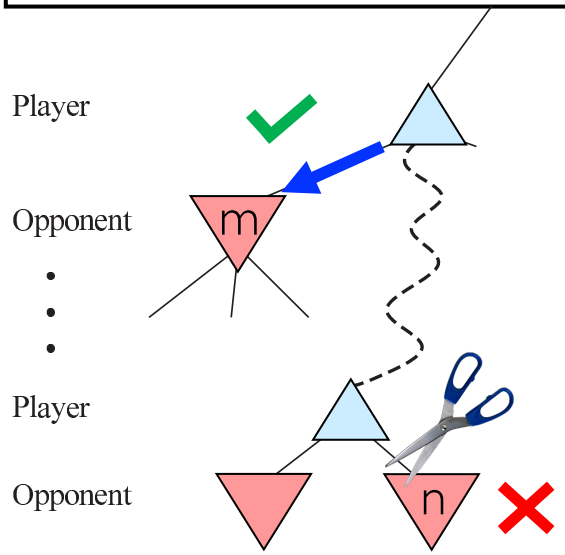
We can prune when: MIN node won't be higher than 2, while parent max has seen something larger in another branch

The order of generation matters: more pruning is possible if good moves come first

Alpha-Beta Pruning

```
function MAX-VALUE(s, α, β) returns value
    if Terminal-Test(s) then return Utility(s)
    initialize v = -∞
    for each successor s′ of state s:
        v = max(v, MIN-VALUE(s′, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v
```

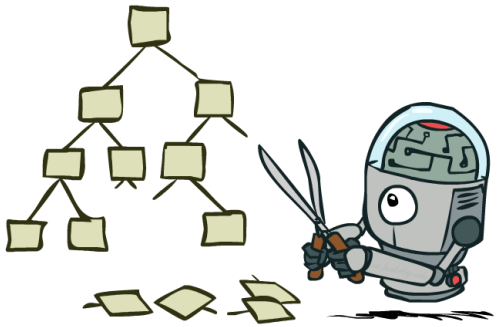
```
function MIN-VALUE(s, α, β) returns value
    if Terminal-Test(s) then return Utility(s)
    initialize v = +∞
    for each successor s′ of state s:
        v = min(v, MAX-VALUE(s′, α, β))
        if v ≤ α
            return v
        β = min(β, v)
    return v
```



- General case for alpha-beta pruning
  - If m is better than n for Player ($n < m = \alpha$), we will never get to n in play m比n好则n被裁剪
    - MIN-VALUE is returned
  - α: MAX's best option on path to root
  - β: MIN's best option on path to root

Move Ordering

- Theorem: This alpha-beta pruning has no effect on the minimax value computed for the root.
- Move ordering: Good child ordering improves effectiveness of pruning
  - Worst ordering: $O(b^m)$ time
  - Best ordering: $O(b^{0.5m})$ time
  - Random ordering: $O(b^{0.75m})$ time



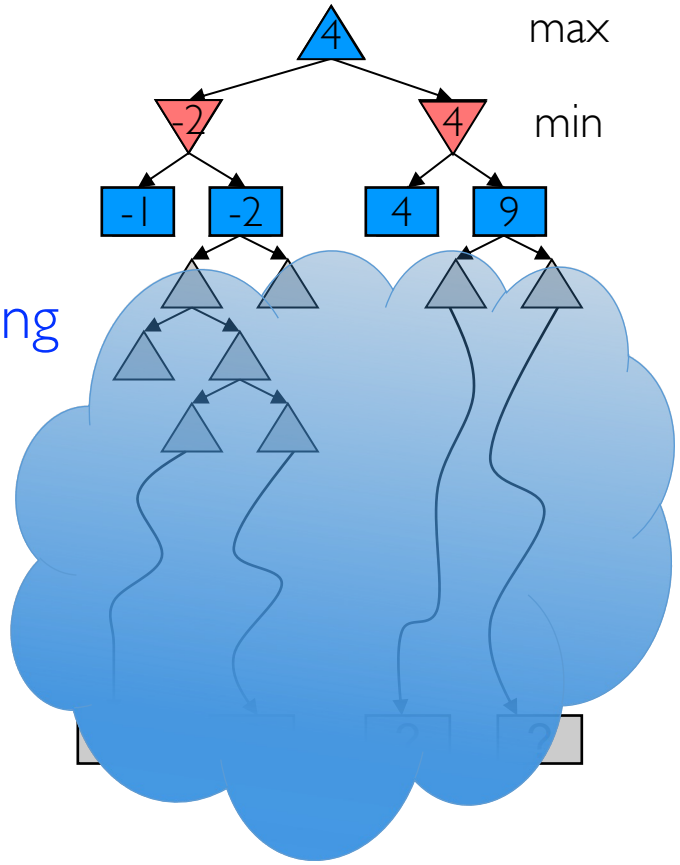
- In practice, can use evaluation function $\mathbf{Eval}(s)$:
  - **MAX nodes:** order successors by decreasing $\mathbf{Eval}(s)$
  - **MIN nodes:** order successors by increasing $\mathbf{Eval}(s)$
- Iterative deepening also helps
  - Searches at shallower depths give move-ordering hints

Outline

- Games
- **Adversarial Search**
  - Minimax Algorithm
  - Alpha-Beta Pruning
  - **Real-Time Decisions**
  - Games with Uncertain Outcomes
- Monte Carlo Tree Search

Resource Limits in Realistic Games

- **Problem:** In realistic games, we cannot search to leaves, too expensive
- Solution 1: Game tree pruning
- Solution 2: Bounded lookahead
  - Search only to a preset depth limit (cutting off search) or horizon (forward pruning)
  - Use evaluation function for non-terminal positions
- No guarantee of optimal play
- More plies make a big difference



Evaluation Function

Definition: Evaluation function

An evaluation function $\mathbf{Eval}(s)$ is a (possibly very weak) estimate of the true value $V_{\mathbf{minimax}}(s)$.

- Typically weighted linear sum of features:

$$\mathrm{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

- Or a more complex nonlinear function (e.g., Neural Network) trained by self-play RL
  - Applying **machine learning (ML)** techniques to chess has confirmed that a bishop is indeed worth about three pawns

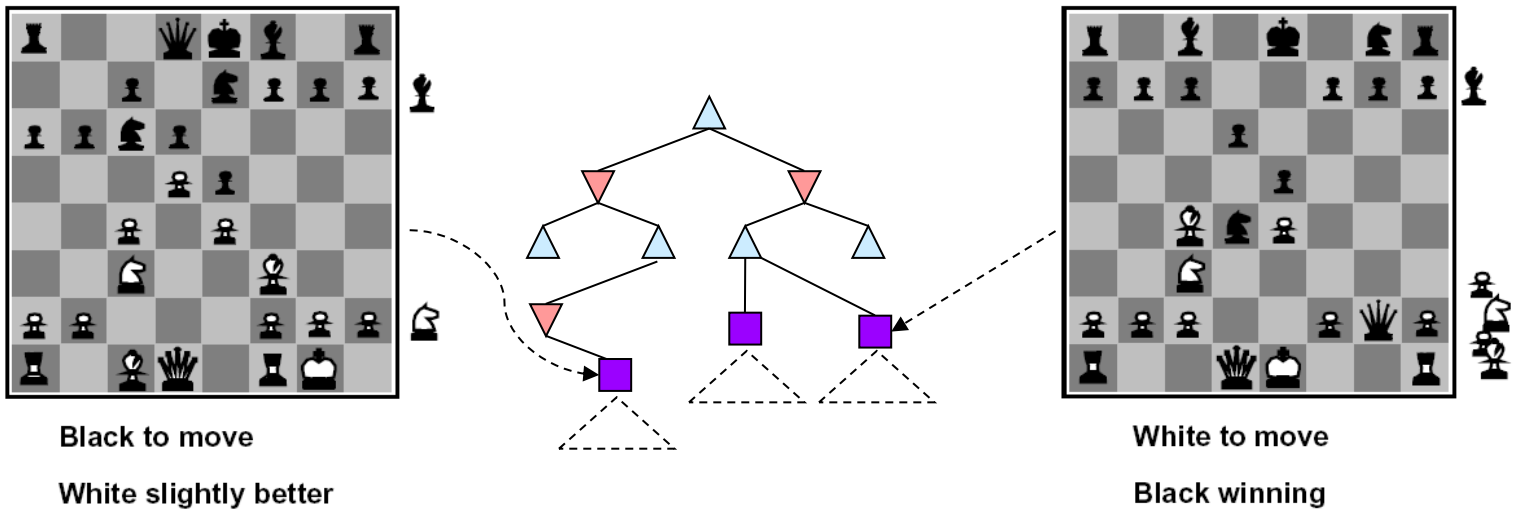
Evaluation Function: Chess

- There is no free lunch, and we have to use domain knowledge

$$\text{Eval}(s) = \text{material} + \text{mobility} + \text{king-safety} + \text{center-control}$$

$$\text{material} = 10^{100}(K - K') + 9\,(Q - Q') + 5\,(R - R') + 3(B - B' + N - N') + 1\,(P - P')$$

$$\text{mobility} = 0.1(\text{num-legal-moves} - \text{num-legal-moves}')$$



Cutting off Search

- Limited depth tree search (stop at maximum depth $d_{\max}$)

$$V_{\text{minimax}}(s, d)$$

$$= \begin{cases} \text{Utility}(s), & \text{Terminal-Test}(s) \\ \text{Eval}(s), & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Result}(s, a), d), & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Result}(s, a), d - 1), & \text{Player}(s) = \text{opp} \end{cases}$$

- Use: at state s, call $V_{\text{minimax}}(s, d_{\max})$
- Convention: decrement depth at last player's turn
  - One move deep = two half-moves, each of which is called a ply

Depth Matters

- **Evaluation functions** are always imperfect
- Deeper search ⟹ better play (usually)

A trade-off between complexity of features and of computation

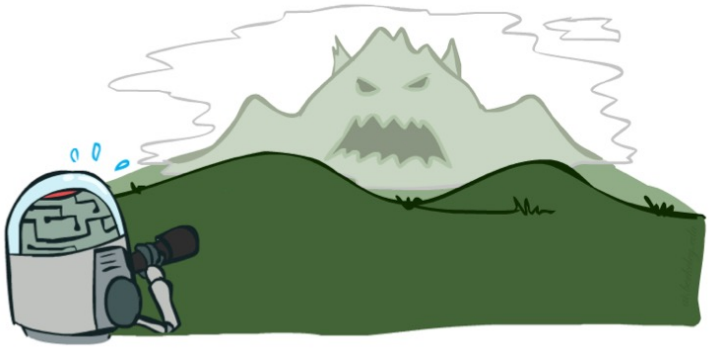
Example:

Suppose we have 100 seconds, and can explore 10K nodes / sec

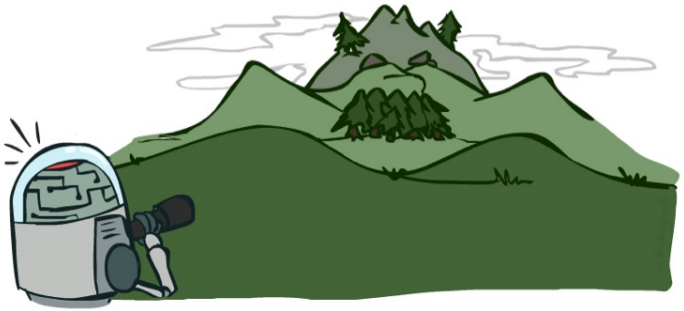
So can check 1M nodes per move

Chess with alpha-beta, $35^{8/2} = 1M$;

Depth 4 (8 plies) is good

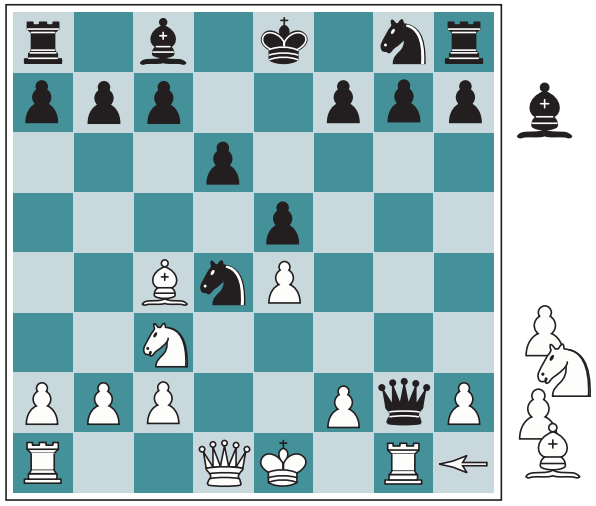


- Use iterative deepening search for an anytime algorithm



Quiescence Search

- Terminate search only in quiescent positions
  - Where no major changes are expected in feature values
  - For example: In chess, positions in which favorable captures can be made are not quiescent.



White to move

Example: Non-quiescent position

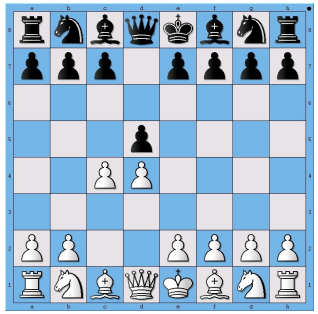
Seems Black has an advantage of a knight and two pawns?

No! White will capture the queen, giving it an advantage that should be strong enough to win.

Acceleration Strategies

- Forward pruning: some moves pruned without further consideration
  - Beam search: on each ply, consider only a "beam" of n best moves
- Transposition Table: a hash table to store the searched positions and its evaluation.
  - Explored list in Graph Search

1. d4 d5
2. c4 ...



1. c4 d5
2. d4 ...

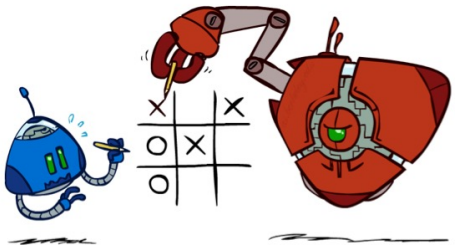
- Opening of games:
  - The best advice of human experts (< 10 moves)
  - Rare position → Switch from table lookup to search
- Ending of games: the computer has the expertise
  - All chess endgames with up to five/six pieces →

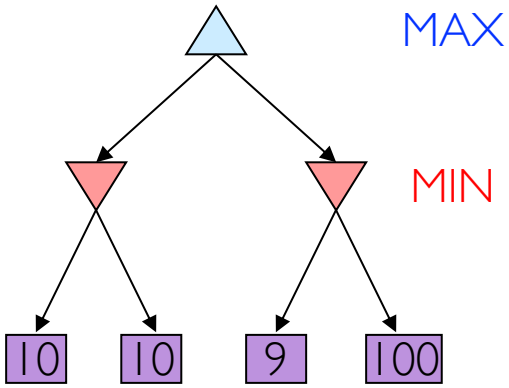
Ken Thompson
(1983 Turing Award)

Outline

- Games
- Adversarial Search
  - Minimax Algorithm
  - Alpha-Beta Pruning
  - Real-Time Decisions
  - **Games with Uncertain Outcomes**
- Monte Carlo Tree Search

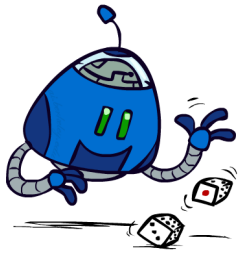
Uncertain Outcomes in Trees

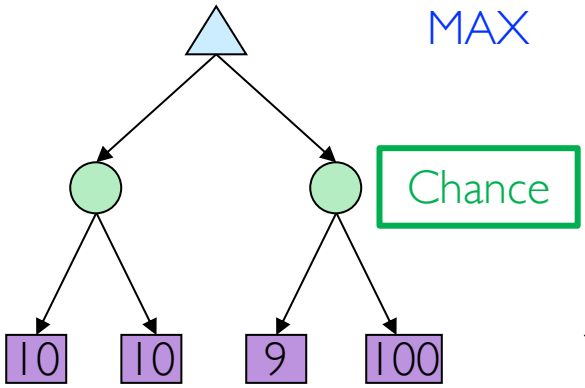




Tictactoe, chess

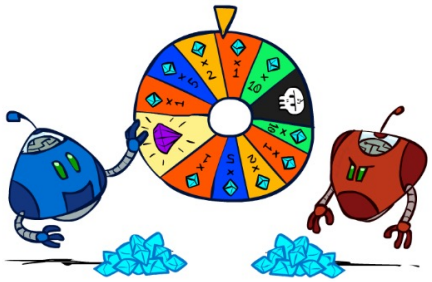
Minimax

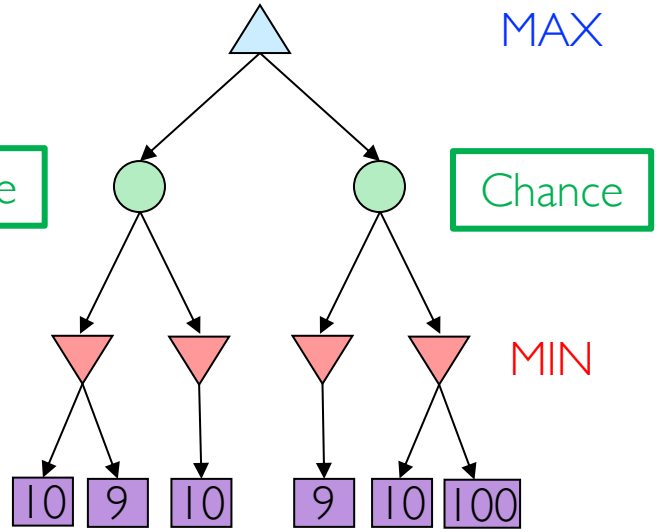




Tetris, investing

Expectimax

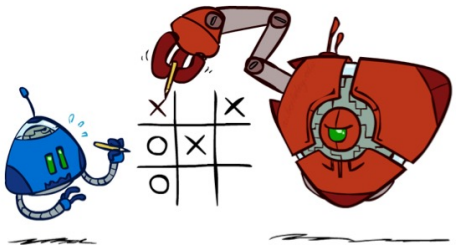




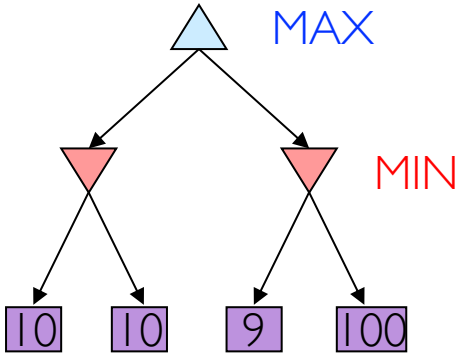
Stochastic Games

Backgammon, Monopoly

Expectiminimax



Minimax



function decision(s) **returns** an *action*

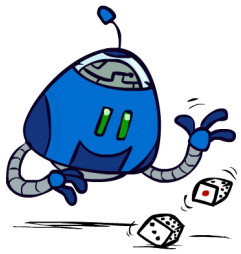
return the action a in Actions(s) with the highest minimax-value(Result(s, a))

function minimax-value(s) **returns** a *value*

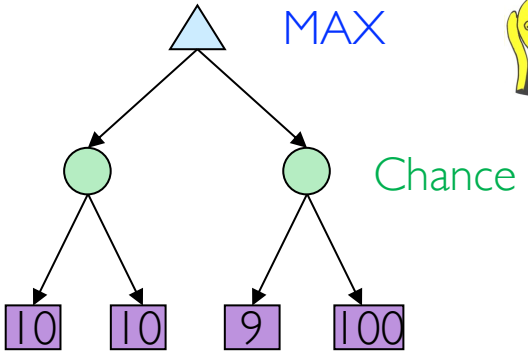
if Terminal-Test(s) **then return** Utility(s)

if Player(s) = MAX **then return** $\max_{a \text{ in Actions}(s)}$ minimax-value(Result(s, a))

if Player(s) = MIN **then return** $\min_{a \text{ in Actions}(s)}$ minimax-value(Result(s, a))



Expectimax



Being rational means maximizing your expected utility

function decision(s) **returns** an *action*

return the action a in Actions(s) with the highest value(Result(s, a))

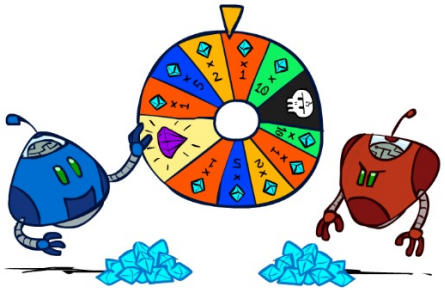
function value(s) **returns** a *value*

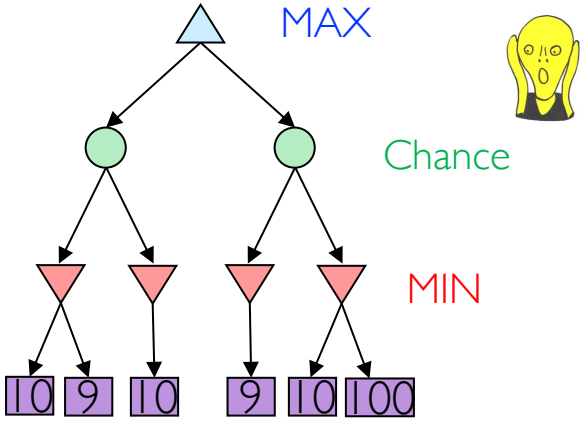
if Terminal-Test(s) **then return** Utility(s)

if Player(s) = MAX **then return** $\max_{a \text{ in Actions}(s)}$ value(Result(s, a))

if Player(s) = CHANCE **then return** $\text{sum}_{a \text{ in Actions}(s)}$ Pr(a)*value(Result(s, a))

Expectiminimax





function decision(s) **returns** an *action*

return the action a in Actions(s) with the highest value(Result(s, a))

function value(s) **returns** a *value*

if Terminal-Test(s) **then return** Utility(s)

if Player(s) = MAX **then return** $\max_{a \text{ in Actions}(s)}$ value(Result(s, a))

if Player(s) = MIN **then return** $\min_{a \text{ in Actions}(s)}$ value(Result(s, a))

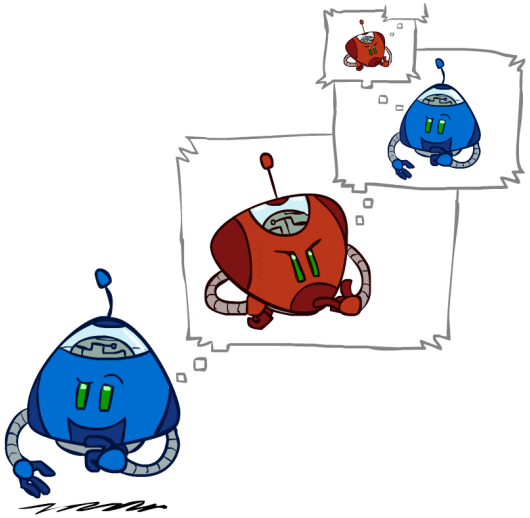
if Player(s) = CHANCE **then return** $\text{sum}_{a \text{ in Actions}(s)}$ Pr(a)*value(Result(s, a))

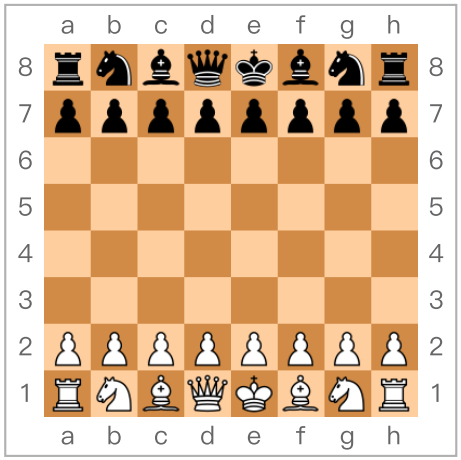
Outline

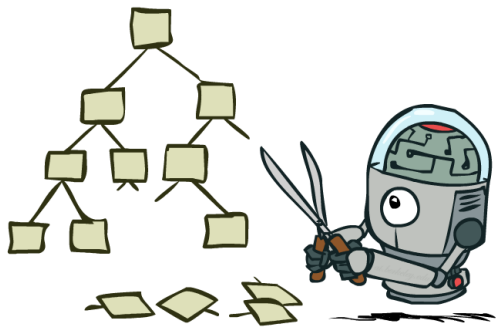
- Games
- Adversarial Search
  - Minimax Algorithm
  - Alpha-Beta Pruning
  - Real-Time Decisions
  - Games with Uncertain Outcomes
- **Monte Carlo Tree Search**

Conventional Game Tree Search

- Minimax with alpha-beta pruning
- Effective for:
  - Modest branching factor
  - A good heuristic value function is known
- **Chess:** Deep Blue, Rybka etc.







Go: Failure for Alpha-Beta

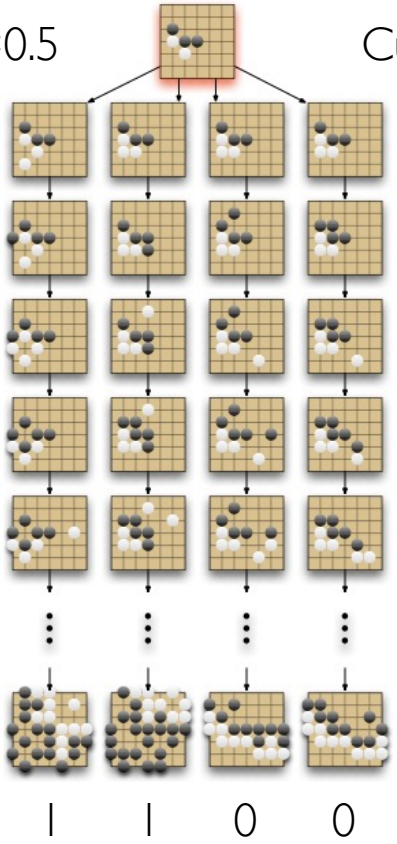
- Decades of research on knowledge-based and alpha-beta methods
- Only reach level weak to intermediate

- Branching factor of Go is very large
  - 250 moves on average, game length > 200 moves
  - Order of magnitude greater than $b \approx 20$ for Chess
  - Alpha-beta search limited to only 4 or 5 plies
- Lack of a good evaluation function
  - Too subtle to model: similar looking positions can have completely different outcomes

Monte Carlo Tree Search (MCTS)

MCTS combines two important ideas

Evaluation by rollouts



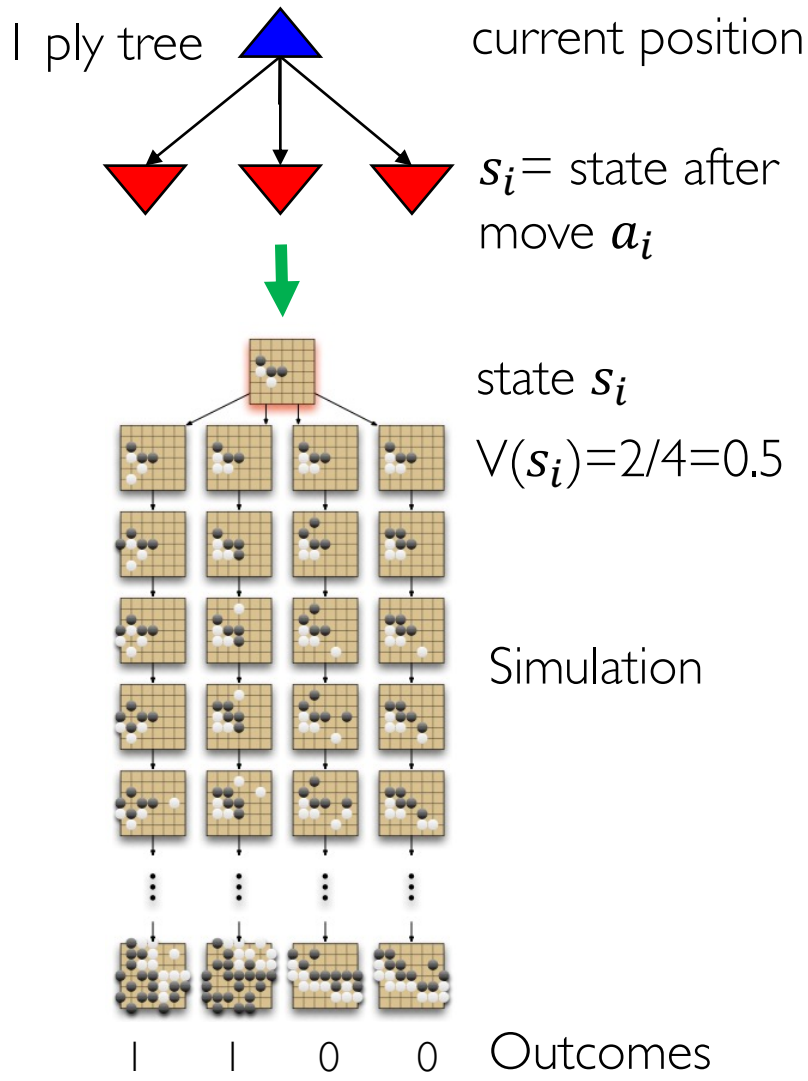
Selective search

Evaluation by Rollouts

- Pure Monte Carlo search:
  - Do N rollouts from each child of the root, record fraction of wins
  - Pick the move that gives the best outcome by this metric

For each rollout:
  Repeat until terminal:
    Play a move according to a fixed, fast rollout policy (usually random)
  Record the result

Having a "better" rollout policy helps



Selective Search 选择性搜索

- Use results of simulations to guide growth of the game tree
  - Exploitation: focus on more promising moves
  - Exploration: focus on more uncertain nodes
- Seems like two contradictory goals
  - There is a well-known exploitation-exploration dilemma
  - Theory of bandits can help

Multi-Armed Bandit Problem

- Assumptions:
  - Choice of several arms
  - Each arm pull is independent of other pulls, either a win (payoff 1) or a loss (payoff 0)
  - Each arm has fixed, unknown average payoff
- Which arm has the best average payoff?

A

P(A wins)=60%

B

P(B wins)=55%

C

P(B wins)=40%

A is the best arm,
but we don't know that

Upper Confidence Bound

- Policy: Strategy for choosing arm to play at some time step t
  - Given arm selections and outcomes of previous trials at times $1, \ldots, t-1$
- Maximize: UCB1 formula (Auer et al 2002)

$$v_i + C \times \sqrt{\frac{\ln(N)}{n_i}}$$

value estimate; tunable parameter; total number of trials; num trials for arm i

Exploit: prefers higher payoff arm

Explore: prefers less played arm

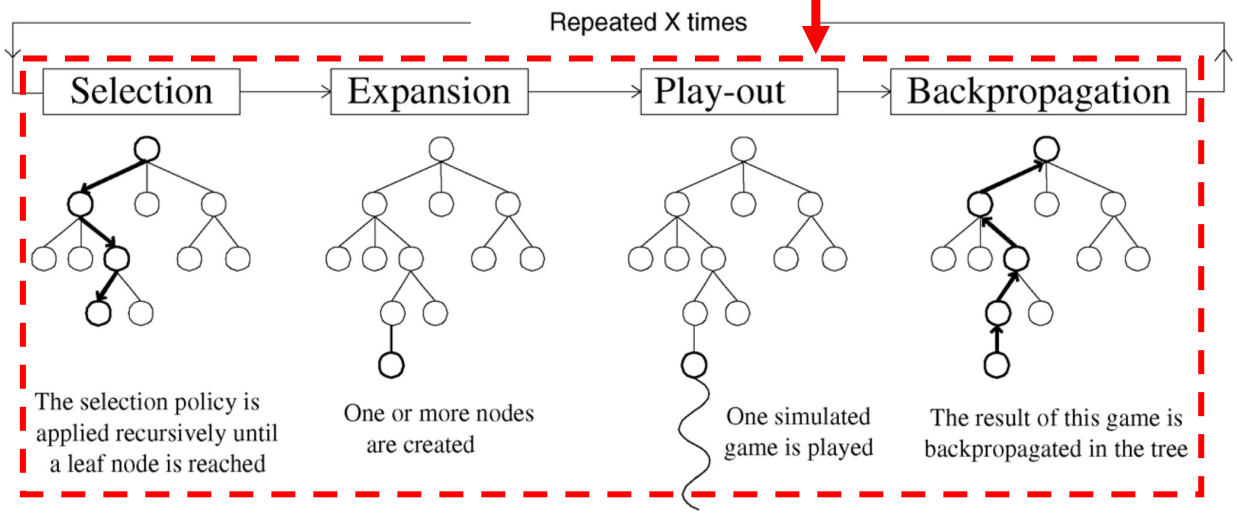
Intuition: Confidence interval

- Expect "true value" to be in some confidence interval around v_i
- The confidence interval is large when number of trials n_i is small, shrinks in proportion to $\sqrt{n_i}$

Monte Carlo Tree Search: UCT

- UCT (Kocsis Szepesvari, 06): Upper Confidence Bounds on Trees

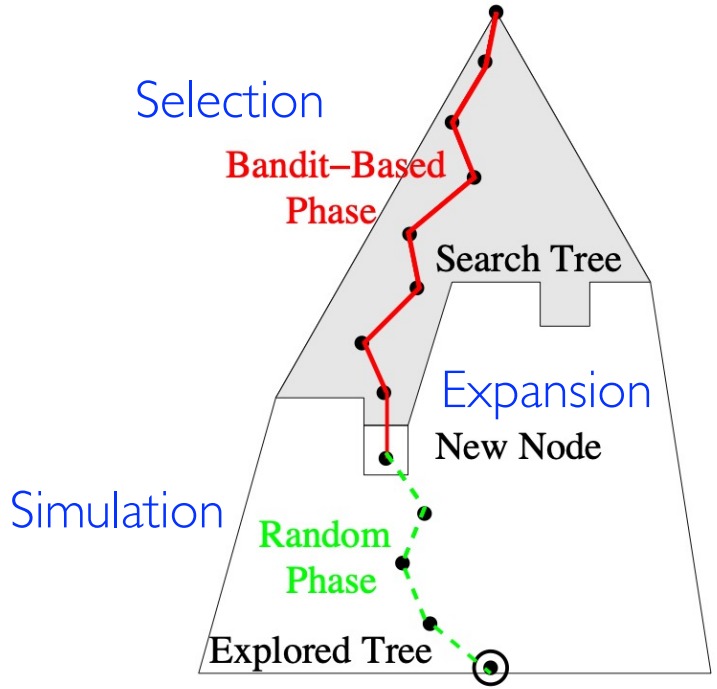
```
01. function Monte-Carlo-Tree-Search(state) returns an action
02.     tree = Node(state)
03.     while Is-Time-Remaining() do
04.         MCTS-sample(tree)
05.     return the move in Actions(state) whose node has highest number of playouts
```



MCTS-sample: Gradually grow the search tree

MCTS: Overview

- Selection
  - Used for nodes we have seen before
  - Pick according to UCB
- Expansion
  - Used when we reach the frontier
  - Add one node per playout
- Simulation
  - Used beyond the search frontier
  - Don't bother with UCB, just play randomly



- Backpropagation
  - After reaching a terminal node, update value and visits for states expanded in selection and expansion

MCTS: Overview

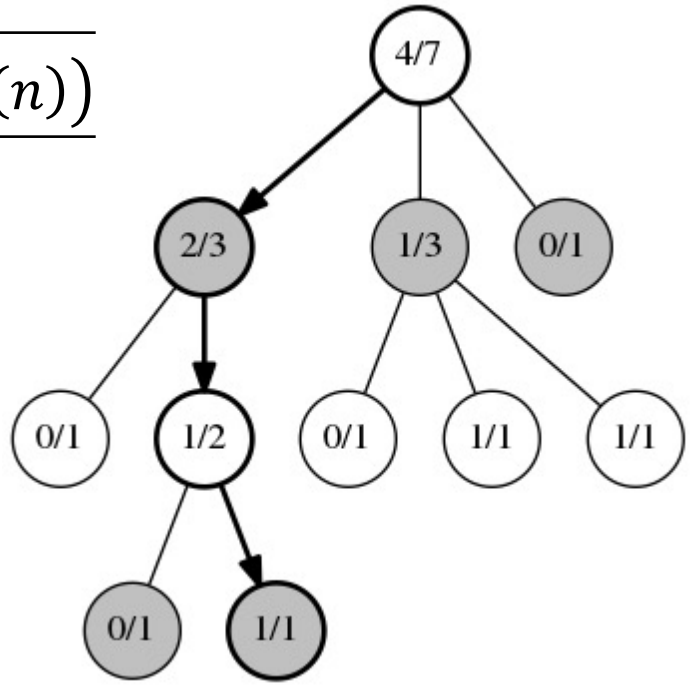
- Gradually grow the search tree:
  - Selection, Expansion, Simulation, Backpropagation

```
01. function MCTS-sample(state)
02.     if all children of state expanded:
03.         next-state = UCB-sample(state)
04.         winner = MCTS-sample(next-state)          (1). Selection
05.     else:
06.         next-state = expand(random unexpanded child)   (2). Expansion
07.         winner = random-playout(next-state)       (3). Simulation
08.         update-value(next-state, winner)
09.     update-value(state, winner)                   (4). Backpropagation
10.     return winner
```

MCTS: Selection

- Selection policy is applied recursively until a node that is not fully expanded is reached

$$\text{UCB1}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{Parent}(n))}{N(n)}}$$



- $U(n)$: the total utility of rollouts (e.g., # wins) that went through node n
- $N(n)$: the number of playouts through node n

MCTS: Selection

```
01. function UCB-sample(state):
02.     weights = []
03.     for child of state:
04.         w = child.value / child.visits + C * sqrt(log(state.visits) / child.visits)
05.         weights.append(w)
06.     distribution = [w / sum(weights) for w in weights]
07.     return child sampled according to distribution
```

$$\text{UCB1}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{Parent}(n))}{N(n)}}$$

$U(n)$: the total utility of rollouts (e.g., # wins) that went through node n ← *state.value*

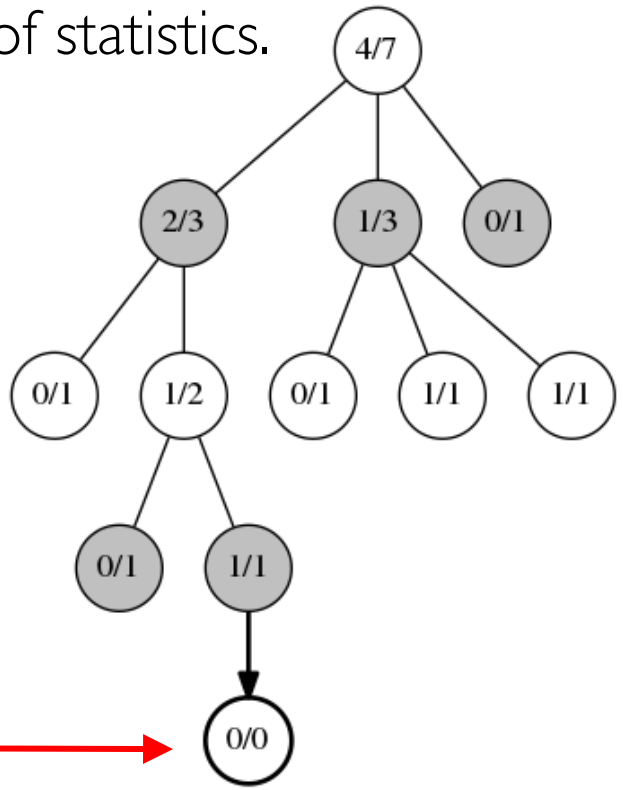
$N(n)$: the number of playouts through node n ← *state.visits*

MCTS: Expansion

```
01. function MCTS-sample(state)
02.     if all chidren of state expanded:
03.         next-state = UCB-sample(state)
04.         winner = MCTS-sample(next-state)
05.     else:
06.         next-state = expand(random
                                unexpanded child)
07.         winner = random-playout(next-state)
08.         update-value(next-state, winner)
09.     update-value(state, winner)
10.     return winner
```

```
01. function expand(state):
02.     state.visits = 0
03.     state.value = 0
```

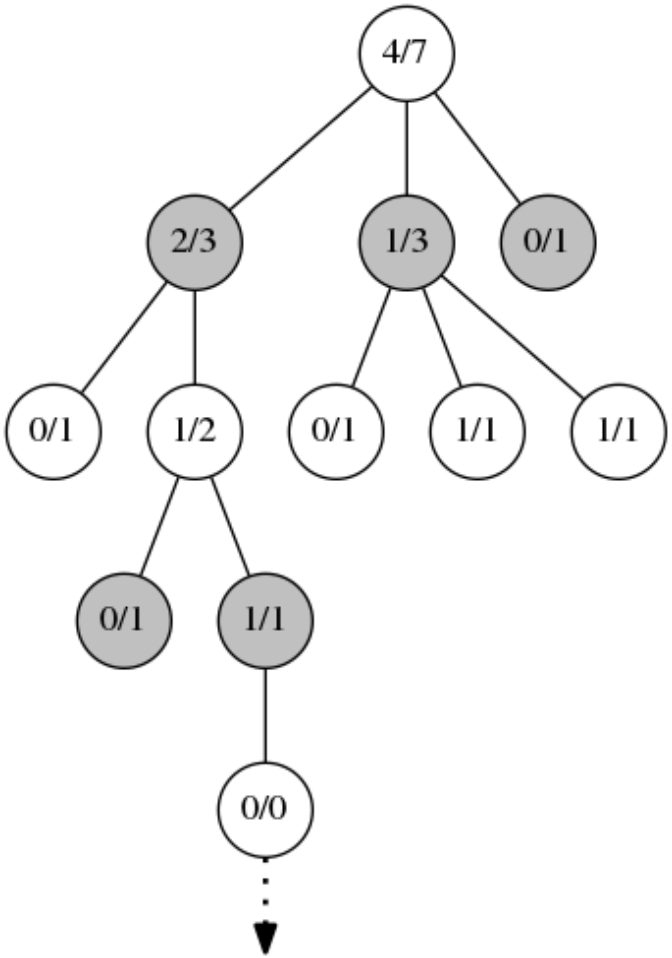
- An unvisited child position is randomly chosen, and a new record node is added to the tree of statistics.



MCTS: Simulation

- One typical Monte Carlo simulated game is played
  - Simulating a real game is hard
  - Let's just play out the game randomly

```
01. function random-playout(state):
02.     if is-terminal(state):
03.         return winner
04.     else:
05.         return random-playout(random-move(state))
```

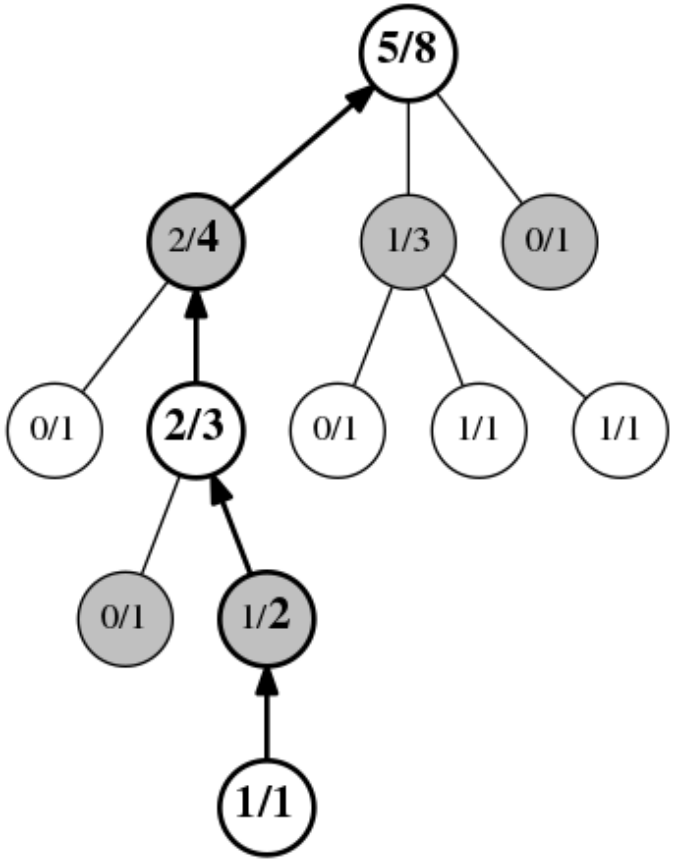


MCTS: Backpropagation

- Result is backpropagated up the tree
  - Update $U(n)$ and $N(n)$

state.value *state.visits*

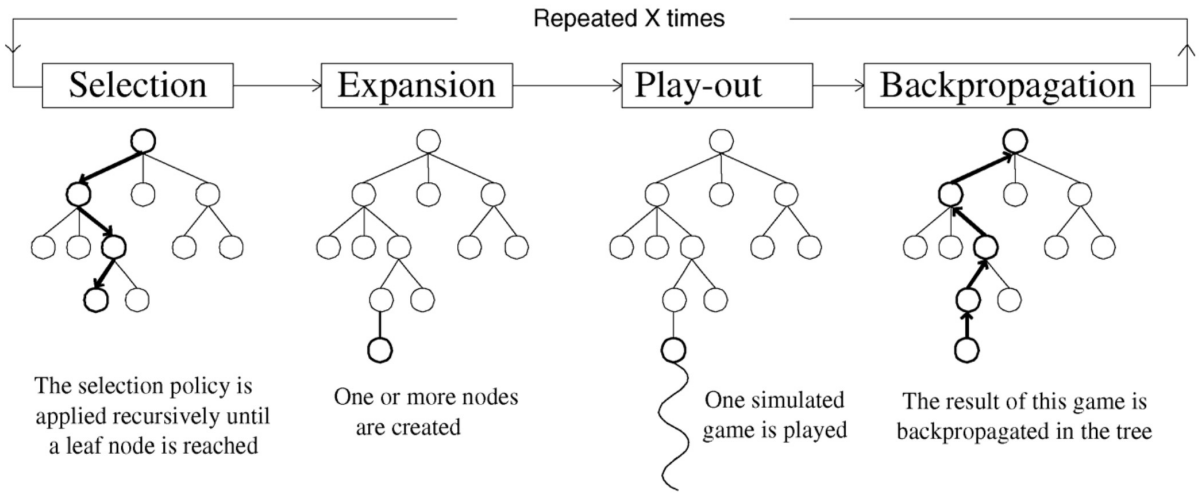
```
01. function update-value(state, winner):
02.     state.visits++
03.     if winner == state.turn:
04.         state.value += 1
```



MCTS: Decision

```
01. function Monte-Carlo-Tree-Search(state) returns an action
02.     tree = Node(state)
03.     while Is-Time-Remaining() do
04.         MCTS-sample(tree)
05.     return the move in Actions(state) whose node has highest number of playouts
```

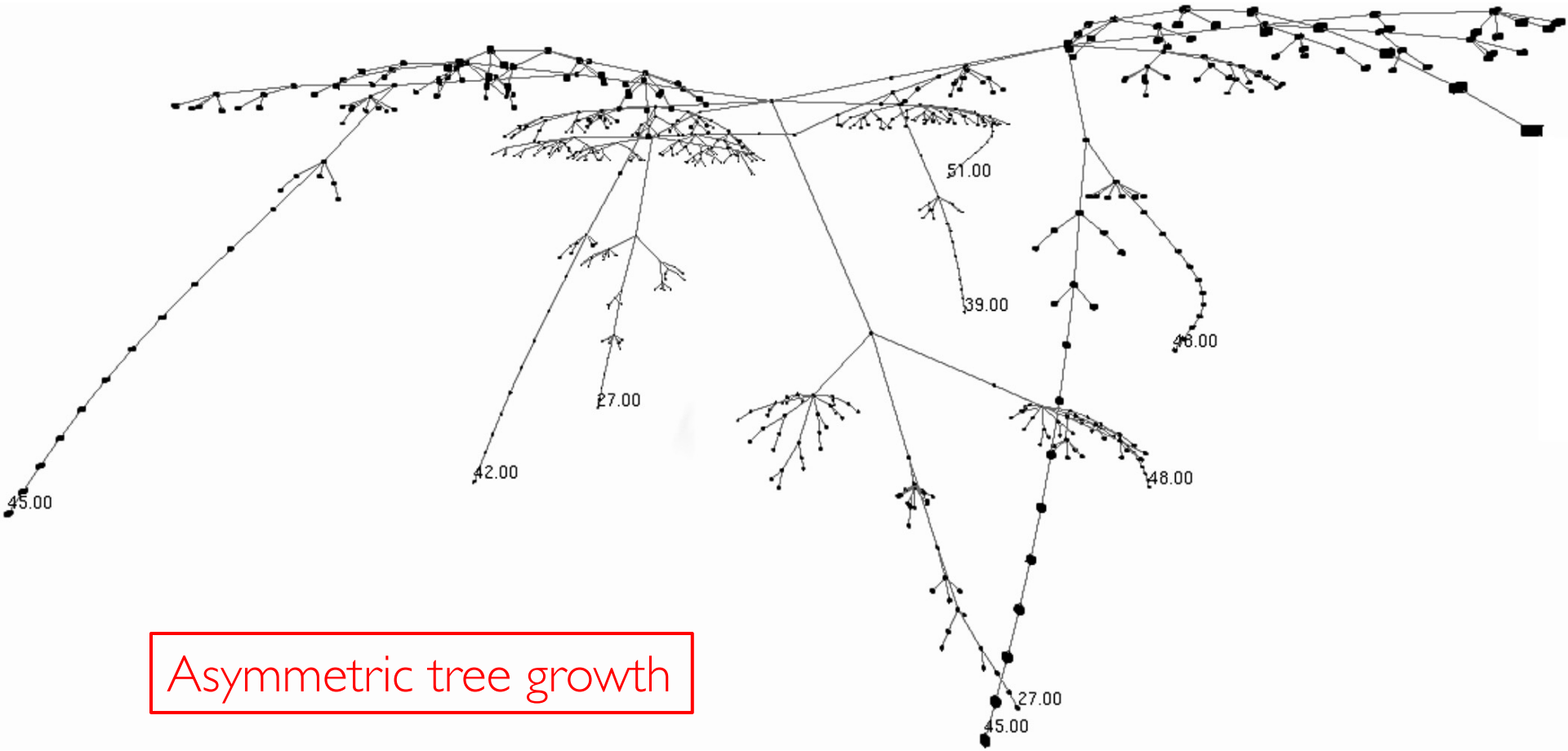
Returned solution: Path visited most often (highest $N(n)$)



Decision

Why not the highest average utility?
A node with 65/100 wins is better than one with 2/3 wins.

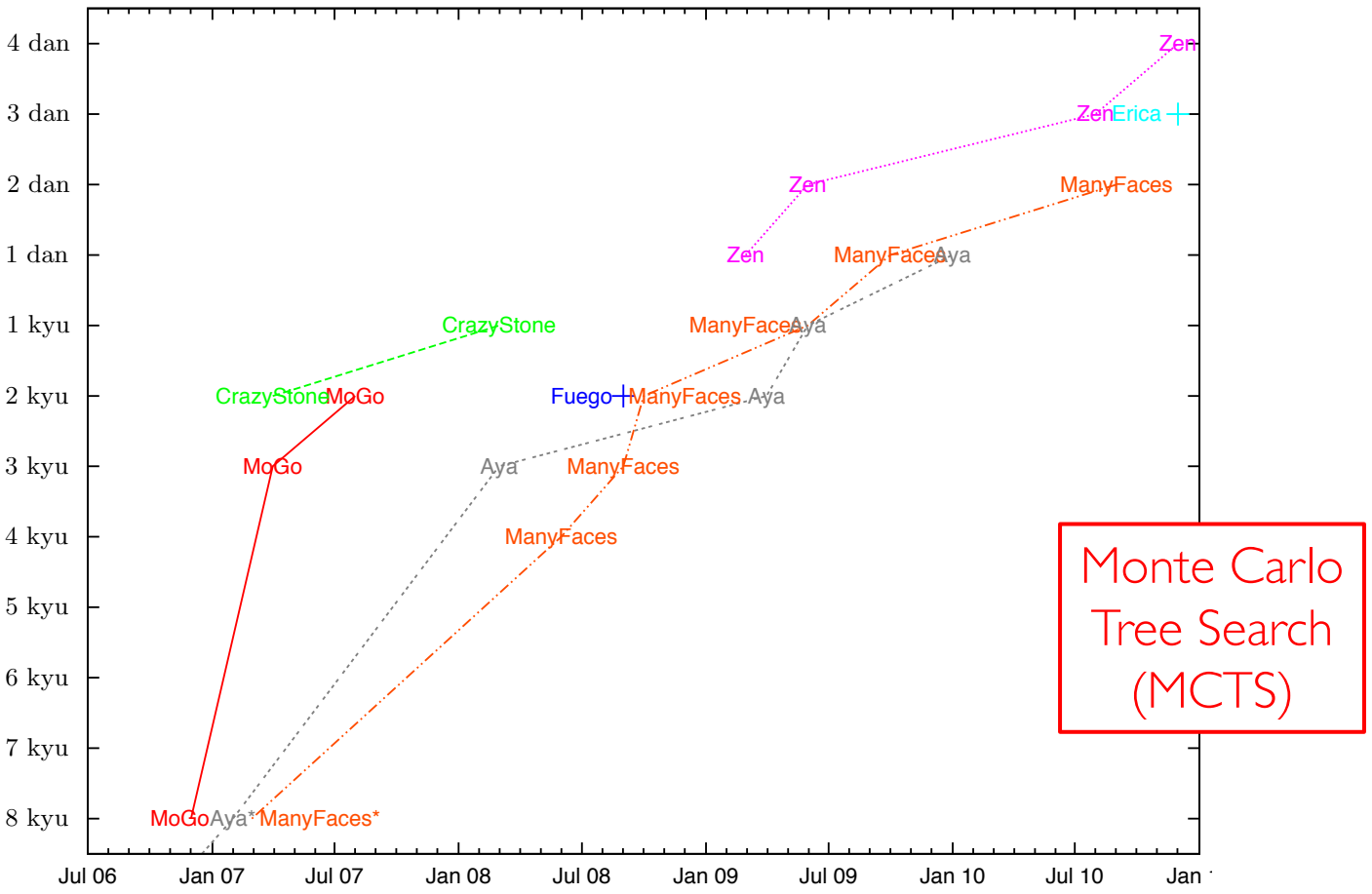
MCTS: Tree Example



Asymmetric tree growth

Figure from CadiaPlayer, Bjornsson and Finsson, IEEE T-CIAIG

MCTS and Early Go Results

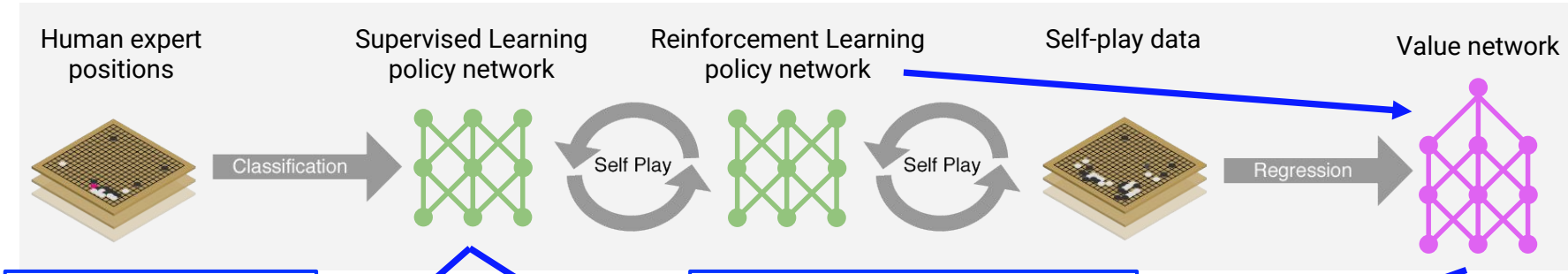


Dates at which several strong MCTS programs on KGS achieved a given rank. Classical versions of these programs, before MCTS was introduced, are marked with an asterisk* (Gelly et al., 2012)

Why MCTS

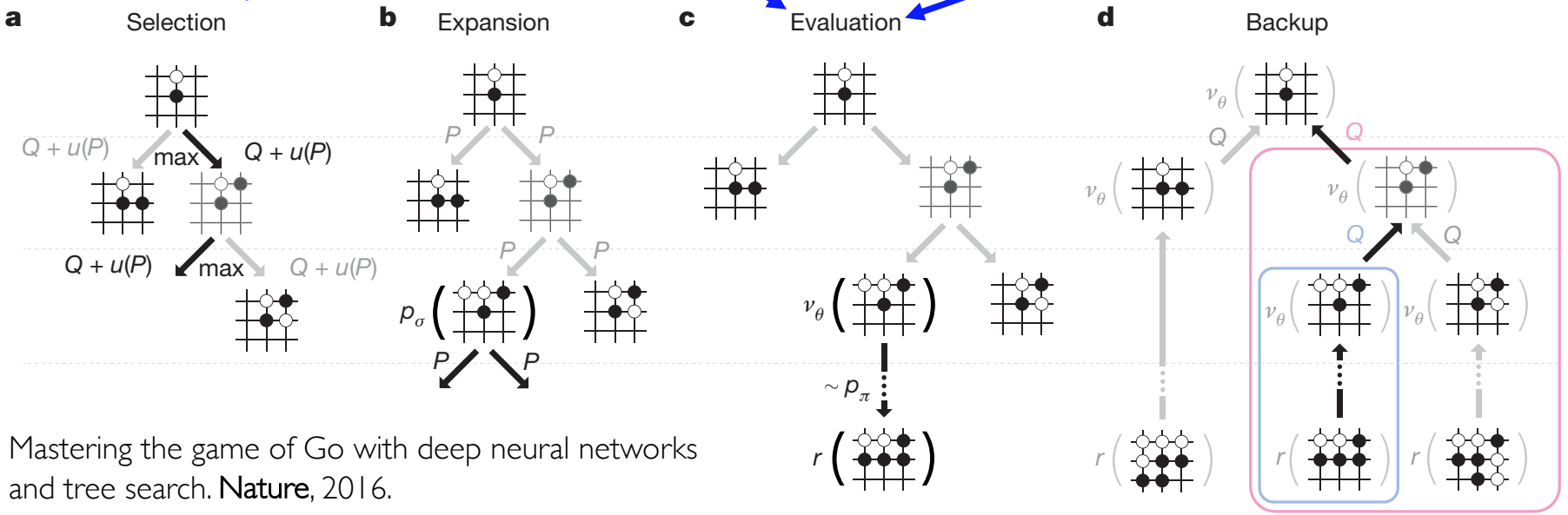
- Pros:
  - Grows tree asymmetrically, balancing expansion and exploration
  - Unaffected by branching factor
  - Easy to adapt to new games
  - Heuristics not required, but can also be integrated
  - Anytime algorithm: can finish on demand
  - Trivially parallelizable
- Cons:
  - Can't handle extreme tree depth
  - Requires ease of simulation, massive computation resources
  - Relies on random play being "weakly correlated"
  - Many variants, need expertise to tune
  - Theoretical properties not yet understood

AlphaGo: MCTS + Learning



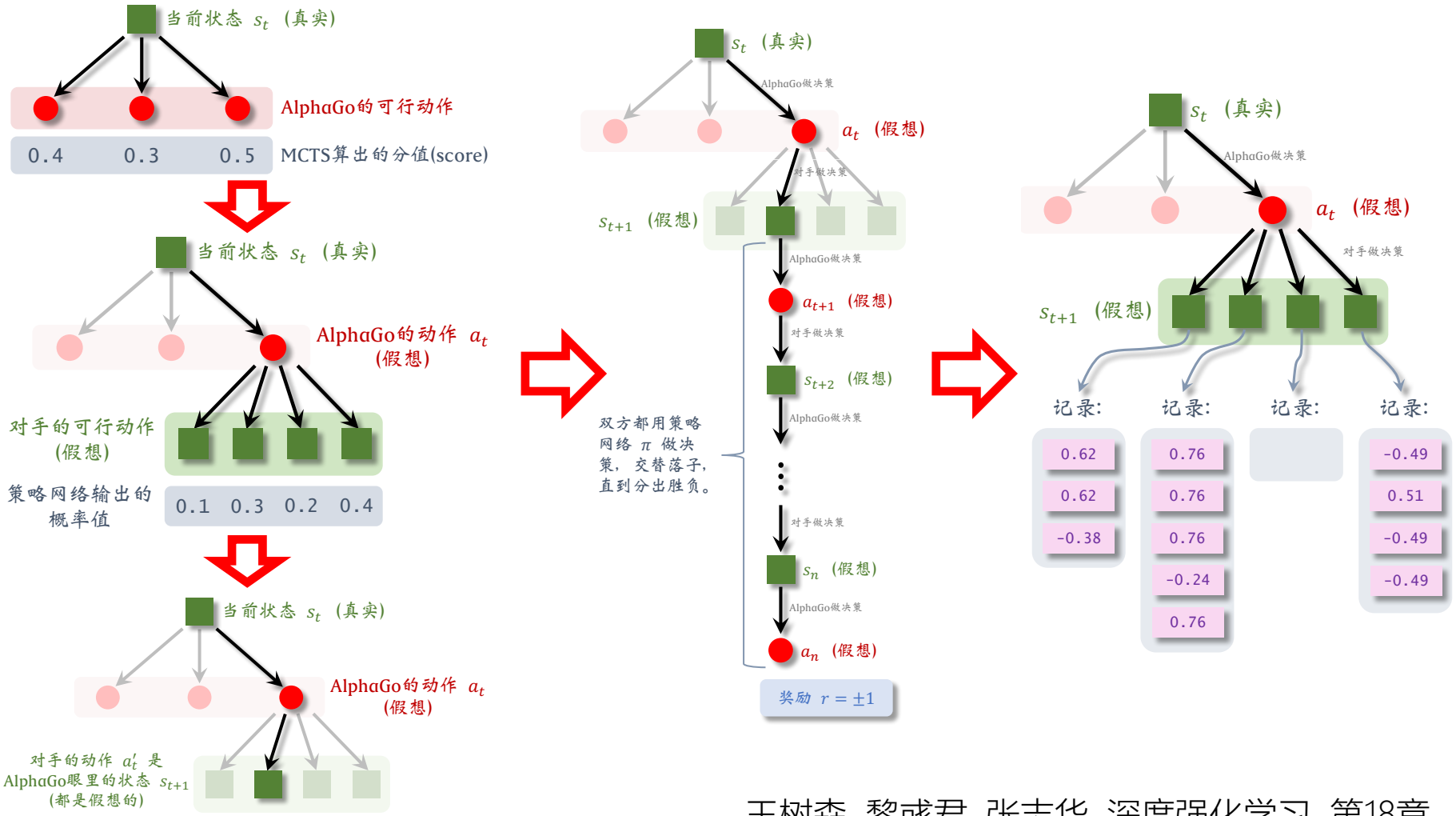
Policy neural net to select moves

Value network to evaluate board positions



Mastering the game of Go with deep neural networks and tree search. **Nature**, 2016.

AlphaZero: MCTS + Learning



王树森, 黎彧君, 张志华. 深度强化学习, 第18章.

Thank You

Questions?

Mingsheng Long
mingsheng@tsinghua.edu.cn
http://ise.thss.tsinghua.edu.cn/~mlong
答疑：东主楼11区413室

[Some slides adapted from Dan Klein and Pieter Abbeel]