Artificial Intelligence

Lecture 3：Adversarial Search

Xiaojin Gong

2022-03-07

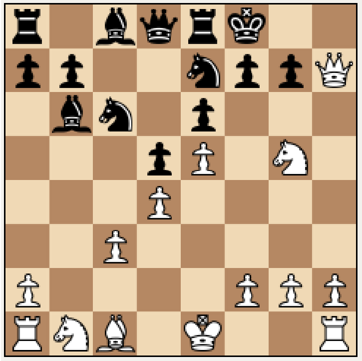
Credits: AI Courses in Berkeley

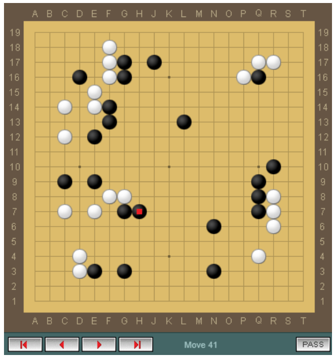
Review

- Problem-Solving Agents
  - Problem Formulation
  - Solving Problems by Searching
    - Uninformed Search
      - Breadth First Search
      - Depth First Search
      - Iterative Deepening Search
      - Cost-sensitive Search
    - Informed Search
      - Best First Search
      - A* Search

Outline

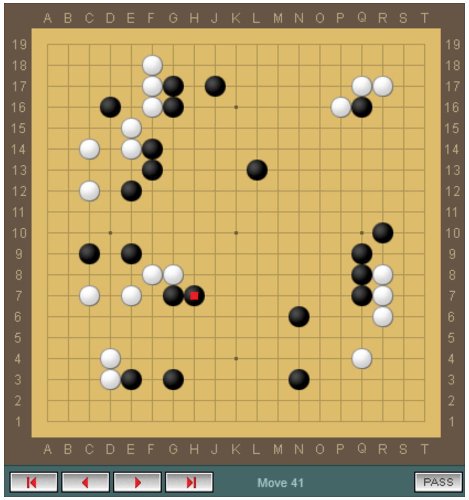
- Adversarial Search Problem (a.k.a Games)
  - Game Formulation
  - Minimax Search
  - Alpha-Beta Pruning
  - Monte Carlo Tree Search
- AlphaGo

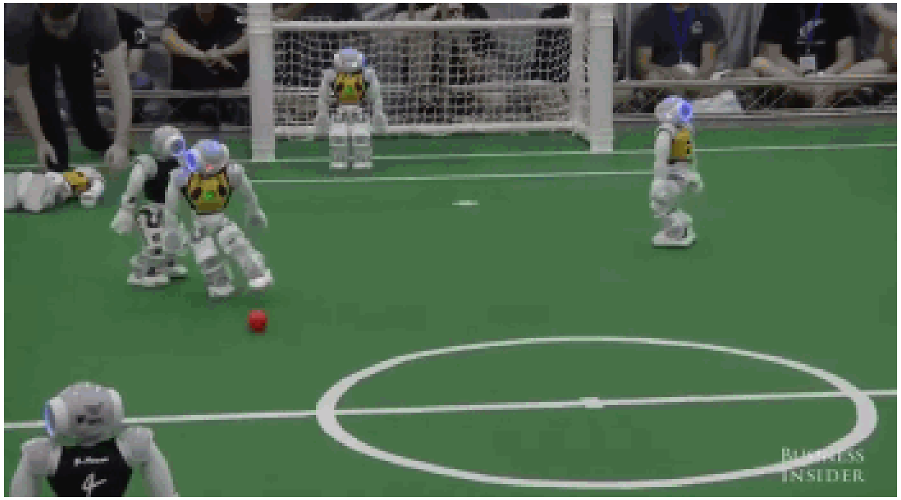




Adversarial Search

- Adversarial search problems (a.k.a Games)
  - Multi-agent environment, competitive
  - Solving by adversarial search
- Types of Games
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?
  - Deterministic or stochastic?



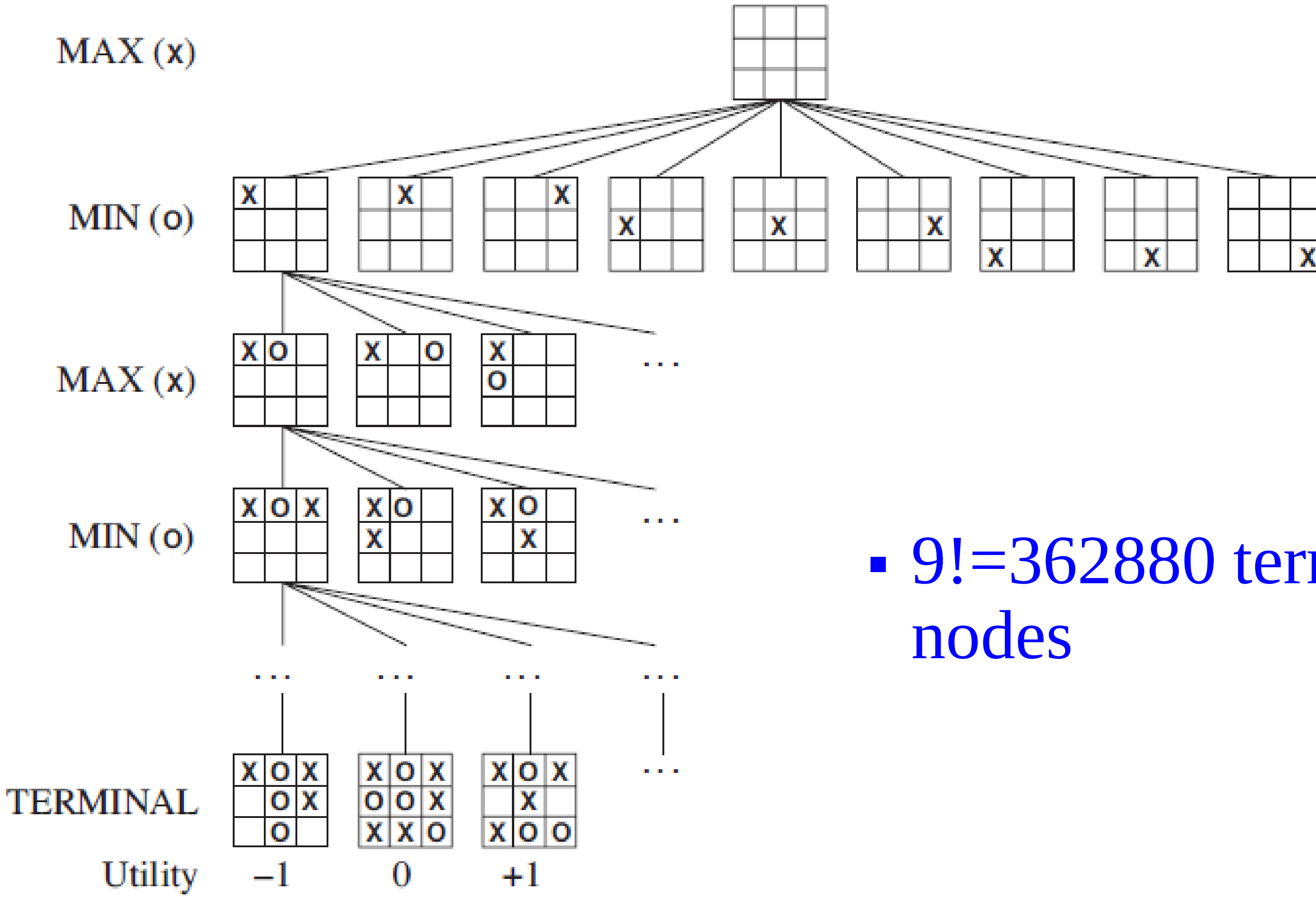


Deterministic Games

- A search problem consists of
  - S_0: the initial state
  - Player(*s*): defines which player has the move in a state
  - Actions(*s*): returns the set of legal moves in a state
  - Result(*s*, *a*): the transition model
  - Terminal-test(*s*): is true when the game is over and false otherwise
  - Utility(*s*, *p*): a utility / objective function
- A solution for a player is a policy which specifies each move.

Game Tree

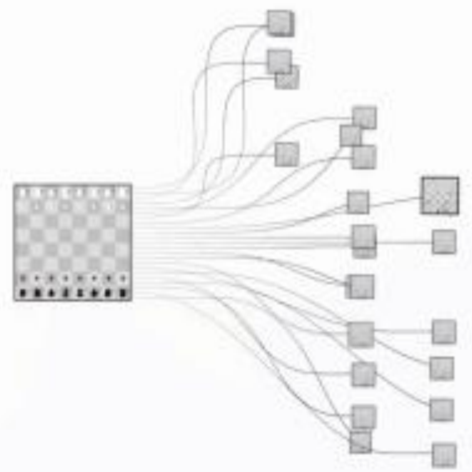
- The initial state, actions, and results define the game tree



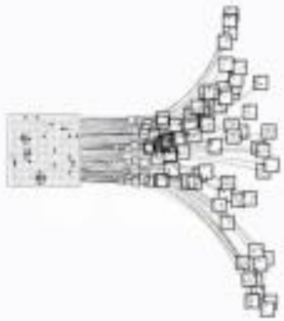
- 9!=362880 terminal nodes

Game Tree

- The initial state, actions, and results define the game tree



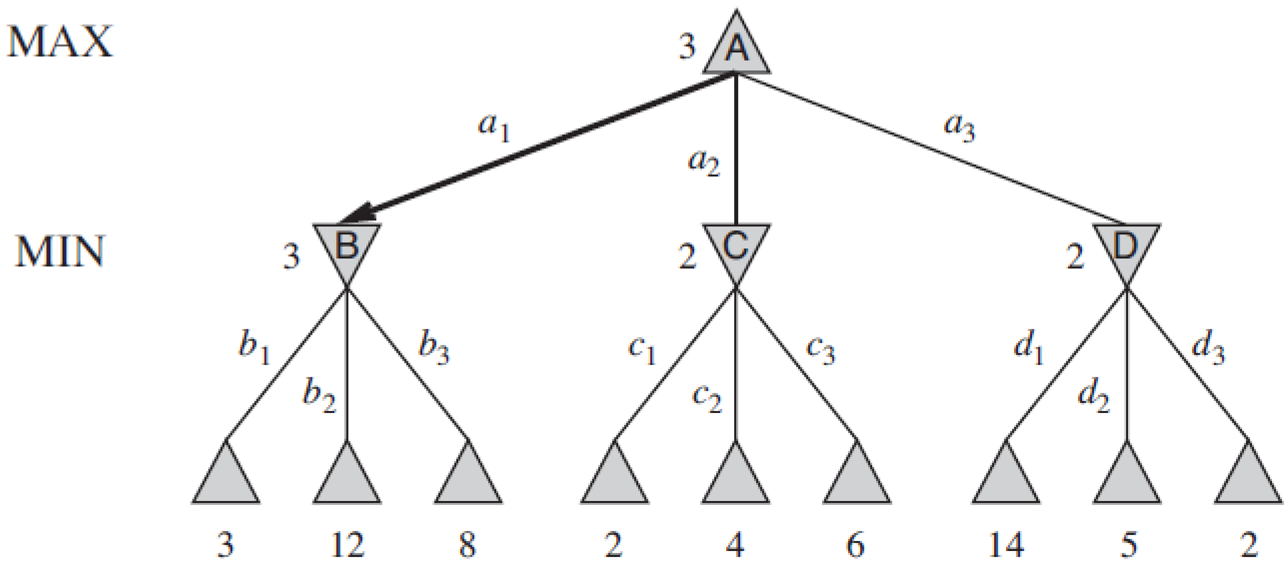
- $b \approx 35, \ \ m \approx 100$
- $O(b^m)$



- $b \approx 250, \ \ m \approx 150$
- $O(b^m)$

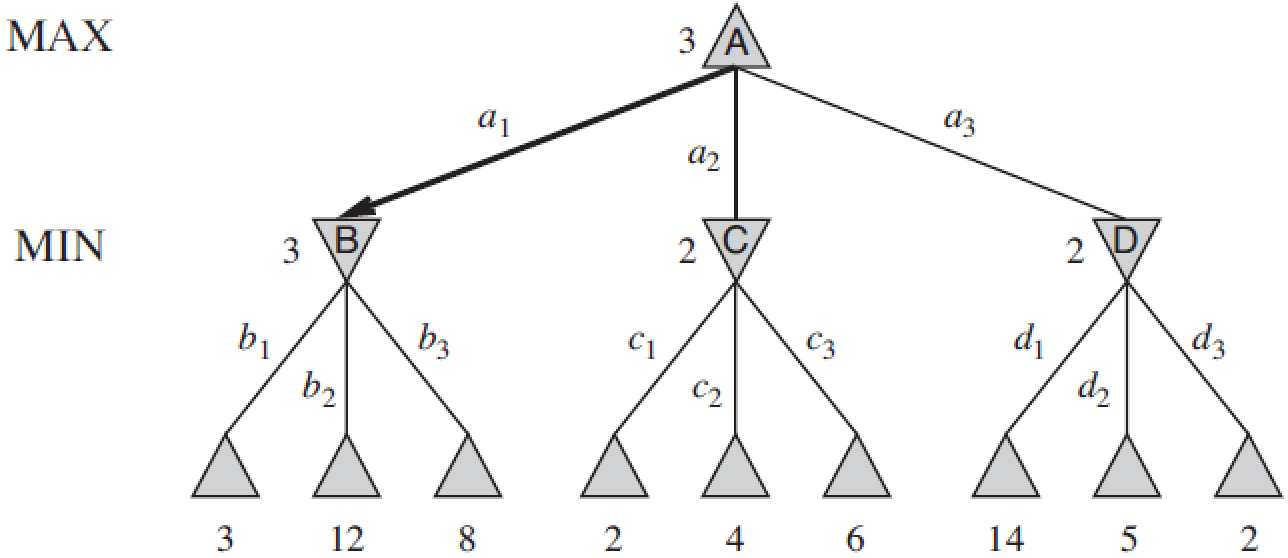
Minimax Search

- A game tree
- Players alternate turns
- Compute each node's minimax value:
  - The minimax value is the utility of being in the state, assuming that both players play optimally from there to the end of the game.



Minimax Search

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$



The Minimax Algorithm

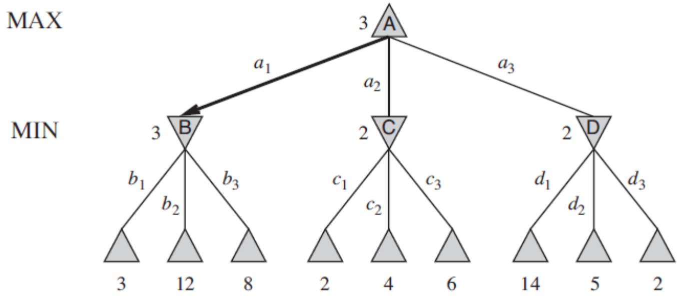
function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

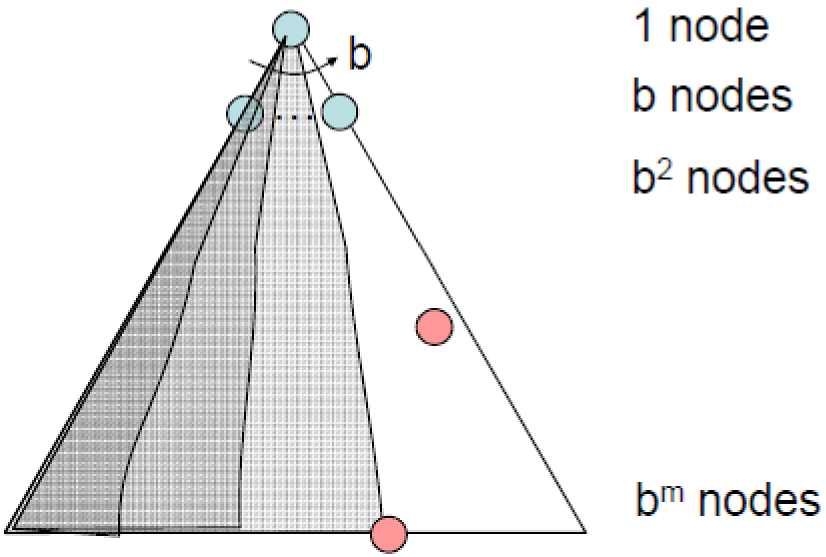
function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v



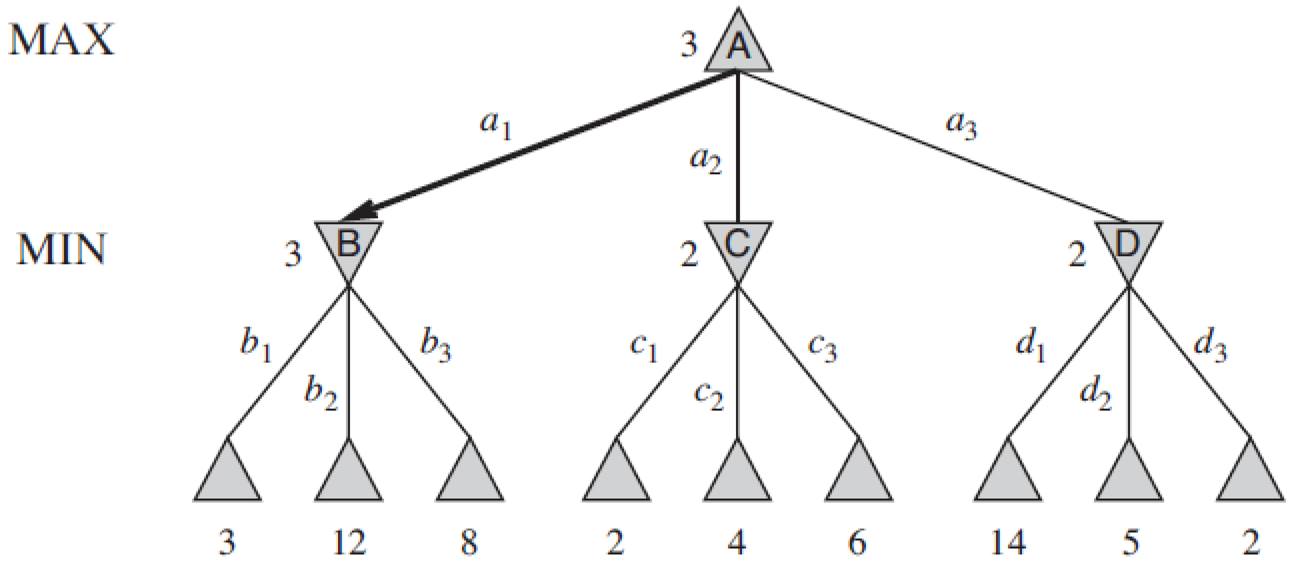
Minimax Search

- Performance
  - Performs a complete depth-first exploration
  - Time complexity: $O(b^m)$
  - Space complexity: $O(bm)$
  - Complete? Yes (if tree is finite)
  - Optimal? Yes (against an optimal opponent)

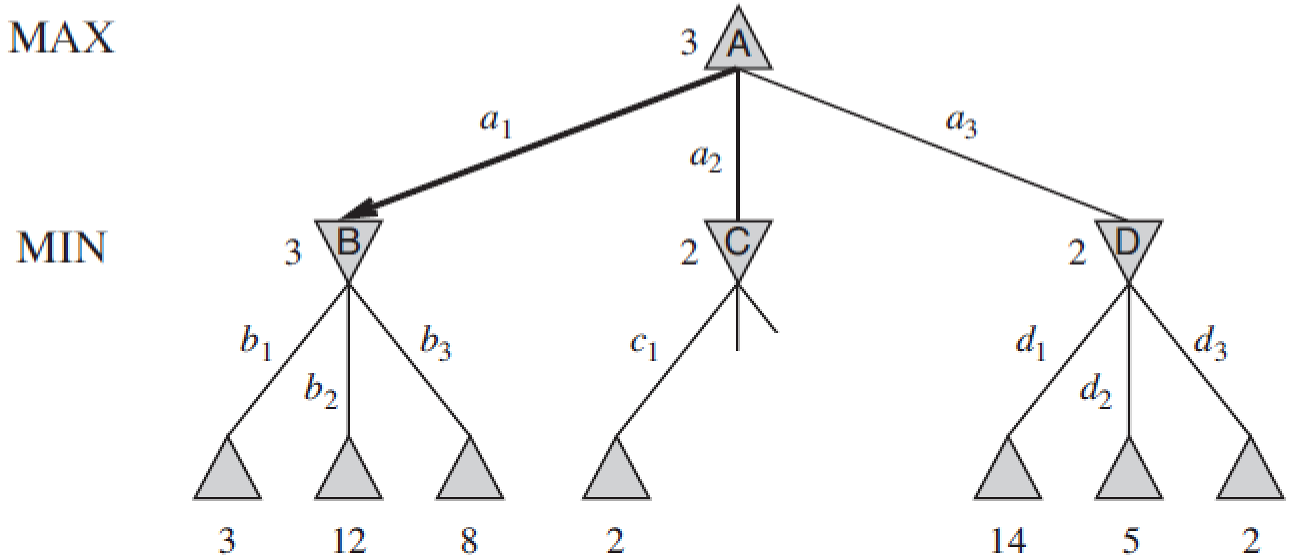
- Examples:
  - Chess, $b \approx 35$, $m \approx 100$
  - Go game, $b \approx 250$, $m \approx 150$



Alpha-Beta Pruning



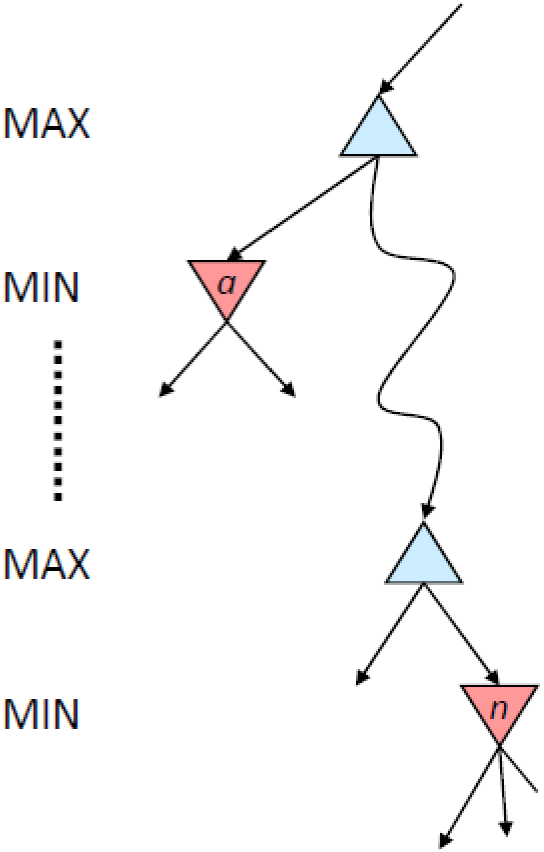
Alpha-Beta Pruning



$$\begin{aligned}
\textsc{Minimax}(\mathit{root}) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3.
\end{aligned}$$

Alpha-Beta Pruning

- General principle:
  - Consider a node n;
  - If MAX has a better choice α either at the parent node of n or at any choice point further up, then n will never be reached in actual play.
  - So once we found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.
- Alpha-Beta Pruning
  - α = the highest value for MAX on path to root
  - β = the lowest value for MIN on path to root



Alpha-Beta Pruning

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

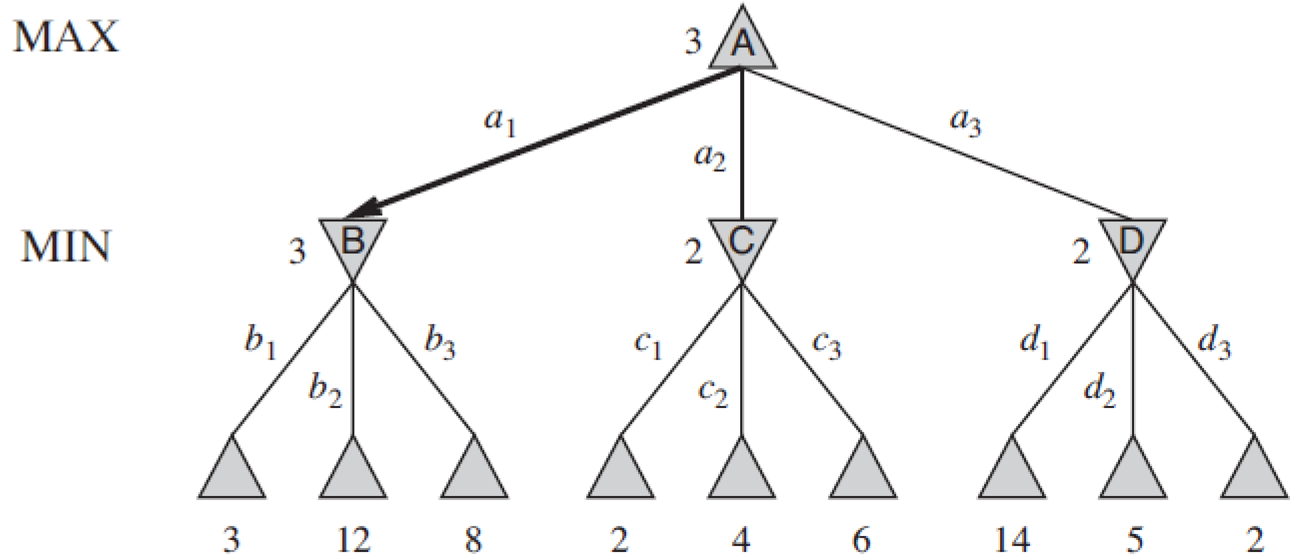
function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-Beta Pruning

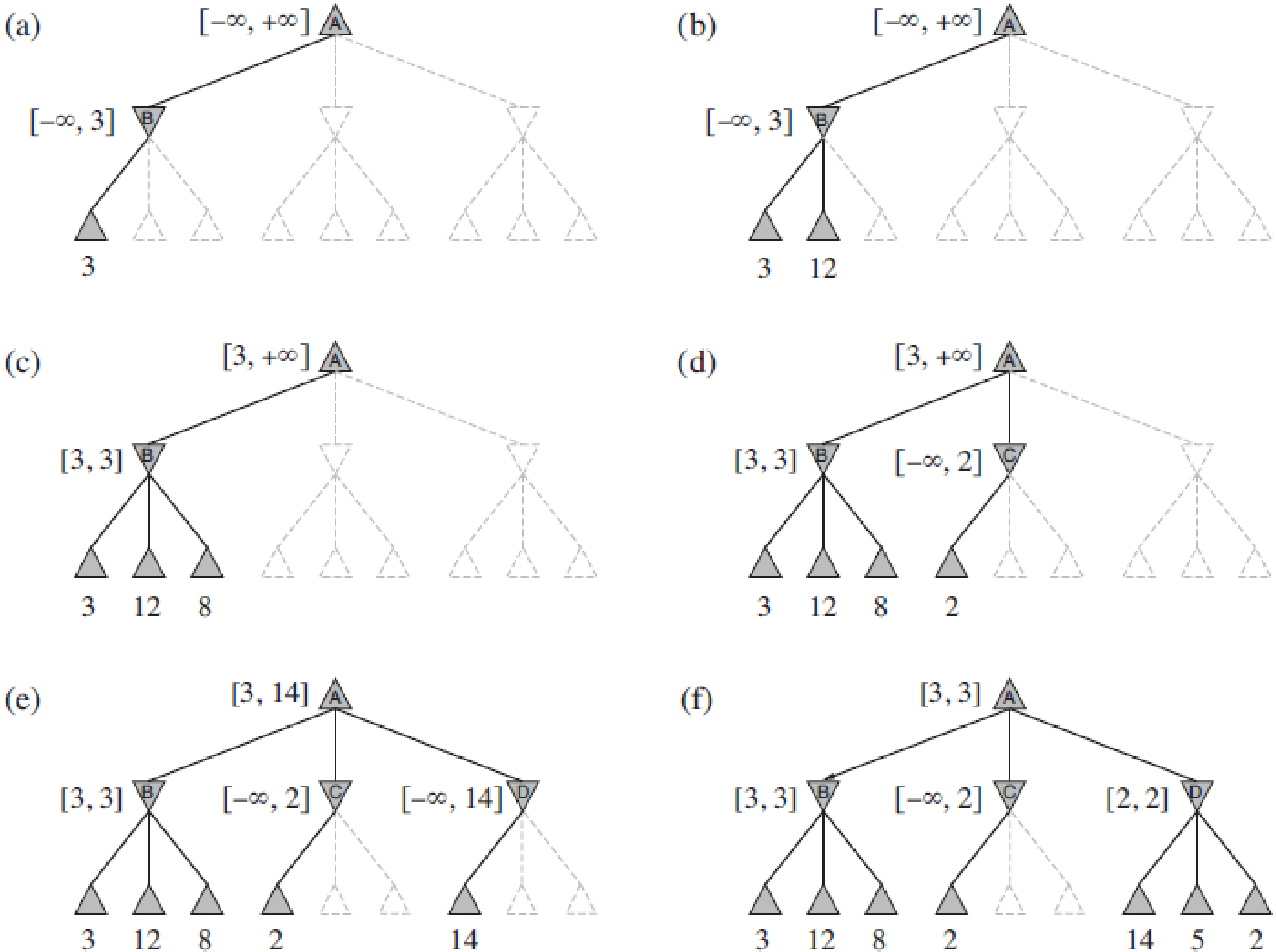
function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

function MAX-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v



Alpha-Beta Pruning



Alpha-Beta Pruning

- The effectiveness of alpha-beta pruning is highly dependent on the order in which the states are examined.
- In perfect case, alpha-beta needs to examine only $O(b^{m/2})$ nodes, instead of $O(b^m)$ for minimax.
- In random order, alpha-beta needs to examine roughly $O(b^{3m/4})$ nodes.

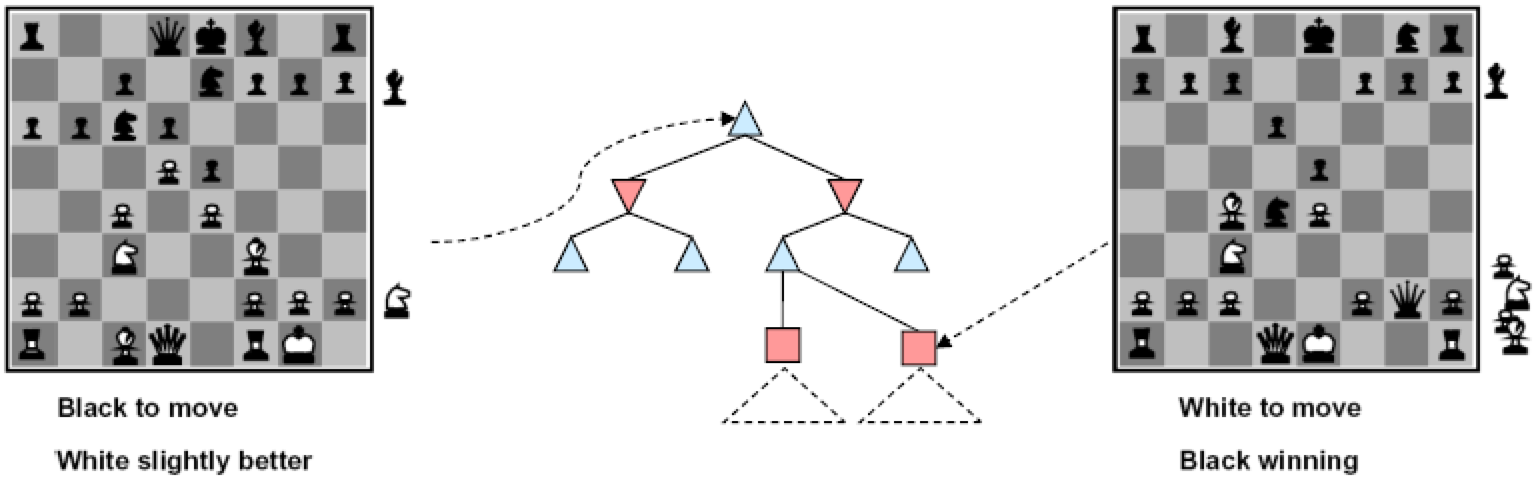
Imperfect Real-time Decisions

- Problem: In realistic games, cannot search to leaves !
- Solution 1: Depth-limited search
  - Search only to a limited depth in the tree
  - Replace terminal utilities with a heuristic evaluation function for non-terminal positions

$$\text{H-MINIMAX}(s, d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

Evaluation Functions

- An evaluation function returns an estimate of the expected utility of the game from a given position.



- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

- E.g. $f_1(s)$ = (num white queens – num black queens)

Evaluation Functions

- Evaluation function are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

Imperfect Real-time Decisions

- Problem: In realistic games, cannot search to leaves !
- Solution 2: Monte Carlo Tree Search
  - MTCS builds a statistics tree that partially maps onto the entire game tree
  - Statistics tree guides to "look only/mostly at the most interesting nodes in the game tree"
  - Value of nodes determined by simulations

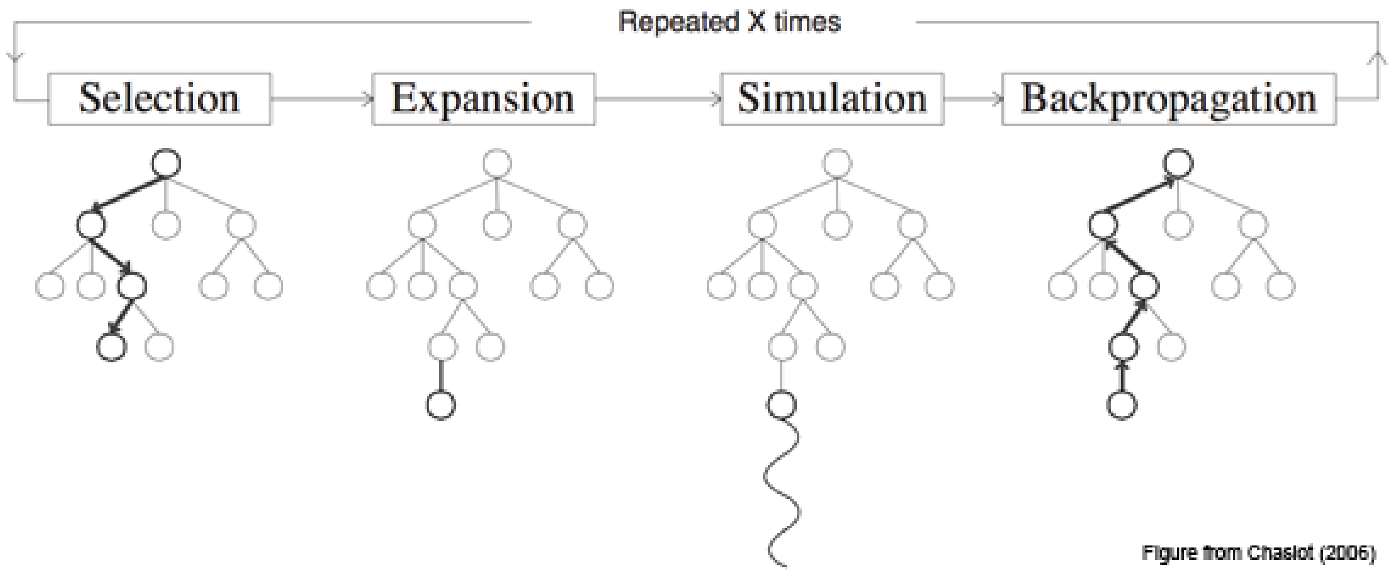
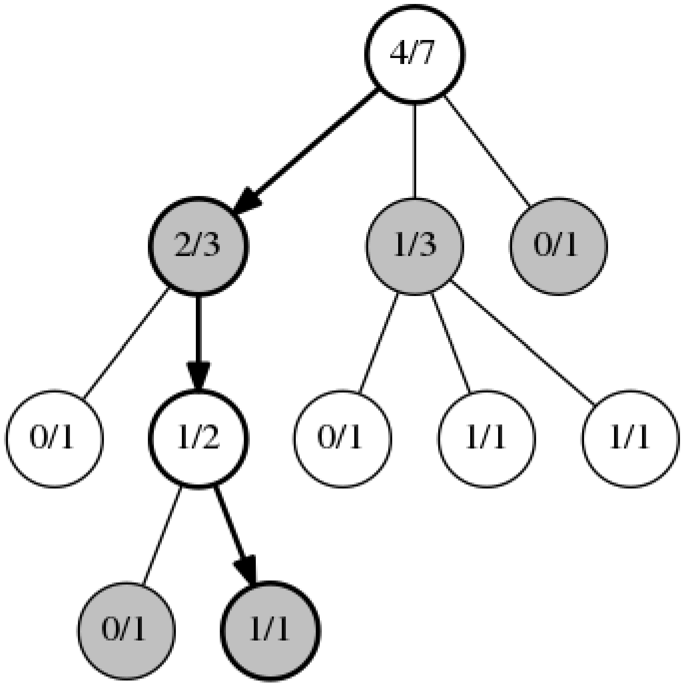


Figure from Chaslot (2006)

Monte Carlo Tree Search

- Selection ➡ Expansion ➡ Simulation ➡ Backpropagation

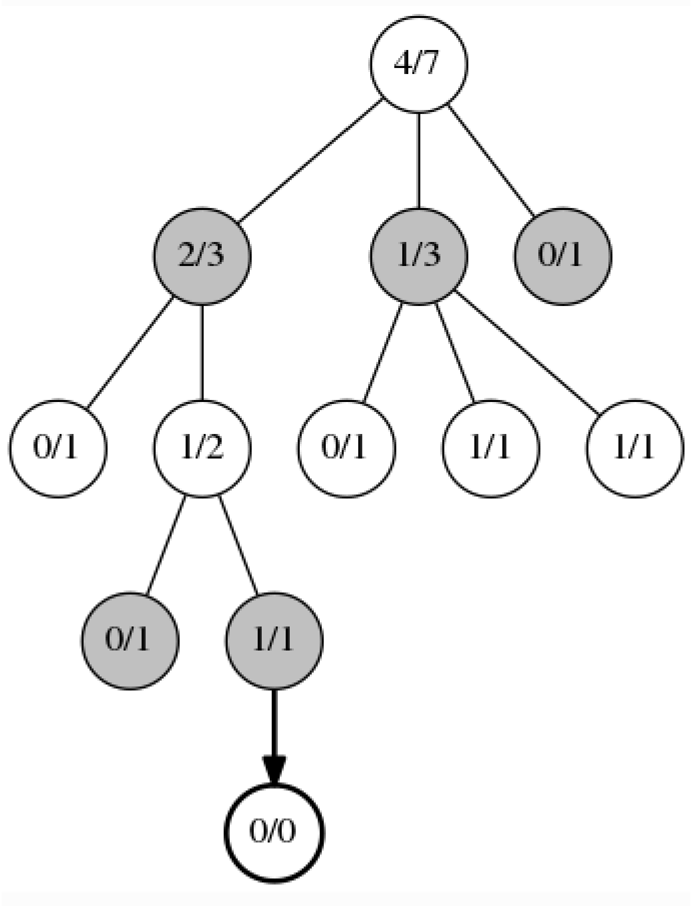


Selection

Starting at root node R, recursively select optimal child nodes until a leaf node L is reached.

Monte Carlo Tree Search

- Selection ➡ Expansion ➡ Simulation ➡ Backpropagation

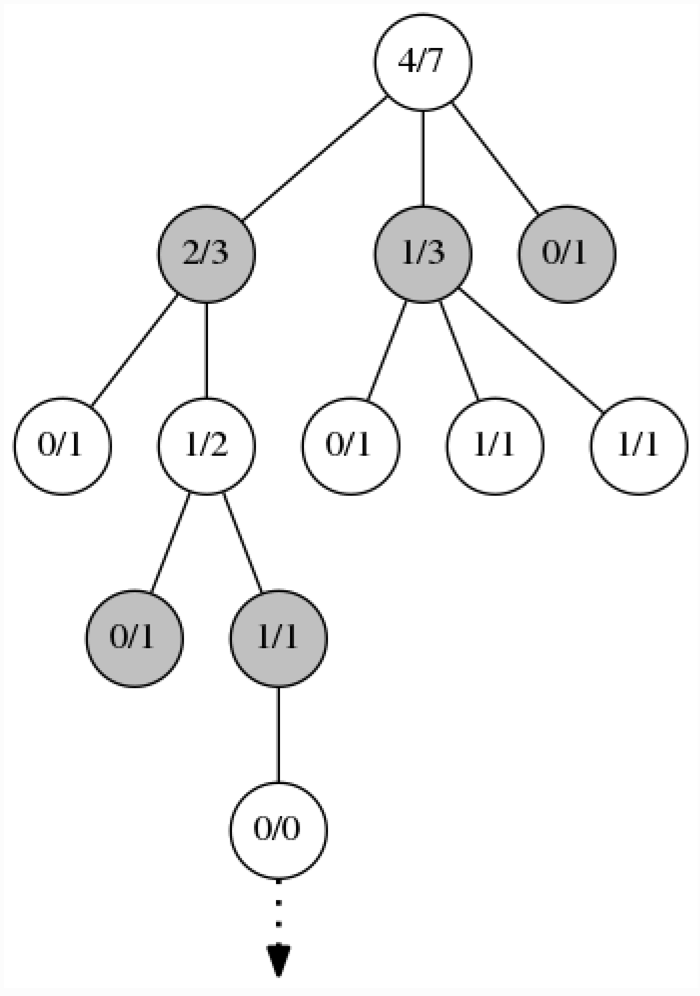


Expansion

If L is a not a terminal node then create one or more child nodes and select one C.

Monte Carlo Tree Search

- Selection ➡ Expansion ➡ Simulation ➡ Backpropagation



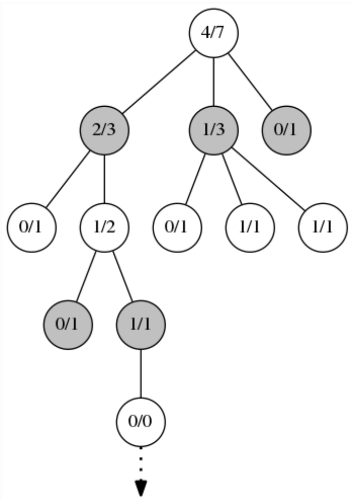
Simulation

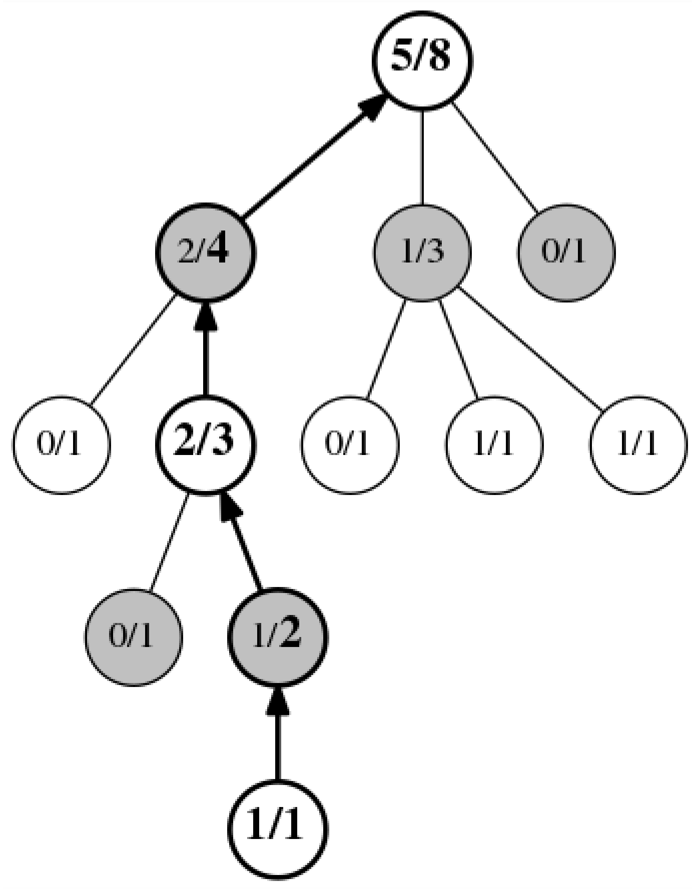
Run a simulated playout from C until a result is achieved.

Rollout: Randomly choose an action at each step and simulate this action to receive an average reward when the game is over.

Monte Carlo Tree Search

- Selection ⟹ Expansion ⟹ Simulation ⟹ Backpropagation





Backpropagation

Update the current move sequence with the simulation result.

Adversarial Search

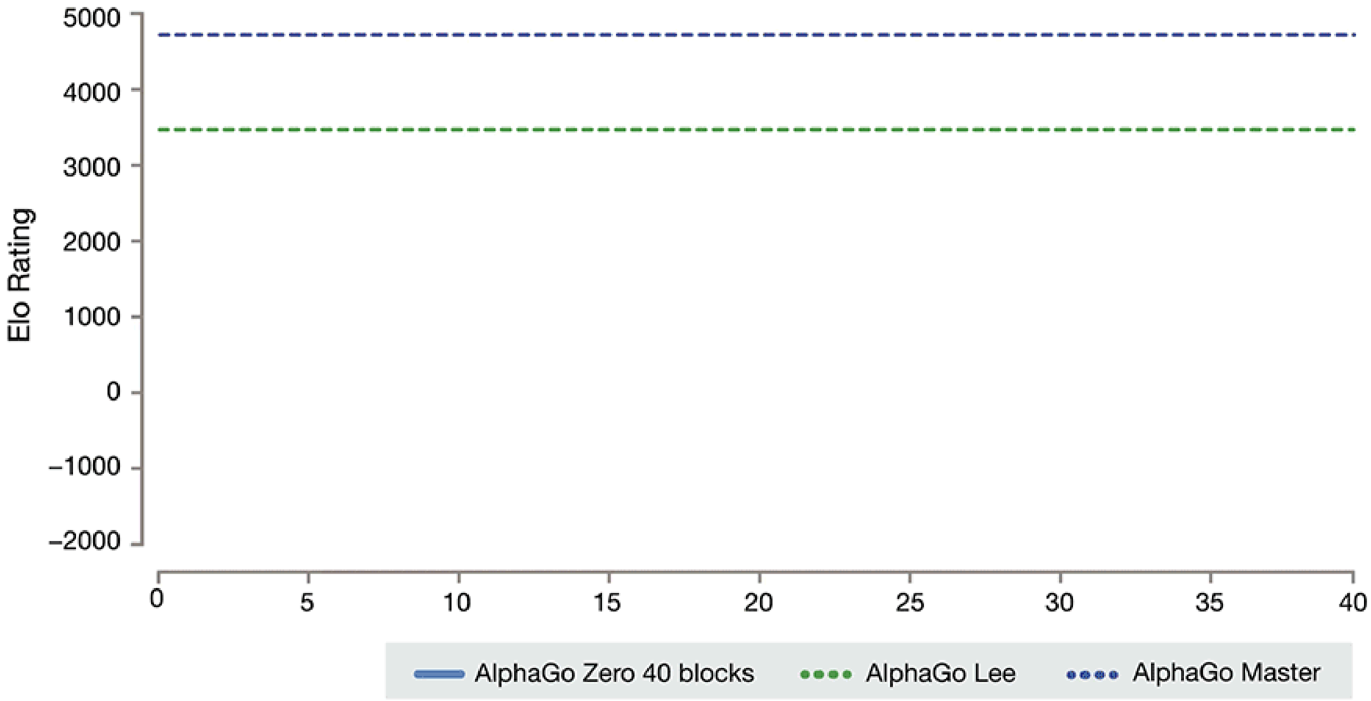
1. A. S. Douglas programmed the **first software that managed to master a game in 1952**. The game? Tic-Tac-Toe! This was part of his doctoral dissertation at Cambridge
2. ***A few years later, Arthur Samuel was the first to use reinforcement learning that to play Checkers by playing against itself***
3. In 1992, Gerald Tesauro designed a now-popular **program called TD-Gammon to play backgammon at a world-class level**
4. For decades, Chess was seen as "the ultimate challenge of AI". **IBM's Deep Blue was the first software that exhibited superhuman Chess capability.** The system famously defeated Garry Kasparov, the reigning grandmaster of chess, in 1997
5. **One of the most popular board game AI milestones was reached in 2016 in the game of Go**. Lee Sedol, a 9-dan professional Go player, lost a five-game match against Google DeepMind's AlphaGo software which featured a deep reinforcement learning approach
6. Notable recent milestones in video game AI include **algorithms developed by Google DeepMind to play several games from the classic Atari 2600** video game console at a super-human skill level
7. Last year, OpenAI built the popular OpenAI Five system that mastered the complex strategy game of DOTA

https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/

AlphaGo

2016. 3

2017. 5



AlphaGo

ARTICLE

NATURE | VOL 529 | 28 JANUARY 2016
doi:10.1038/nature16961

Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

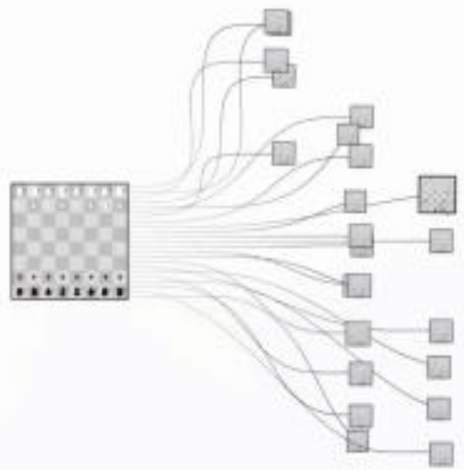
ARTICLE

NATURE | VOL 550 | 19 OCTOBER 2017
doi:10.1038/nature24270

Mastering the game of Go without human knowledge

David Silver[1]*, Julian Schrittwieser[1]*, Karen Simonyan[1]*, Ioannis Antonoglou[1], Aja Huang[1], Arthur Guez[1], Thomas Hubert[1], Lucas Baker[1], Matthew Lai[1], Adrian Bolton[1], Yutian Chen[1], Timothy Lillicrap[1], Fan Hui[1], Laurent Sifre[1], George van den Driessche[1], Thore Graepel[1] & Demis Hassabis[1]

AlphaGo



- $b \approx 35, \quad m \approx 100$
- $O(b^m)$

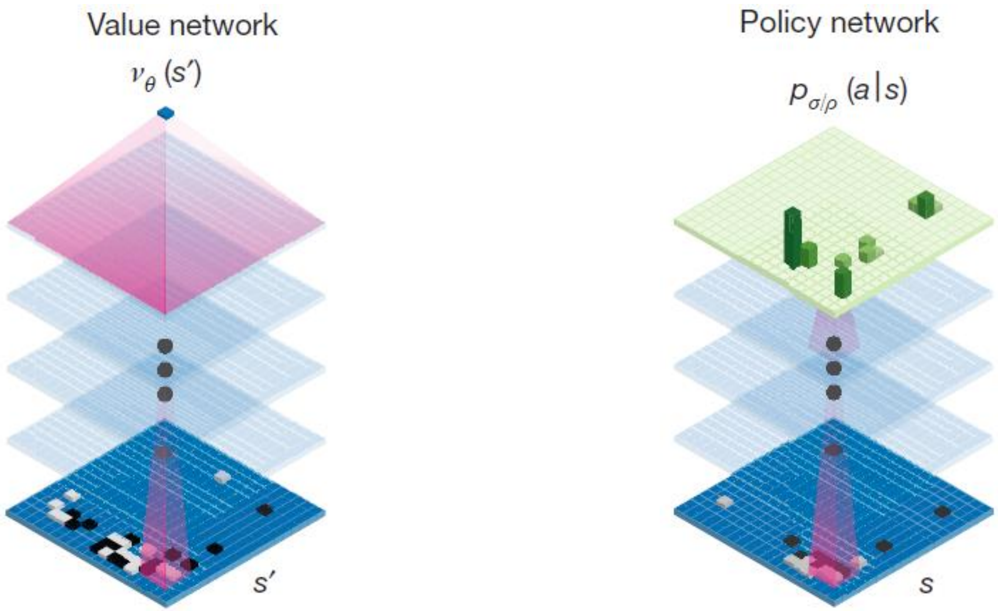
- $b \approx 250, \quad m \approx 150$
- $O(b^m)$

AlphaGo

- The core parts of AlphaGo:
  - Convolutional Neural Networks:
    - Evaluates new positions & moves
  - Reinforcement learning:
    - Trains the AI by using the current best agent to play against itself
  - Monte Carlo Tree Search:
    - Chooses the next move

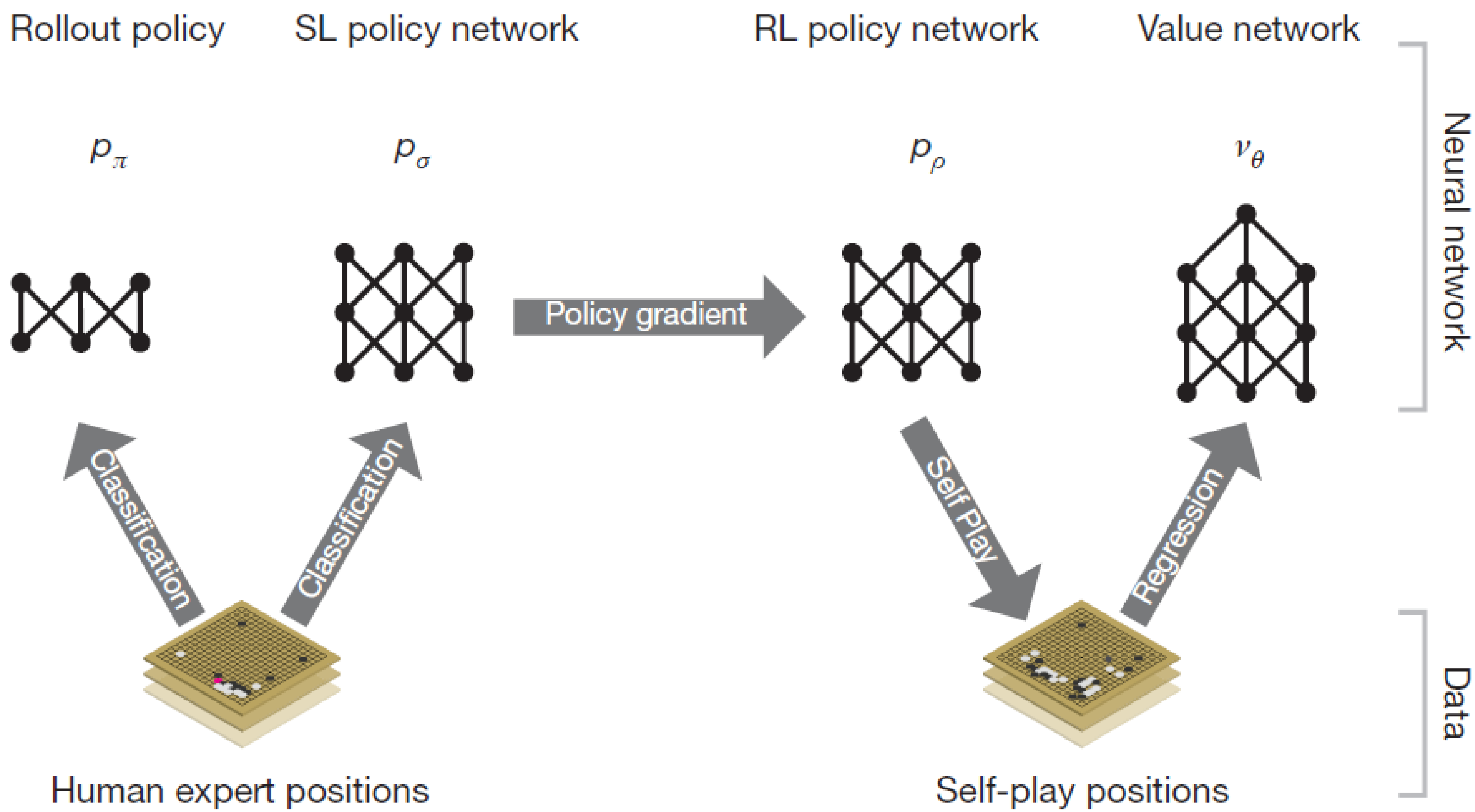
AlphaGo

- Two policies:
  - The depth of the search is reduced by position evaluation.
  - The breath of the search is reduced by sampling actions from a policy $p(a|s)$.
- Two networks:
  - Value network to evaluate board position.
  - Policy network to select moves.



AlphaGo

- Reinforcement learning



AlphaGo

- Monte Carlo tree search in AlphaGo

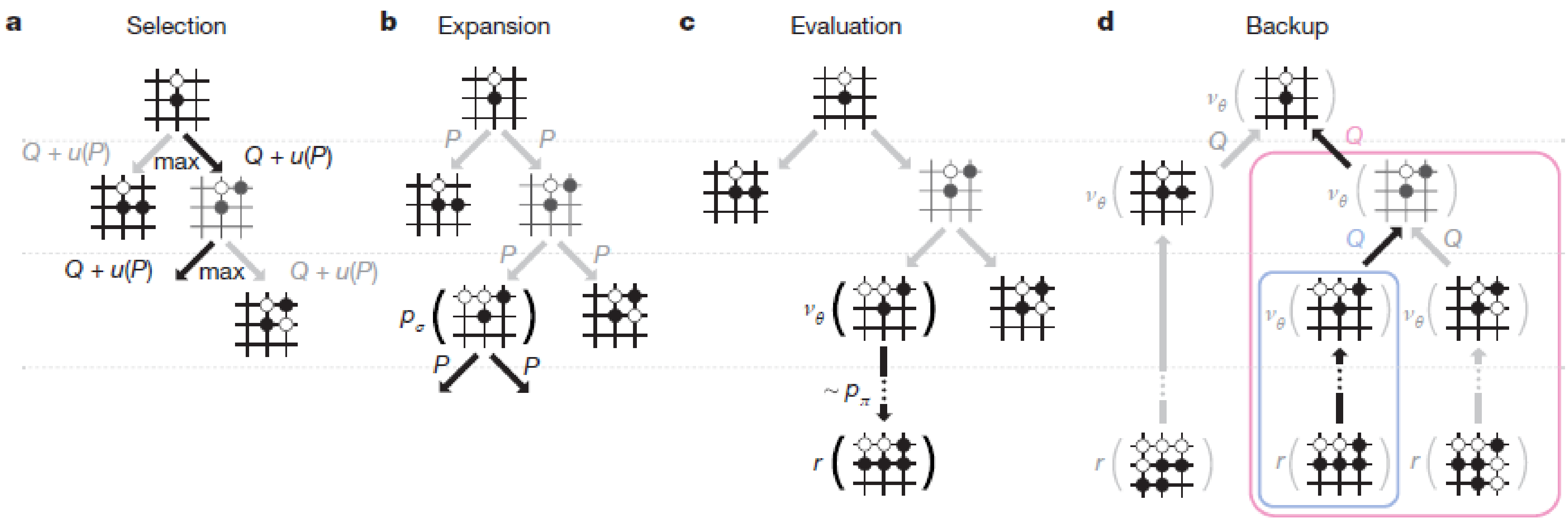
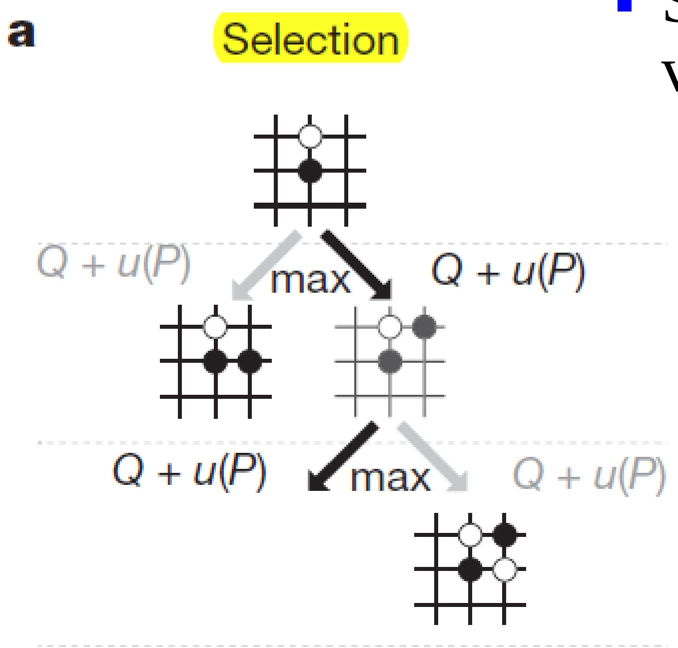


Figure 3 | Monte Carlo tree search in AlphaGo.

AlphaGo

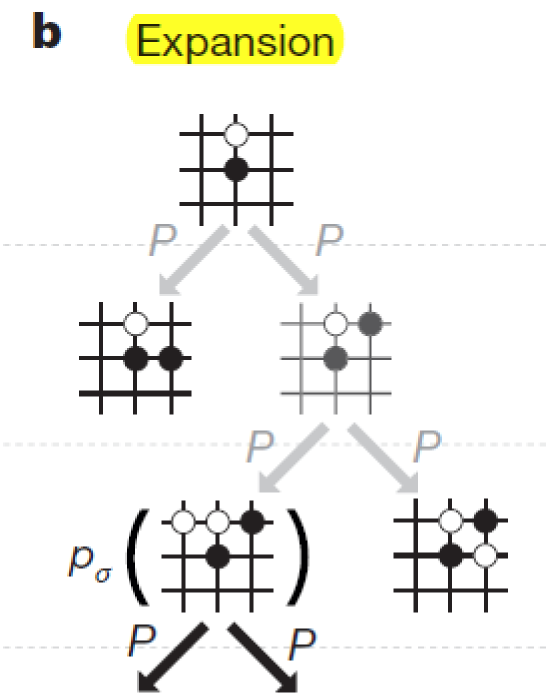
- Monte Carlo tree search in AlphaGo



- Selecting the edge with maximum action value *Q* plus a bonus *u(P)*.

AlphaGo

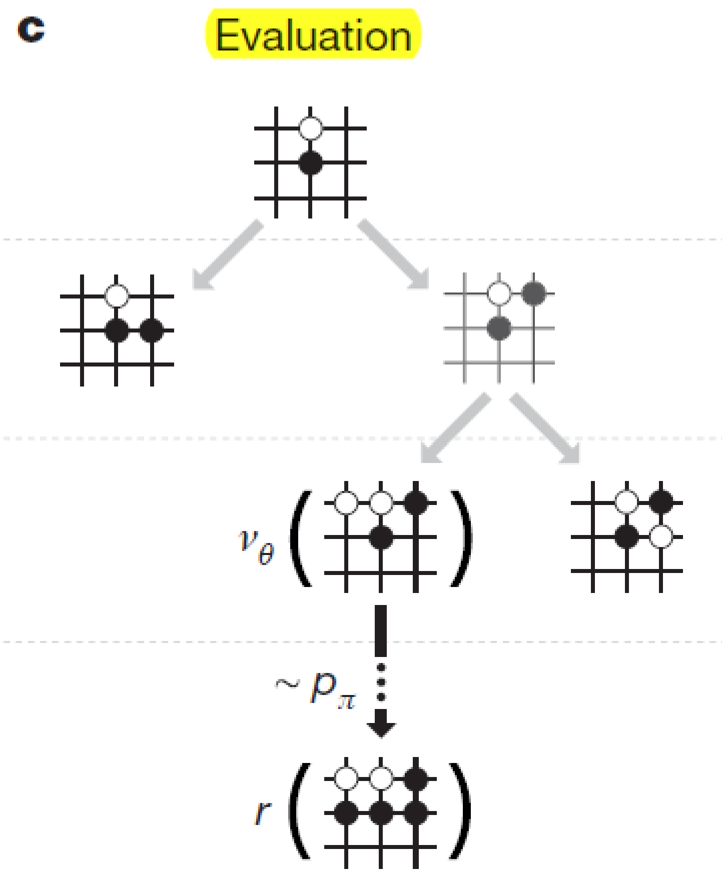
- Monte Carlo tree search in AlphaGo



- The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action.

AlphaGo

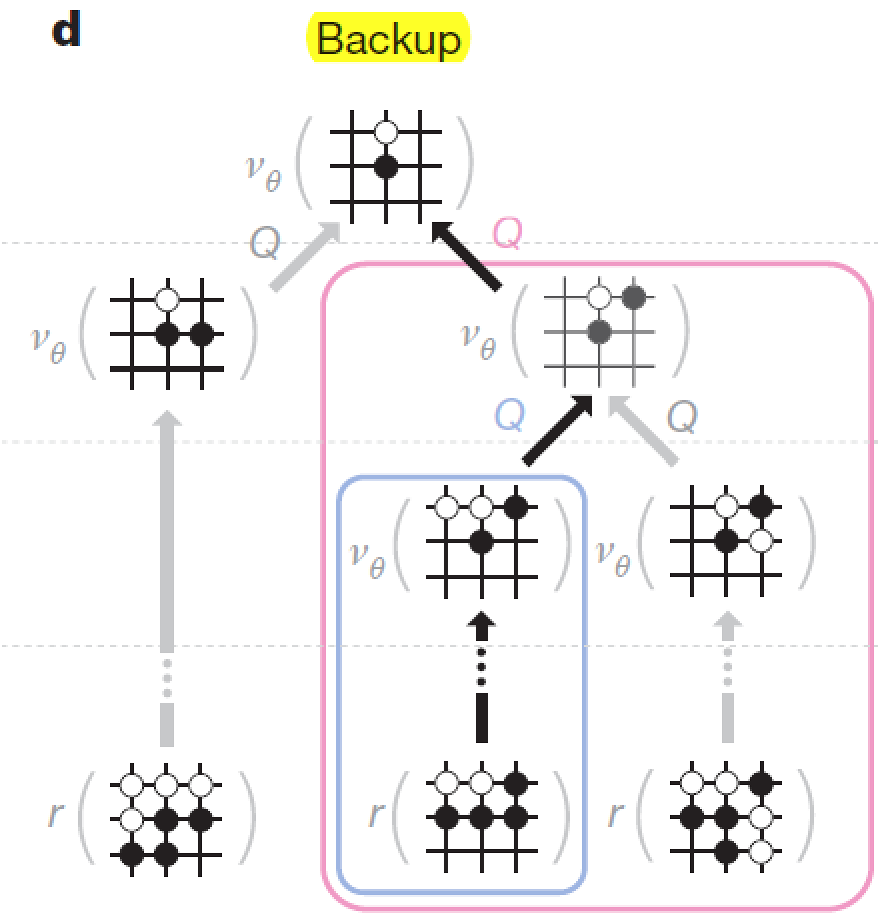
- Monte Carlo tree search in AlphaGo



- At the end of a simulation, the leaf node is evaluated in two ways:
  - using the value network v_θ;
  - running a rollout to the end of the game with the fast rollout policy p_π;
  - then computing the winner with function r.

AlphaGo

- Monte Carlo tree search in AlphaGo



- Action values *Q* are updated to track the mean value of all evaluations *r(·)* and $v_\theta(\cdot)$ in the subtree below that action.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s_L^i)$$

$$N(s,a) = \sum_{i=1}^{n} 1(s,a,i)$$

$$V(s_L) = (1-\lambda) v_\theta(s_L) + \lambda z_L$$

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$$

$$P(s,a) = p_\sigma(a|s)$$

Assignment

- Reading assignment:
  - Ch. 5.1-5.5
- Homework 2:
  - Due by Mar. 14, 2022.
- Project 1:
  - Due by Mar. 21, 2022.