Announcement

- **Correction**: HW3 will be out after HW2 due
- Quiz is this Thur – will be light, ~half size of Quiz 1

DATA 37200: Learning, Decisions, and Limits (Winter 2025)

Solving Zero-Sum Sequential Games

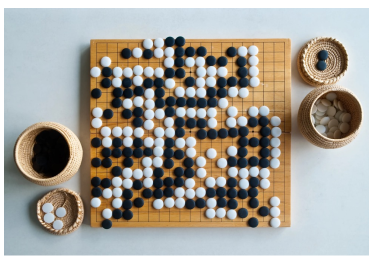
Instructor: Haifeng Xu

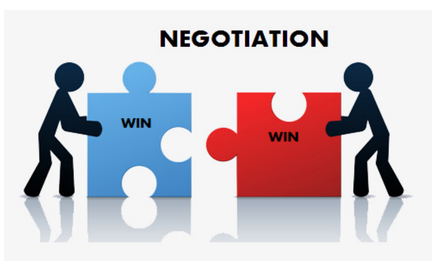
Outline

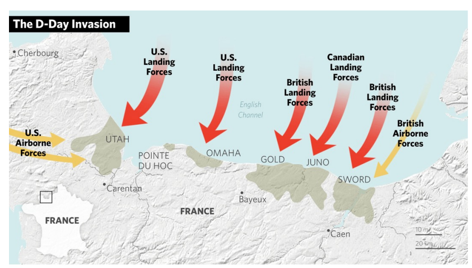
- Sequential Games and Extensive-Form Representations
- Solving Complete-Information Games
- Solving Incomplete-Information Games

Many "Real" Games Are Sequential

- Entertainment games: Checker, Chess, Go, Poker, StarCraft, etc.
- Negotiation
- Interactions in adversarial/military environments
- Political campaigns ...







Many "Real" Games Are Sequential

- Entertainment games: Checker, Chess, Go, Poker, StarCraft, etc.
- Negotiation
- Interactions in adversarial/military environments
- Political campaigns …

This lecture focuses on strictly competitive situations – **zero-sum**.

- ✓ Appears widely
- ✓ A great ground for applying online/reinforcement learning
- ✓ General-sum games are much more difficult to solve

To Begin With…

Sequential games do cricially differ from simultaneous-move games

- What is the NE if A,B move simultaneously?
  - (a_2, b_2) is the unique Nash, resulting in utility pair (1,2)
- If A moves first; B sees A's move and then best responds, how should A play?
  - Play action a_1 deterministically!
  - B will respond optimally with b_1

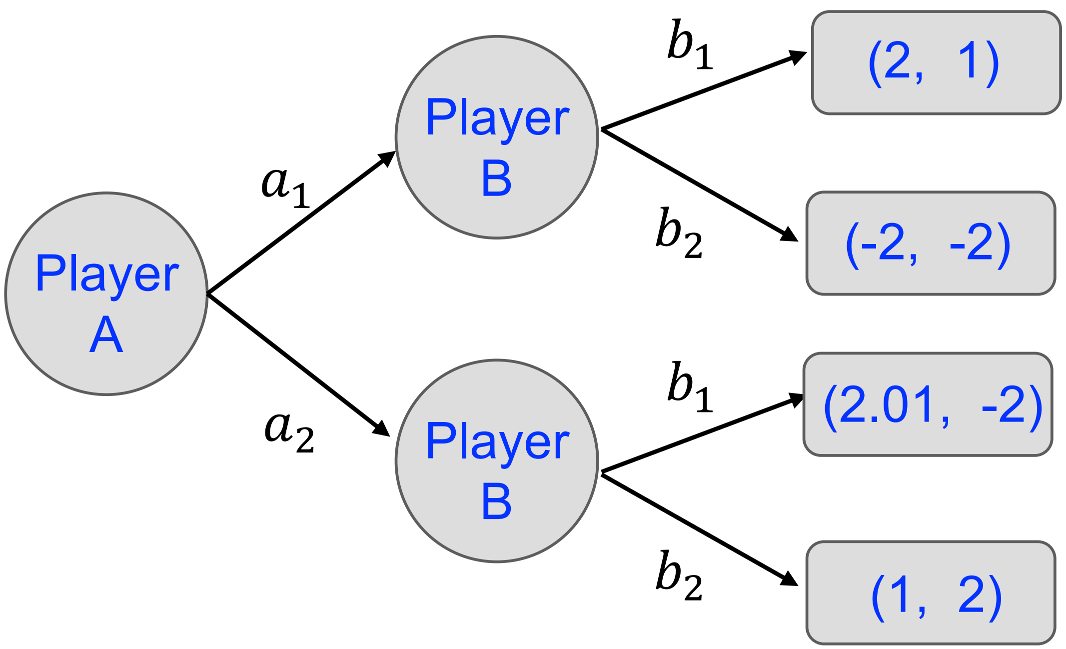
B

A		b_1	b_2
	a_1	(2, 1)	(-2, -2)
	a_2	(2.01, -2)	(1, 2)

Representing Sequential Games in Extensive Form

Also known as **extensive-form games**

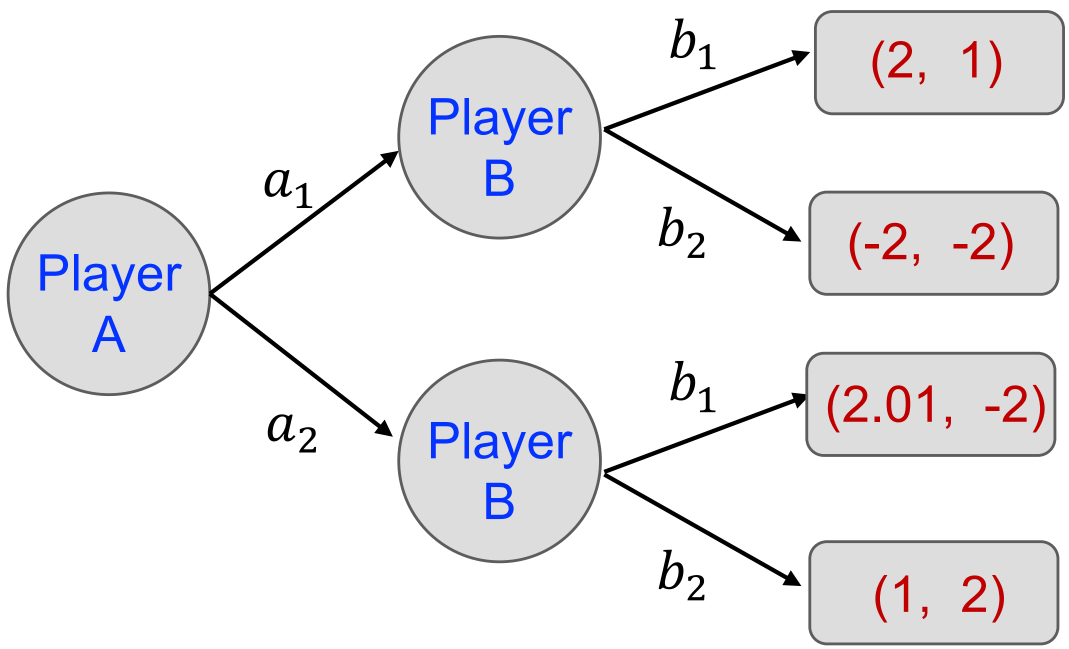
Represented via a tree structure in which directions indicate move orders



A \ B	b_1	b_2
a_1	(2, 1)	(-2, -2)
a_2	(2.01, -2)	(1, 2)

Representing Sequential Games in Extensive Form

- Each leaf node is called terminal state $z \in Z$
  - I.e., game terminates here
  - In Go, this is where game ends
  - Player i'th utility function $u_i(z)$
  - Two-player zero-sum: $u_A(z) + u_B(z) = 0, \forall z$



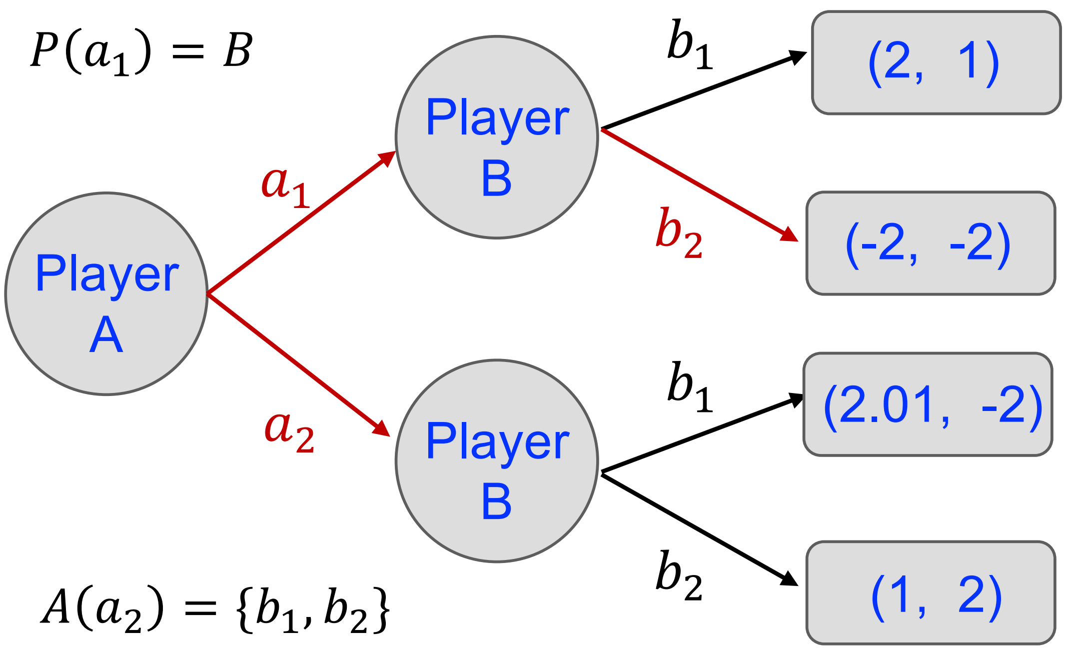
B

A	b_1	b_2
a_1	(2, 1)	(-2, -2)
a_2	(2.01, -2)	(1, 2)

Representing Sequential Games in Extensive Form

- Any (possibly partial) trajectory is called a history $h \in H$
  - A history can consist of moves by multiple players
  - Let $H_i = \{h \in H : P(h) = i\}$ denote those associated with i
  - Notably, can think of terminal states $Z \subset H$
- Each *non-terminal* history h corresponds to
  - a player $P(h) \in \{A, B\}$ who moves next
  - An action set $A(h)$ available to player $P(h)$

$P(a_1) = B$



$A(a_2) = \{b_1, b_2\}$

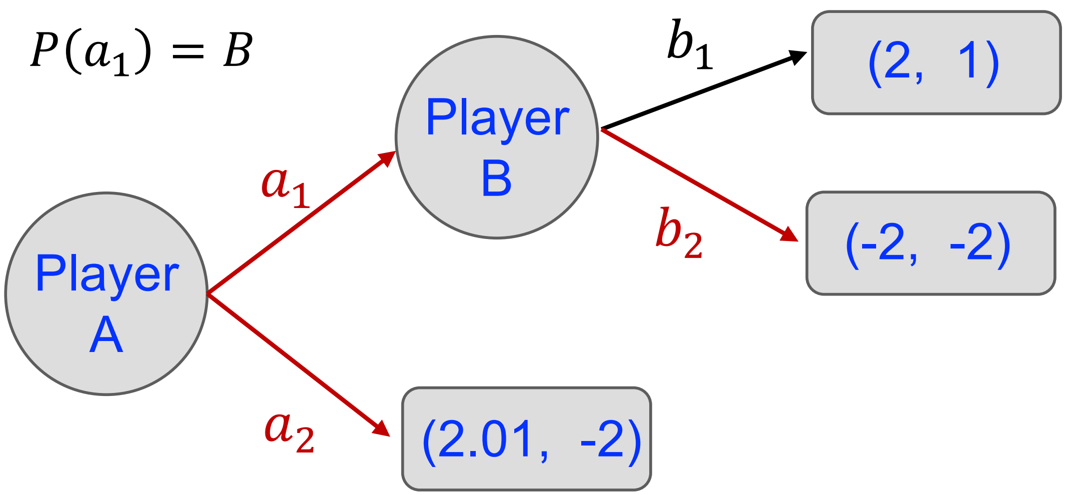
A \ B	b_1	b_2
a_1	(2, 1)	(-2, -2)
a_2	(2.01, -2)	(1, 2)

An EFG does not need to be symmetric

Representing Sequential Games in Extensive Form

- Any (possibly partial) trajectory is called a history $h \in H$
  - A history can consist of moves by multiple players
  - Let $H_i = \{h \in H: P(h) = i\}$ denote those associated with i
  - Notably, can think of terminal states $Z \subset H$
- Each *non-terminal* history h corresponds to
  - a player $P(h) \in \{A, B\}$ who moves next
  - An action set $A(h)$ available to player $P(h)$

$P(a_1) = B$



History a_2 is a terminal state here

B

A	b_1	b_2
a_1	(2, 1)	(-2, -2)
a_2	(2.01, -2)	(1, 2)

An EFG does not need to be symmetric

From Extensive Form to Normal Form

Claim 1. Any extensive form game can be converted to an "equivalent" normal-form game

Idea: enumerate each player's action choices for **every associated history**

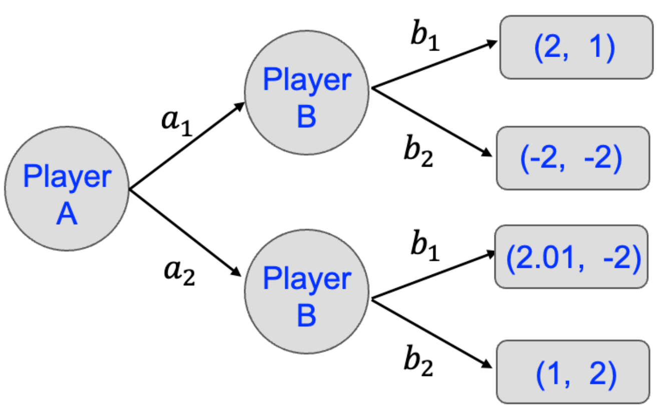
B

A		(b_1, b_1)	(b_1, b_2)	(b_2, b_1)	(b_2, b_2)
	a_1	(2, 1)	(2, 1)	(-2, -2)	(-2, -2)
	a_2	(2.01, -2)	(1, 2)	(2.01, -2)	(1, 2)

B's action under a_1

B's action under a_2

- B acts upon two possible histories
- Two choices at each situation



From Extensive Form to Normal Form

Claim 1. Any extensive form game can be converted to an "equivalent" normal-form game

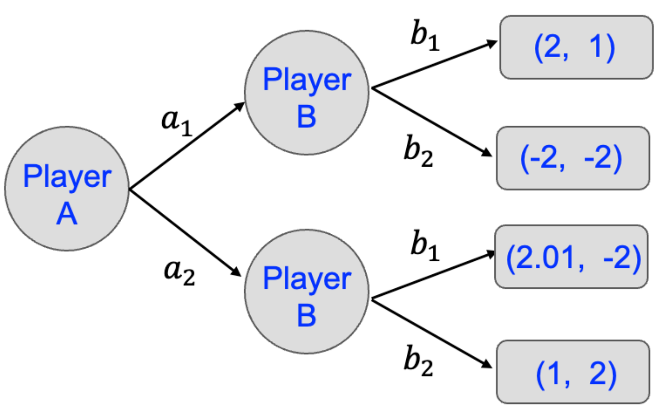
Idea: enumerate each player's action choices for **every associated history**

B

A		(b_1, b_1)	(b_1, b_2)	(b_2, b_1)	(b_2, b_2)
	a_1	(2, 1)	(2, 1)	(-2, -2)	(-2, -2)
	a_2	(2.01, -2)	(1, 2)	(2.01, -2)	(1, 2)

B's action under a_1

B's action under a_2



What's the NE for the above normal-form game?

- ✓ Recall: in previous sequential move, $a_1, (b_1, b_2)$ is a Nash equilibrium
- ✓ $a_1, (b_1, b_2)$ is also a NE in the above game
- ✓ However, $a_1, (b_1, b_2)$ is not the unique NE

From Extensive Form to Normal Form

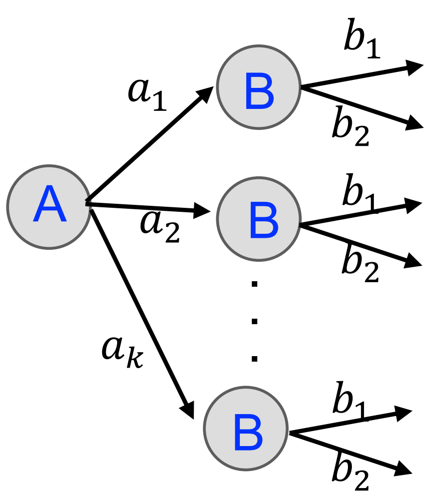
Claim 1. Any extensive form game can be converted to an "equivalent" normal-form game whose size is exponential in the number of nodes

Idea: enumerate each player's action choices for **every associated history**

This is why we need smarter ways to solve extensive-form games

What about this game?

- B's strategy in normal-form representation needs to enumerate choices under every a_i
- Blow up exponentially: 2^k many!

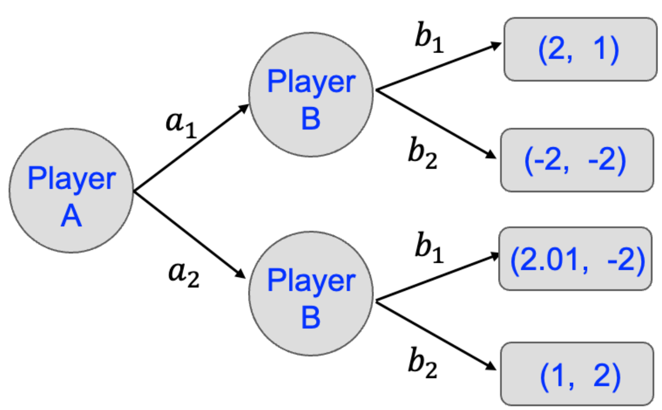


From Normal Form to Extensive Form

Claim 2. Any normal form game can be converted to an equivalent extensive-form game

Idea: allow incomplete information in the extensive form game

- Recall previous representation under sequential move
- To allow simultaneous move, we need the concept of information set



B

A		b_1	b_2
	a_1	(2, 1)	(-2, -2)
	a_2	(2.01, -2)	(1, 2)

From Normal Form to Extensive Form

Claim 2. Any normal form game can be converted to an equivalent extensive-form game

Def. An **information set** I_i is a subset of histories that share the same next-move player $i \in \{A, B\}$ and the same action set. Formally,

$$\forall\, h, h' \in I_i, \qquad P(h) = i \text{ and } A(h) = A(h')$$

Player i cannot distinguish which $h \in I_i$ she is at, hence has to use the same strategy for every $h \in I_i$.

Why cannot distinguish? → There are states that i cannot observe

From Normal Form to Extensive Form

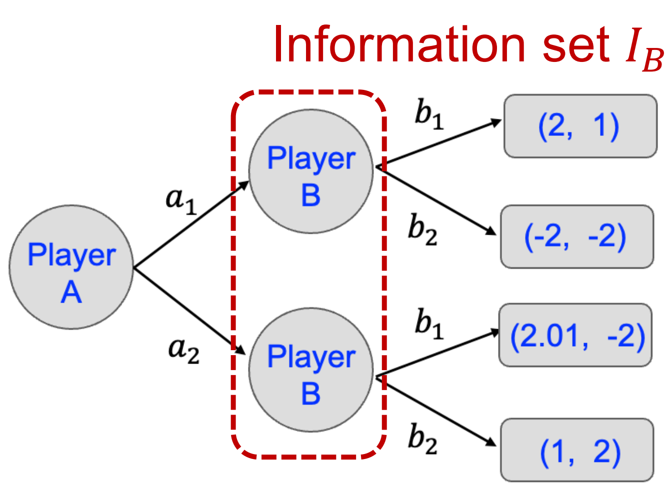
Claim 2. Any normal form game can be converted to an equivalent extensive-form game with incomplete information

Def. An **information set** I_i is a subset of histories that share the same next-move player $i \in \{A, B\}$ and the same action set. Formally,

$$\forall\, h, h' \in I_i, \qquad P(h) = i \text{ and } A(h) = A(h')$$

Player i cannot distinguish which $h \in I_i$ she is at, hence has to use the same strategy for every $h \in I_i$.

Why cannot distinguish? → There are states that i cannot observe

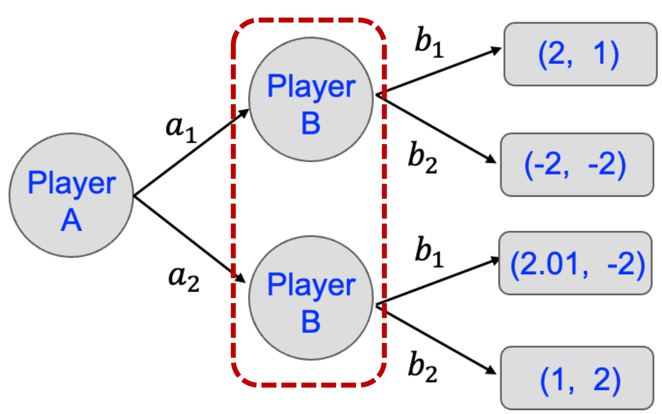


→ B cannot observe what action A took, making the game effectively simultaneous move

A \ B	b_1	b_2
a_1	(2, 1)	(-2, -2)
a_2	(2.01, -2)	(1, 2)

Recap on What We Have Thus Far

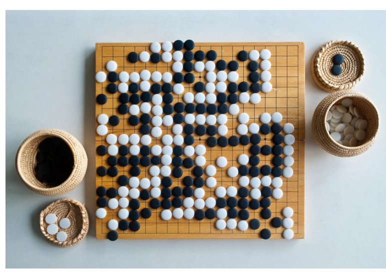
- Extensive form game (EFG) with incomplete information
  - ✓ A powerful class of games that capture most entertainment games and many games in real life (e.g., negotiation, military planning, etc.)
- Consists of
  - ✓ Terminal states, and associated player utilities
  - ✓ History of moves, associated next-to-move player and available actions
  - ✓ Information set $I_i \subset H_i$, which captures a player i's incomplete information
- EFG can be converted to a normal form game but inefficient, and any normal-form game can be converted to an EFT



A \ B	b_1	b_2
a_1	(2, 1)	(-2, -2)
a_2	(2.01, -2)	(1, 2)

Solving EFGs

- Had a long history in AI
- Techniques are useful for improving reasoning (even for LLMs)
- Similar in spirit to RL in MDPs
  - ✓ Player strategies → policy
  - ✓ Utility → rewards
  - ✓ Information set → uncertainty of future states
- Having incomplete information (i.e., information set) or not matters a lot to the problem's complexity



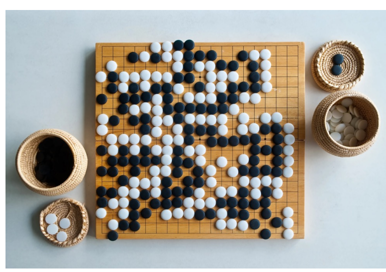
Complete information EFGs


Incomplete information EFGs

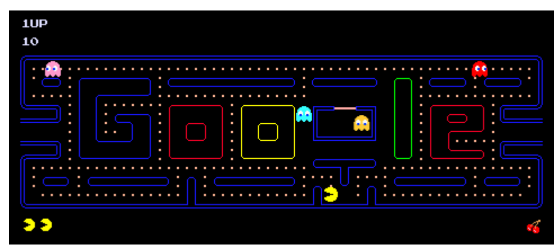
Outline

- Sequential Games and Extensive-Form Representations
- Solving Complete-Information Games
- Solving Incomplete-Information Games

Chess



Go



Pacman

Tic-Tac-Toe

Will Cover Two Algorithms

“Solving” = finding Nash equilibrium strategy (i.e., Maximin) for one player

1. Minimax Search
   - The core algorithm framework for IBM’s deep blue
   - Real implementation has lots of speed-up improvements via expert knowledge
2. Monte-Carlo Tree Search (MCTS)
   - The core algorithmic framework for AlphaGo
   - Deep RL played a key role

Will Cover Two Algorithms

"Solving" = finding Nash equilibrium strategy (i.e., Maximin) for one player

1. Minimax Search
   - The core algorithm framework for IBM's deep blue
   - Real implementation has lots of speed-up improvements via expert knowledge
2. Monte-Carlo Tree Search (MCTS)
   - The core algorithmic framework for AlphaGo
   - Deep RL played a key role

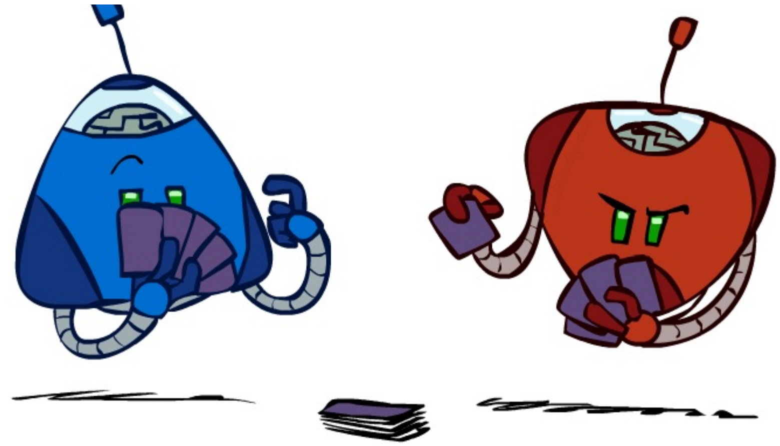
Remarks.

- ➢ Go is much more complex to play than Chess
- ➢ Minimax is applicable to games with not-to-big size, but is "more optimal"
- ➢ MCTS scales to games with very large size, but less optimal
- ➢ To beat human champions, it is not necessary to find NE strategy, but just need to find superhuman strategies (e.g., AlphaGo)

Vanilla Minimax for Small-Size EFGs

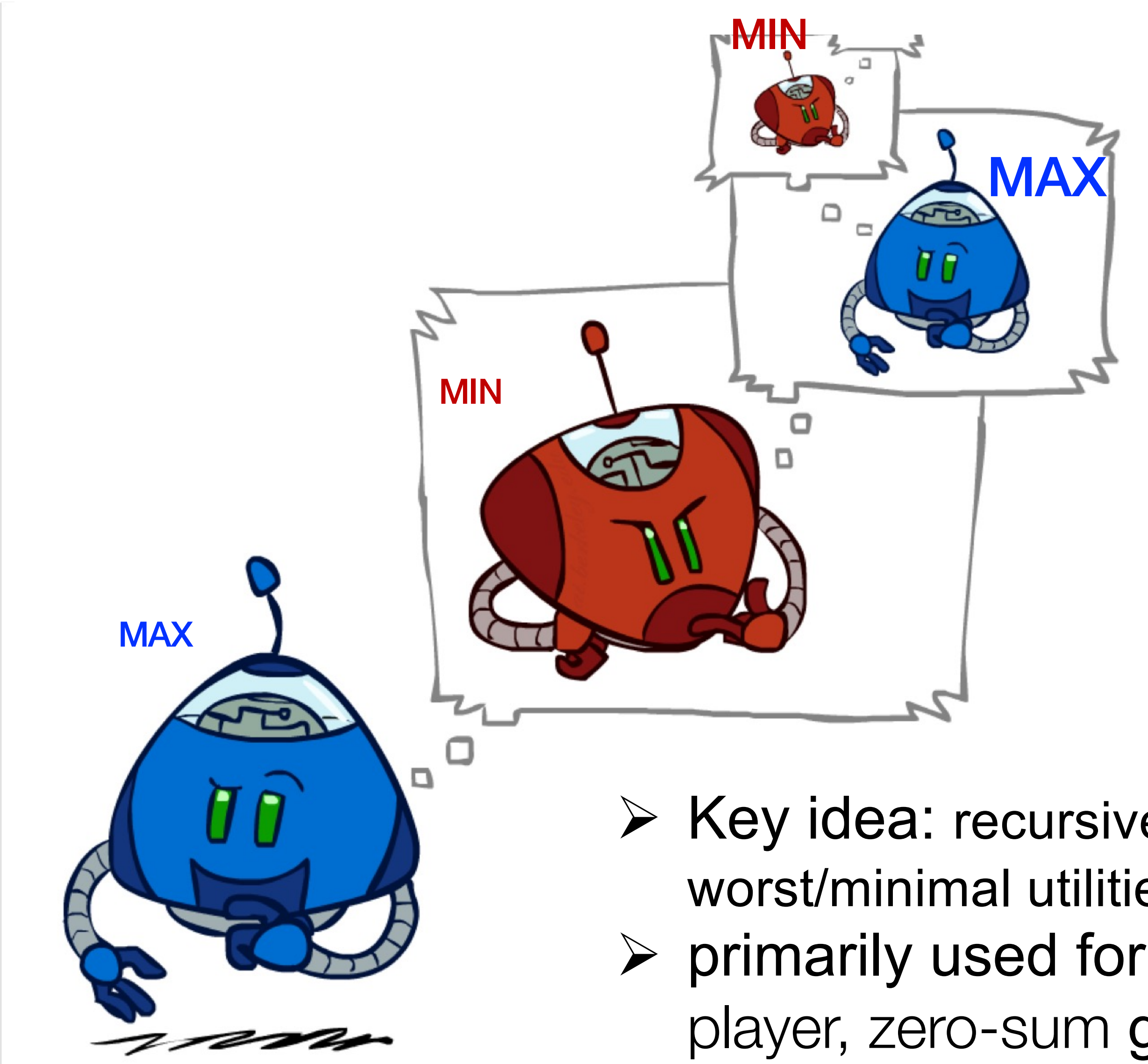
- Two-player zero-sum sequential game with complete information
- Different from single-agent search as in MDP!

Vanilla Minimax for Small-Size EFGs



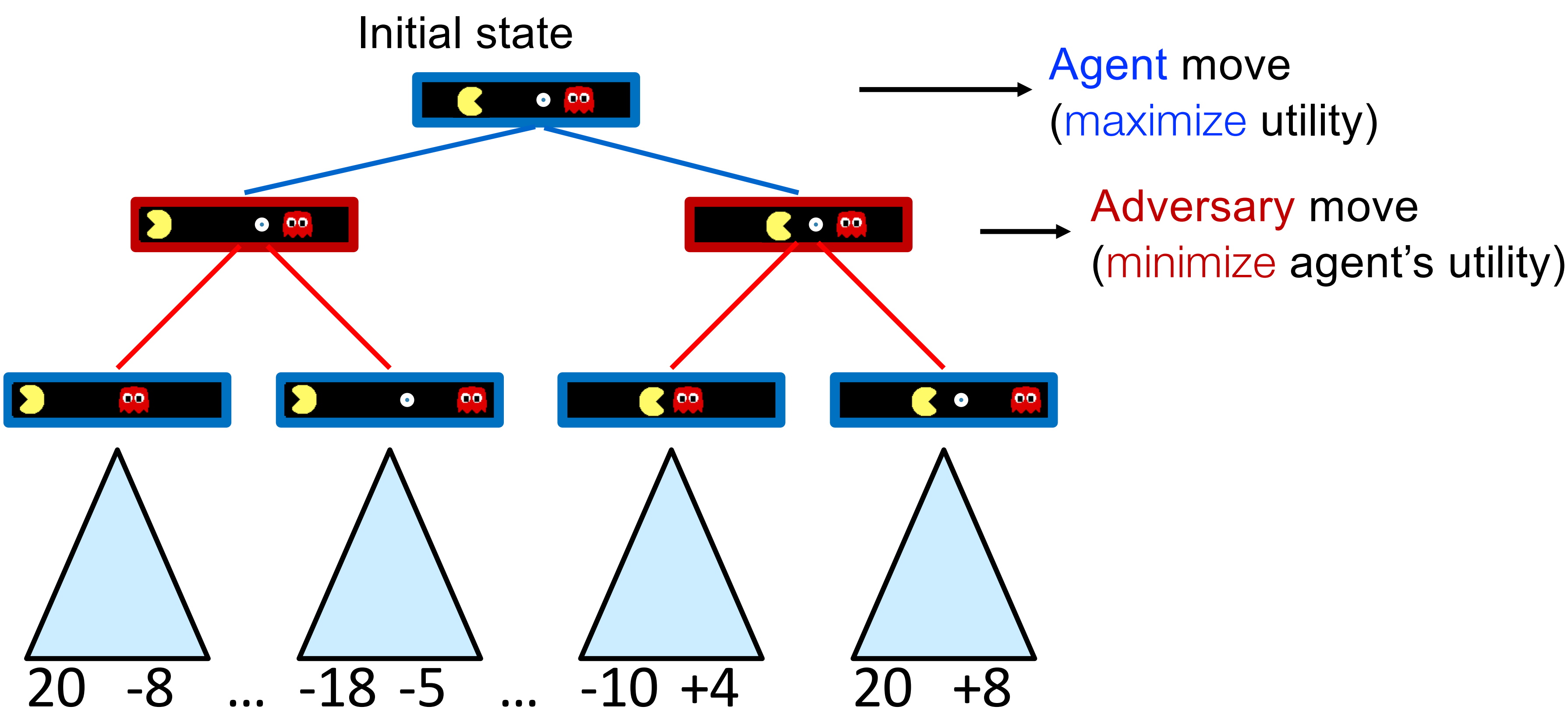
- Goal: design algorithms for calculating a policy which recommends a move at each node (i.e., a game history)

Minimax Search



- **Key idea:** recursively maximize worst/minimal utilities
- **primarily used for** deterministic, two-player, zero-sum **games**

The EFT's Game Tree



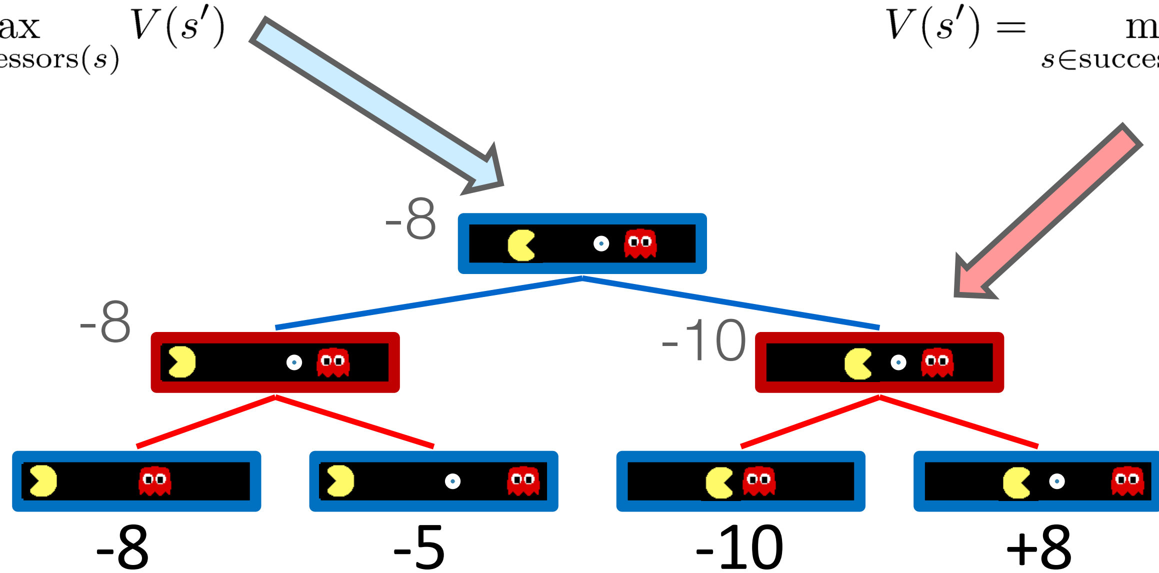
Key Concept: Minimax Values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

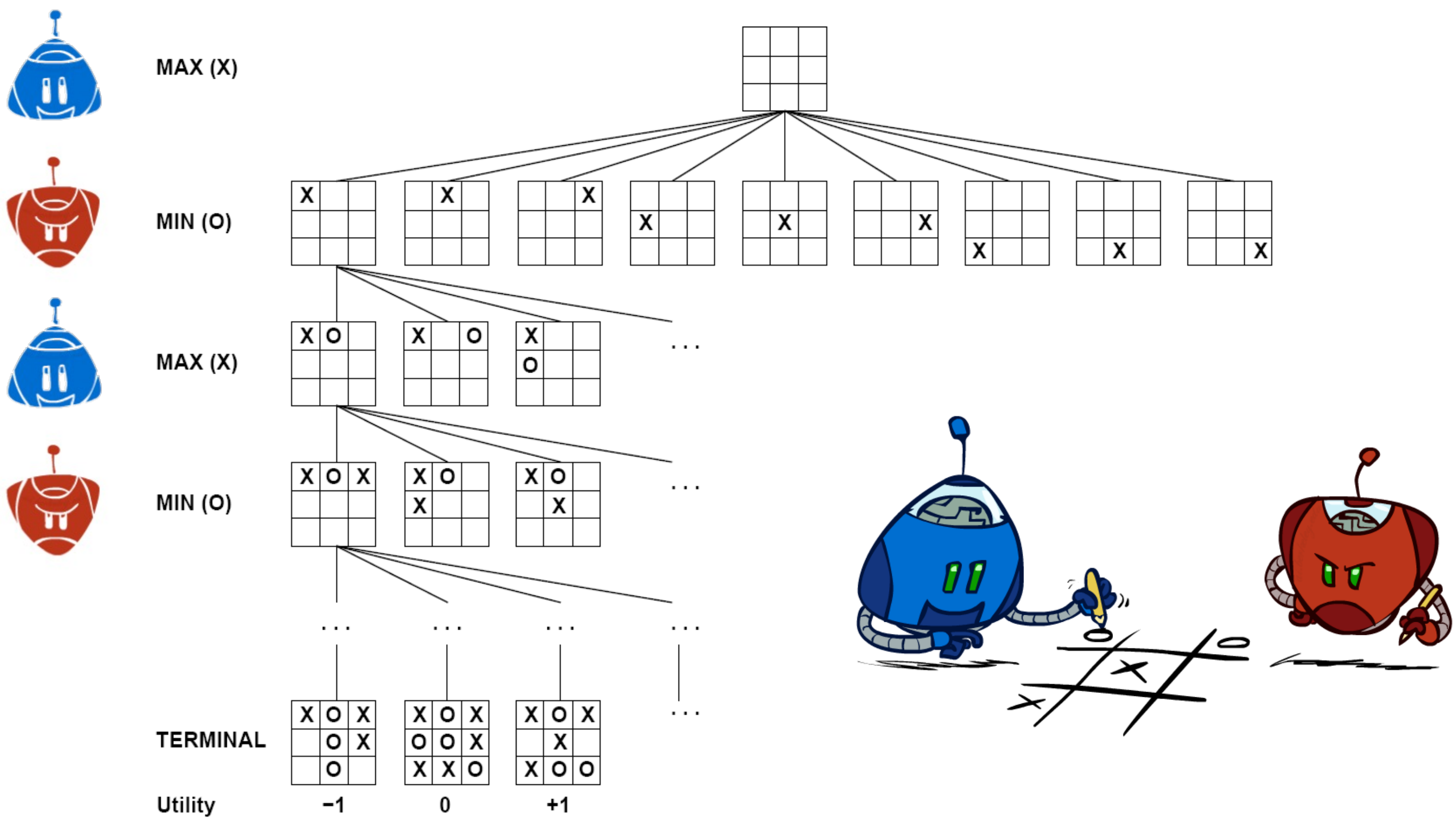


Terminal States:

$$V(s) = \text{terminal utility}$$

Minimax value of initial state = Agent's best achievable utility against an optimal adversary = Agent's utility at equilibrium

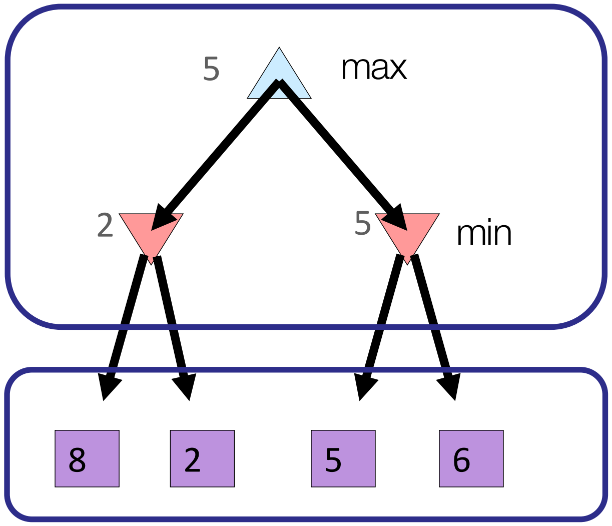
Example: Tic-Tac-Toe



Minimax Search

- ➢ Goal: compute minimax value for the initial state
  - Usually also need to record the path that achieves the value
- ➢ **Minimax** — the basic algorithm
  - Players alternate turns
  - Expand a game tree
  - Recursively compute each node's minimax value

Minimax values: computed recursively



Terminal values

Easy to Implement Minimax

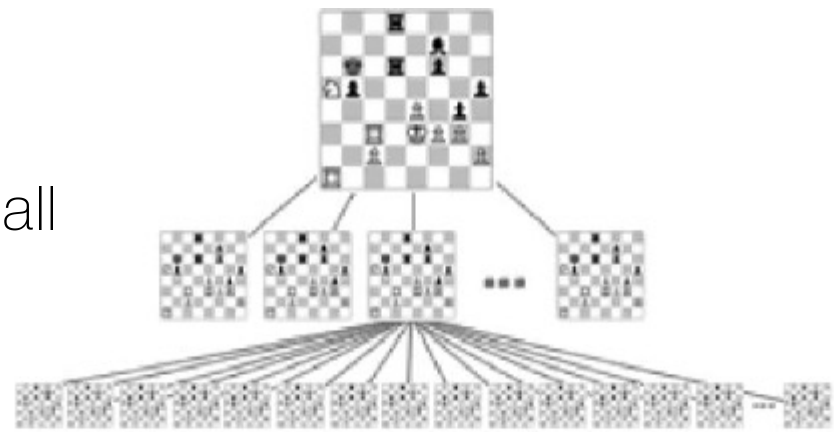
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

Complexity and Limitations

- Is it optimal? Yes, against an optimal adversary, and even better if adversary is sub-optimal
- Computational efficiency
  - Need to visit every node
  - Only feasible when game tree is small

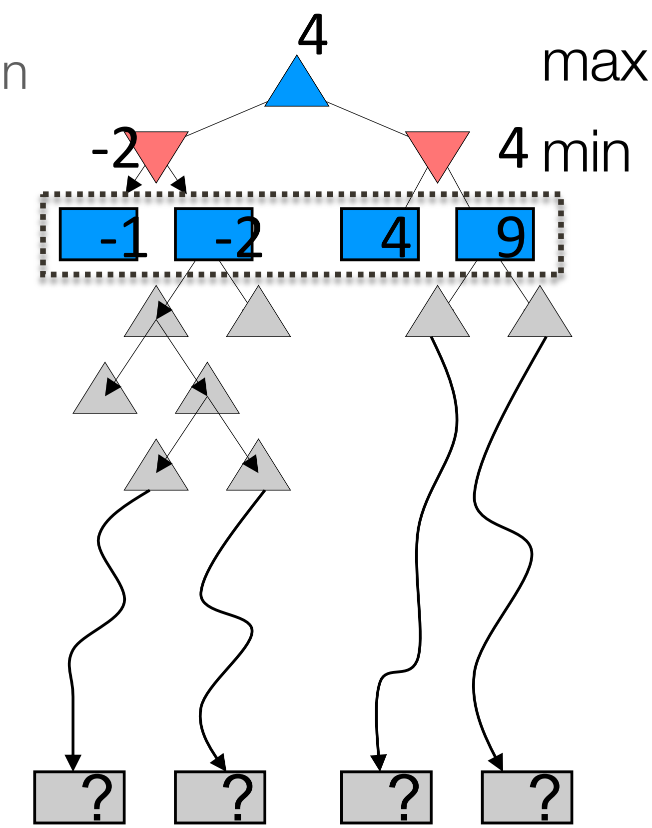


- Example: for chess, $b \approx 35$, $m \approx 80$
  - Exact solution is infeasible
- Drawbacks: high time complexity, cannot reach leaves in most interesting games

Speed-up Idea 1: Depth-Limited

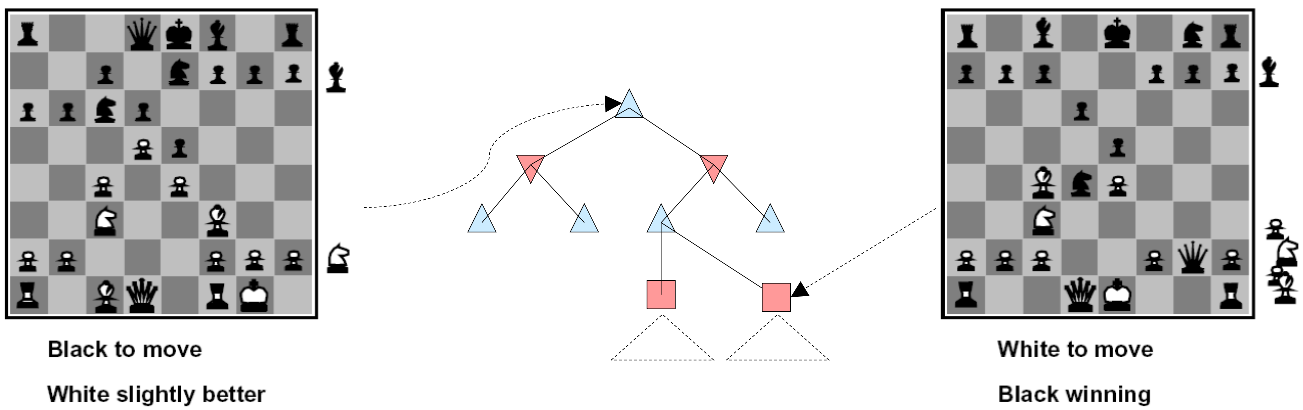
- Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal nodes
- Performance relies on two key factors
  - Depth: typically deeper search is better
  - Evaluation function: optimal if given perfect evaluation
- Example:
  - E.g., given 100 sec, can explore 10K nodes/sec
  - So can check 1M nodes per move

    Search reaches about depth 8 – decent chess program
- *No guarantee of optimality*



Value Function Approximation

- Evaluation functions “score” non-terminals in depth-limited search



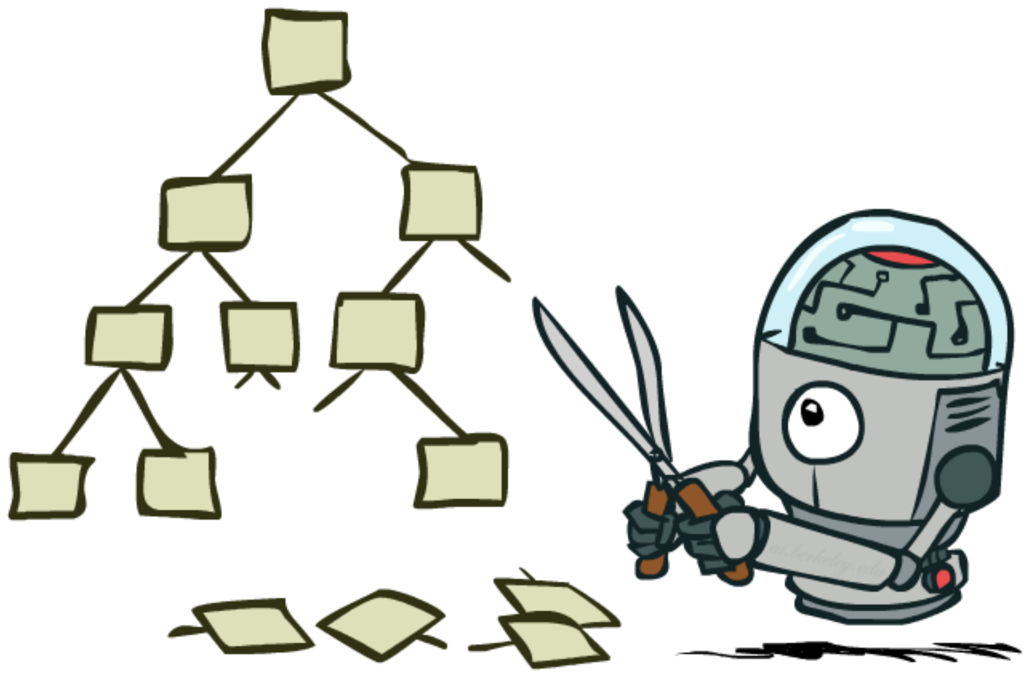
- Ideally: returns the actual minimax value of the state
- In practice: a simple heuristic is weighted linear sum of features
  - e.g. $f_1(s)$ = (num white queens – num black queens), etc.

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- Fashionable idea: use deep neural networks (this is how AlphaGo works)

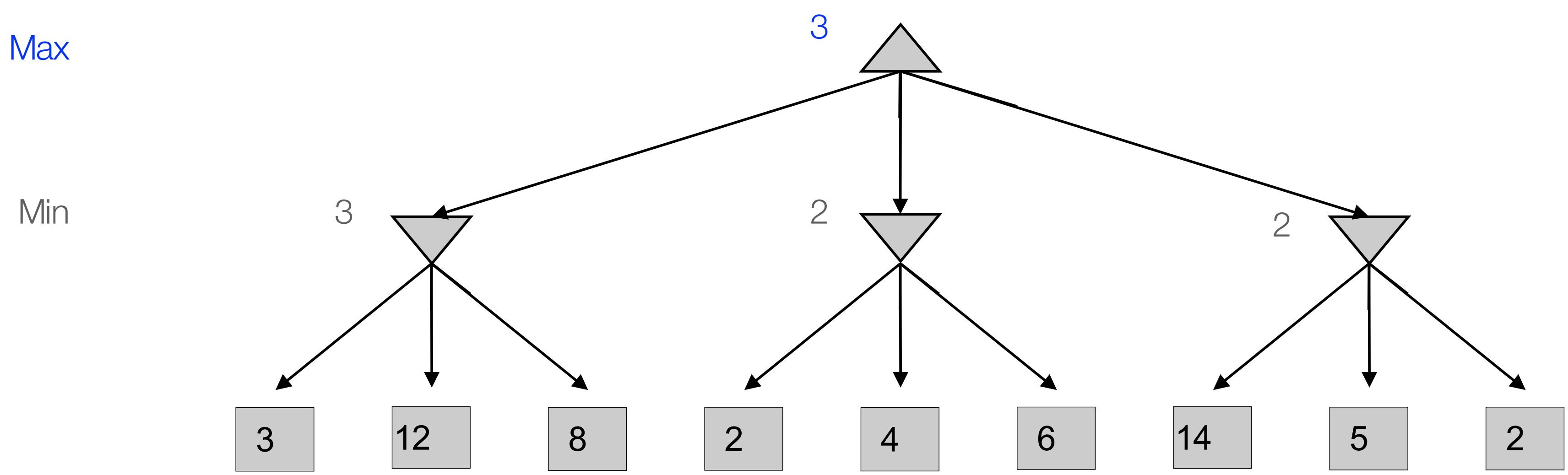
Speed-up Idea 2: Pruning

Key fact: to compute minimax value of initial state, no need to look at every branches → eliminate unnecessary computation



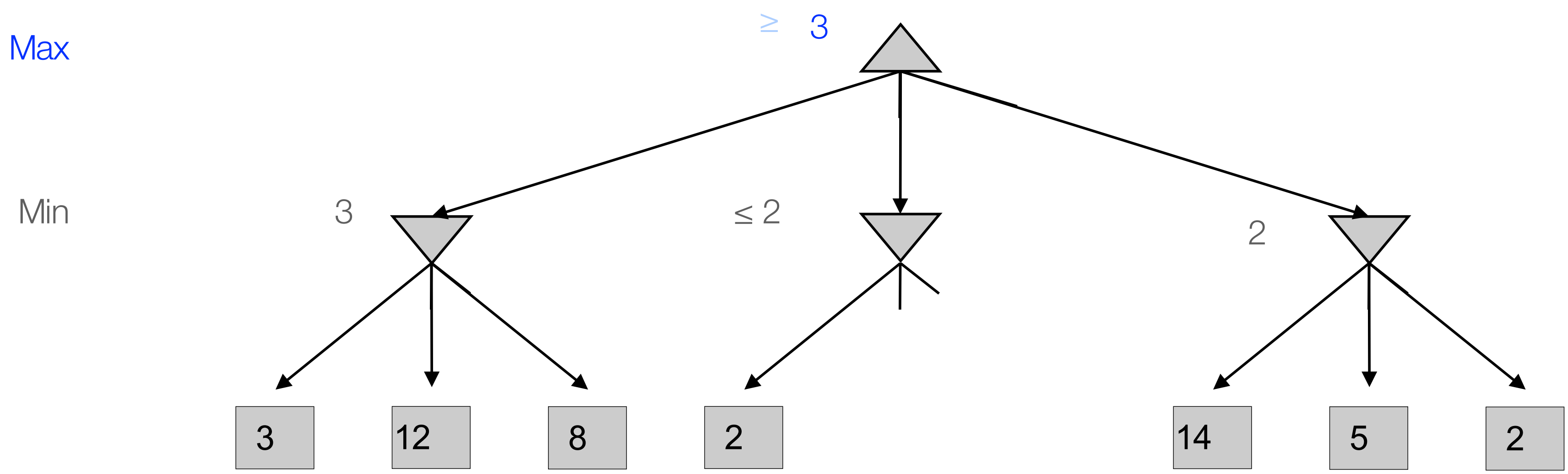
Example: Standard Minimax

From a Max player's perspective



Example: Pruning in Minimax

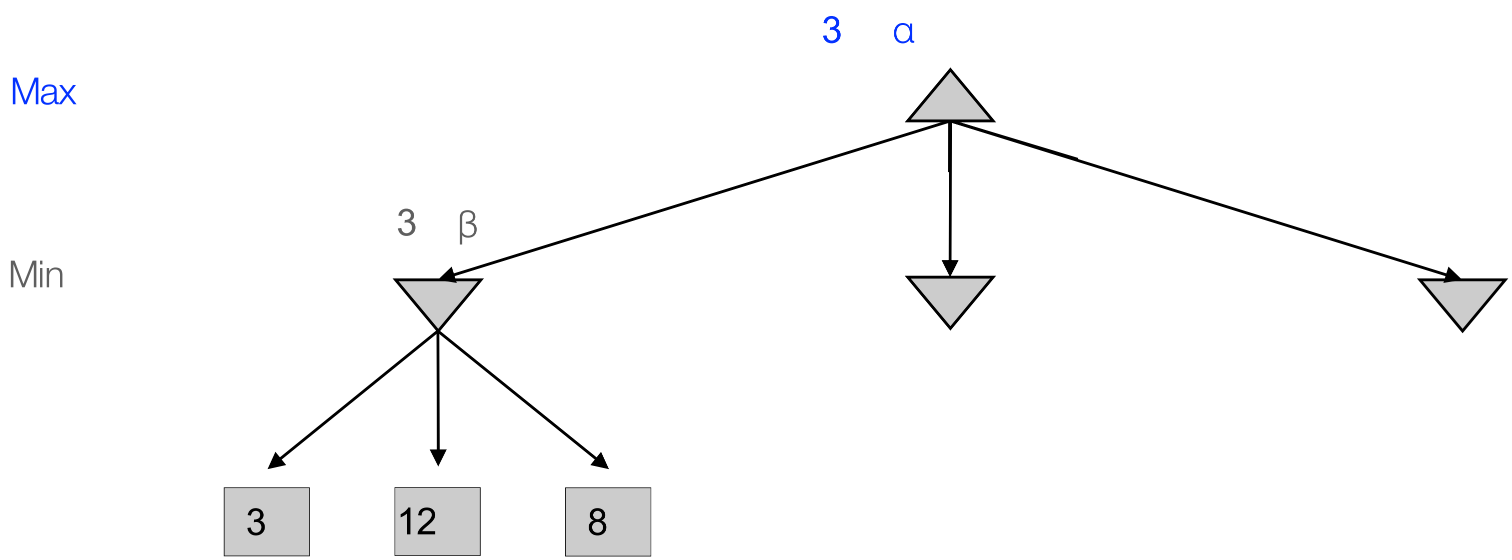
From a Max player's perspective



Formalizing This Procedure

α: MAX's maximum possible value so far

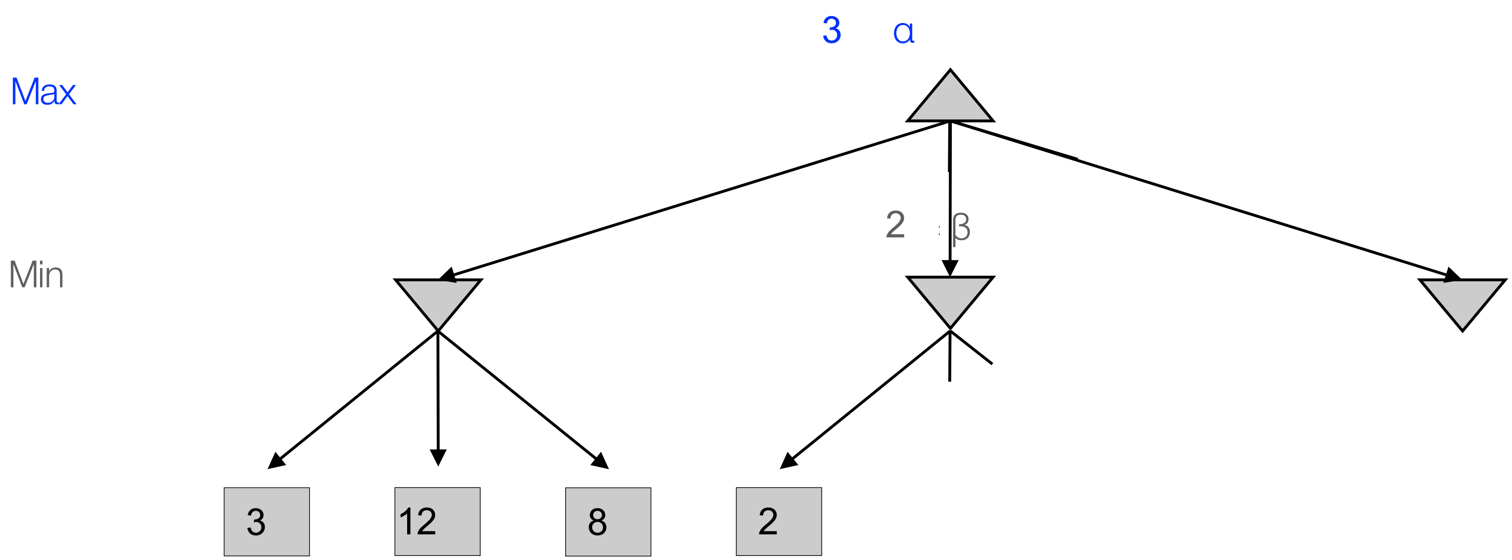
β: MIN's minimum possible value so far



Formalizing This Procedure

α: MAX's maximum possible value so far

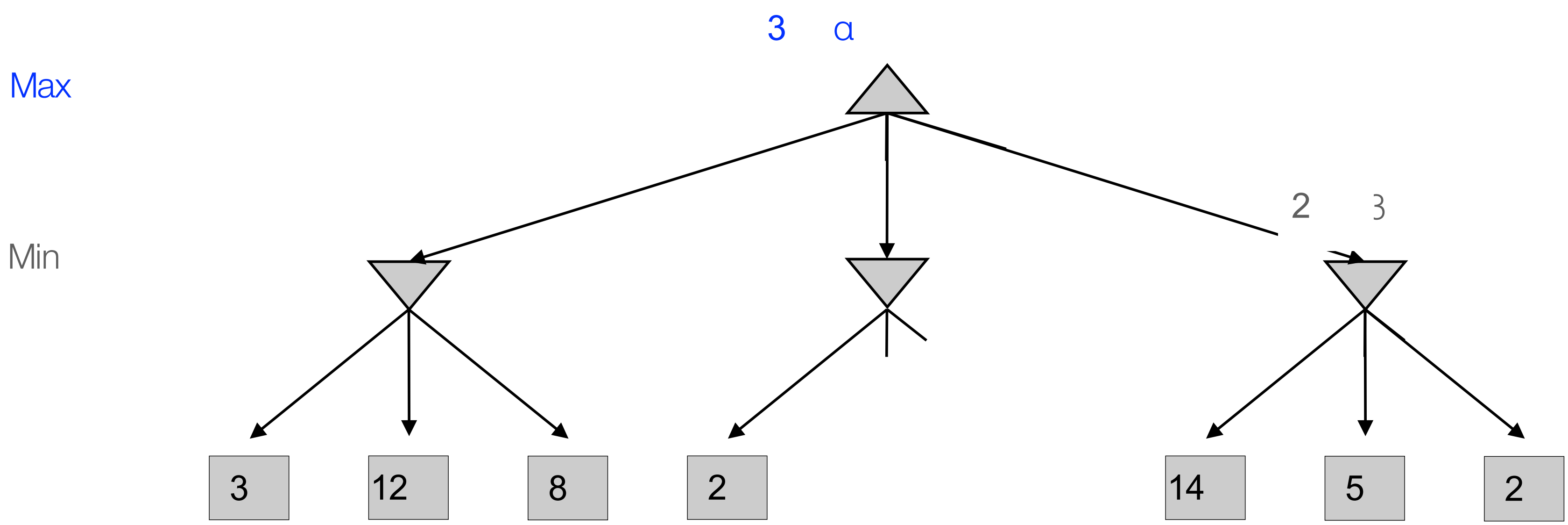
β: MIN's minimum possible value so far



Formalizing This Procedure

α: MAX's maximum possible value so far

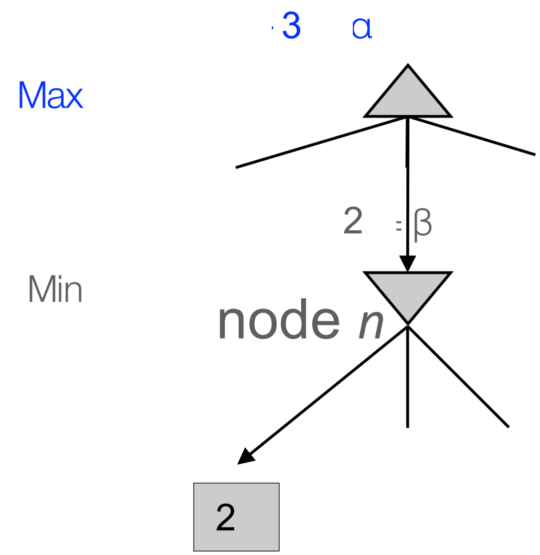
β: MIN's minimum possible value so far



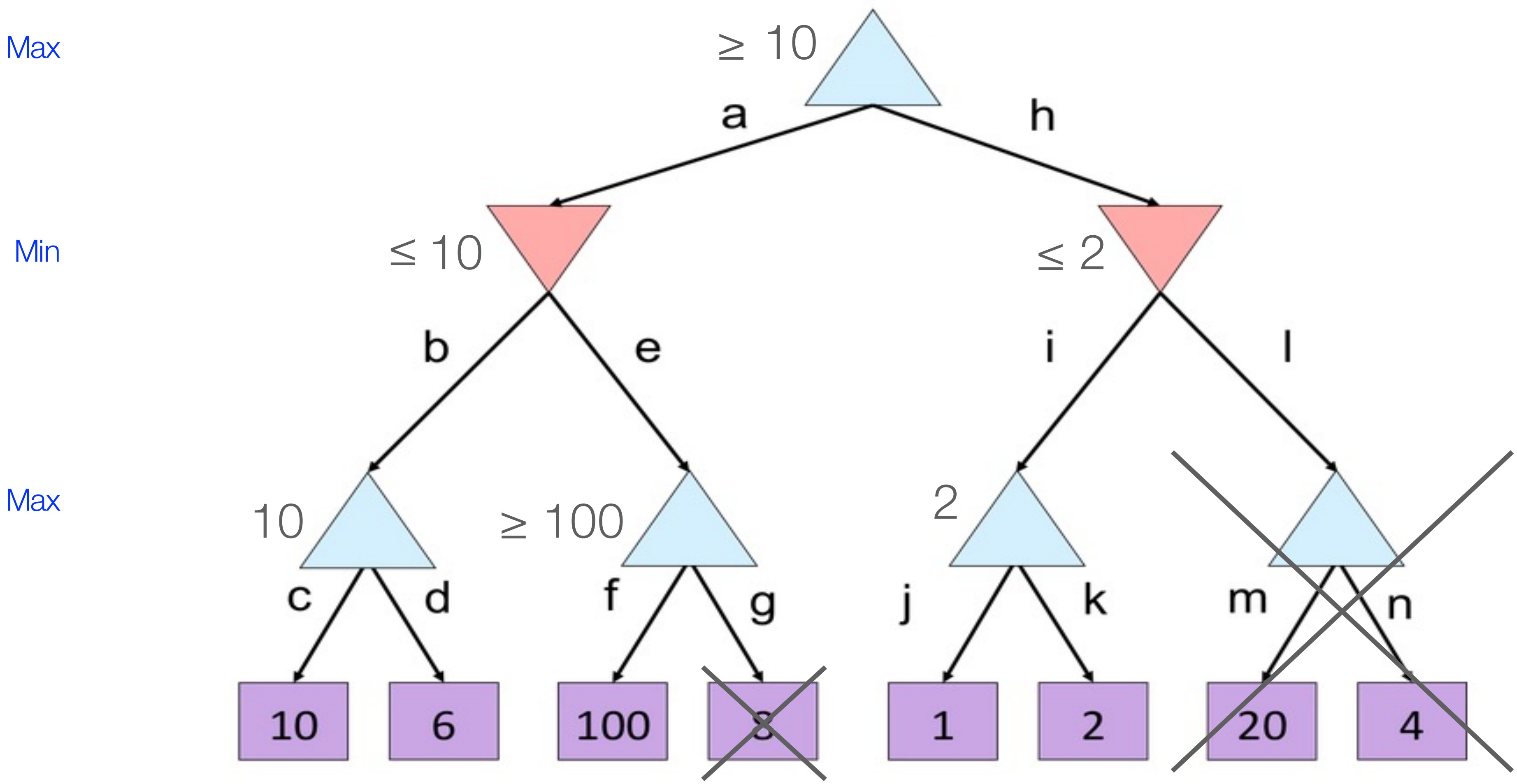
Alpha-Beta Pruning

- General configuration (**`MIN`** version)
  - We're computing the min value at some node ***n***
  - Loop over *n*'s children
  - *n's* estimate **β** of the children's min is decreasing
  - Who processes *V(n)*? **`MAX`**
  - Let ***α*** be the best value that **`MAX`** can get so far
  - If *α* ≥ **β**, **`MAX`** will avoid node ***n***, so we can stop considering *n*'s other children (it's already bad enough that it won't be played by **`MAX`**)

- **`MAX`** version is symmetric



Alpha-Beta Pruning: Example



Implementing Alpha-Beta Pruning

α: MAX's maximum possible value so far
β: MIN's minimum possible value so far

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

- In both cases, if state is a terminal, simply return its *utility*

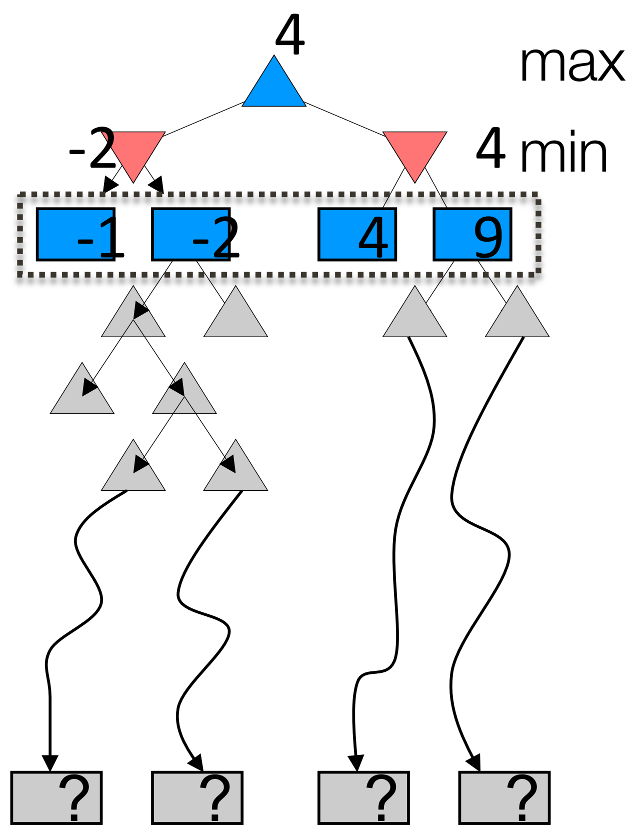
Will Cover Two Algorithms

"Solving" = finding Nash equilibrium strategy (i.e., Maximin) for one player

1. Minimax Search
   - The core algorithm framework for IBM's deep blue
   - Real implementation has lots of speed-up improvements via expert knowledge
2. Monte-Carlo Tree Search (MCTS)
   - The core algorithmic framework for AlphaGo
   - Deep RL played a key role

Monte Carlo Tree Search (MCTS)

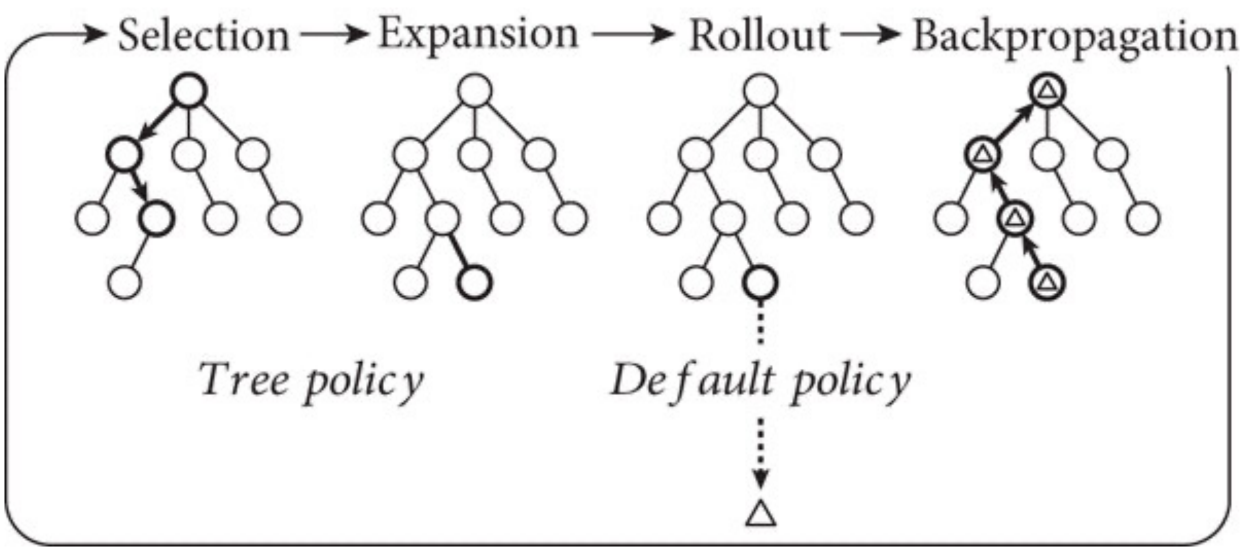
➢ Key idea: estimating the value of a node via Monte Carlo simulations



Monte Carlo Tree Search (MCTS)

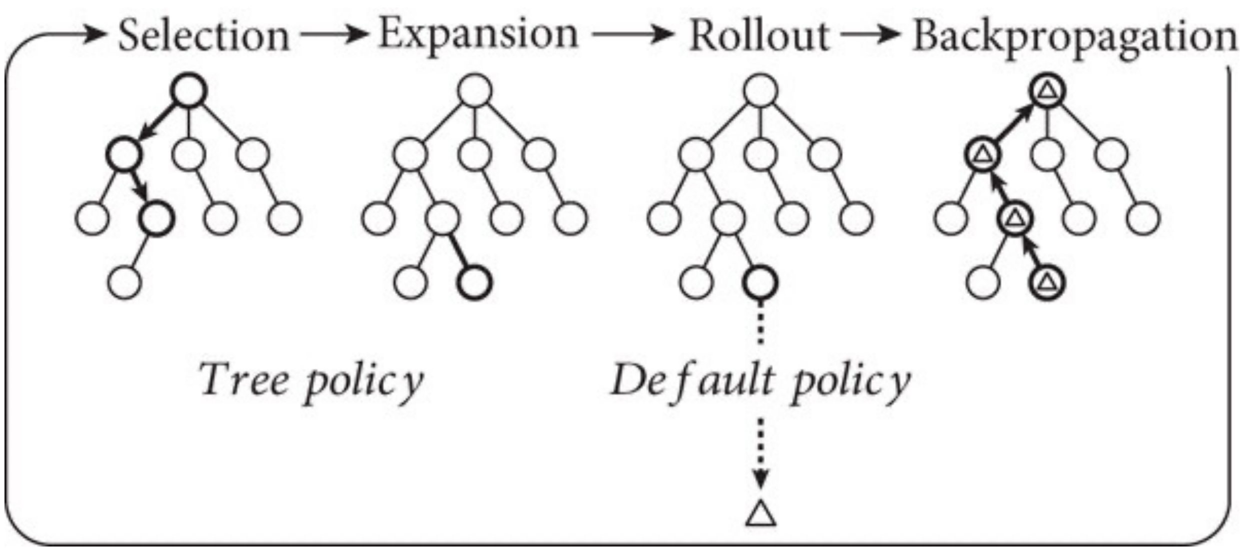
- Key idea: estimating the value of a node via Monte Carlo simulations

Overview of MCTS



Policies

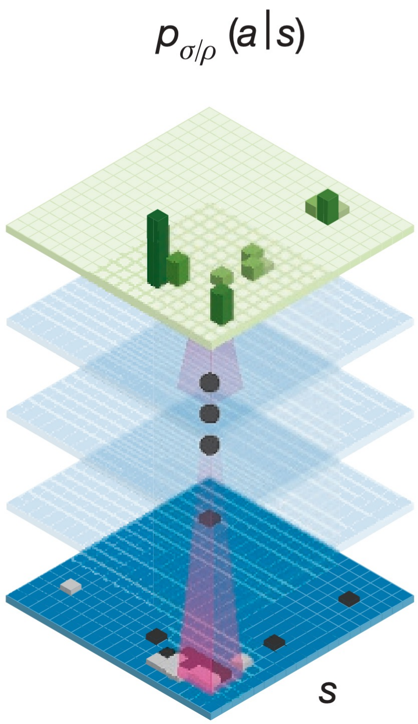
- Policies are crucial for how MCTS operates
- Tree policy
  - Used to determine how children are selected
- Default policy
  - Used to determine how MC simulations are run (e.g., randomized)
  - Result of simulation is backpropagated to update values

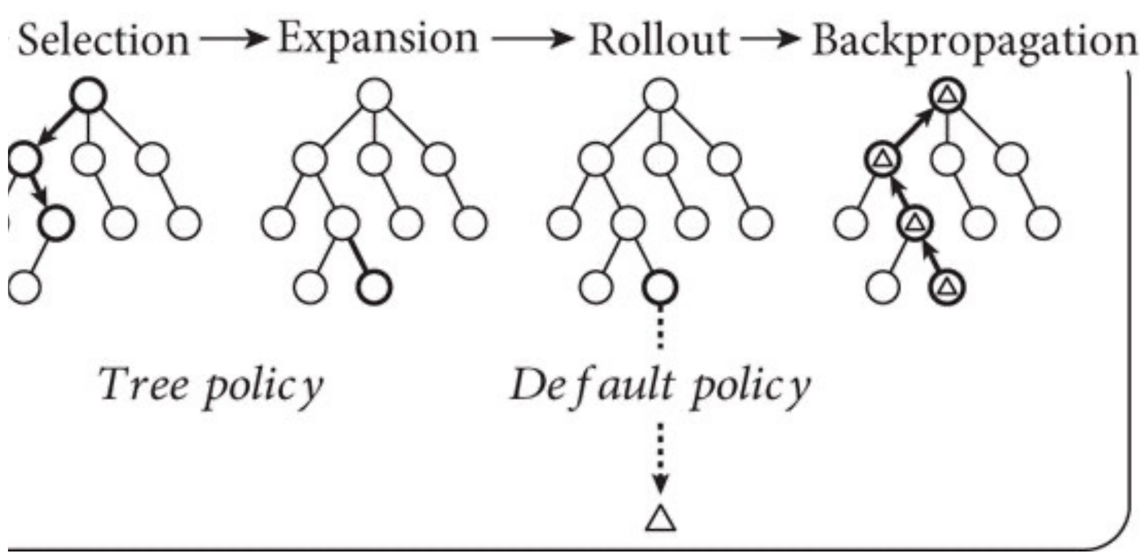


Selection

- Start at root node
- Based on tree policy select child
  - This is where deep learning comes in – when tree policy is very complex, use a neural network to output selection

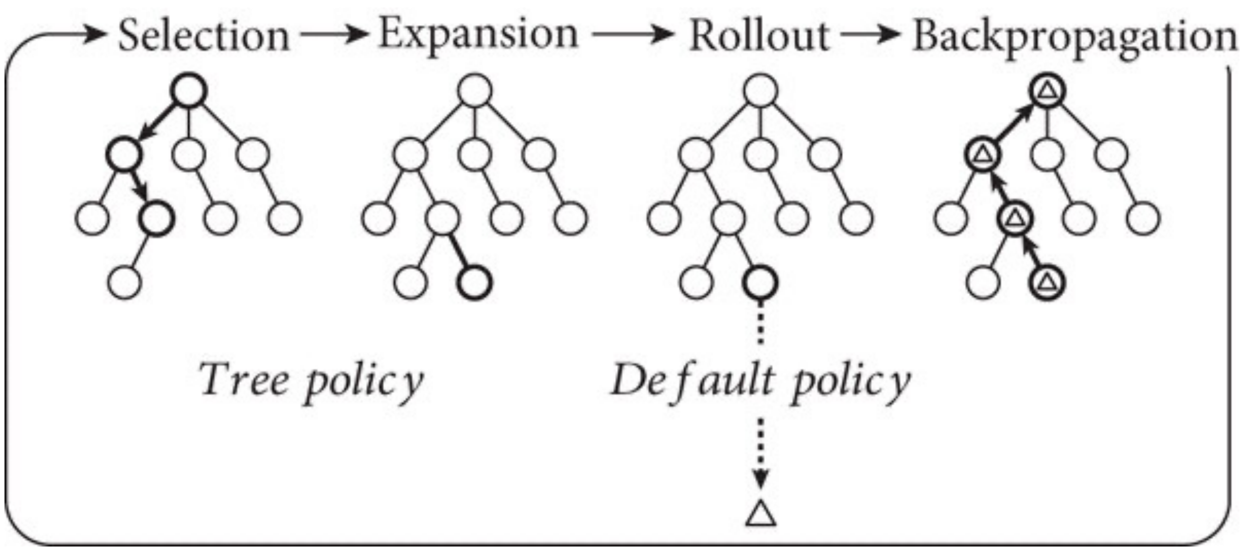
Policy network





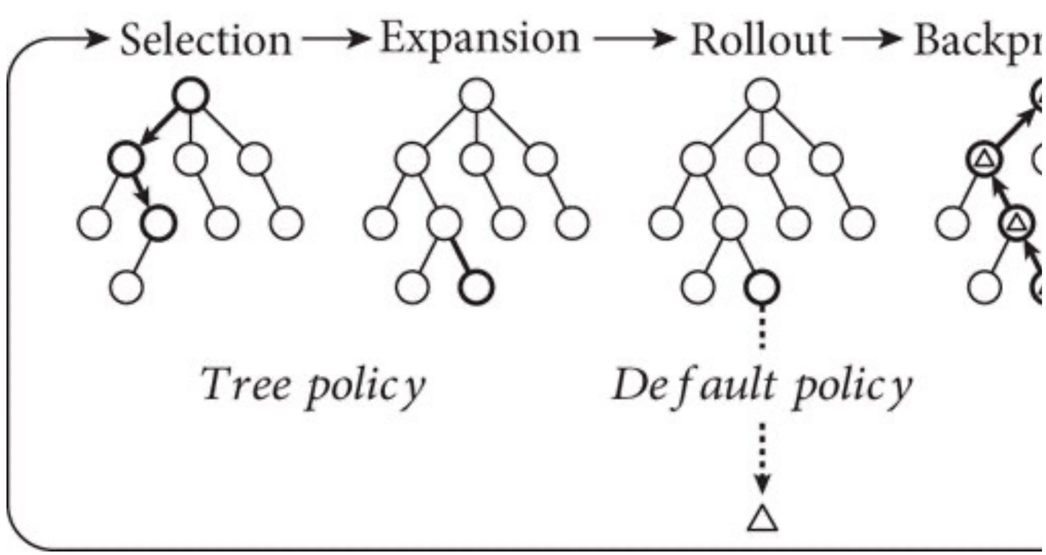
Expansion

- Expand to next one (or a few) child nodes in the tree

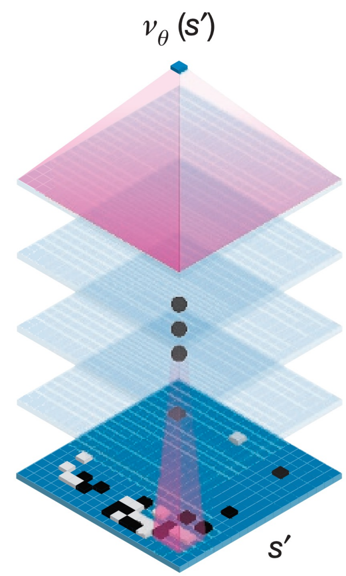


Rollout vis MC Simulation

- Run simulations of path based on **default policy**
- Get values at end of of simulation
  - For board games, board outcomes determine the value
  - Can use UCB to encourage exploration
  - This is where deep learning comes in – can use **value network** to estimate the value of a state (trained from expert data as in AlphaGo or pure simulation data as in AlphaZero)

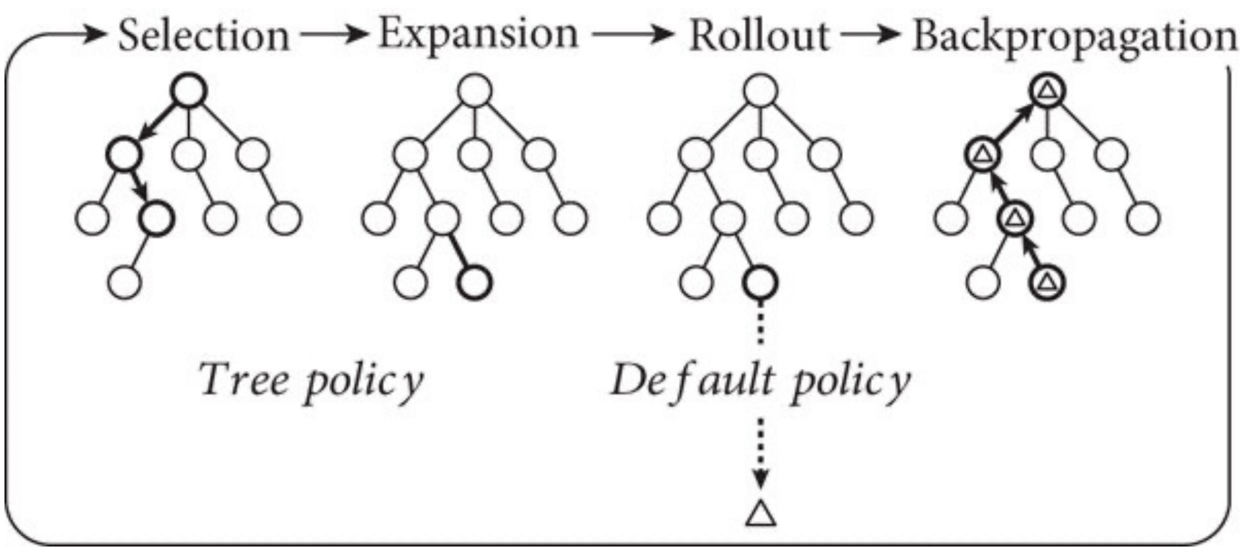


Value network



Backpropagation

- Like that in Minimax search

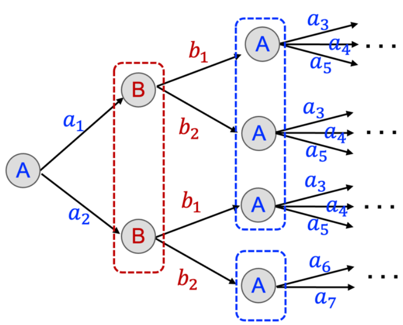


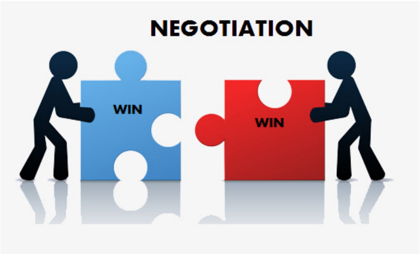
Outline

- Sequential Games and Extensive-Form Representations

- Solving Complete-Information Games

- Solving Incomplete-Information Games

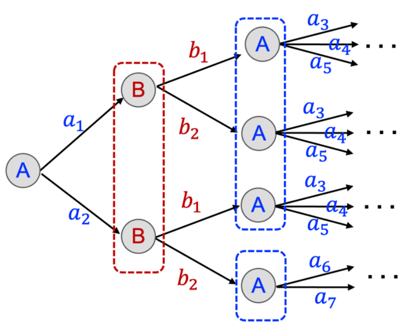




How to Represent a Strategy/Policy Here?

Policy representation for player $i \in \{A, B\}$

- For each information set I_i, i uses a mixed strategy $\sigma_i[I_i] \in \Delta(A(I_i))$
  - $\sigma_i[I_i](a) =$ prob of taking action a
- This is not a trivial statement, since a mixed strategy generally should be a distribution over all possible move combinations
  - Recall the (b_1, b_2) action in matrix representation
  - [Kuhn, 1953] shows that it is *without loss* to consider the above policies, which decompose joint moves into a randomized move at each information set
  - Need to assume every player remembers all the past ("perfect recall")



Policy Re-Formulation in Sequence-form

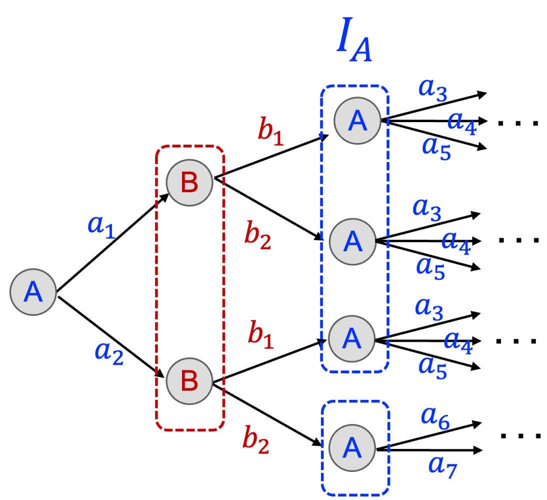
Policy representation for player $i \in \{A, B\}$

- For each information set I_i, i uses a mixed strategy $\sigma_i[I_i] \in \Delta(A(I_i))$
  - $\sigma_i[I_i](a) =$ prob of taking action a

✓ Prob (a sequence of actions) = $\Pi_{a \text{ in sequence}} \sigma_i(a)$

✓ Above policy can be equivalently represented as probabilities over all sequences

⇒ any policy induce a distribution over sequences
⇐ any distribution over sequence induces a policy like above



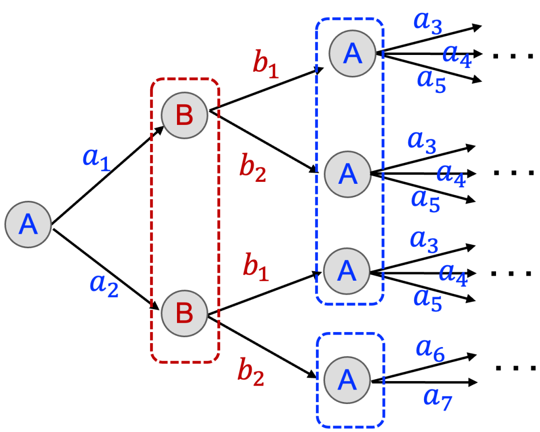
For example:

$$\sigma_A[I_A](a_3) = \frac{\Pr(a_1, a_3)}{\Pr(a_1, a_3) + \Pr(a_1, a_4) + \Pr(a_1, a_5)}$$

Hence, Can Solve Small Games by LPs

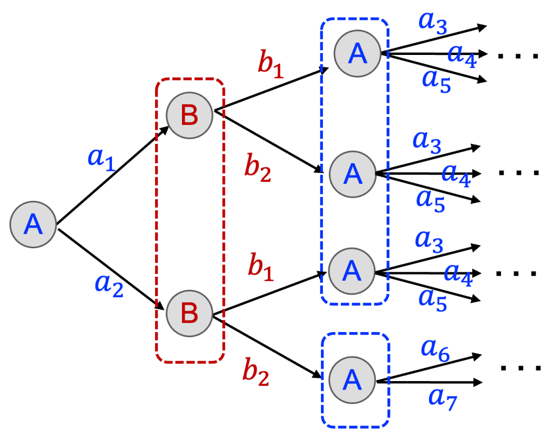
- Representing each player's mixed strategy as probabilities over that player's action sequence (at most #nodes many variables)
- Expected utilities can be written as linear functions of these probabilities
- Can be solved by LPs similar to that of the matrix form

Naturally, everything here applies to complete-information EFGs as well as they are special cases



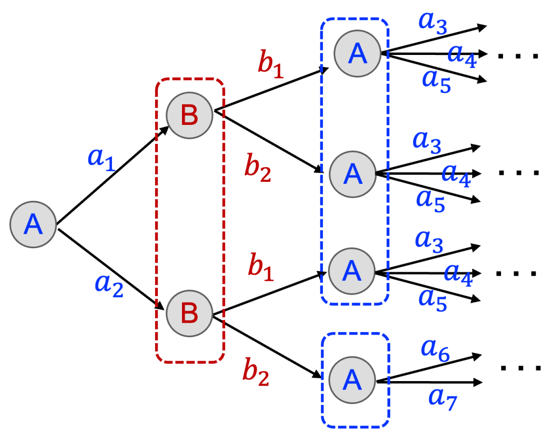
What About Large Games Like Pokers

- LP approaches do not work any more, since #variables too large
- Not easy to extend previous tree search methods, with information set and randomized actions
  - Note: no need to randomize in complete-info EFTs
- Practically successful approaches are based on no-regret learning



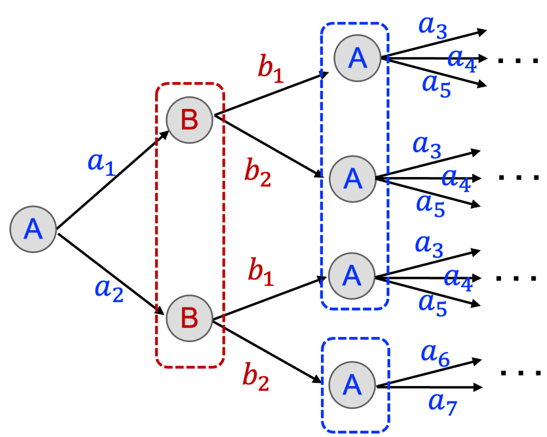
The Core Idea

- When game tree is extremely large…
  - No hope to compute probabilities for each action sequence in the tree
  - Hence, can only optimize "local" moves – i.e., optimizing the mixed strategy $\sigma_i[I_i]$ before for each information set I_i
- However, regret of a policy is a global notion – question is, how to ensure each local move reduces "global regret"



The Core Idea

- Core idea is **regret decomposition** – decomposing total regret into (hopefully) the sum of “local regrets” for each local moves
  - Key is to find a notion of “local regret”, the sum of which upper bounds total regret
  - One choice is CounterFactual Regret (CFR) for each local move
- Suppose we can do so… we basically decomposed policy design to each local move, which is much more manageable
  - You can run any no-regret learning algorithm as you liked, just using the right reward value and regret notion
  - MCTS shares similar spirit, but uses very different approaches



Definition of Counterfactual Regrets (CFR)

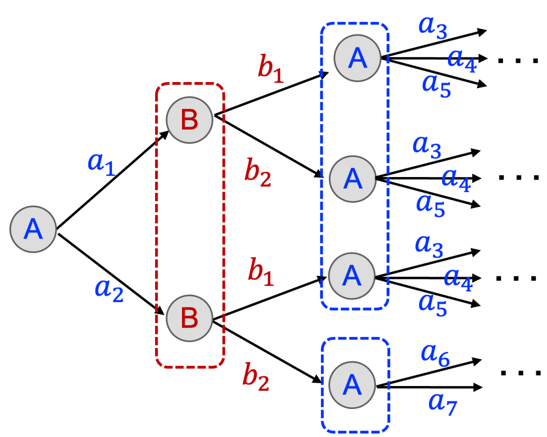
Defined for each information set I_i

- Suppose the game is played repeatedly for T times
- Player i used strategy σ_i^t (i.e., $\sigma_i^t[I_i]$ at info set I_i)
  - Let σ^t denote their joint action profile

Where the term "counterfactual" comes from

$U_i(\sigma, I) =$ Expected i'th utility, conditioned on (1) all other players play σ_{-i}; and (2) player i plays to reaches I

$$= \frac{\sum_{h \in I} \sum_{z \in Z} \Pr(\text{play reach } h \text{ under } \sigma_{-i}) \Pr(h \to z) u_i(z)}{\Pr(\text{play reach } I \text{ under } \sigma_{-i})}$$



A local deviation from σ

✓ Policy $\sigma|I_i \to a$ is the same as σ except that player i always plays action $a \in A(I_i)$ at info set I_i

Definition of Counterfactual Regrets (CFR)

Defined for each information set I_i

- Suppose the game is played repeatedly for T times
- Player i used strategy σ_i^t (i.e., $\sigma_i^t[I_i]$ at info set I_i)
  - Let σ^t denote their joint action profile

Where the term "counterfactual" comes from

$U_i(\sigma, I) =$ Expected i'th utility, conditioned on (1) all other players play σ_{-i}; and (2) player i plays to reaches I

$$CFR_i(I_i \to a) = \frac{1}{T}\sum_{t=1}^{T}[U_i(\sigma^t | I \to a, I_i) \; - U_i(\sigma^t, I_i)\;] \times \Pr(\text{reach } I_i \text{ under } \sigma_{-i})$$

Regret minimization picks the a to minimize it

Thank You

Haifeng Xu

University of Chicago

haifengxu@uchicago.edu