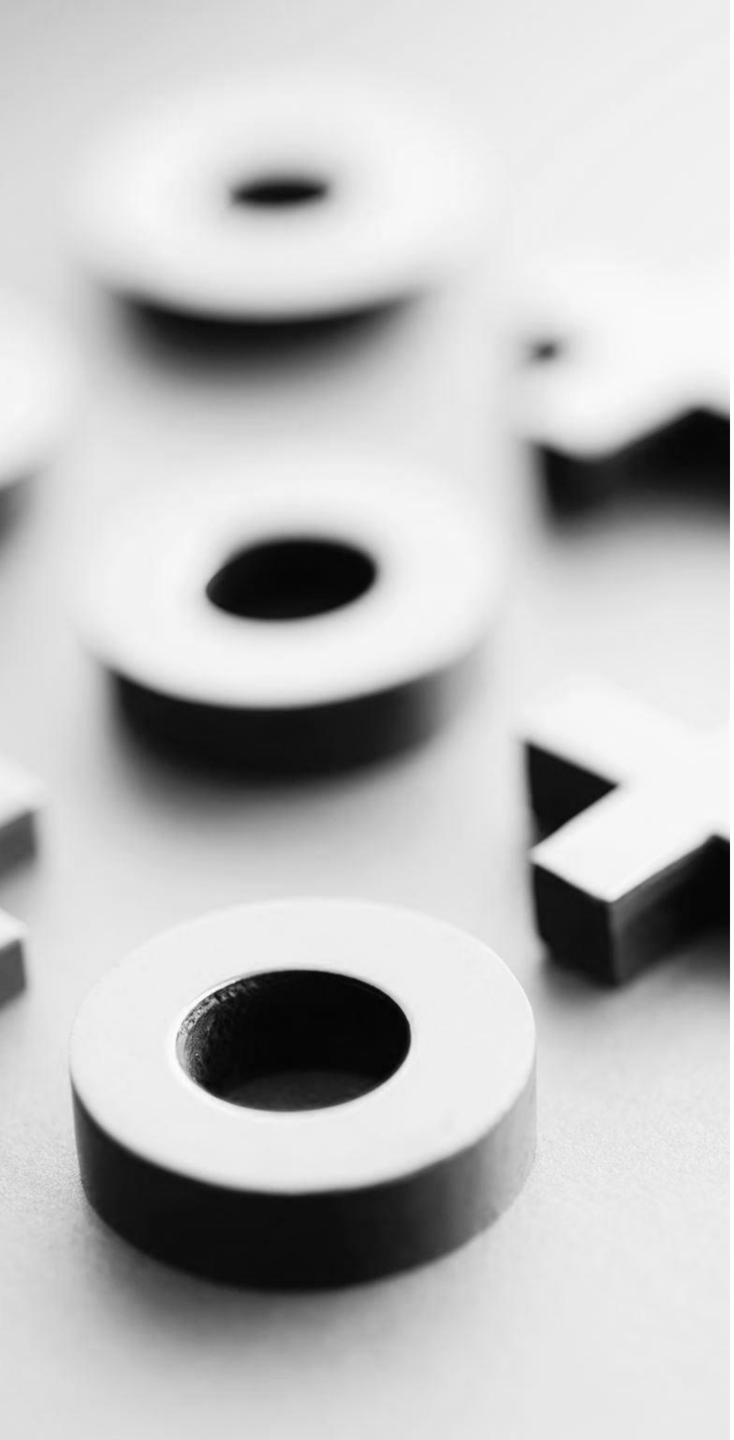
CS 5/7320
Artificial
Intelligence

Adversarial Search and Games

AIMA Chapter 5

Slides by Michael Hahsler



Games

- Games typically confront the agent with a competitive (adversarial) environment affected by an opponent.

- We will focus on deterministic two-player zero-sum games with perfect information.

- We call the two players:
  1) **Max** tries to maximize his utility.
  2) **Min** tries to minimize Max's utility since it is a zero-sum game.

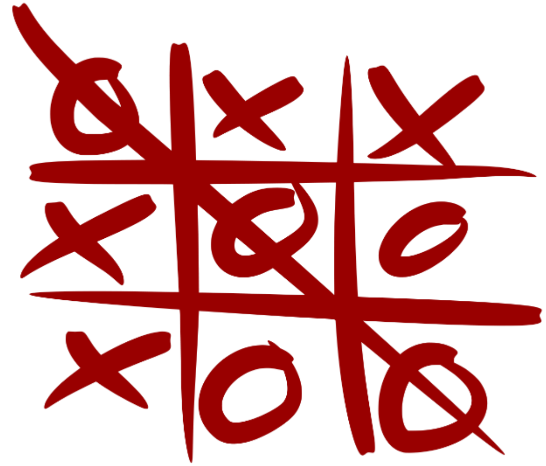
Definition of a Game

- **Definition:**

s_0	The initial state (position, board).
$Actions(s)$	Legal moves in state s.
$Result(s, a)$	Transition model.
$Terminal(s)$	Test for terminal states.
$Utility(s)$	Utility for player Max.

- **State space**: a graph defined by the initial state and the transition function containing all reachable states (e.g., chess positions).
- **Game tree**: a search tree superimposed on the state space. A complete game tree follows every sequence from the current state to the terminal state (the game ends).

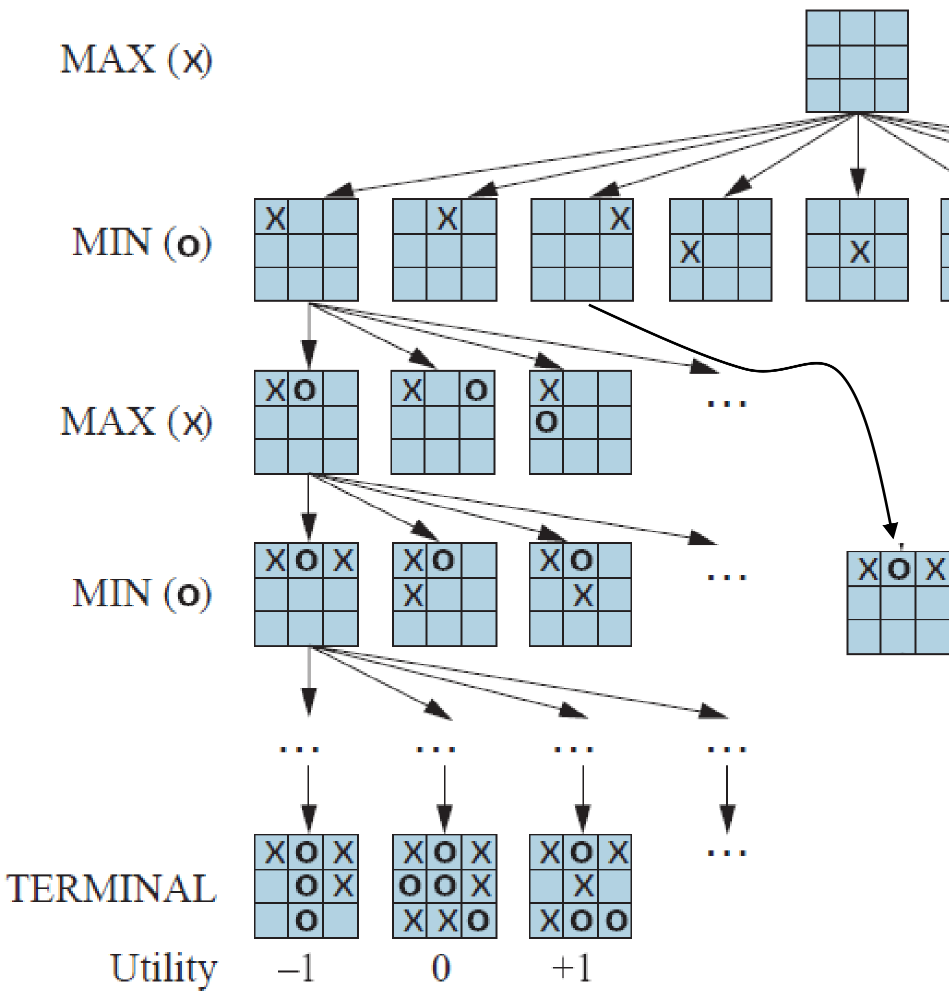
Example: Tic-tac-toe



s_0	Empty board.
$Actions(s)$	Empty squares.
$Result(s, a)$	Place symbol (x/o) on empty square.
$Terminal(s)$	Did a player win or is the game a draw?
$Utility(s)$	+1 if x wins, -1 if o wins and 0 for a draw.
	Utility is only defined for terminal states.

Here player x is Max and player o is Min.

Tic-tac-toe: Partial Game Tree



The state space is:

$$3^9 = 19{,}683$$

Why is the complete game tree size much larger? A little smaller than:

$$9! = 362{,}880$$

Methods for Adversarial Games

Exact Methods

- **Model as nondeterministic actions**: The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. We **consider all possible moves** by the opponent.
- **Find optimal decisions**: Minimax search and Alpha-Beta pruning where **each player plays optimal** to the end of the game.

Heuristic Methods

(state space is too large)

- **Heuristic Alpha-Beta Tree Search**:
  a. Cut off search tree and use heuristic for utility.
  b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate utility of a state by simulating complete games and average the utility.

Nondeterministic Actions

- Recall And-Or Search from AIMA Chapter 4

Methods for Adversarial Games

Exact Methods

- **Model as nondeterministic actions**: The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. We **consider all possible moves** by the opponent.
- **Find optimal decisions**: Minimax search and Alpha-Beta pruning where **each player plays optimal** to the end of the game.

Heuristic Methods
(state space is too large)

- **Heuristic Alpha-Beta Tree Search**:
  a. Cut off search tree and use heuristic for utility.
  b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate utility of a state by simulating complete games and average the utility.

Nondeterministic Actions

Each action consists of the move by the player and all possible (i.e., nondeterministic) responses by the opponent.

Outcome of actions in the environment is nondeterministic = **transition model need to describe uncertainty about the opponent's behavior.**

Example transition:

$$Results(s_1, a) = \{s_2, s_4, s_5\}$$

i.e., action a in s_1 can lead to one of several states.

The set of possible states $\mathrm{b} = \{s_2, s_4, s_5\}$ is called a **belief state** of the agent.

AND-OR DFS Search Algorithm

= nested If-then-else statements

function AND-OR-SEARCH(*problem*) **returns** a conditional plan, or *failure*
  return OR-SEARCH(*problem*, *problem*.INITIAL, [ ])

function OR-SEARCH(*problem*, *state*, *path*) **returns** *a conditional plan, or failure*
  if *problem*.IS-GOAL(*state*) **then return** the empty plan
  if IS-CYCLE(*path*) **then return** *failure* // don't follow loops
  for each *action* **in** *problem*.ACTIONS(*state*) **do** // check all possible actions
    plan ← AND-SEARCH(*problem*, RESULTS(*state*, *action*), [*state*] + *path*])
    if *plan* ≠ *failure* **then return** [*action*] + *plan*]
  return *failure*

my moves

function AND-SEARCH(*problem*, *states*, *path*) **returns** *a conditional plan, or failure*
  for each s_i **in** *states* **do** // check all states in belief state
    $plan_i$ ← OR-SEARCH(*problem*, s_i, *path*)
    if $plan_i$ = *failure* **then return** *failure*
  return [**if** s_1 **then** $plan_1$ **else if** s_2 **then** $plan_2$ **else** ... **if** s_{n-1} **then** $plan_{n-1}$ **else** $plan_n$]

opponent moves

- And-Or Search searches the whole tree till it finds a subtree that leads only to goal nodes.
- BFS and A* search can also be used to search an AND-OR tree.

Optimal Decisions

Minimax Search and Alpha-Beta Pruning

Methods for Adversarial Games

Exact Methods

- **Model as nondeterministic actions**: The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. We **consider all possible moves** by the opponent.
- **Find optimal decisions**: Minimax search and Alpha-Beta pruning where **each player plays optimal** to the end of the game.

Heuristic Methods
(state space is too large)

- **Heuristic Alpha-Beta Tree Search**:
  a. Cut off search tree and use heuristic for utility.
  b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate utility of a state by simulating complete games and average the utility.

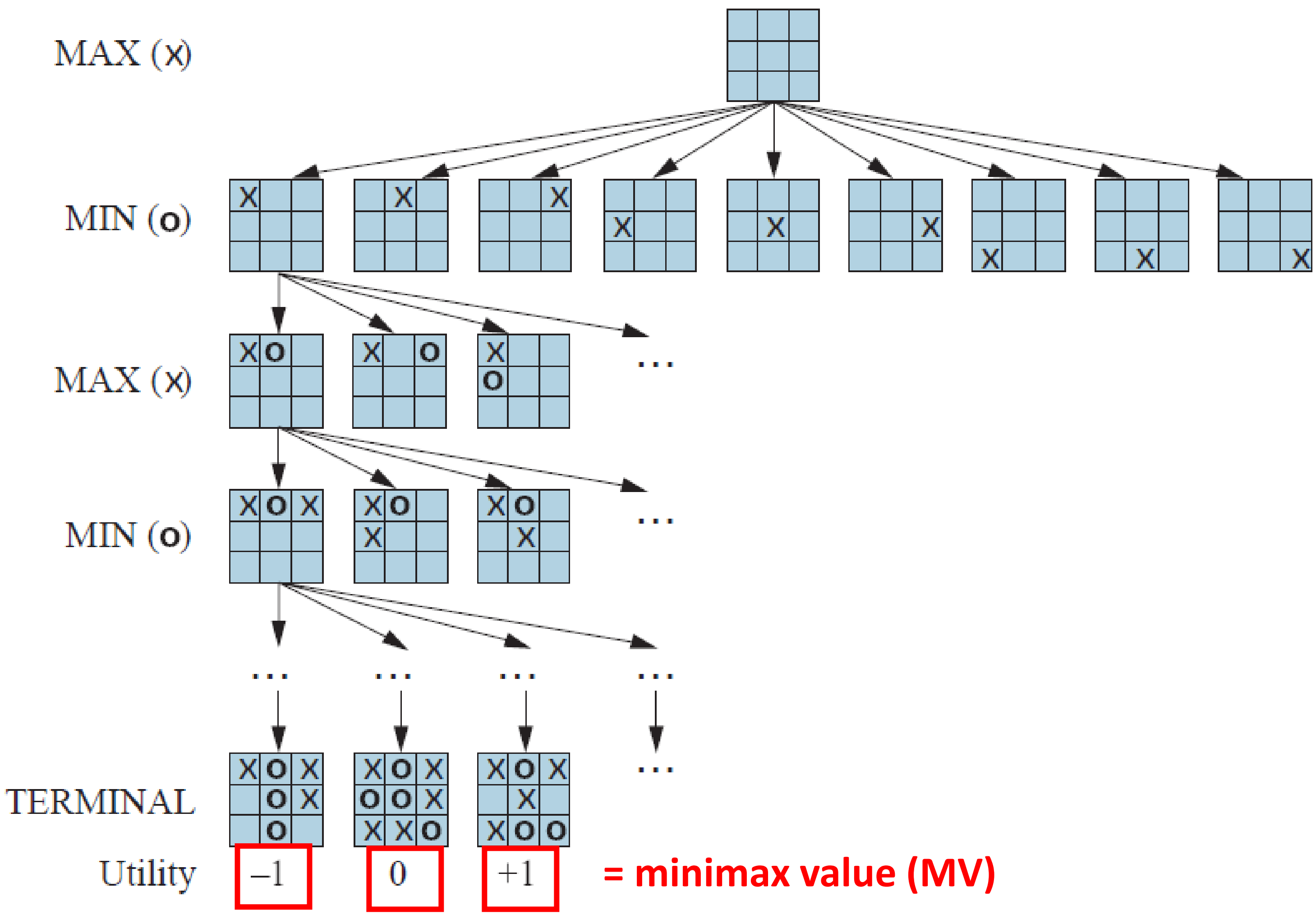
Idea: Minimax Decision

- Assign each state a **minimax value** that reflects how much Max prefers the state (= Min dislikes the state).

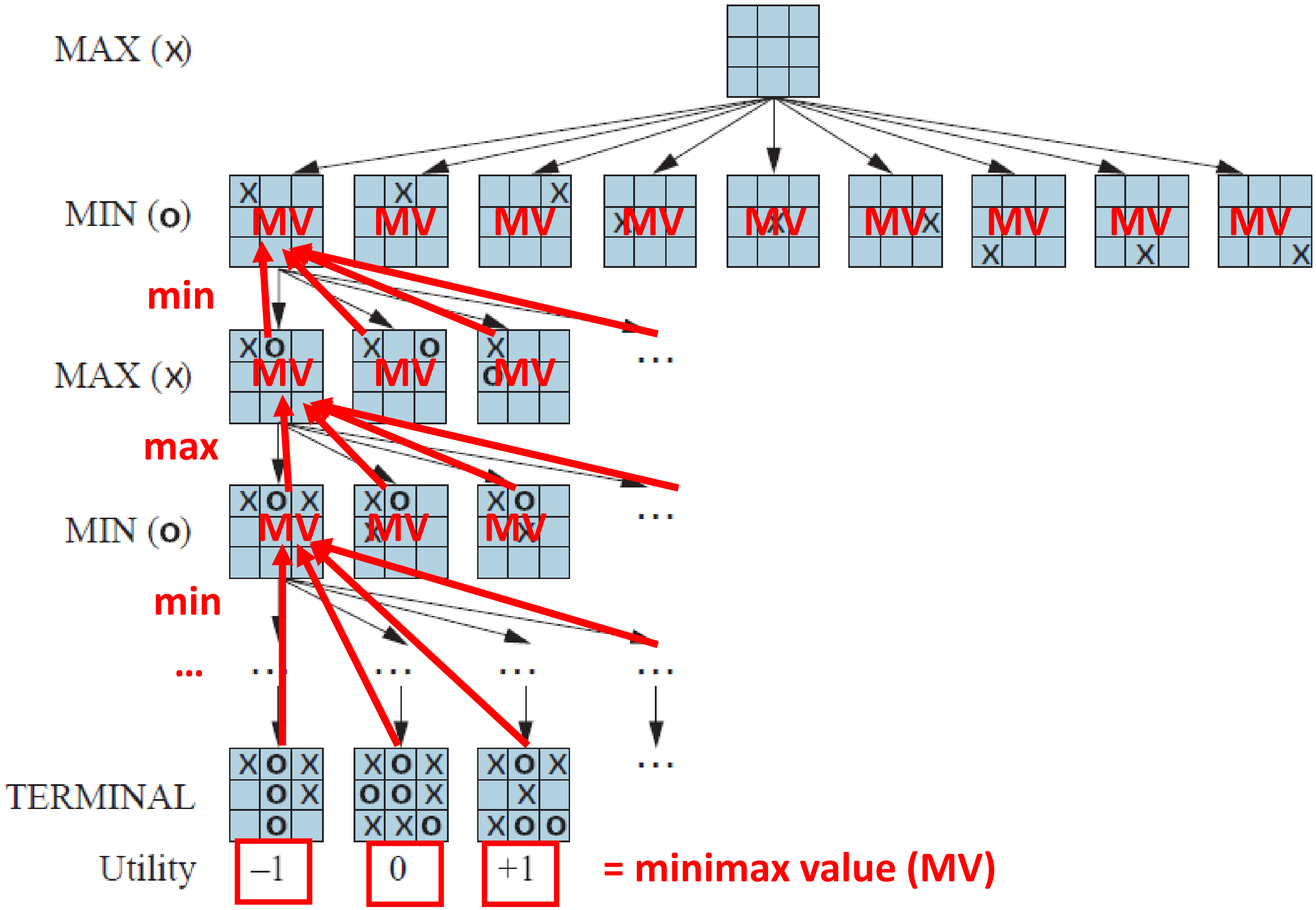
$$Minimax(s) = \begin{cases} Utility(s) & \text{if } terminal(s) \\ \max_{a \in Actions(s)} Minimax\big(Result(s,a)\big) & \text{if } move = Max \\ \min_{a \in Actions(s)} Minimax\big(Result(s,a)\big) & \text{if } move = Min \end{cases}$$

- The minimax value is the utility for Max in state s assuming that **both players play optimally** from s to the end of the game.
- The **optimal decision** for Max is the action that leads to the state with the largest minimax value.

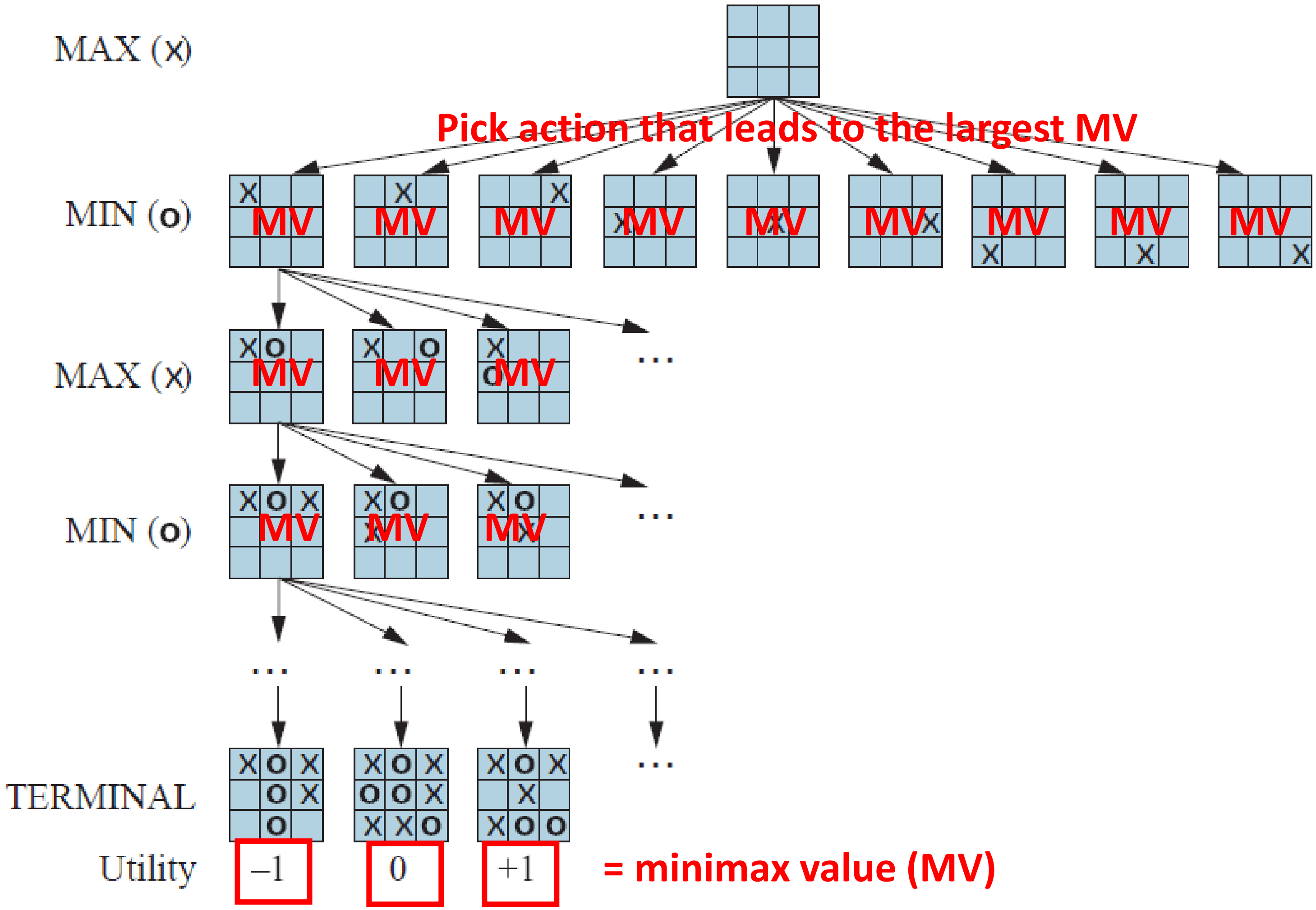
Minimax Search



Minimax Search : Back-up Minimax Values



Minimax Search: Decision



Approach: Follow tree to each terminal node and back up minimax value.

Note: This is just a generalization of the And-Or Tree Search and returns first action of the conditional plan.

function MINIMAX-SEARCH(*game*, *state*) **returns** *an action*
  player $\leftarrow$ *game*.TO-MOVE(*state*)
  value, *move* $\leftarrow$ MAX-VALUE(*game*, *state*)
  return *move*

function MAX-VALUE(*game*, *state*) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow -\infty$
  for each a **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* $\leftarrow$ MIN-VALUE(*game*, *game*.RESULT(*state*, *a*))
    if $v2 > v$ **then**
      v, *move* $\leftarrow$ *v2*, *a*
  return v, *move*

Represents OR Search

function MIN-VALUE(*game*, *state*) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow +\infty$
  for each a **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* $\leftarrow$ MAX-VALUE(*game*, *game*.RESULT(*state*, *a*))
    if $v2 < v$ **then**
      v, *move* $\leftarrow$ *v2*, *a*
  return v, *move*

Represents AND Search

Issue: Game Tree Size

- This traverses the complete game tree using DFS!

Time complexity: $O(b^m)$

- Only feasible for very simple games!
- Example: Tic-tac-toe

  $b = 9, m = 9 \rightarrow O(9^9) = O(387{,}420{,}489)$

  b decreases from 9 to 8, 7, …

  $\rightarrow$ actually less than $O(9!) = O(362{,}880)$

- We need to reduce the search space! -> Pruning

Alpha-Beta Pruning

- **Idea**: Do not search parts of the tree if they do not make a difference to the outcome.
- **Observations**:
  - $\min(3, x, y)$ can never be more than 3
  - $\max(5, \min(3, x, y, \dots))$ does not depend on the values of x or y.
  - Minimax search applies alternating min and max.
- **Approach**: maintain for each state bounds for the minimax value $[alpha, beta]$ and prune subtrees that cannot be part of the solution.
  - Alpha is used by Max and means "$Minimax(s)$ is at least."
  - Beta is used by Min and means "$Minimax(s)$ is at most."

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

=minimax search + pruning.

Note: Pruning can be made more effective by **move ordering**: Check known good moves first to get a good bound early.
Optimal decision algorithms still scale poorly!

Heuristic Alpha-Beta Tree Search

Methods for Adversarial Games

Exact Methods

- **Model as nondeterministic actions**: The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. We **consider all possible moves** by the opponent.
- **Find optimal decisions**: Minimax search and Alpha-Beta pruning where **each player plays optimal** to the end of the game.

Heuristic Methods
(state space is too large)

- **Heuristic Alpha-Beta Tree Search**:
  a. Cut off search tree and use heuristic for utility.
  b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate utility of a state by simulating complete games and average the utility.

Idea: Cutting off search

Stop search in a state before the terminal node is reached. Use a heuristic evaluation function $Eval(s)$ to approximate the utility for that state.

Properties of the evaluation function:

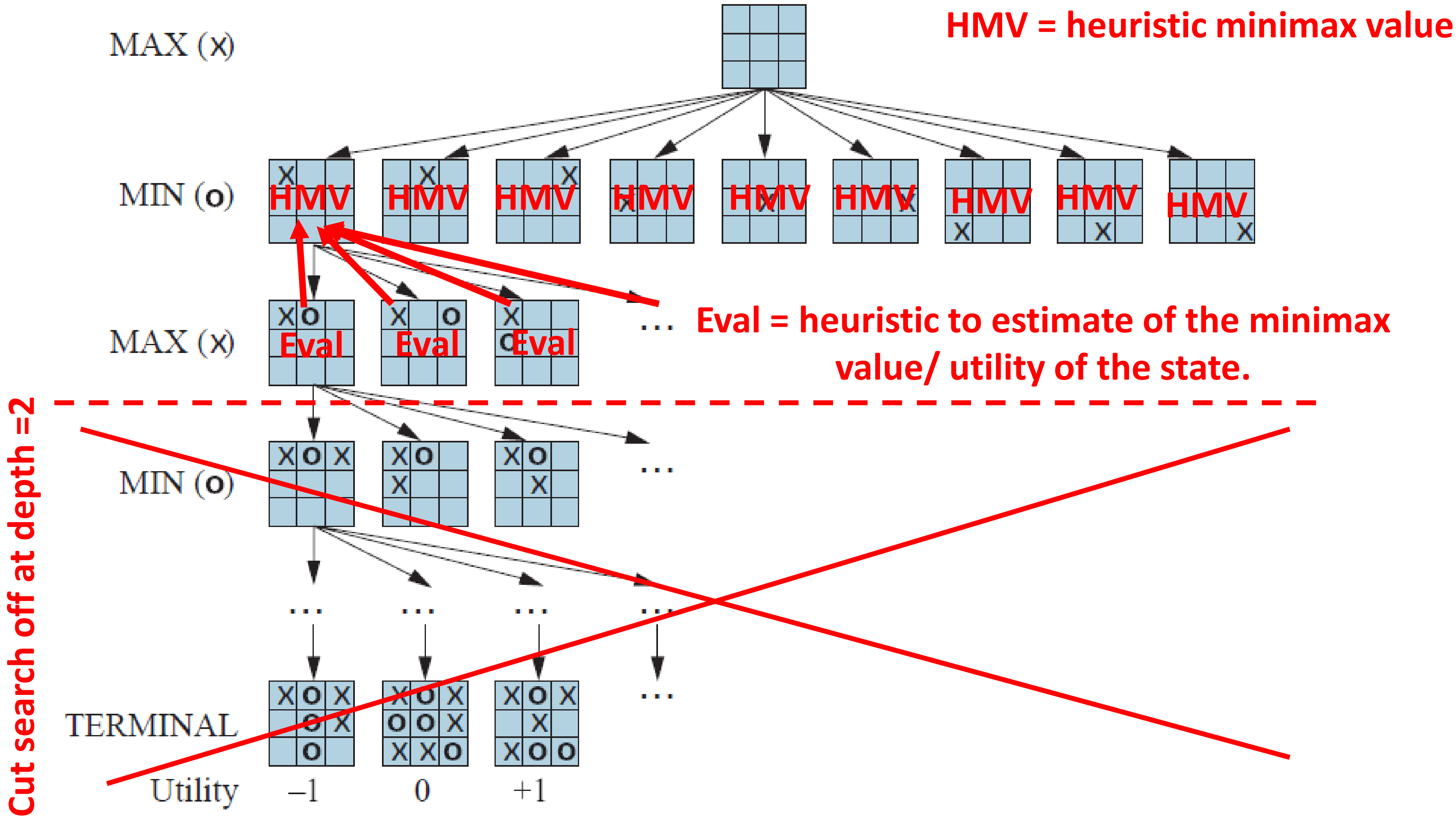
- Fast to compute.
- $Eval(s) \in [Utility(loss), Utility(win)]$
- Correlated with the actual chance of winning (e.g., using features of the state).

Example: A weighted linear function

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

where f_i is a feature of the state.

Heuristic Alpha-Beta Tree Search: Cutting off search



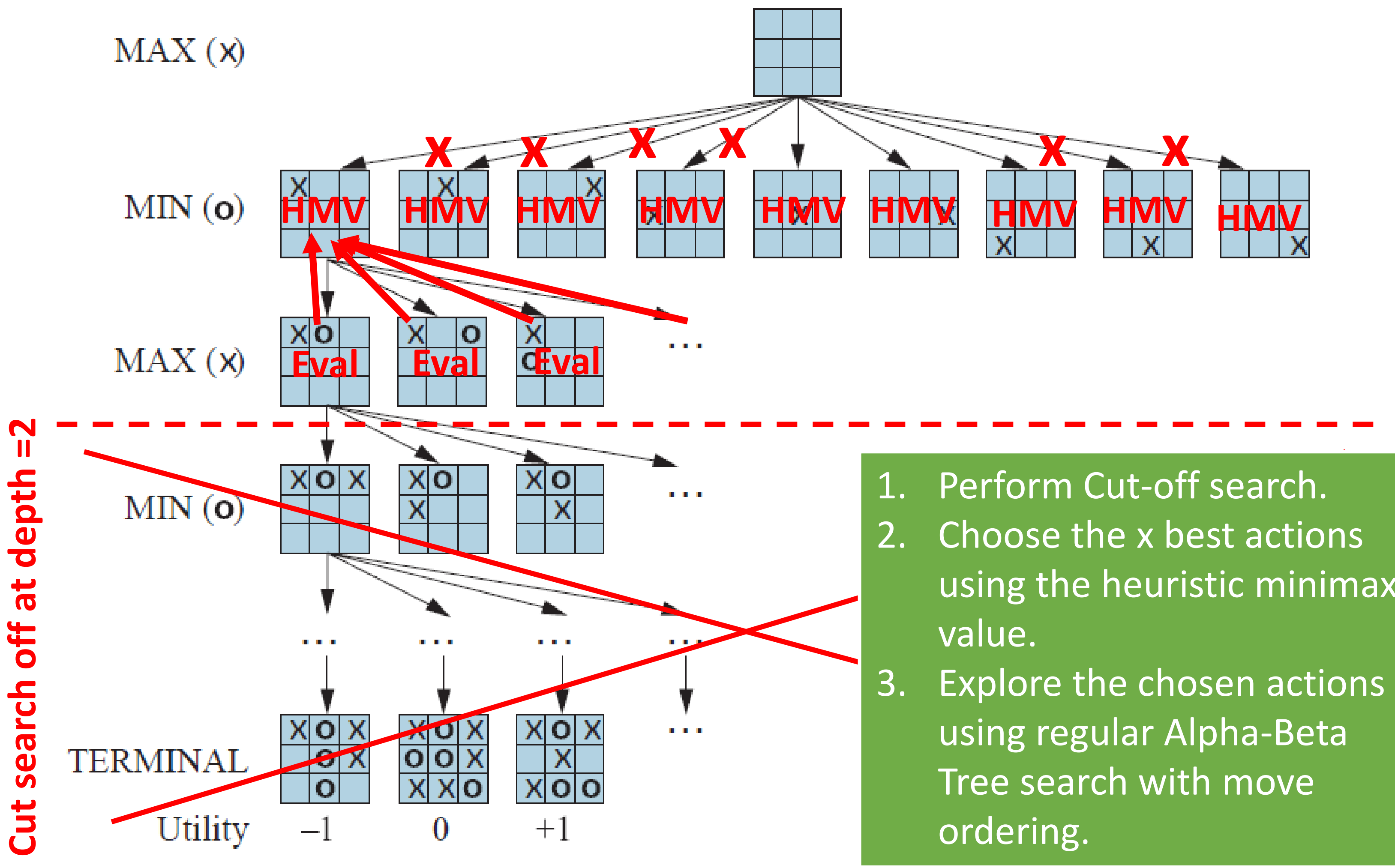
Idea: Forward pruning

Prune moves that appear poor. Poor can be evaluated in several ways:

- Low evaluation value after shallow search.
- Past experience.

Issue: May prune important moves.

Heuristic Alpha-Beta Tree Search: Forward Pruning Example



1. Perform Cut-off search.
2. Choose the x best actions using the heuristic minimax value.
3. Explore the chosen actions using regular Alpha-Beta Tree search with move ordering.

Monte Carlo Tree Search (MCTS)

Methods for Adversarial Games

Exact Methods

- **Model as nondeterministic actions**: The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent. We **consider all possible moves** by the opponent.
- **Find optimal decisions**: Minimax search and Alpha-Beta pruning where **each player plays optimal** to the end of the game.

Heuristic Methods
(state space is too large)

- **Heuristic Alpha-Beta Tree Search**:
    a. Cut off search tree and use heuristic for utility.
    b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate utility of a state by simulating complete games and average the utility.

Idea

- **Approximate** $\boldsymbol{Eval(s)}$ as the average utility of a number of simulation runs to the terminal state (called playouts).
- **Playout policy**: How to choose moves during the simulation runs? Example policies:
  - Random.
  - Heuristics for good moves developed by experts.
  - Learn good moves from self-play (e.g., with deep neural networks).

- Typically used for problems with
  - High branching factor (many possible moves).
  - Unknown or hard to define good evaluation functions.

- **Questions**:
  - How many simulations/playouts should we run each time?
  - From what state should the playouts start?

Pure Monte Carlo Search

- Method
    1. Simulate N playouts from **current state**.
    2. Select the move with the highest win percentage.

- Converges to optimal play for stochastic games as N increases.
- **Do as many playouts as you can** given the available time.

Selection Policy

- Select the starting state for playouts to **focus** on important parts of the game tree. It is a tradeoff between:

  a) **Exploration**: search from states that have few playouts.

  b) **Exploitation**: more playouts for states that have done well to get more accurate estimates.

Selection using Upper Confidence Bounds applied to Trees (UCT)

Tradeoff constant ($\approx \sqrt{2}$) can be optimizes using experiments

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

Average utility (=exploitation)

High for nodes with few playouts (=exploration)

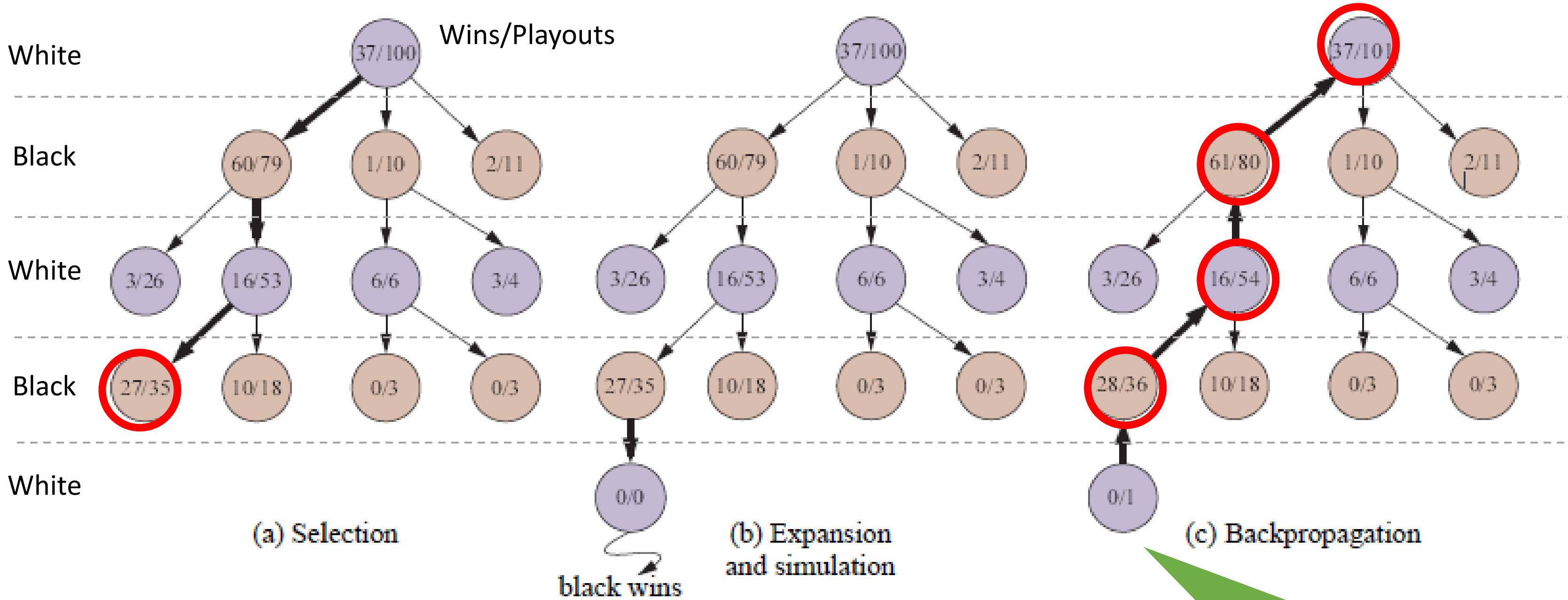
$U(n)$ ... total utility of all playouts going through node n
$N(n)$ ... number of playouts through n

Policy: Select leaf with highest UCB1 score.

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

Highest UCB1 score

UCB1 selection favors win percentage more and more.



(a) Selection

(b) Expansion and simulation

(c) Backpropagation

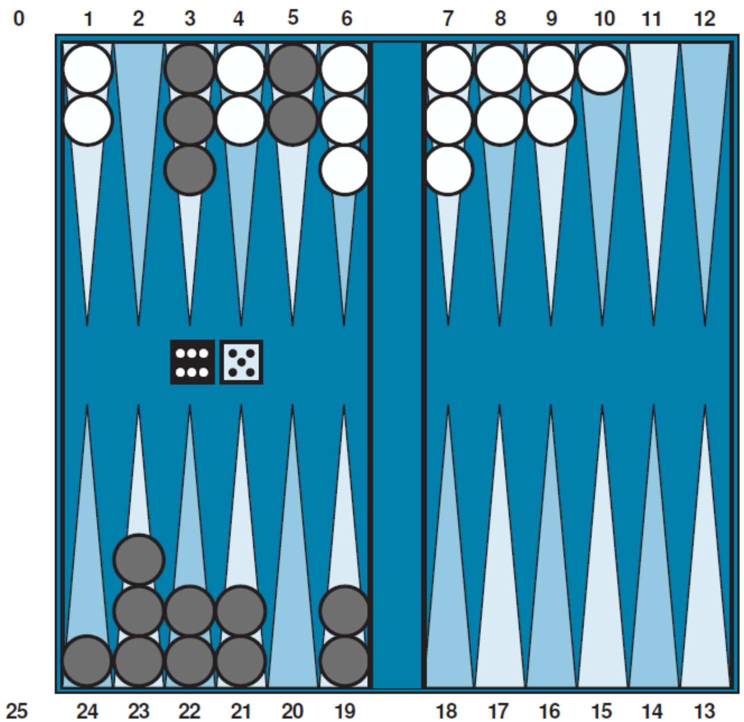
Note: the simulation path is not recorded!

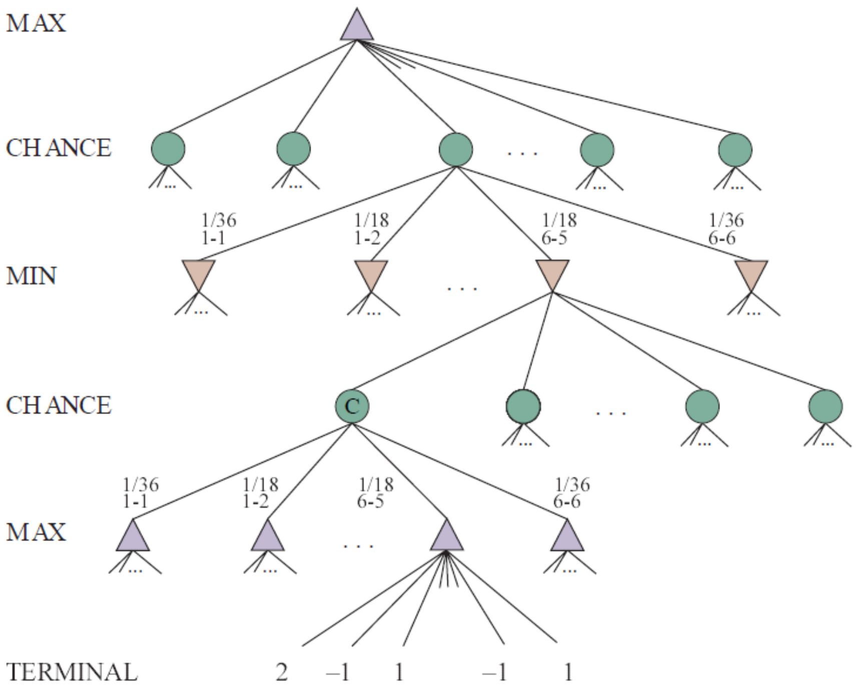
Stochastic Games

Games With Random Events

Stochastic Games

- Game includes a "random action" r (e.g., dice, dealt cards)
- Add **chance nodes** that calculate the expected value.





Backgammon

Expectiminimax

- Game includes a "random action" r (e.g., dice, dealt cards)
- Add **chance nodes** that calculate the expected value.

$$Expectiminimax(s) =$$

$$\begin{cases} Utility(s) & \text{if } terminal(s) \\ \max_{a \in Actions(s)} Expectiminimax\big(Result(s,a)\big) & \text{if } move = Max \\ \min_{a \in Actions(s)} Expectiminimax\big(Result(s,a)\big) & \text{if } move = Min \\ \boxed{\sum_r P(r) Expectiminimax\big(Result(s,r)\big) \qquad \text{if } move = Chance} \end{cases}$$

- Options:
  - Use Minimax algorithm. Issue: Search tree size explodes if the number of "random actions" is large. Think of drawing cards!
  - Approximate Expectiminimax with an evaluation function.
  - Perform Monte Carlo Tree Search.

Conclusion

Nondeterministic actions:

- The opponent is seen as part of an environment with nondeterministic actions. Non-determinism is the result of the unknown moves by the opponent.

Optimal decisions:

- Minimax search and Alpha-Beta pruning where each player plays optimal to the end of the game.
- Choice nodes and Expectiminimax for stochastic games.

Scale only for tiny problems!

Heuristic Alpha-Beta Tree Search:

- Cut off search tree and use heuristic evaluation function for utility (based on state features).
- Forward Pruning: ignore poor moves.

Monte Carlo Tree search:

- Simulate complete games and average the results.
- Learn playout policy using self-play and deep learning.
- Use modified UCB1 scores.

State of the Art