

CS 5/7320
Artificial Intelligence

Adversarial Search and Games

AIMA Chapter 5

Slides by Michael Hahsler
with figures from the AIMA textbook



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

"Reflected Chess pieces" by Adrian Askew

A black and white photograph showing a stack of three white rings with black centers, and a black L-shaped block, all resting on a light-colored surface. The rings are stacked vertically, and the L-shaped block is positioned to the right of the stack.

Games

- Games typically confront the agent with a competitive (adversarial) environment affected by an opponent (strategic environment).
- We will focus on planning for
 - two-player zero-sum games with
 - deterministic game mechanics and
 - perfect information (i.e., fully observable environment).
- 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.



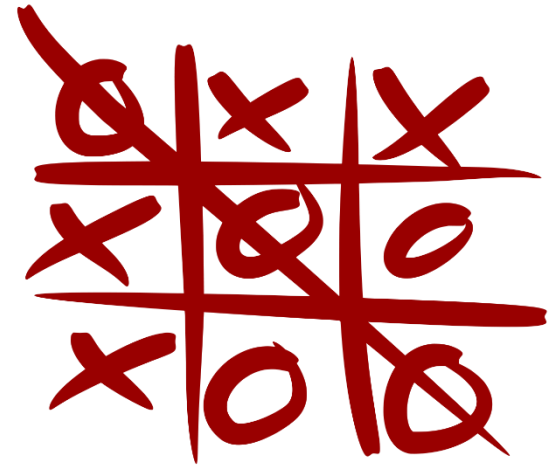
Definition of a Game

- **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?

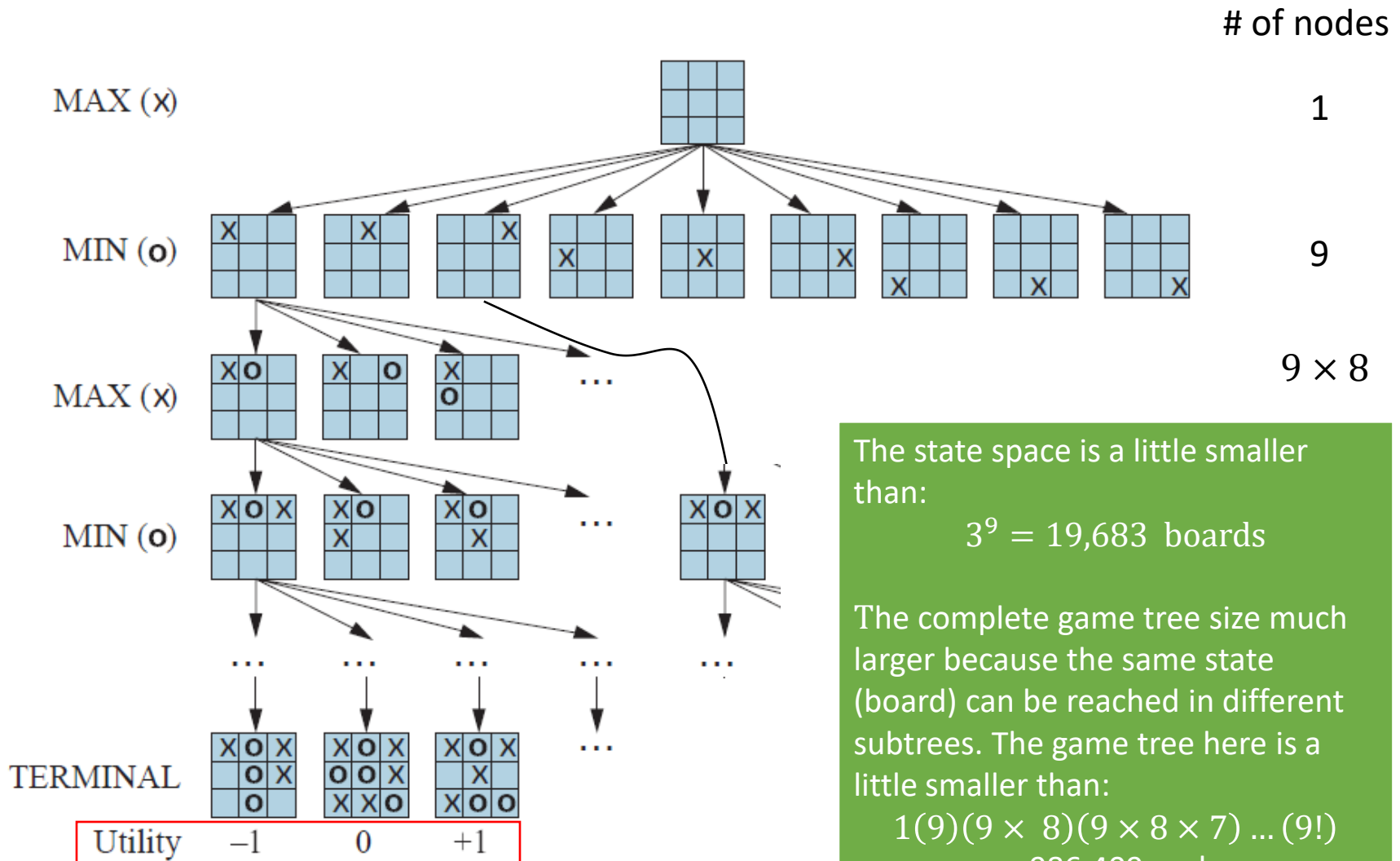
$Utility(s)$

+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



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

(game tree is too large)

- **Heuristic Alpha-Beta Tree Search:**
 - a. Cut off game 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.

A dynamic background image showing a bright yellow powder or dust exploding or dispersing against a solid black background. The powder forms a large, billowing cloud that fills the right side and top of the frame, with many fine particles visible in the air.

Nondeterministic Actions

Recall AND-OR Search from AIM A 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

(game tree is too large)

- **Heuristic Alpha-Beta Tree Search:**
 - a. Cut off game 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

For **planning**, we do not know what the opponents moves will be. This is the same situation as not being to sense the opponents moves during a real game which we have already modeled using 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 (which 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 \leftarrow AND-SEARCH(*problem*, RESULTS(*state*, *action*), [*state*] + *path*)
 if *plan* \neq *failure* **then return** [*action*] + *plan*
return failure

my
moves

all states that can result from
opponent's moves

function AND-SEARCH(*problem*, *states*, *path*) **returns** a conditional plan, or *failure*
for each s_i **in** *states* **do** // check all possible current states
 *plan*_{*i*} \leftarrow 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}]

abandon subtree if a loss is found

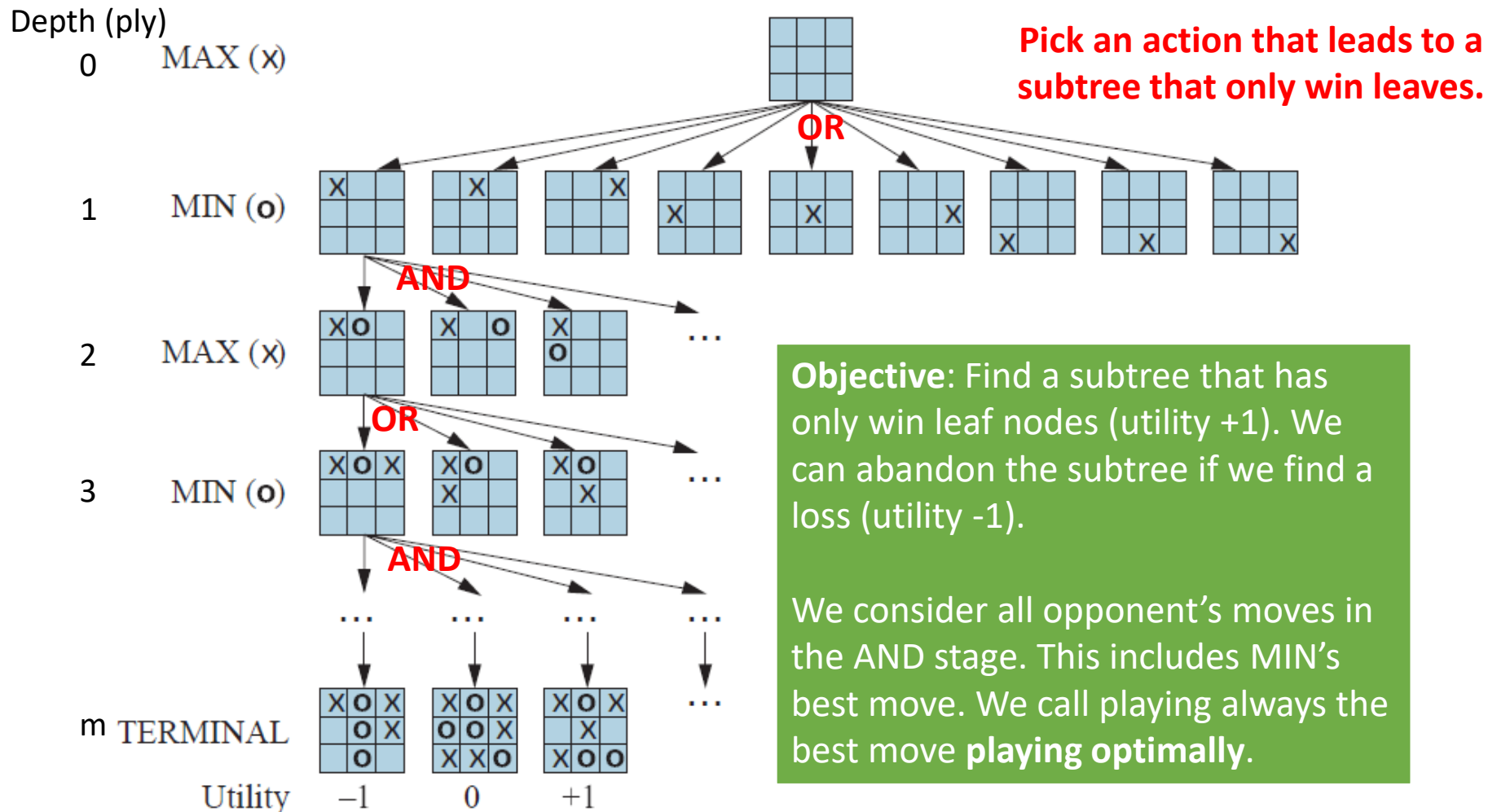
Go through
opponent
moves

- 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.

Tic-tac-toe: AND-OR Search

We play MAX and decide on our actions (OR).

MIN's actions introduce non-determinism (AND).





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

(game tree is too large)

- **Heuristic Alpha-Beta Tree Search:**
 - a. Cut off game 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.

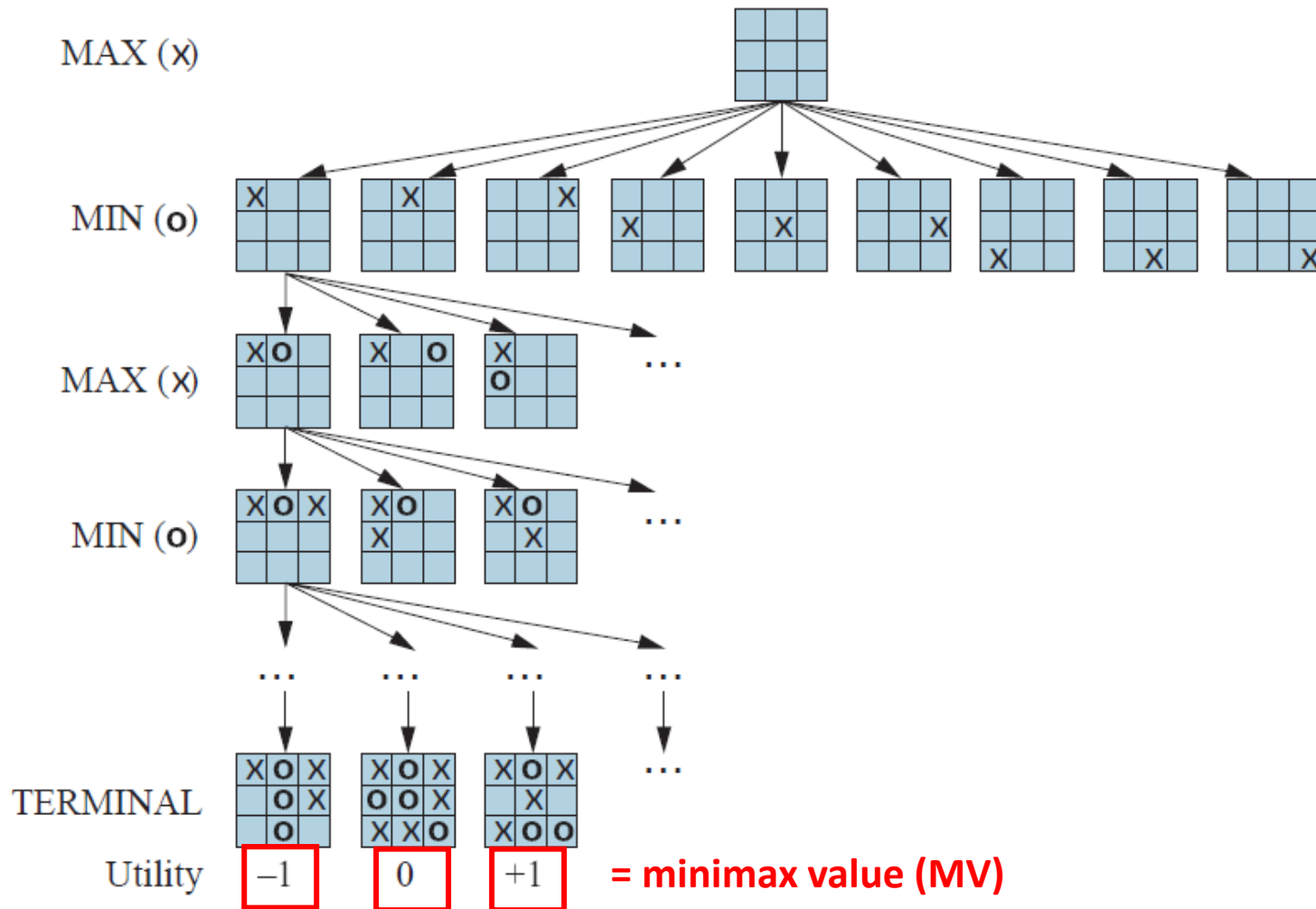
Idea: Minimax Decision

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

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

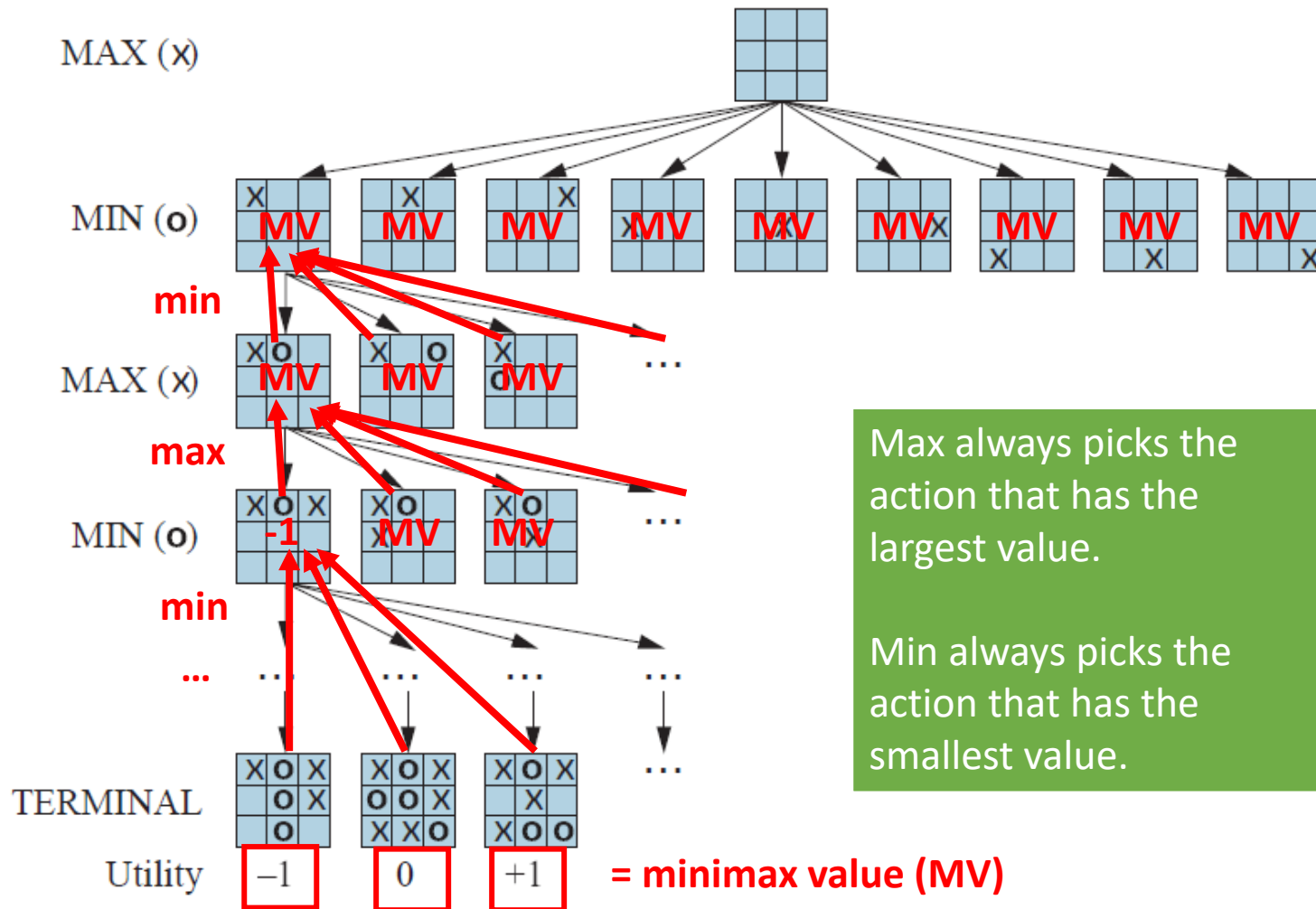
- The minimax value is the utility for Max in state s assuming that **both players play optimally** from s to the end of the game.
- The **optimal decision** for Max is the action that leads to the state with the largest minimax value.

Minimax Search

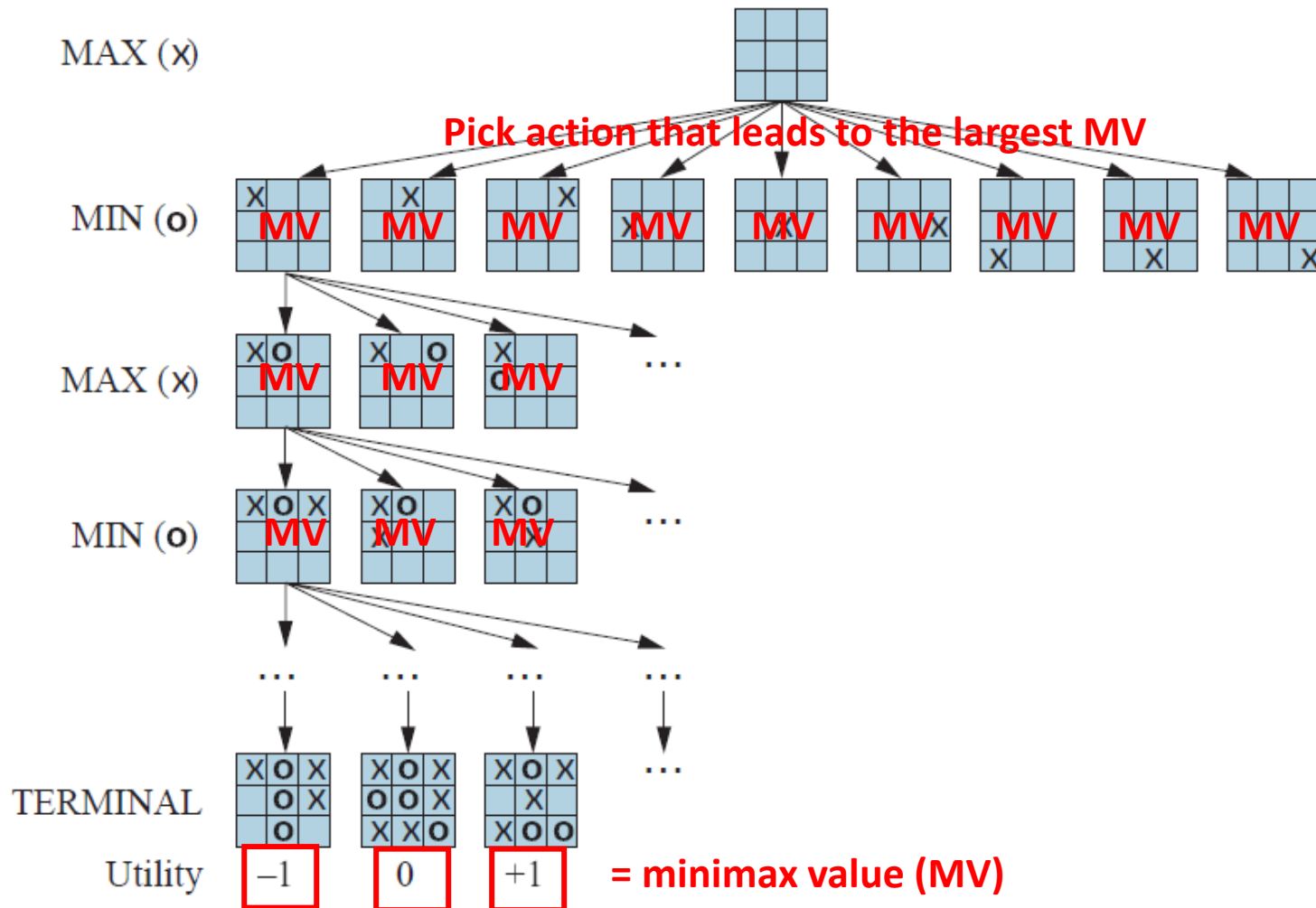


Minimax Search : Back-up

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

b: branching factor
m: max depth of tree

Issue: Game Tree Size

- **Minimax search traverses the complete game tree using DFS!**

Time complexity: $O(b^m)$

- Only feasible for very simple games with small branching factor!

- Example: Tic-tac-toe

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

b decreases from 9 to 8, 7, ... the actual size is smaller than:

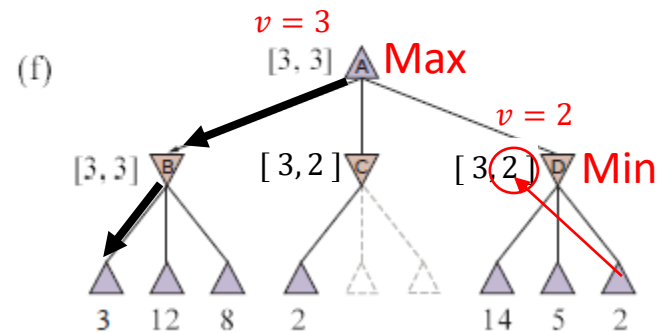
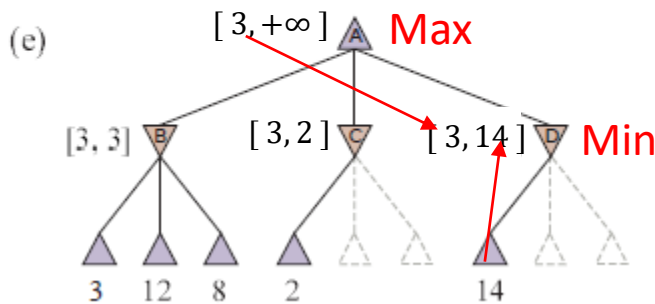
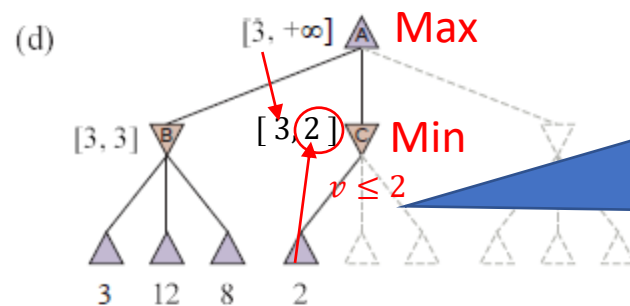
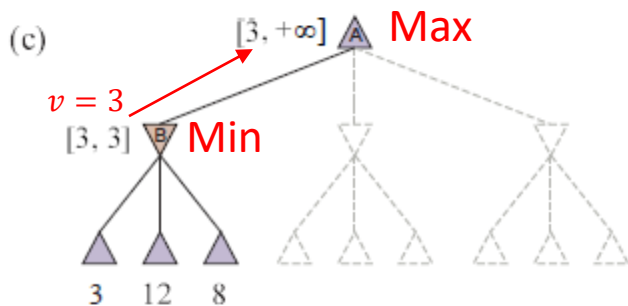
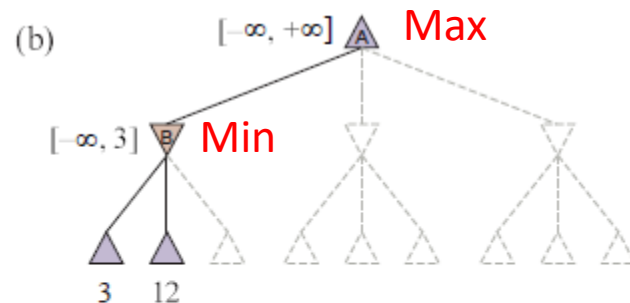
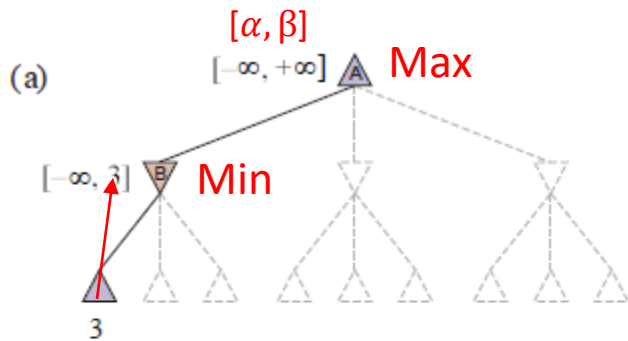
$$1(9)(9 \times 8)(9 \times 8 \times 7) \dots (9!) = 986,409 \text{ nodes}$$

- **We need to reduce the search space! → Game tree 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 bounds for the minimax value $[\alpha, \beta]$ and prune subtrees (i.e., don't follow actions) that do not affect the current minimax value bound.
 - Alpha is used by Max and means "*Minimax*(s) is at least α ."
 - Beta is used by Min and means "*Minimax*(s) is at most β ."

Example: Alpha-Beta Search



Max updates α
 (utility is at least)

Min updates β
 (utility is at most)

Utility cannot be more than 2 in the subtree, but we already can get 3 from the first subtree. Prune the rest.

Once a subtree is fully evaluated, the interval has a length of 0 ($\alpha = \beta$).

```
function ALPHA-BETA-SEARCH(game, state) returns an action  
  player  $\leftarrow$  game.TO-MOVE(state)  
  value, move  $\leftarrow$  MAX-VALUE(game, state,  $-\infty$ ,  $+\infty$ )  
  return move
```

= minimax search + pruning

```
function MAX-VALUE(game, state,  $\alpha$ ,  $\beta$ ) returns a (utility, move) pair  
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null  
  v  $\leftarrow$   $-\infty$  // v is the minimax value  
  for each a in game.ACTIONS(state) do  
    v2, a2  $\leftarrow$  MIN-VALUE(game, game.RESULT(state, a),  $\alpha$ ,  $\beta$ )  
    if v2 > v then Found a better action?  
      v, move  $\leftarrow$  v2, a  
       $\alpha \leftarrow$  MAX( $\alpha$ , v)  
      if v  $\geq$   $\beta$  then return v, move Stop searching if Max finds an actions  
that has more value than the best  
move Min has in another subtree.  
  return v, move
```

```
function MIN-VALUE(game, state,  $\alpha$ ,  $\beta$ ) 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),  $\alpha$ ,  $\beta$ )  
    if v2 < v then Found a better action?  
      v, move  $\leftarrow$  v2, a  
       $\beta \leftarrow$  MIN( $\beta$ , v)  
      if v  $\leq$   $\alpha$  then return v, move Stop searching if Min finds an actions  
that has less value than the best  
move Max has in another subtree.  
  return v, move
```

Notes:

- 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!

A collection of colorful wooden Tetris blocks scattered on a wooden surface. The blocks are in various colors including purple, blue, green, orange, red, pink, grey, brown, and yellow. They are arranged in a way that suggests a game of Tetris, with some blocks already placed and others waiting to be dropped. The text "Heuristic Alpha-Beta Tree Search" is overlaid in the center in a white, sans-serif font.

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

(game tree is too large or search takes too long)

- **Heuristic Alpha-Beta Tree Search:**
 - a. Cut off game 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.

Cutting off search

Reduce the search cost by restricting the search depth:

1. Stop search at a non-terminal node.
2. Use a heuristic evaluation function $Eval(s)$ to approximate the utility for that node/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

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

where f_i is a feature of the state (e.g., # of pieces captured in chess).

Heuristic Alpha-Beta Tree Search: Cutting off search

HMV = heuristic minimax value

Depth (ply)

0 MAX (x)

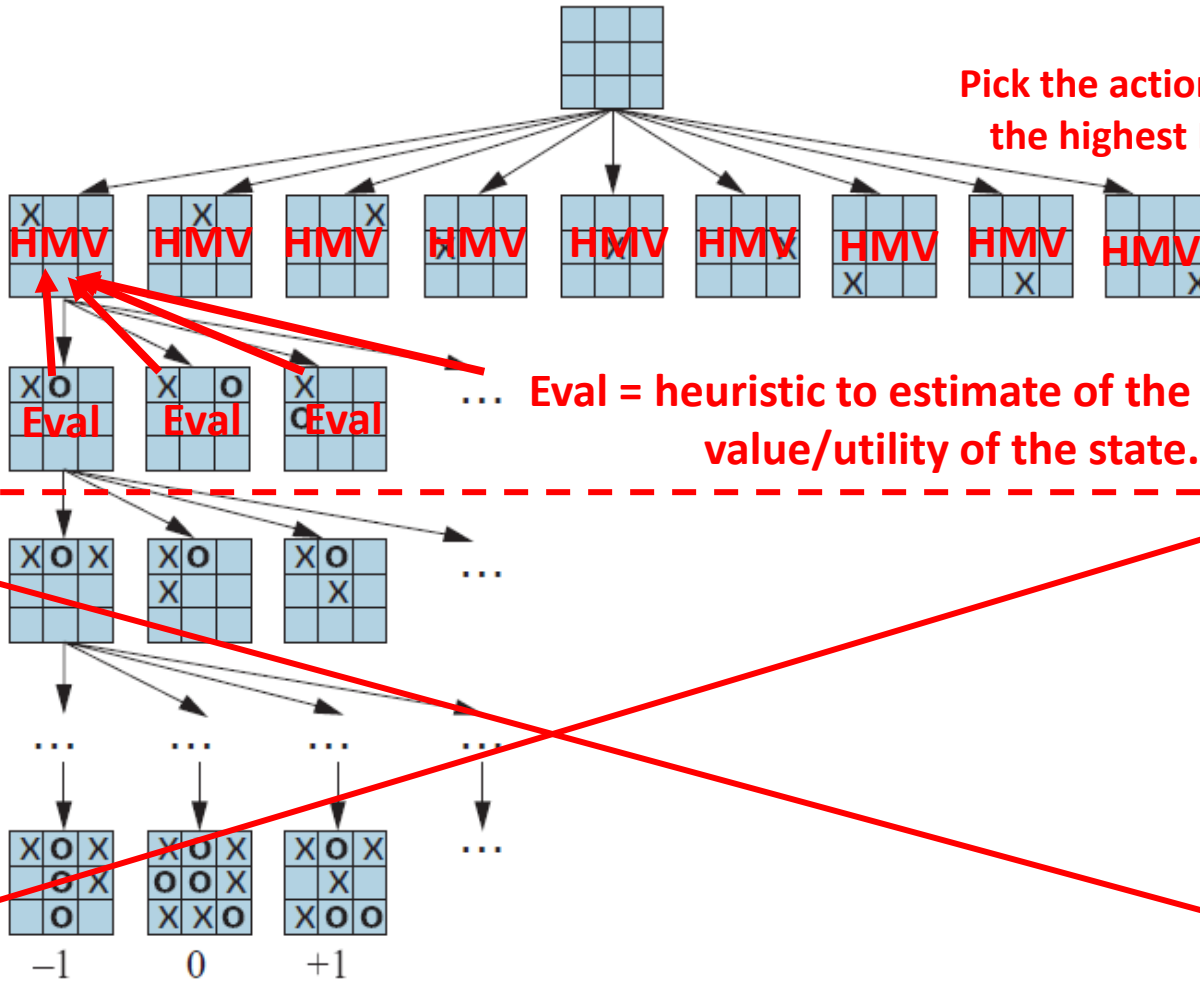
1 MIN (o)

2 MAX (x)

3 MIN (o)

TERMINAL

Utility



Forward pruning

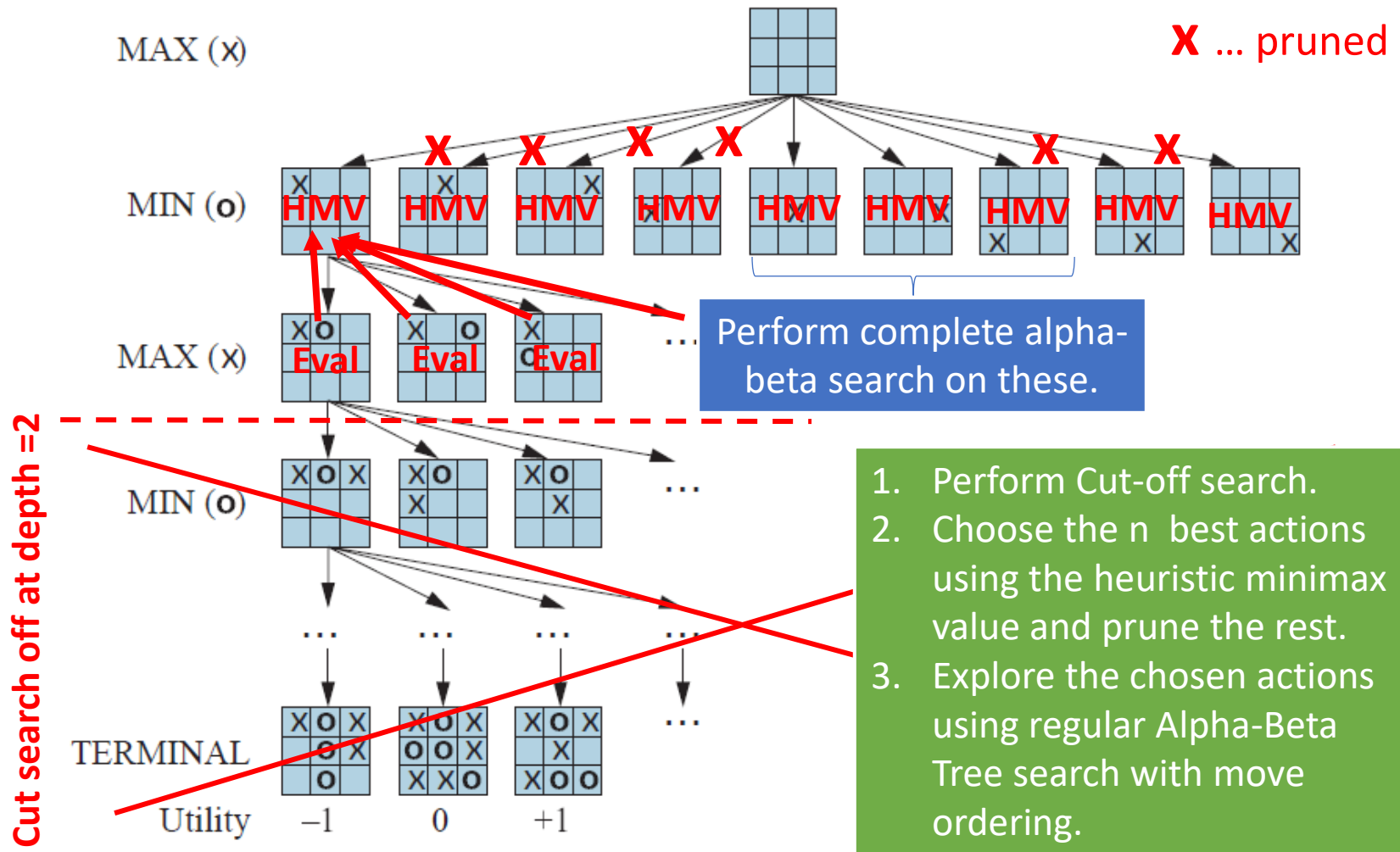
To save time, we can prune moves that appear bad.

There are many ways move quality can be evaluated:

- Low heuristic value.
- Low evaluation value after shallow search (cut-off search).
- Past experience.

Issue: May prune important moves.

Heuristic Alpha-Beta Tree Search: Example for Forward Pruning



A close-up, slightly blurred image of a roulette wheel. The wheel is red with black numbers and green pockets. The text "Monte Carlo Tree Search (MCTS)" is overlaid in white, centered on the wheel. The wheel is tilted, and the numbers are visible in a circular pattern. The text is in a clean, sans-serif font.

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

(game tree is too large or search takes too long)

- **Heuristic Alpha-Beta Tree Search:**
 - a. Cut off game 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 $Eval(s)$** as the average utility of several 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). We will talk about this when we talk about “Learning from Examples.”
- Typically used for problems with
 - High branching factor (many possible moves).
 - Unknown or hard to define good evaluation functions.

Pure Monte Carlo Search

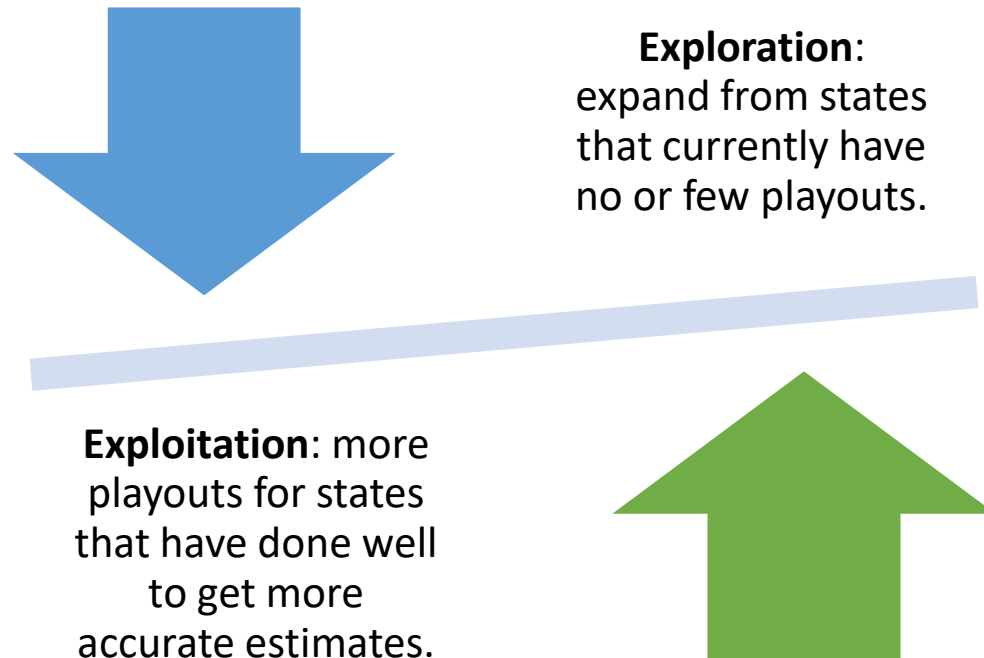
Find the next best move.

- Method
 1. Simulate N playouts from **current state**.
 2. Select the move that leads the highest win percentage.
- **Guarantee:** Converges to optimal play for stochastic games as N increases.
- **Do as many playouts as you can** given the available time.

Playout Selection Strategy

Pure Monte Carlo Search spends a lot of time to create playouts for bad moves.

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



Selection using Upper Confidence Bounds (UCB1)

Tradeoff constant $\approx \sqrt{2}$
can be optimized using experiments

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

Average utility
(=exploitation)

High for nodes with few playouts relative
to the parent node (=exploration)

n ... node in the game tree

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

$N(n)$... number of playouts through n

Playout strategy: Select node with highest UCB1 score.

Monte Carlo Tree Search

We do not want to always start playouts from the current node, so we build a partial game tree and simulate from a node in that tree.

Important considerations:

- We can only store a small part of the game tree.
- We can use UCB1 so decide what part of the tree should focus on.

function MONTE-CARLO-TREE-SEARCH(*state*) *returns an action*

tree \leftarrow NODE(*state*)

while IS-TIME-REMAINING() **do**

leaf \leftarrow SELECT(*tree*)

Highest UCB1 score

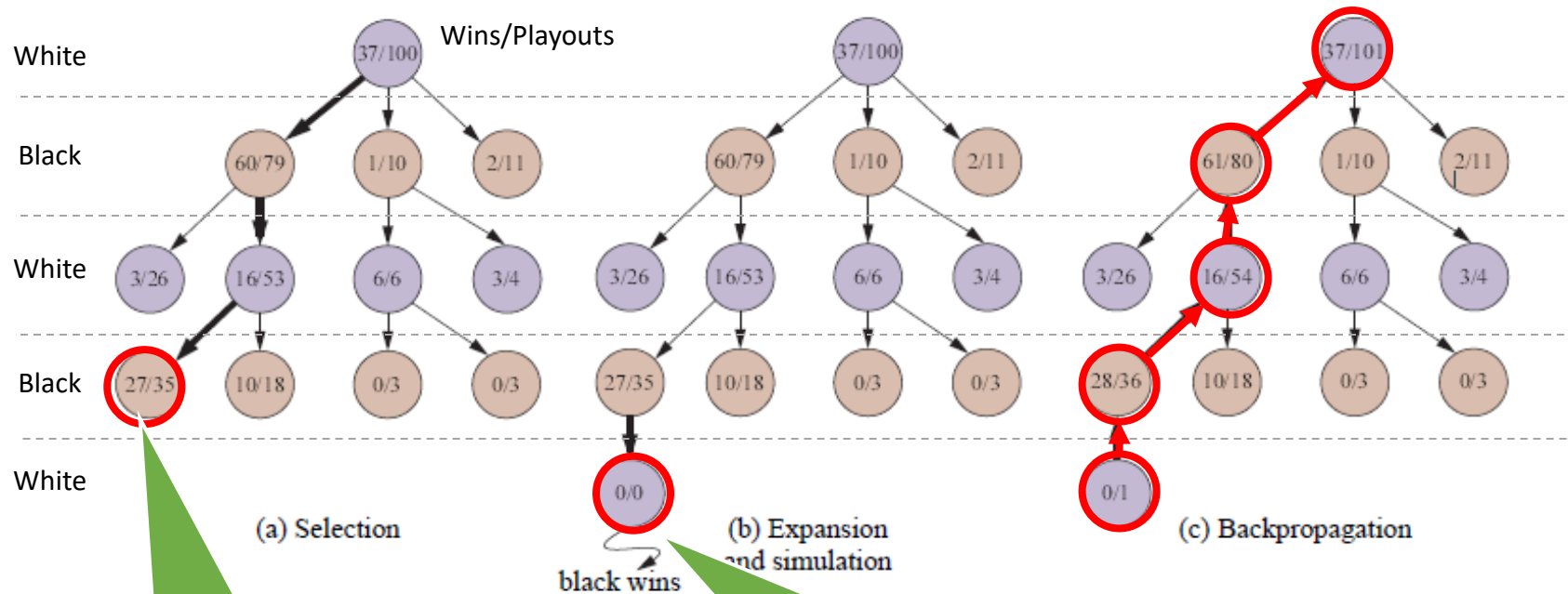
child \leftarrow EXPAND(*leaf*)

result \leftarrow SIMULATE(*child*)

BACK-PROPAGATE(*result*, *child*)

UCB1 selection favors win percentage more and more.

return the move in ACTIONS(*state*) whose node has highest number of playouts

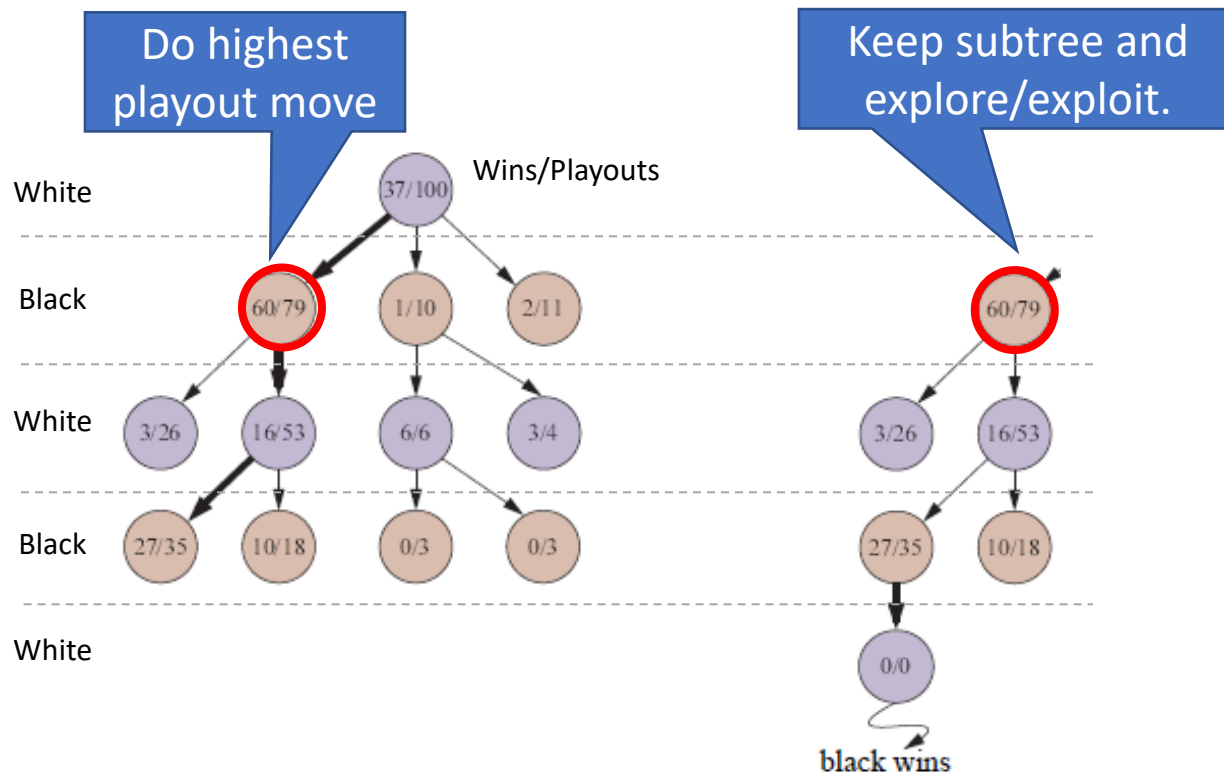


Select highest UCB1 score node

Note: the simulation path is not recorded to preserve memory!

Online Play Using MCTS

- Search to use up the time budget for the move.
- Keep the relevant subtree from move to move and expand from there.



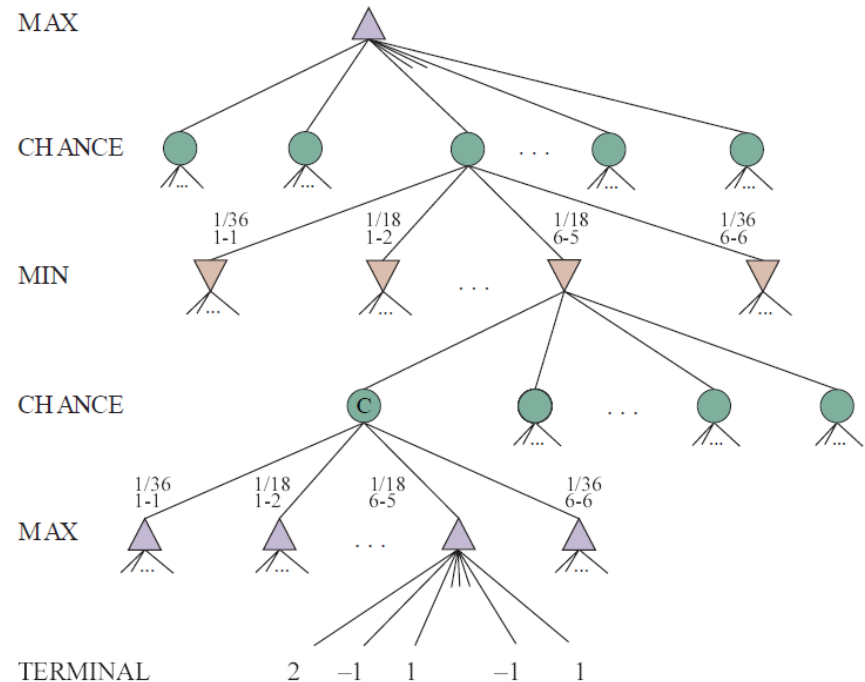
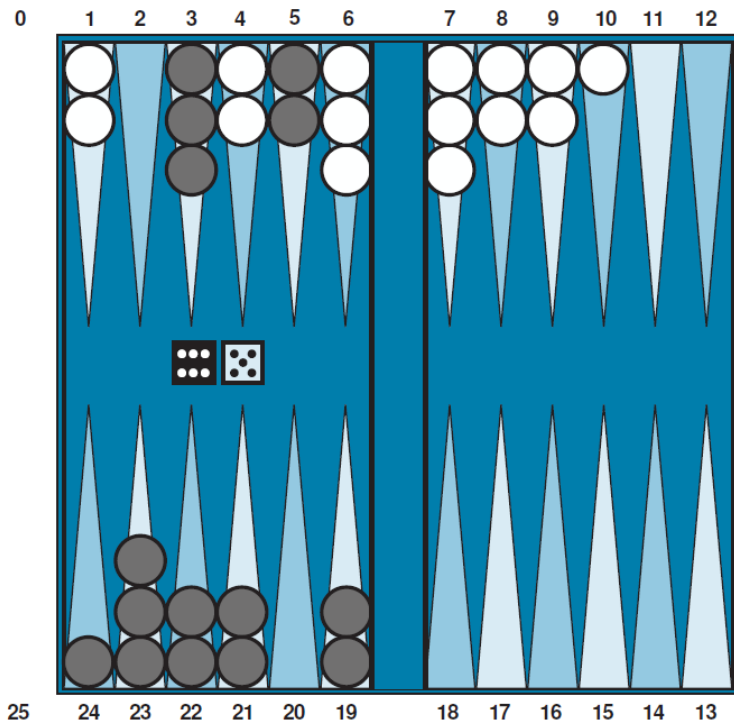
The background of the slide is a close-up photograph of four dice resting on a wooden surface with a diagonal grain. There are two black dice with white pips and two white dice with black pips. The dice are arranged in a loose cluster, with one black die in the lower-left foreground, one white die in the center, and two more dice (one white, one black) in the upper-right background. The lighting is soft, creating gentle shadows.

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)
- For **chance nodes** we calculate the expected minimax value.

$Expectiminimax(s) =$

$$\left\{ \begin{array}{ll} Utility(s) & \text{if } terminal(s) \\ \max_{a \in Actions(s)} Expectiminimax(Result(s, a)) & \text{if } move = Max \\ \min_{a \in Actions(s)} Expectiminimax(Result(s, a)) & \text{if } move = Min \\ \sum_r P(r) Expectiminimax(Result(s, r)) & \text{if } move = Chance \end{array} \right.$$

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

Other approaches

The high branching factor favors:

- Heuristic Expectiminimax Search
- Monte Carlo Tree Search
- Learning evaluation functions from data with self-play (see machine learning).

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. *All possible moves are considered.*

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.

Heuristic Alpha-Beta Tree Search:

- Cut off game 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 calculate proportion of wins.
- Use modified UCB1 scores to expand the game tree.
- Learn playout policy using self-play and deep learning.

Scale only for tiny problems!

State of the Art