

CS 5/7320
Artificial Intelligence

Adversarial Search and Games

AIMA Chapter 5

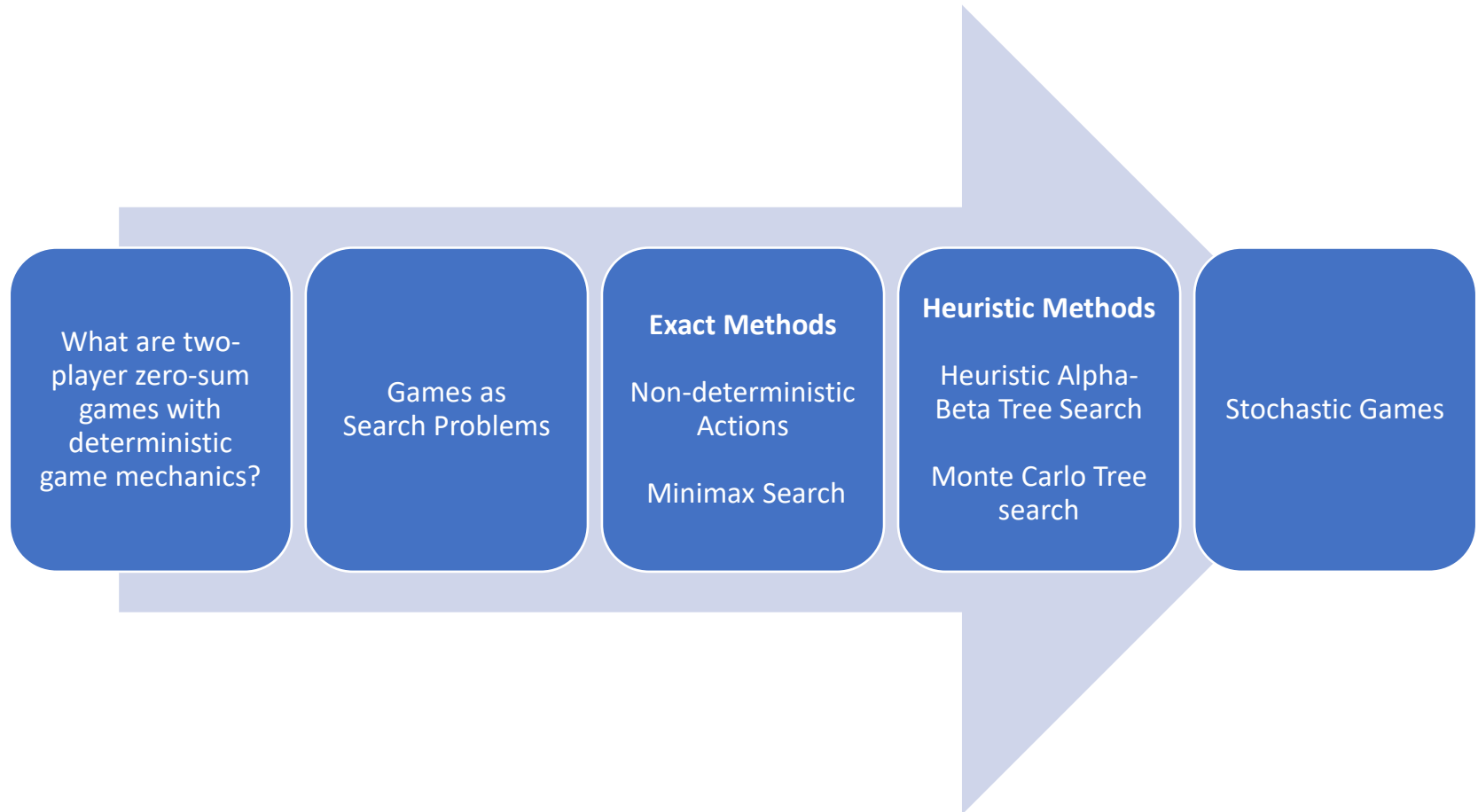
Slides by Michael Hahsler
with figures from the AIMA textbook



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

Contents



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).
- Games are episodic.
- 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.

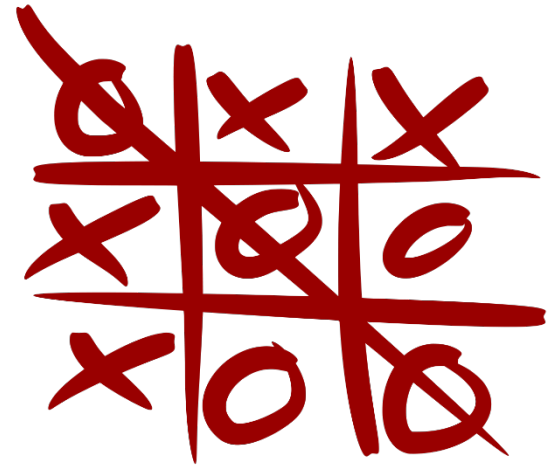


Definition of a Game

Definition:

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

Example: Tic-tac-toe



s_0

Empty board.

$Actions(s)$

Play empty squares.

$Result(s, a)$

Symbol (x/o) is placed 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.

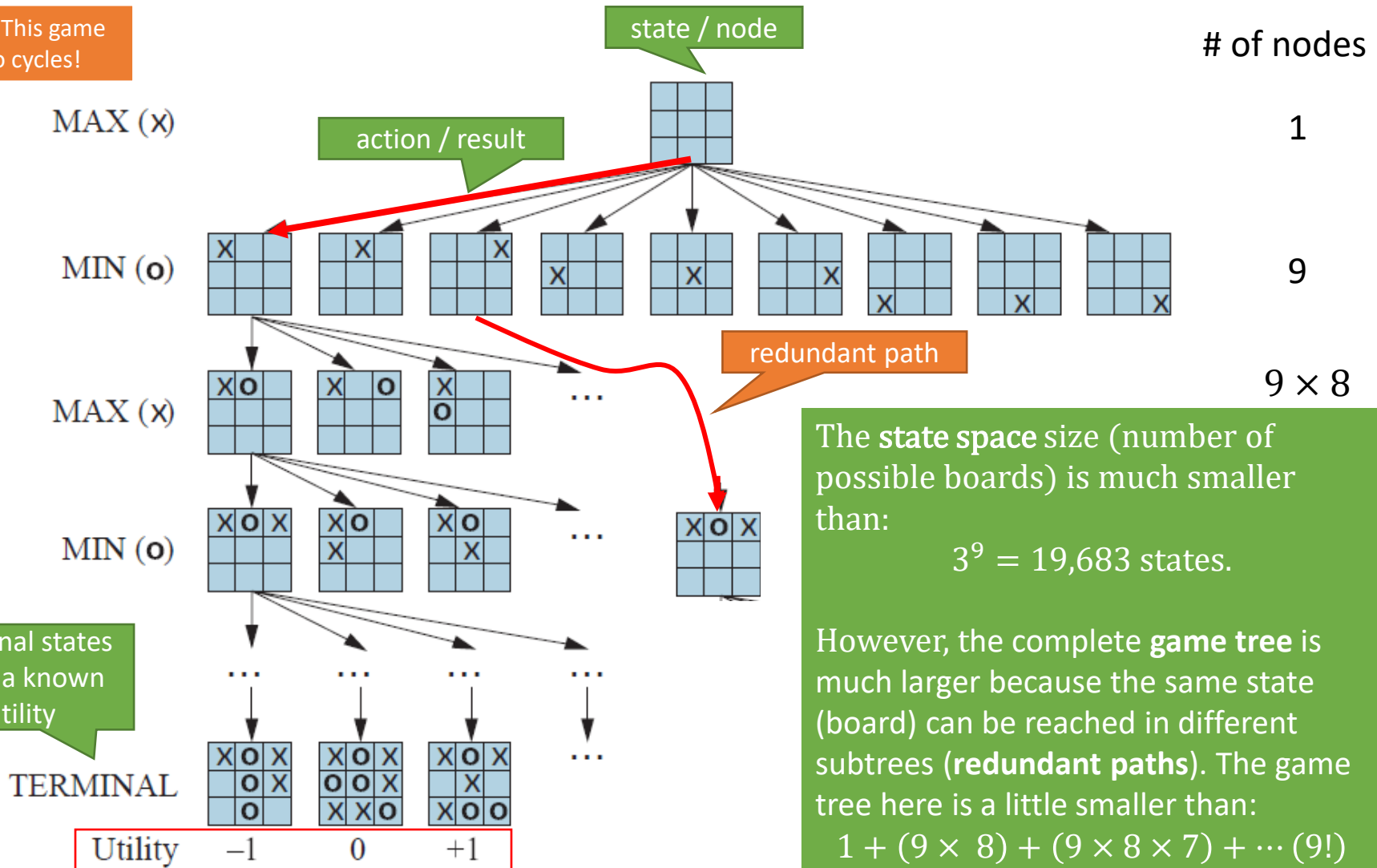
Note: This game still uses a goal-based agent that plans actions to reach a winning terminal state!

Games as Search Problems

- Making a move is a **decision problem** that can be addressed as a **search problem**. We need to search for sequences of moves that lead to a winning position.
- **Search problems have a state space**: a graph defined by the initial state and the transition function containing all reachable states (e.g., chess positions).
- **For games we consider a game tree**: A complete game tree follows every sequence from the current state to the terminal state (the game ends). It consists of the set of paths through the state space representing all possible games that can be played.

Tic-tac-toe: Partial Game Tree

Note: This game has no cycles!



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 smoke explosion against a black background. The particles are concentrated on the right side and spread out towards the left, creating a sense of motion and energy.

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


(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.

Recall: Nondeterministic Actions

For **planning**, we do not know what the opponents moves will be. We have already modeled this issue using nondeterministic actions.

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



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

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).

Recall: 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}]

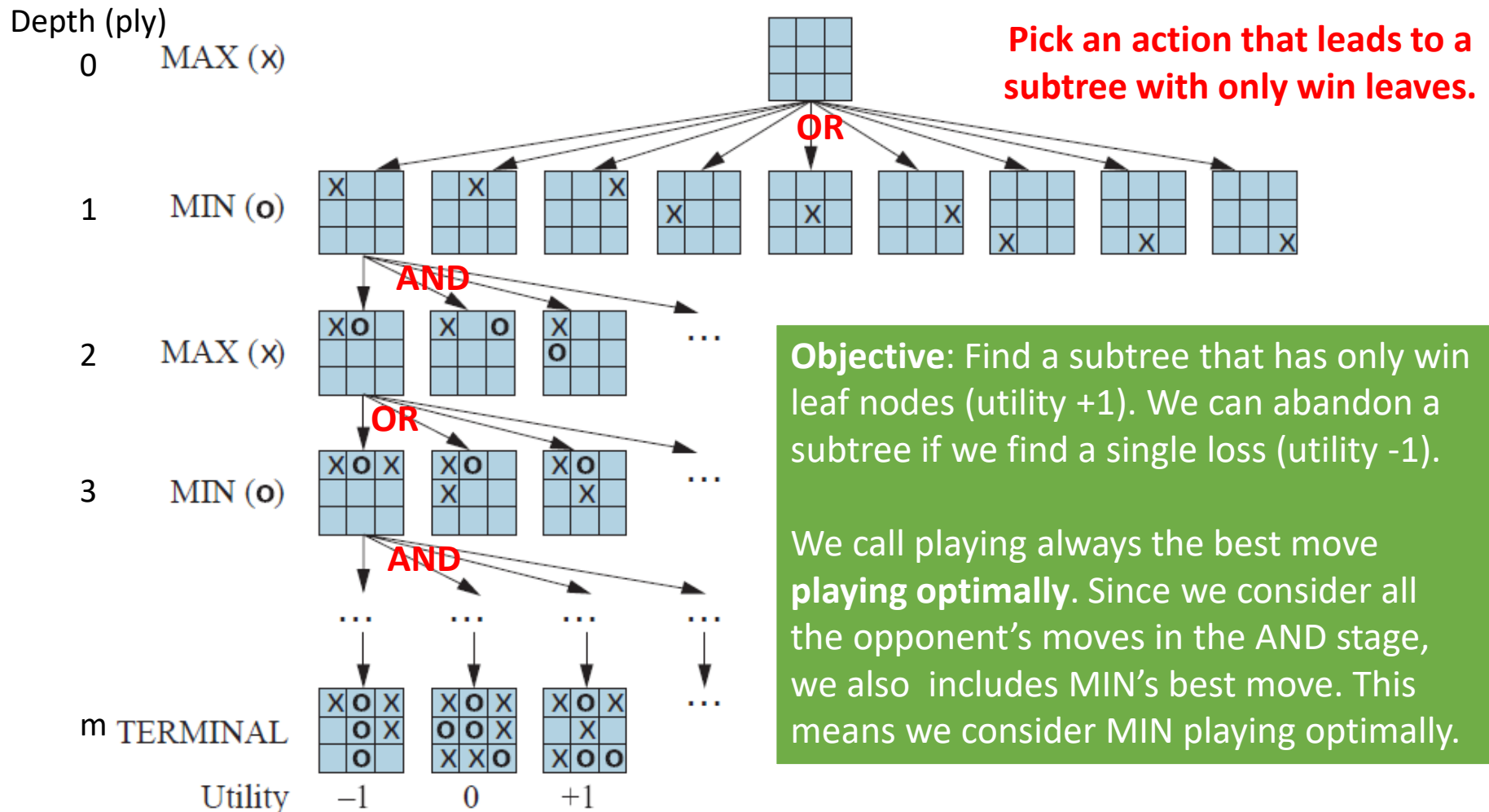
abandon subtree if a loss is found

Go through
opponent
moves

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

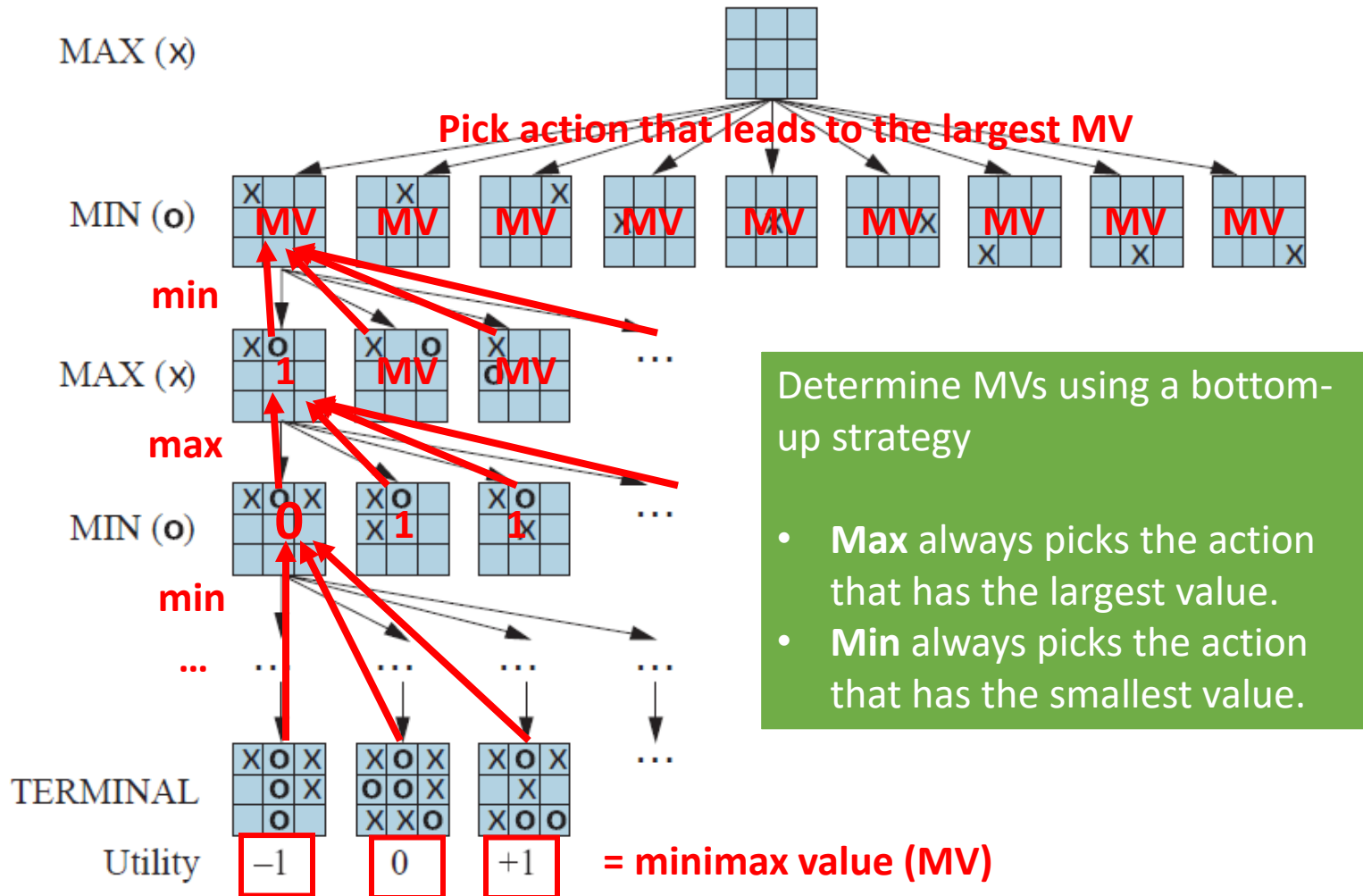
- Assign each state s a **minimax value** that reflects the utility realized if **both players play optimally** from s to the end of the game:

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

- This is a recursive definition which can be solved from terminal states backwards.
- The **optimal decision** for Max is the action that leads to the state with the largest minimax value. That is the largest possible utility if both players play optimally.

Minimax Search: Back-up

Minimax Values



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 the 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
```

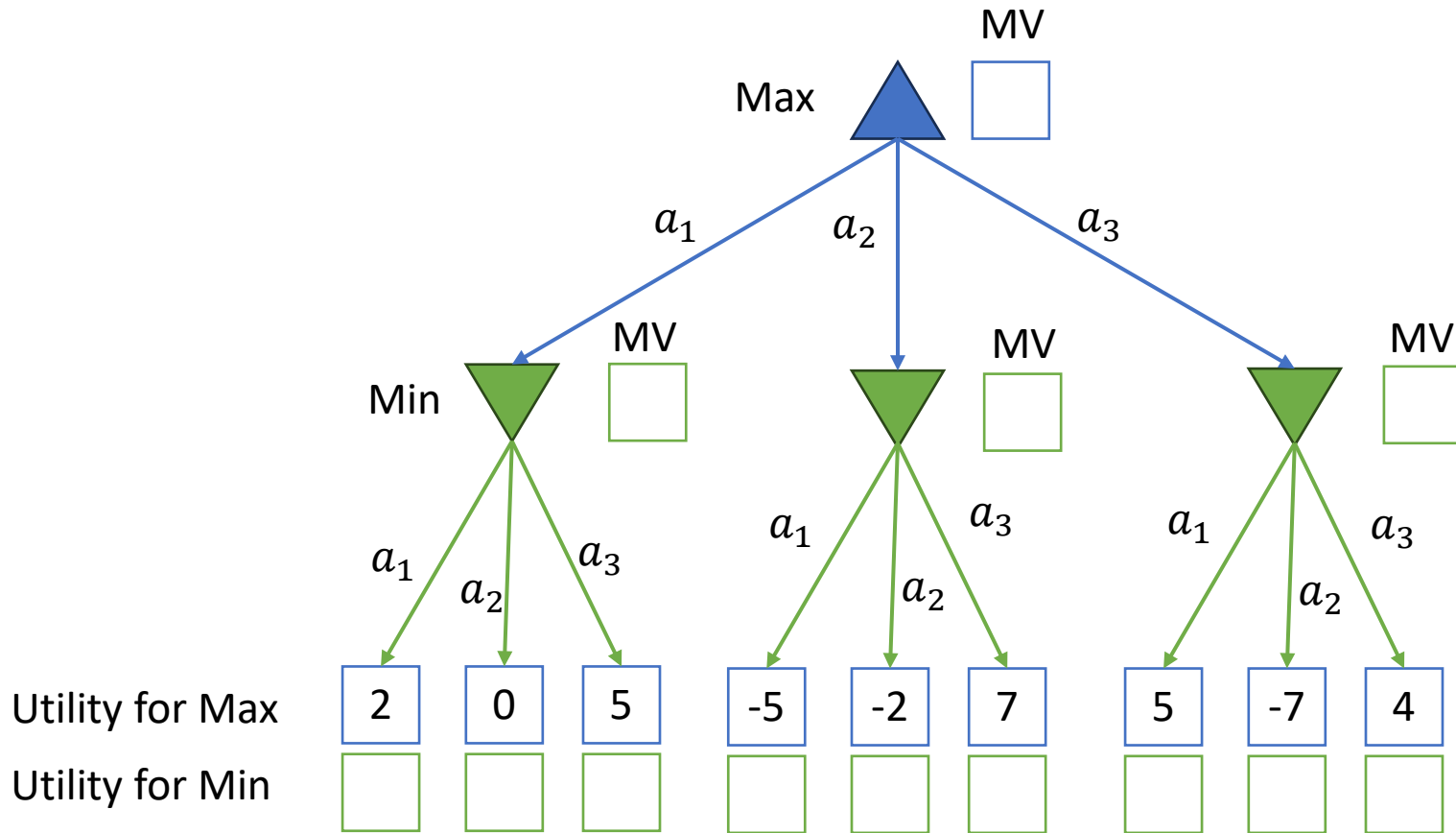
Find the action that leads to the best value.

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

Exercise: Simple 2-Ply Game



- What are the terminal state utilities for Min?
- Compute all MV (minimax values).
- How do we traverse the game tree? What is the Big-O notation for time and space?
- What is the optimal action for Max?

b: max branching factor
m: max depth of tree

Issue: Game Tree Size

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

Space complexity: $O(bm)$

Time complexity: $O(b^m)$

- Fast solution is 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:

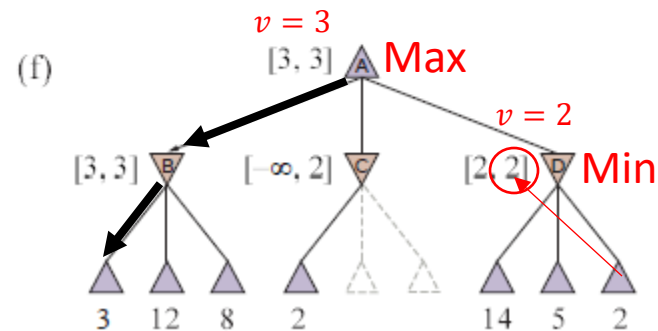
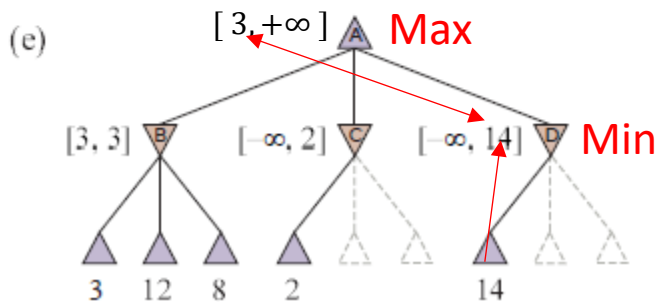
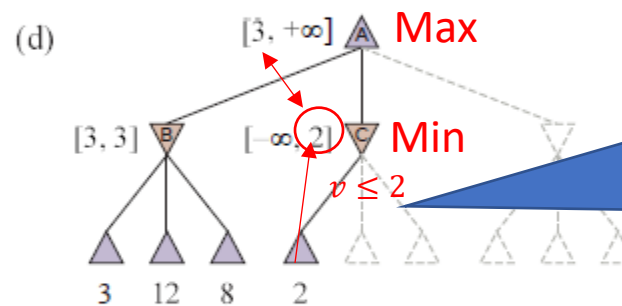
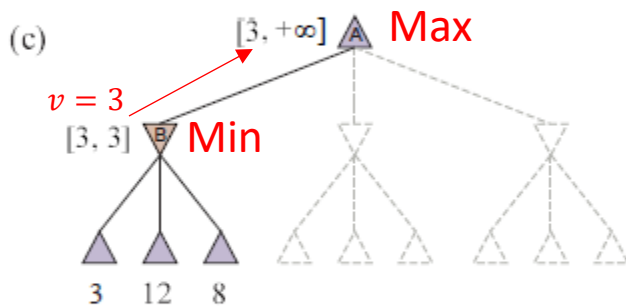
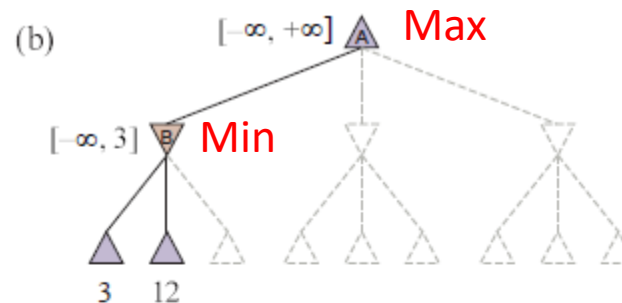
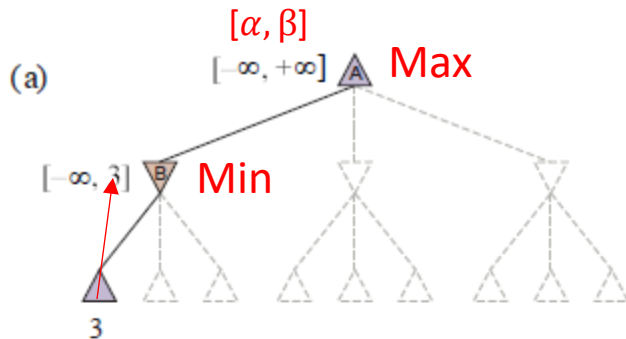
$$1(9)(9 \times 8)(9 \times 8 \times 7) \dots (9!) = 986,409 \text{ nodes}$$

- 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   
      v, move  $\leftarrow$  v2, a  
       $\alpha \leftarrow$  MAX( $\alpha$ , v)  
      if v  $\geq$   $\beta$  then return v, move   
  return v, move
```

Found a better action?

Abandon subtree if Max finds an actions that has more value than the best-known move Min has in another subtree.

```
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   
      v, move  $\leftarrow$  v2, a  
       $\beta \leftarrow$  MIN( $\beta$ , v)  
      if v  $\leq$   $\alpha$  then return v, move   
  return v, move
```

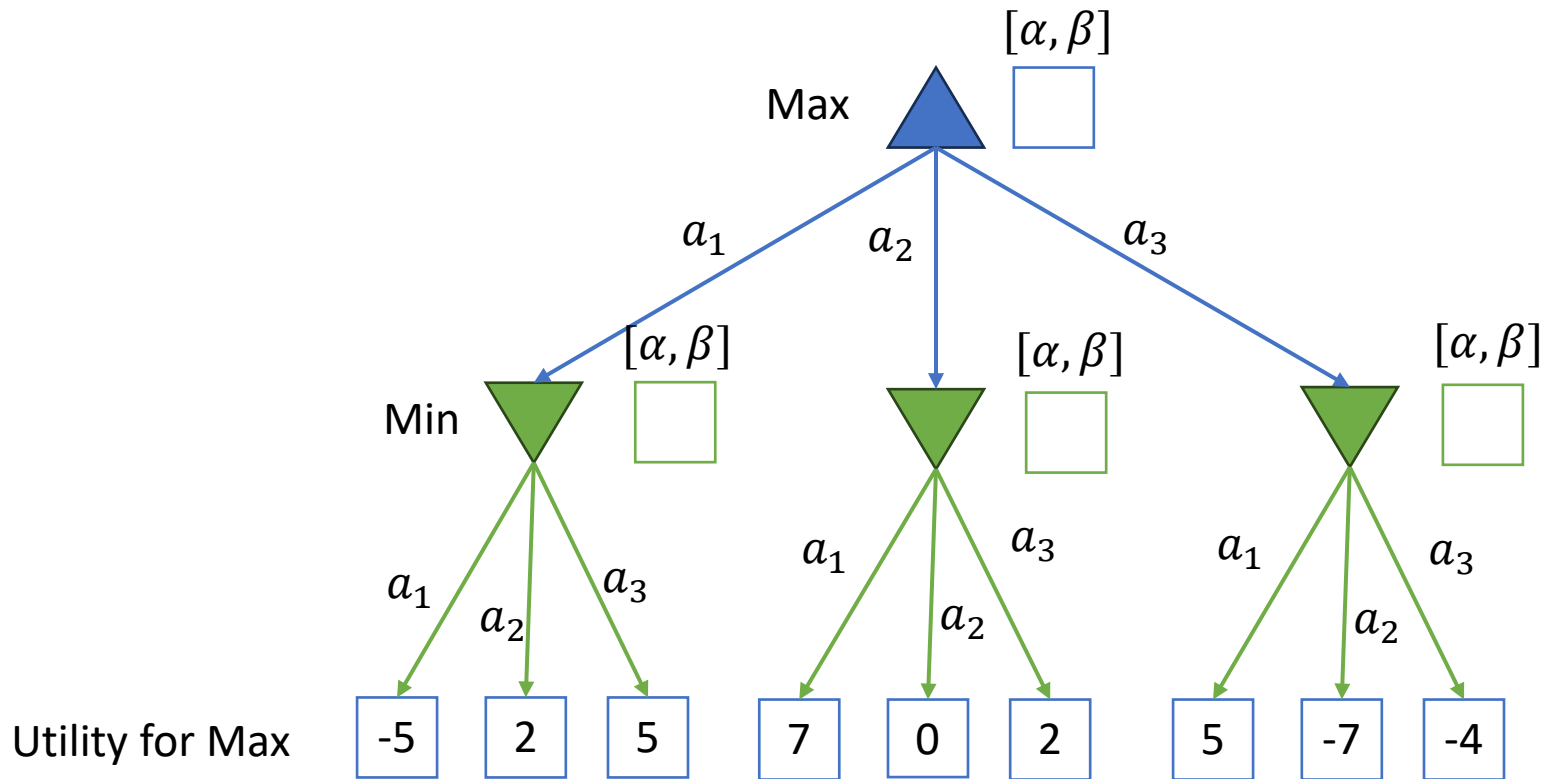
Found a better action?

Abandon subtree if Min finds an actions that has less value than the best-known move Max has in another subtree.

Move Ordering for Alpha-Beta Search

- **Idea:** Pruning is more effective if good alpha-beta bounds can be found in the first few checked subtrees.
- **Move ordering for DFS** = Check good moves for Min and Max first.
- We need expert knowledge or some heuristic to determine what a good move is.
- **Issue:** Optimal decision algorithms still scale poorly even when using alpha-beta pruning with move ordering.

Exercise: Simple 2-Ply Game



- Find the $[\alpha, \beta]$ intervals for all nodes.
- What is the optimal move sequence?
- What part of the tree can be pruned?
- What would be the optimal move ordering?

A collection of colorful wooden blocks, some shaped like the letter 'T', scattered on a wooden surface. The blocks are in various colors including purple, blue, green, orange, red, pink, yellow, and brown. 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.

Needed 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).

Examples:

1. 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).

2. A deep neural network trained on complete games.

Heuristic Alpha-Beta Tree Search: Cutting off search

HMV = heuristic minimax value

Depth (ply)

0 MAX (x)

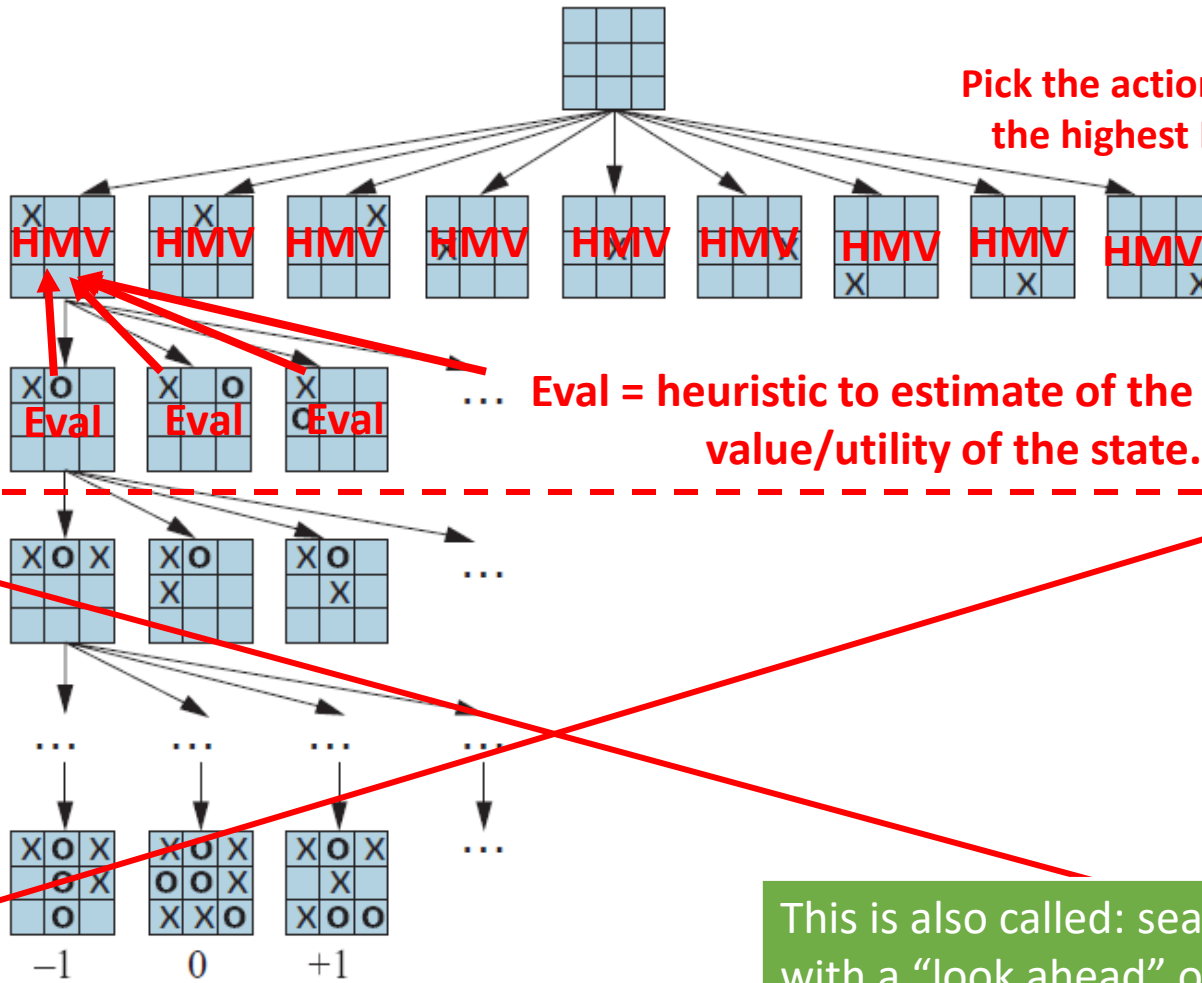
1 MIN (o)

2 MAX (x)

3 MIN (o)

TERMINAL

Utility



Forward pruning

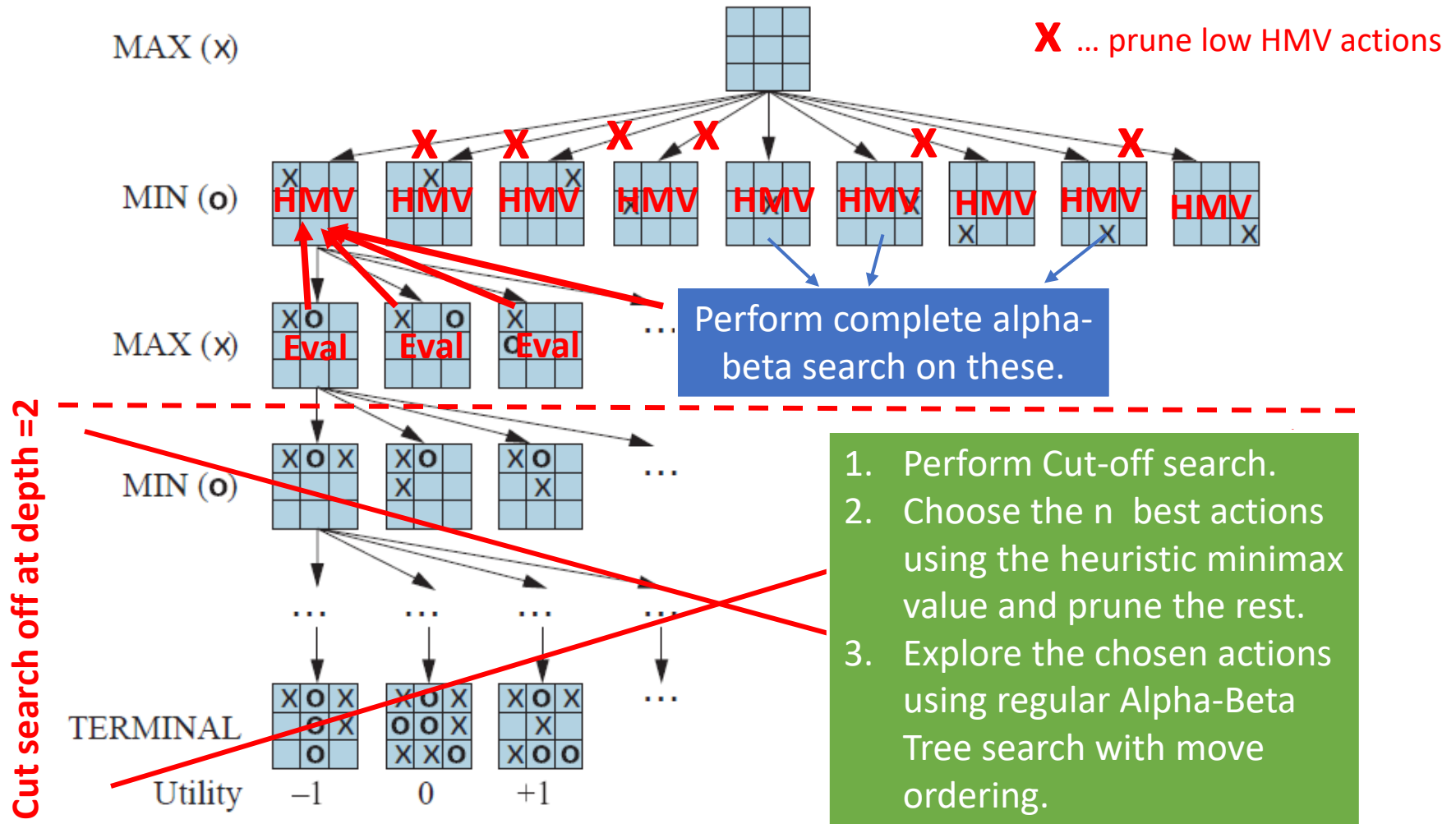
To save time, we can prune moves that appear bad.

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





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

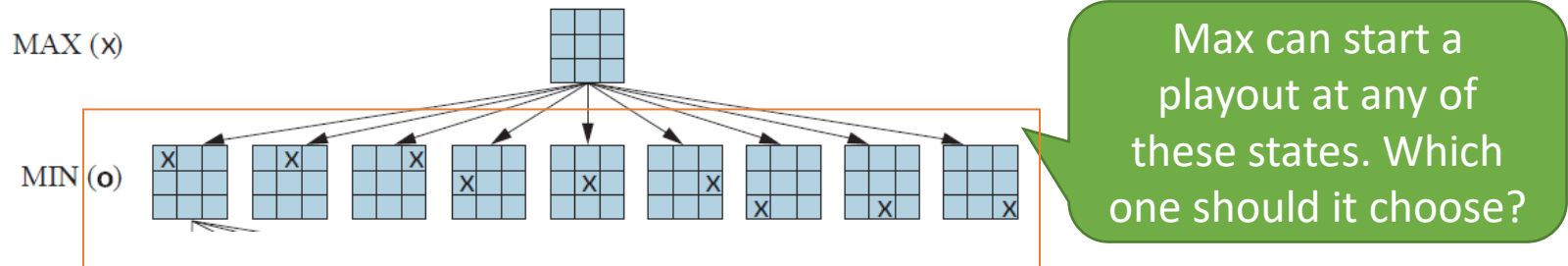
- **Approximate *Eval*(*s*)** as the average utility of several simulation runs to the terminal state (called playouts).
- **Playout policy:** How to choose moves during the simulation runs?
Example playout policies:
 - Random.
 - Heuristics for good moves developed by experts.
 - Learn good moves from 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 make the tree very wide).
 - Unknown or hard to define good evaluation functions.

Pure Monte Carlo Search

Find the next best move.

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

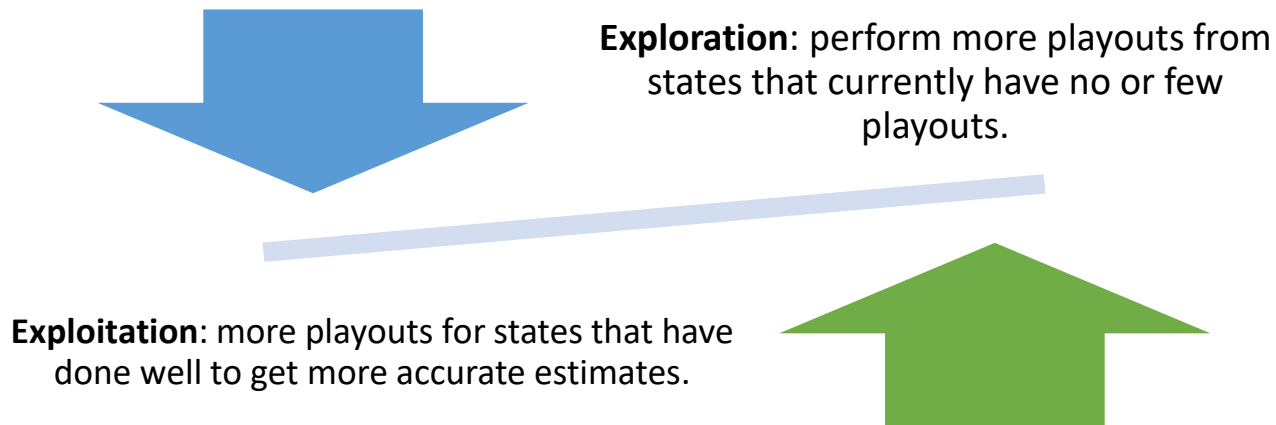
Playout Selection Strategy



Issue: Pure Monte Carlo Search spends a lot of time to create playouts for bad move.

Better: Select the starting state for playouts to focus on important parts of the game tree (i.e., good moves).

This presents the following tradeoff:



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). Goes to 0 for large $N(n)$

n ... node in the game tree

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

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

Selection strategy: Select node with highest UCB1 score.

Monte Carlo Tree Search (MCTS)

Pure Monte Carlo search always start playouts from a given state.

Monte Carlo Tree Search builds a **partial game tree** and can start playouts from any state (node) in that tree.

Important considerations:

- We can use UCB1 as the **selection strategy** to decide what part of the tree we should focus on for the next playout. This balances exploration and exploitation.
- We typically can only store a small **part of the game tree**, so we do not store the complete playout runs.

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

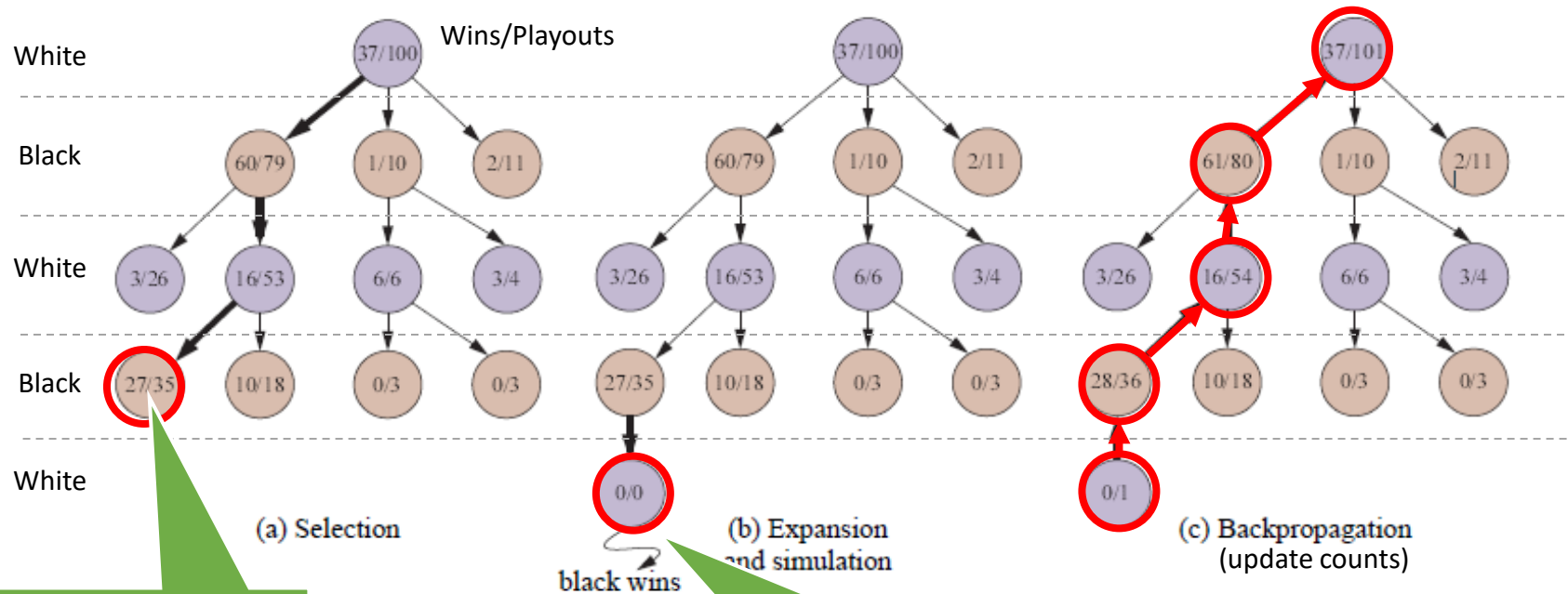
child \leftarrow EXPAND(*leaf*)

result \leftarrow SIMULATE(*child*)

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

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

UCB1 selection favors win percentage more and more.

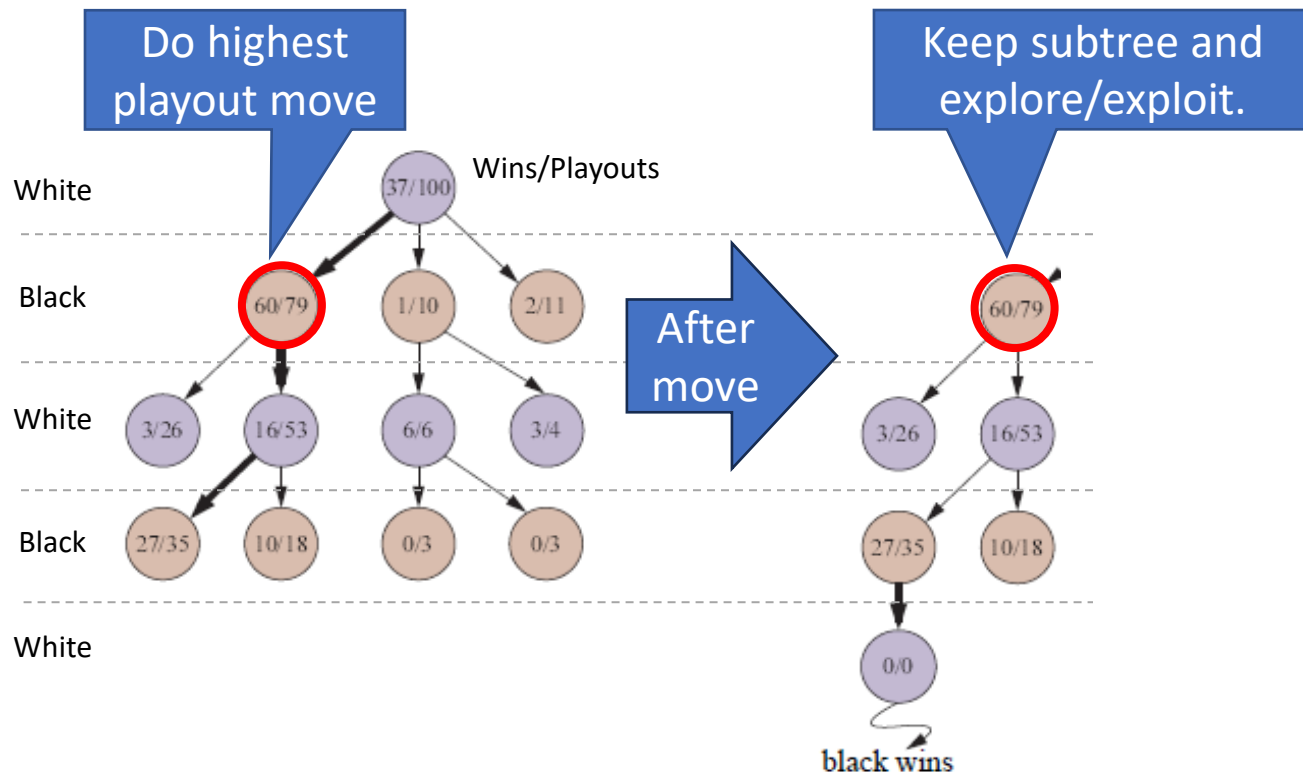


Select leaf with highest UCB1 score

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

Online Play Using MCTS

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



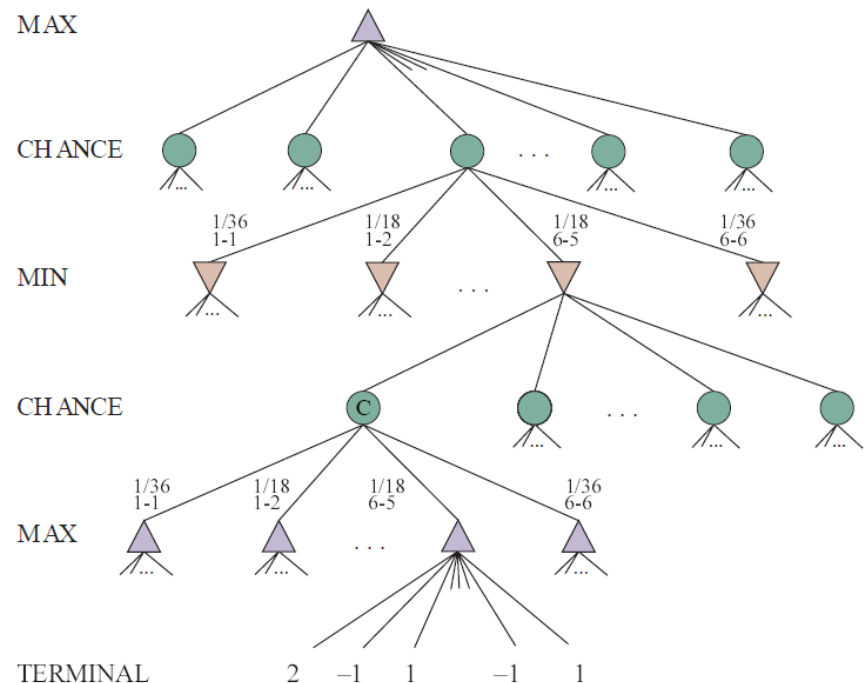
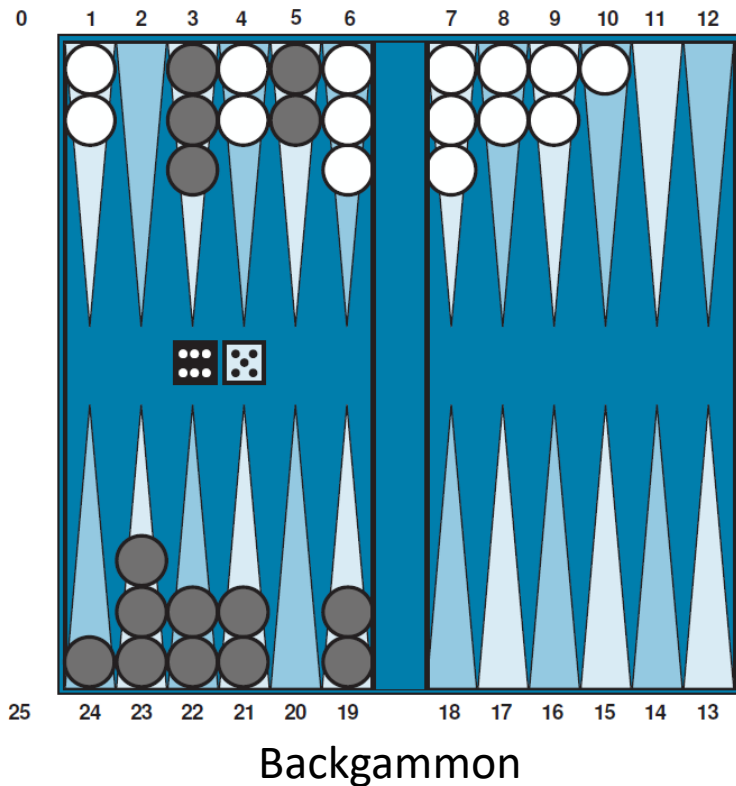
The background of the slide is a close-up photograph of a wooden game board with a light-colored, diagonal striped pattern. Four dice are scattered on the board: two are black with white pips, and two are white with black pips. The dice are positioned around the central text, with one black die in the lower-left, one white die in the upper-left, one white die in the lower-right, and one black die in the upper-right.

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.



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!
 - Cut-off search and 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. *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.
- Learn heuristic from data using MCTS

Monte Carlo Tree search:

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

Scale only for tiny problems!

State of the Art