CS 5/7320
Artificial Intelligence

Adversarial Search and Games

AIMA Chapter 5

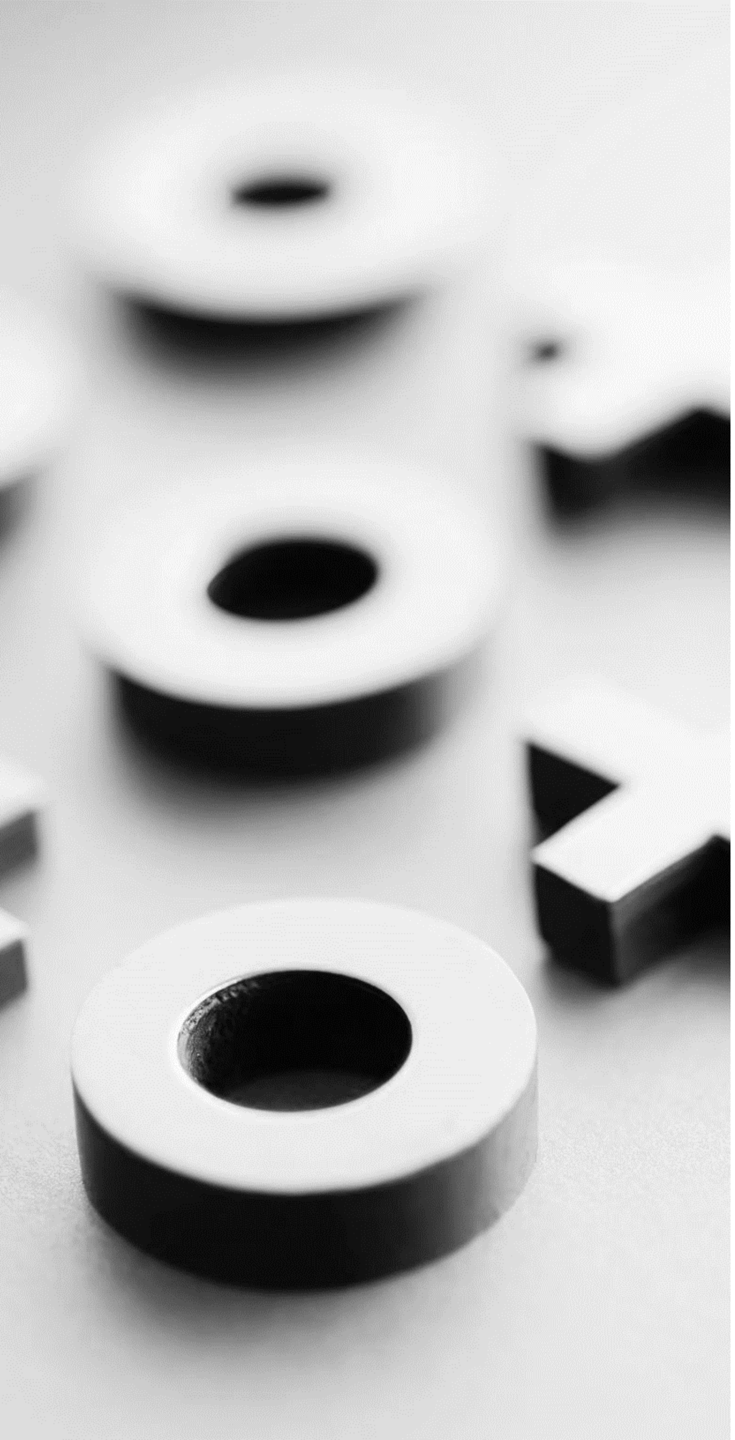
Slides by Michael Hahsler
with figures from the AIMA textbook

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Image: "Reflected Chess pieces" by Adrian Askew

Online Material

Contents



Games

- **Strategic environment**: Games typically feature an environment containing an opponent who wants to win against the agent.
- **Episodic environment**: One game does not affect the next.
- We will focus on planning for
    - two-player zero-sum games with
    - deterministic game mechanics and
    - perfect information (i.e., fully observable environment).

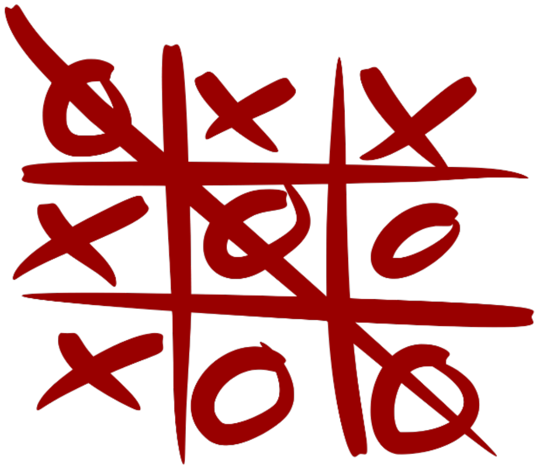
- We call the two players:
    1) **Max** tries to maximize its utility.
    2) **Min** tries to minimize Max's utility (zero-sum game).

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!

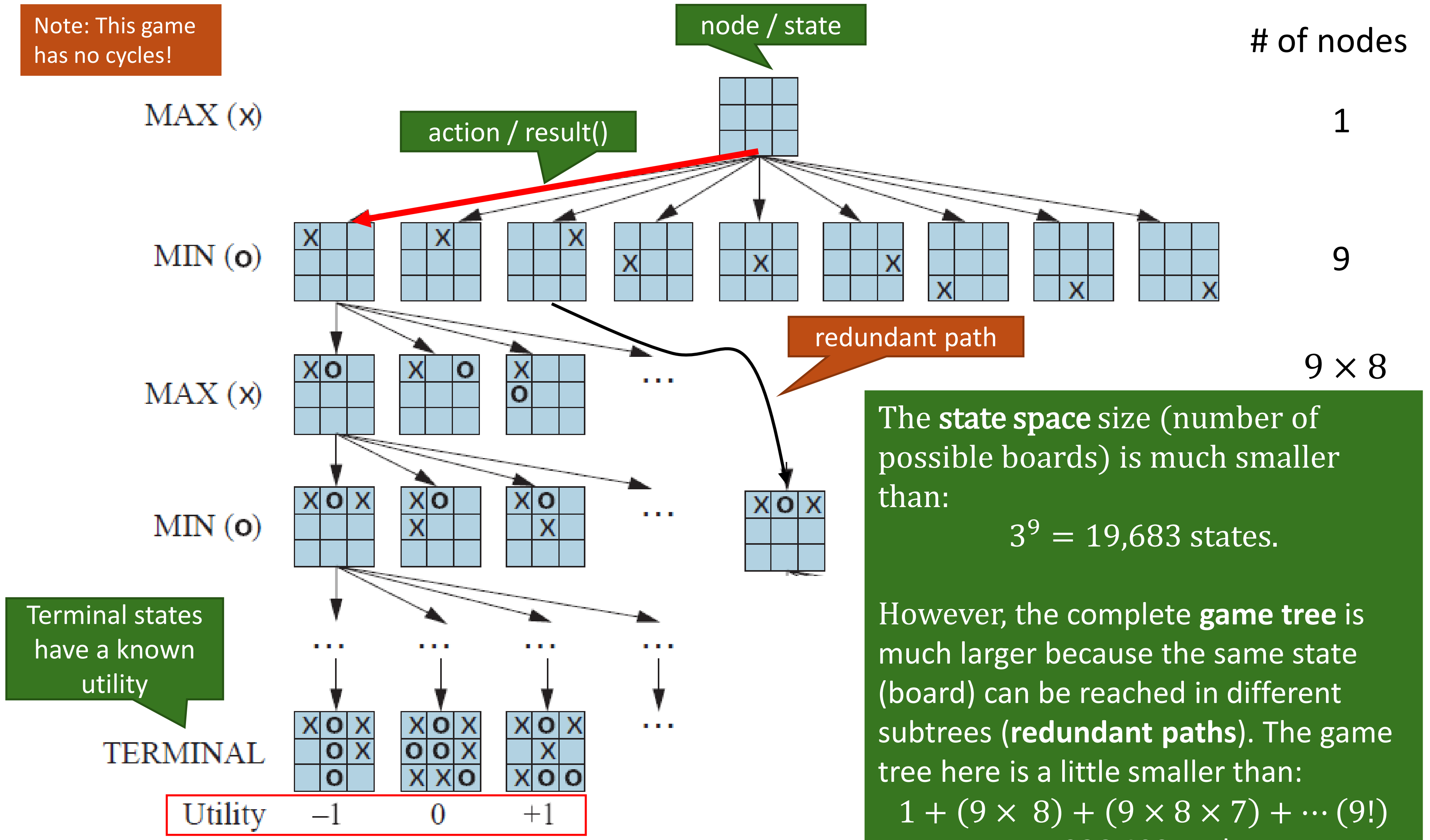
Games as Search Problems

- 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).

- **The search tree is called game tree:** A complete game tree follows every sequence from the current state to the end of the game (the terminal state). 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



The **state space** size (number of possible boards) is much smaller than:

$$3^9 = 19{,}683 \text{ states.}$$

However, the complete **game tree** is much larger because the same state (board) can be reached in different subtrees (**redundant paths**). The game tree here is a little smaller than:

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

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 optimally** 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 a heuristic for utility.
    b. Forward Pruning: ignore poor moves.
- **Monte Carlo Tree search**: Estimate the utility of a state by simulating complete games and averaging the utility.

Exact Method: Nondeterministic Actions

Nondeterministic Actions

- The opponent is considered part of the strategic environment.
- Each "round" consists of
  a) the deterministic move by the agent, and
  b) a **non-deterministic response by the opponent** (the environment).

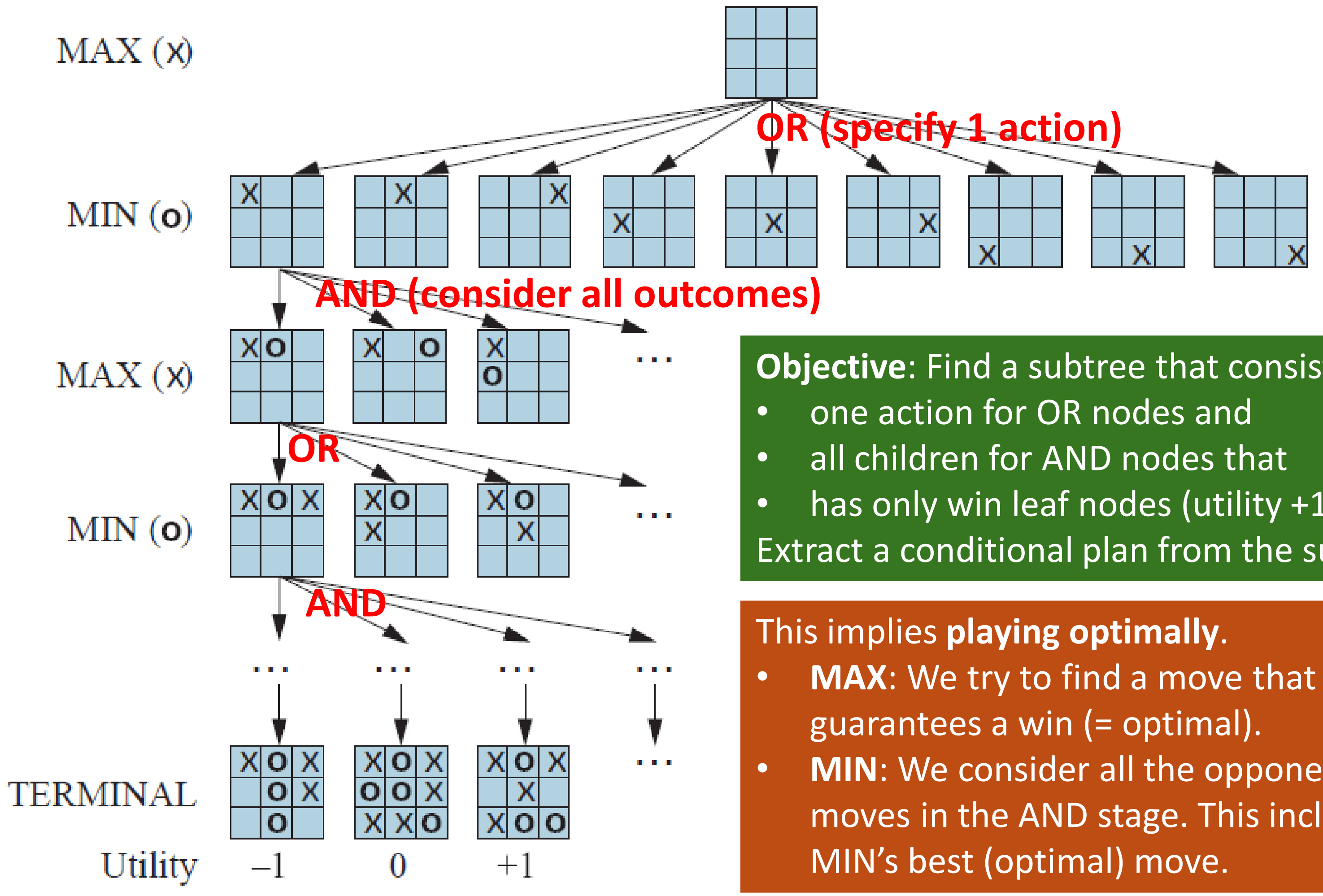
Recall: Nondeterministic actions (AIMA Chapter 4)
We can use a **stochastic transition model that describes uncertainty about the opponent's behavior.**
Example transition:

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

i.e., action a in s_1 can lead to one of several states depending on the opponents move. This set of states is called a *belief state*.

Tic-tac-toe: AND-OR Search

We play MAX and decide on our actions (OR).
MIN's actions introduce non-determinism (AND).



Objective: Find a subtree that consists of

- one action for OR nodes and
- all children for AND nodes that
- has only win leaf nodes (utility +1).

Extract a conditional plan from the subtree.

This implies **playing optimally**.

- **MAX**: We try to find a move that guarantees a win (= optimal).
- **MIN**: We consider all the opponent's moves in the AND stage. This includes MIN's best (optimal) move.

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 ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
      if plan ≠ failure then return [action] + plan]
   return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
   for each $s_i$ in states do                              // check all possible resulting states
      $plan_i$ ← 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 $plan_n$]
```

Try agent's moves

all states that can result from opponent's moves

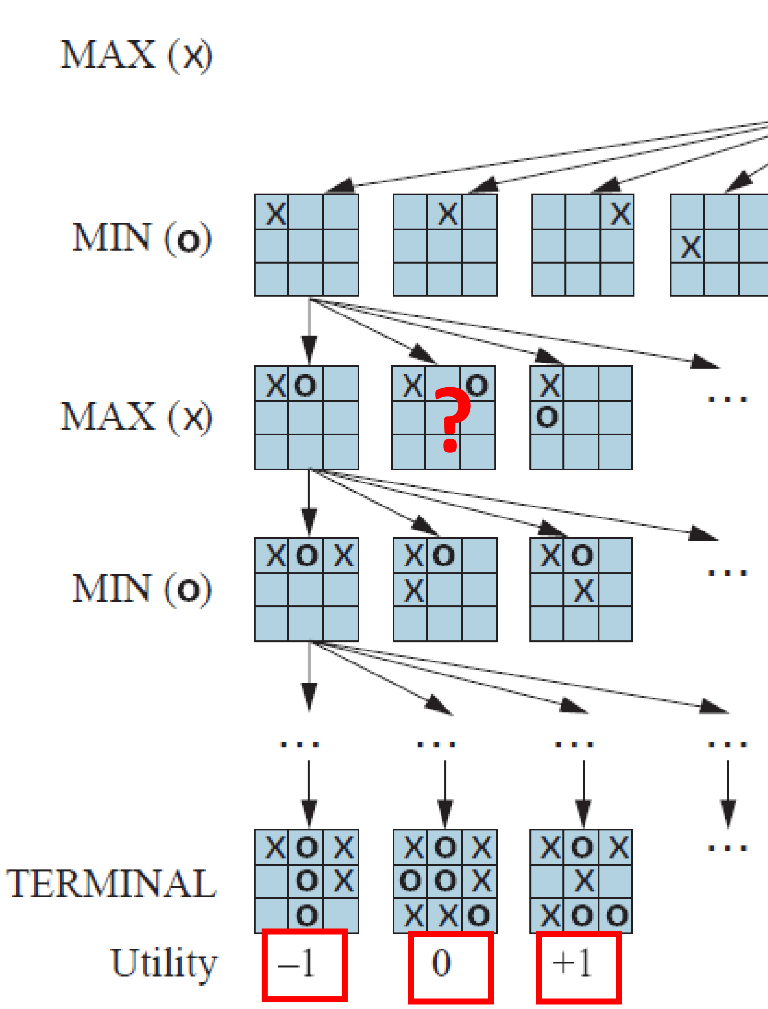
Go through opponent moves

abandon subtree if a loss is possible

Construct a partial conditional plan for the subtree

Optimal Decisions: Minimax Search

Immediate vs. Long-Term Rewards



The **immediate reward** of a state is the utility that the agent receives for being in/getting to that state.

Terminal states: The immediate reward is the known utility.

Non-terminal states: The immediate reward is 0.
Issue: How good is it to be in a non-terminal state? We need to complete the game and see. This is called the (expected) **long-term reward** of the state.

The **optimal decision** is to always choose the action that leads to the highest long-term reward state.

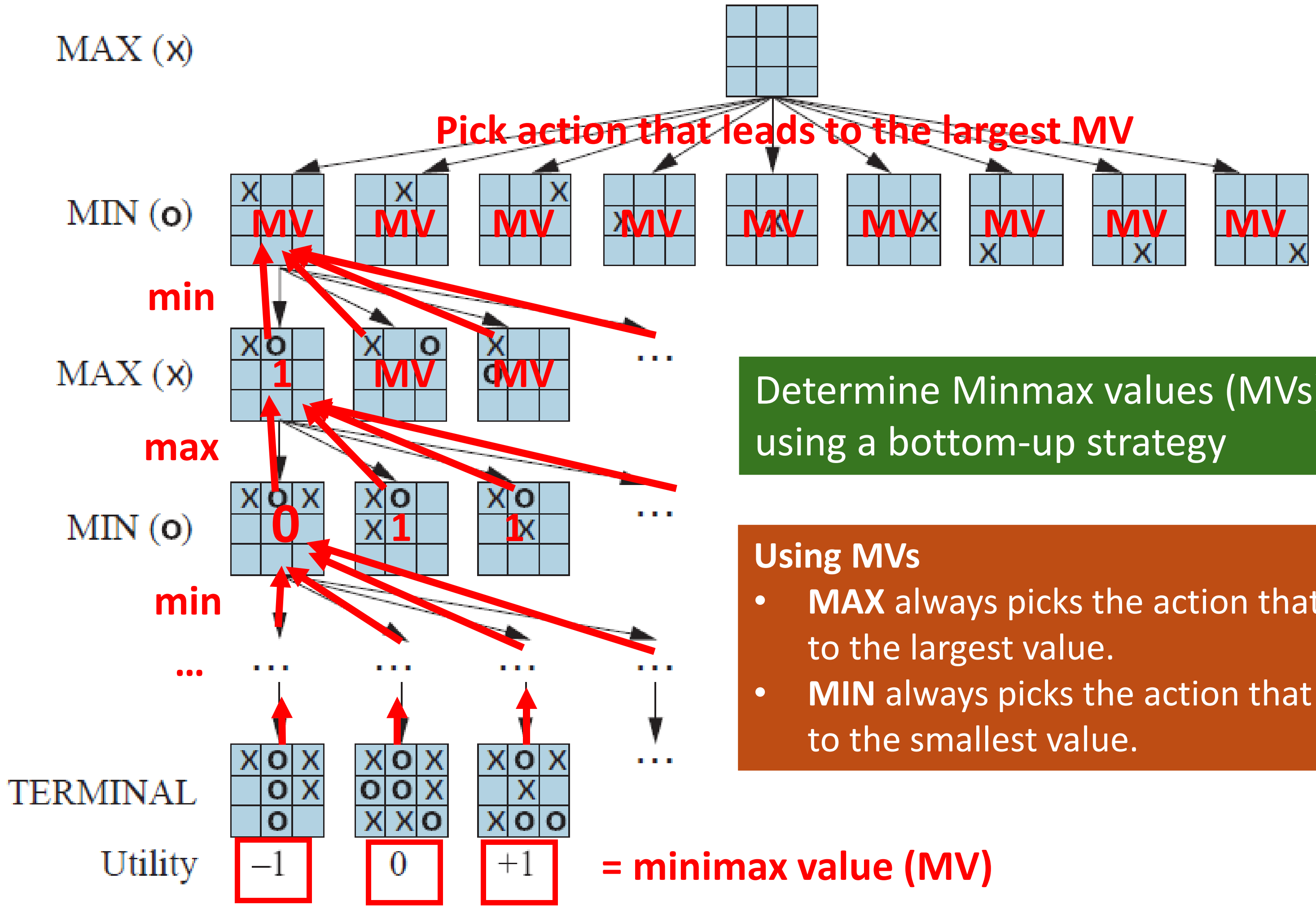
Idea: Minimax Decision

- 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 (i.e., the long-term reward):

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

- This is a recursive definition which can be solved from terminal states backwards.
- **Optimal decision** for Max: Choose the action that leads to the state with the largest minimax value (highest long-term reward).

Minimax Search: Back-up Minimax Values



Determine Minmax values (MVs) using a bottom-up strategy

Using MVs

- **MAX** always picks the action that leads to the largest value.
- **MIN** always picks the action that leads to the smallest value.

Minimax DFS

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

Note: This is just a modification of the AND-OR Tree Search and returns the first action of the conditional plan.

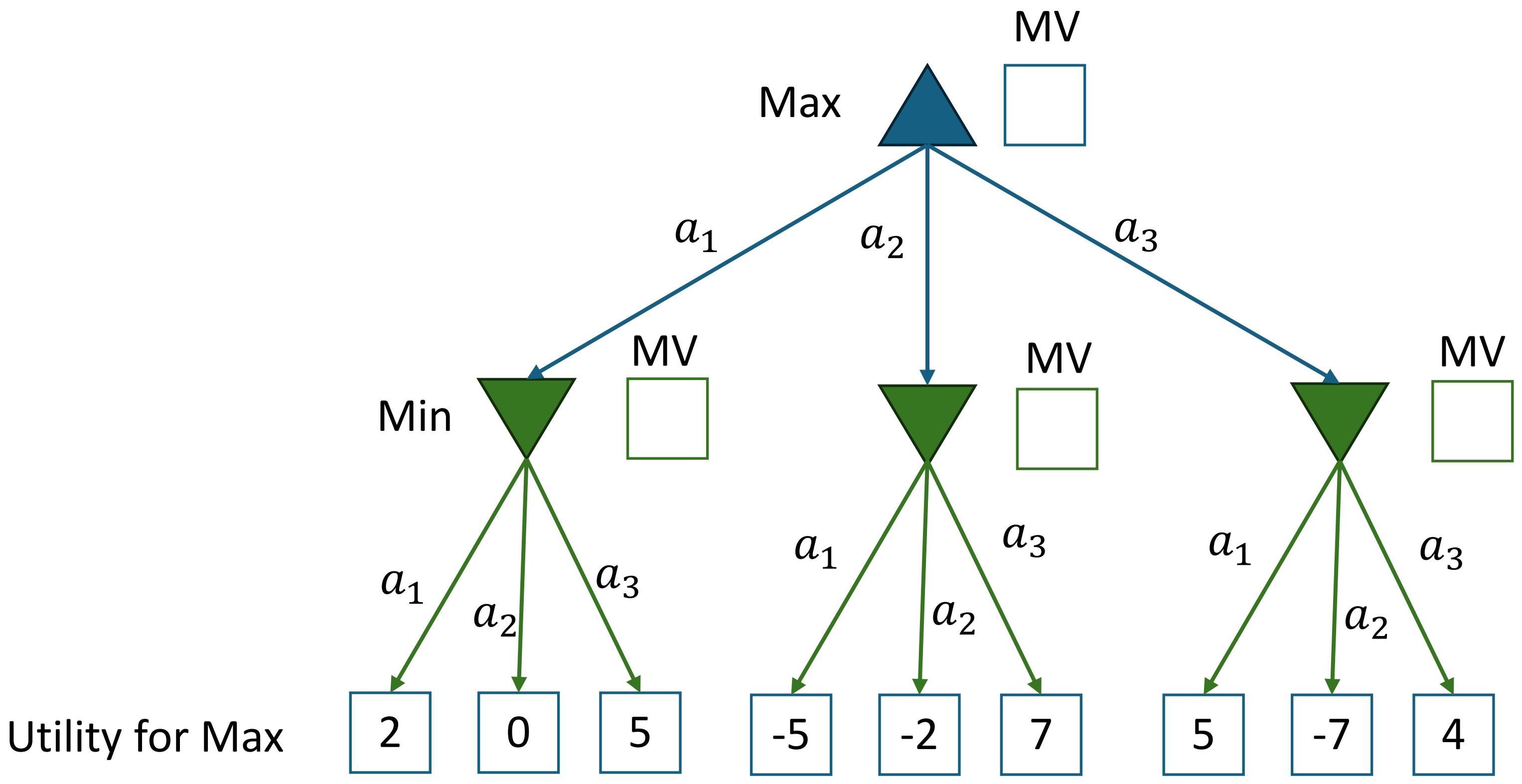
```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

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

function MIN-VALUE(game, state) 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))
        if v2 < v then
            v, move ← v2, a
    return v, move
```

Find the action that leads to the best value.

Represents OR Search

Represents AND Search

Exercise: Simple 2-Ply Game



- Compute all MV (minimax values).
- What is the optimal action for Max?

b: max. branching factor
m: max. depth of tree

Issue: Search Time

- Complexity

  Space complexity: $O(bm)$ - Function call stack + best value/action

  Time complexity: $\boldsymbol{O(b^m)}$ - **Minimax search is worse than regular DFS for finding a goal! It traverses the entire game tree using DFS!**

- A fast solution is only feasible for very simple games with few possible moves (=small branching factor) and few moves till the game is over (=low maximal depth)!

- **Example**: Time complexity of Minimax Search for 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:

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

- We need to reduce the time complexity! $\rightarrow$ **Game tree pruning**

Improvements for Minimax Search

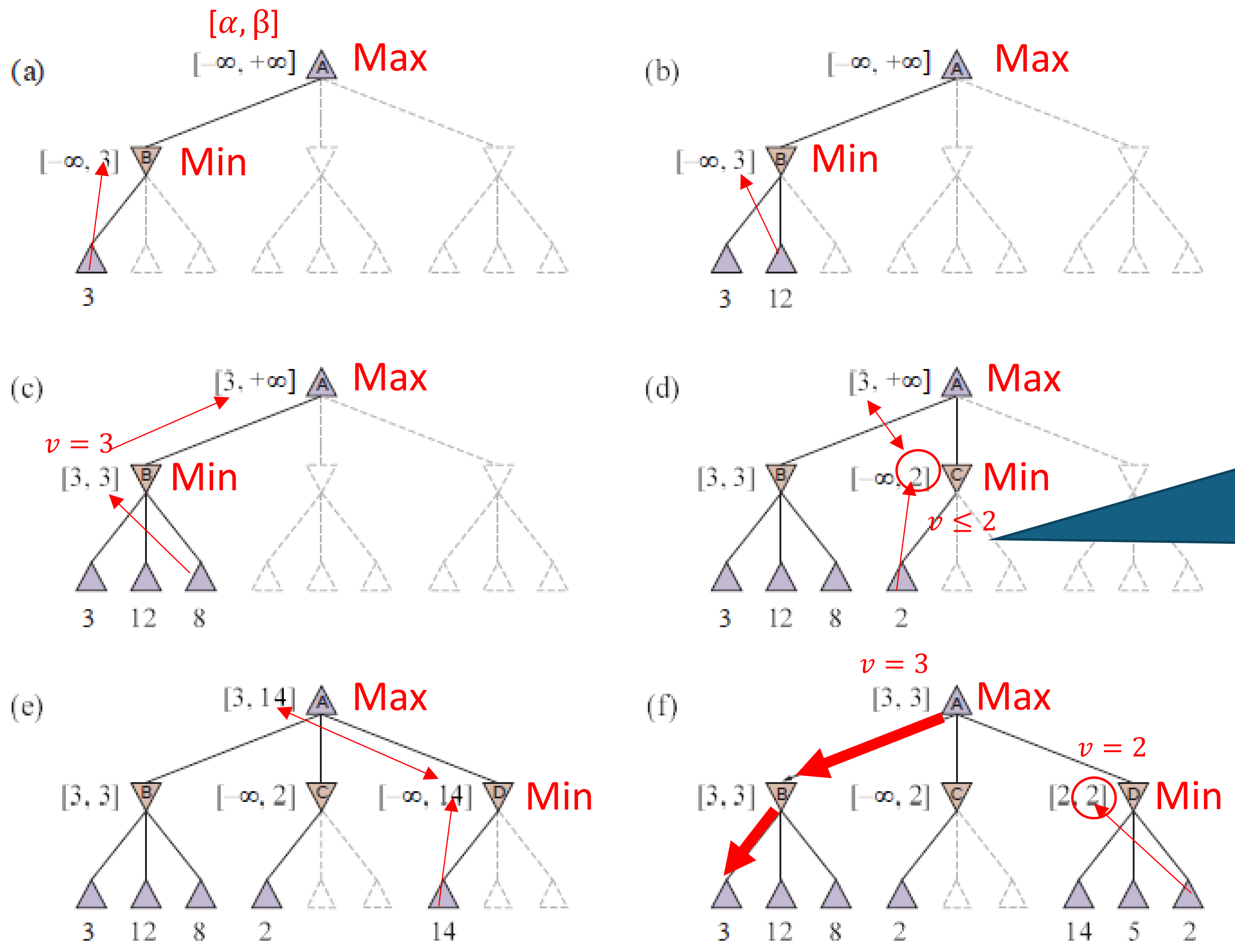
Alpha-Beta Pruning Search and Move Ordering

Alpha-Beta Pruning Search

- **Issue:** Minimax search traverses the entire game tree.
- **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.
  - $\max(5, \min(3, x, y, \dots))$ is always 5 and does not depend on the values of x or y.
  - Minimax search applies alternating min and max.
- **Approach**: maintain bounds for the minimax value $[\alpha, \beta]$. Prune subtrees (i.e., don't follow actions) that do not affect the current minimax value bound.

  - Alpha is used by Max and means "$Minimax(s)$ will be at least α."
  - Beta is used by Min and means "$Minimax(s)$ will be at most β."

Example: Alpha-Beta Pruning



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 ($\alpha = \beta$).

= minimax search + pruning

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

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 ← MIN-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
    if $v2 > v$ then
      v, move ← v2, a
      $\alpha \leftarrow$ MAX($\alpha$, v)
    if $v \geq \beta$ then return v, move
  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 ← MAX-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
    if $v2 < v$ then
      v, move ← 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 would not go there because it has a better choice (represented by β)

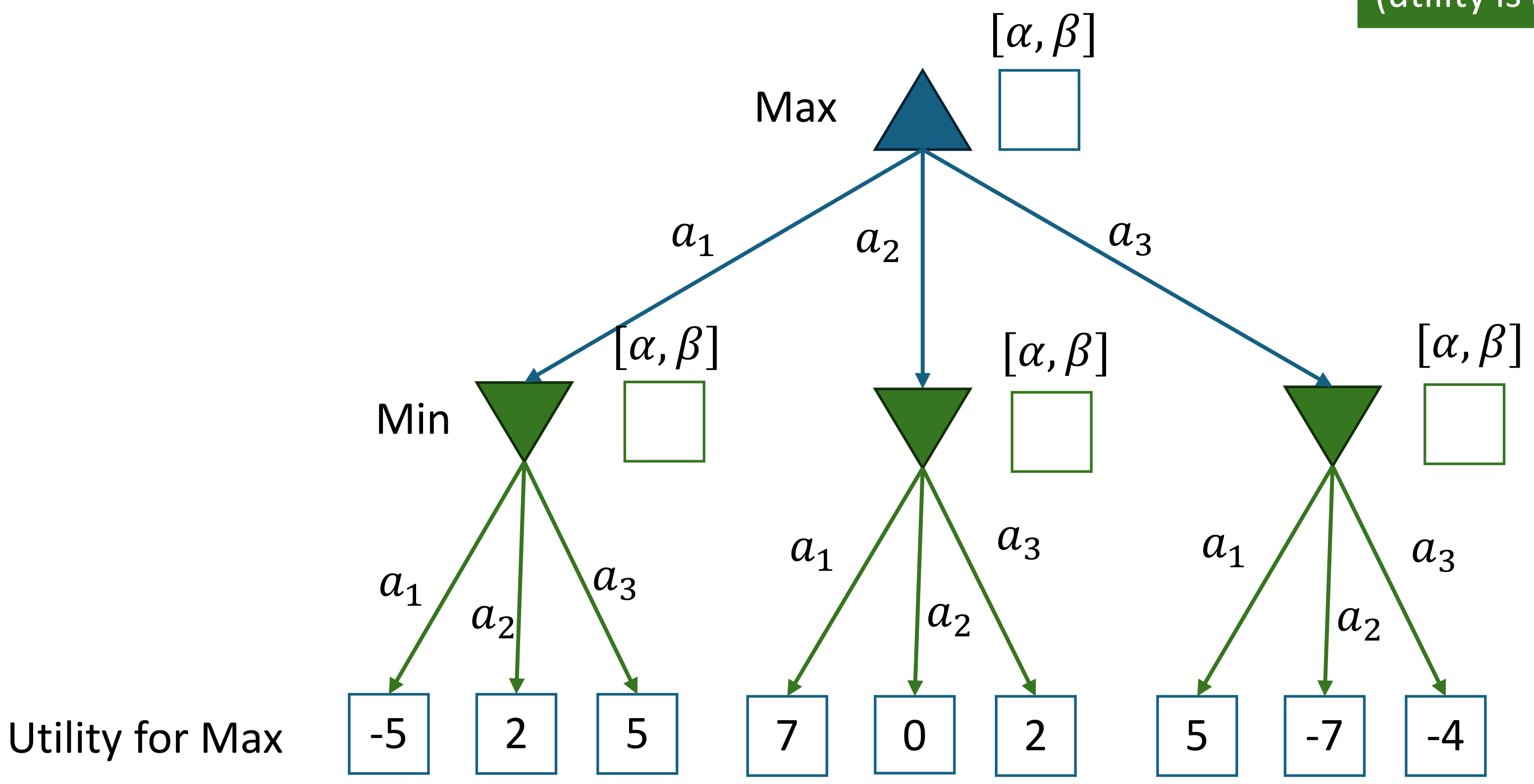
Found a better action?

Abandon subtree if Max would not go there because it has a better choice (represented by α)

Exercise: Simple 2-Ply Game with Alpha-Beta Pruning

Max updates α (utility is at least)

Min updates β (utility is at most)



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

Move Ordering for Alpha-Beta Pruning

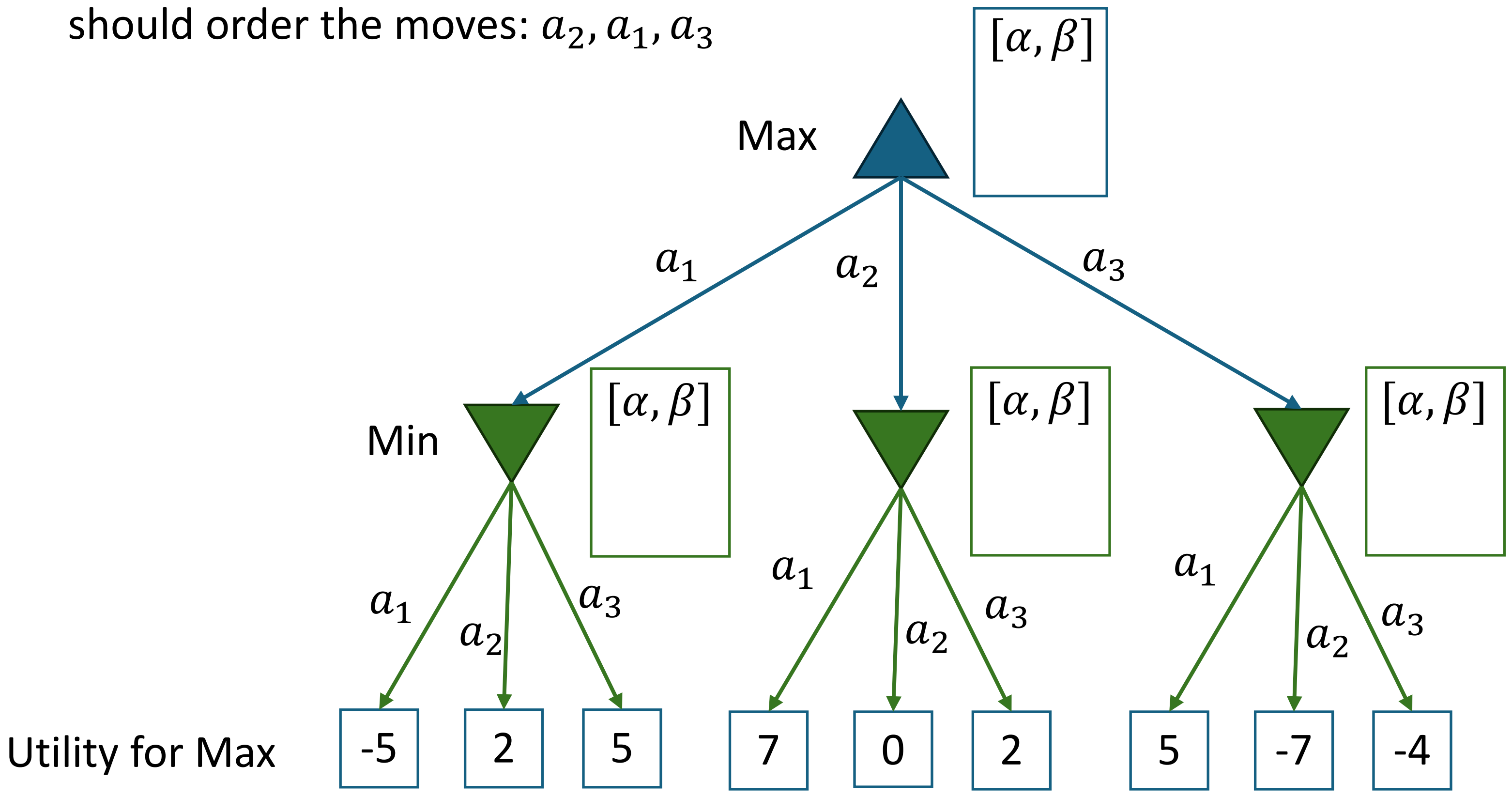
- **Idea:** Pruning is more effective if good alpha-beta bounds can be found in the first few checked subtrees.
- **Move ordering for DFS** = Check good moves for Min and Max first.
- This is very similar to Greedy Best-first Search. We need expert knowledge (a **heuristic**) to determine what a good move is.

Exercise: Simple 2-Ply Game with Alpha-Beta Pruning and Move Ordering

Max updates α (utility is at least)

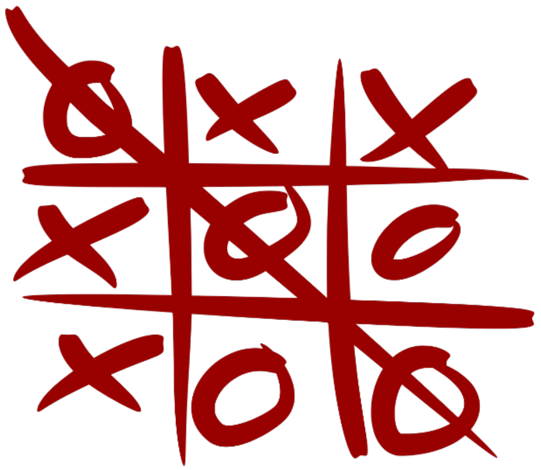
Min updates β (utility is at most)

- Assume a heuristic shows that we should order the moves: a_2, a_1, a_3



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

The Effect of Alpha-Beta Pruning



Tic-tac-toe

Method	Searched Nodes	Search Time
Minimax Search	549,946	13 s
+ Alpha-Beta Pruning	18,297	660 ms
+ Move ordering (heuristic: center, corner, rest)	7,275	202 ms

Issue With Minimax Search

- Optimal decision-making algorithms **scale poorly** for large game trees.

- Alpha-beta pruning and move ordering are often not sufficient to reduce the search time.

- **Fast approximate methods are needed**.
  We may lose the optimality guarantee, but we can work with larger problems.

Heuristic Methods

Heuristic Alpha-Beta Tree Search

Heuristic Alpha-Beta Tree Search

Issue: The game tree is too large to use optimal Alpha-Beta Search.

Approach: Search only part of the tree by replacing missing information using a heuristic evaluation function.

Options:

a. Cut off game tree and use a heuristic for utility.
b. Forward Pruning: ignore poor moves.

Option A: Heuristic Cut 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 based on features of the state.

Needed properties of the evaluation function:

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

Examples:

1. A weighted linear function

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

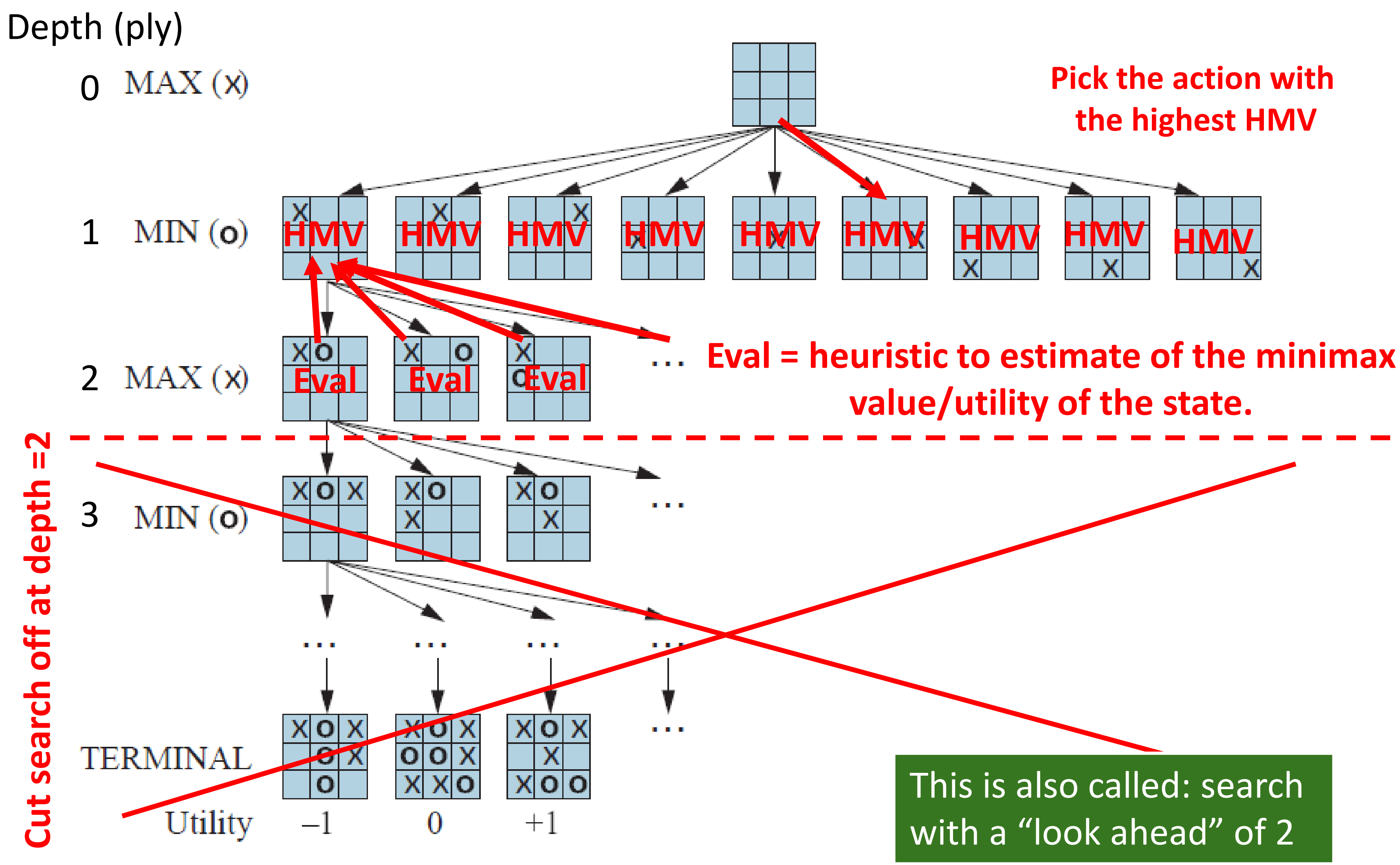
   where f_i is a feature of the state (e.g., # of pieces captured in chess).

2. A deep neural network (or other ML method) trained on complete games.

Heuristic Alpha-Beta Tree Search: Cut Off Search

HMV = heuristic minimax value

Pick the action with the highest HMV



Eval = heuristic to estimate of the minimax value/utility of the state.

This is also called: search with a "look ahead" of 2

Option B: Heuristic Forward Pruning

Idea: Focus search on good moves (= prune the others).

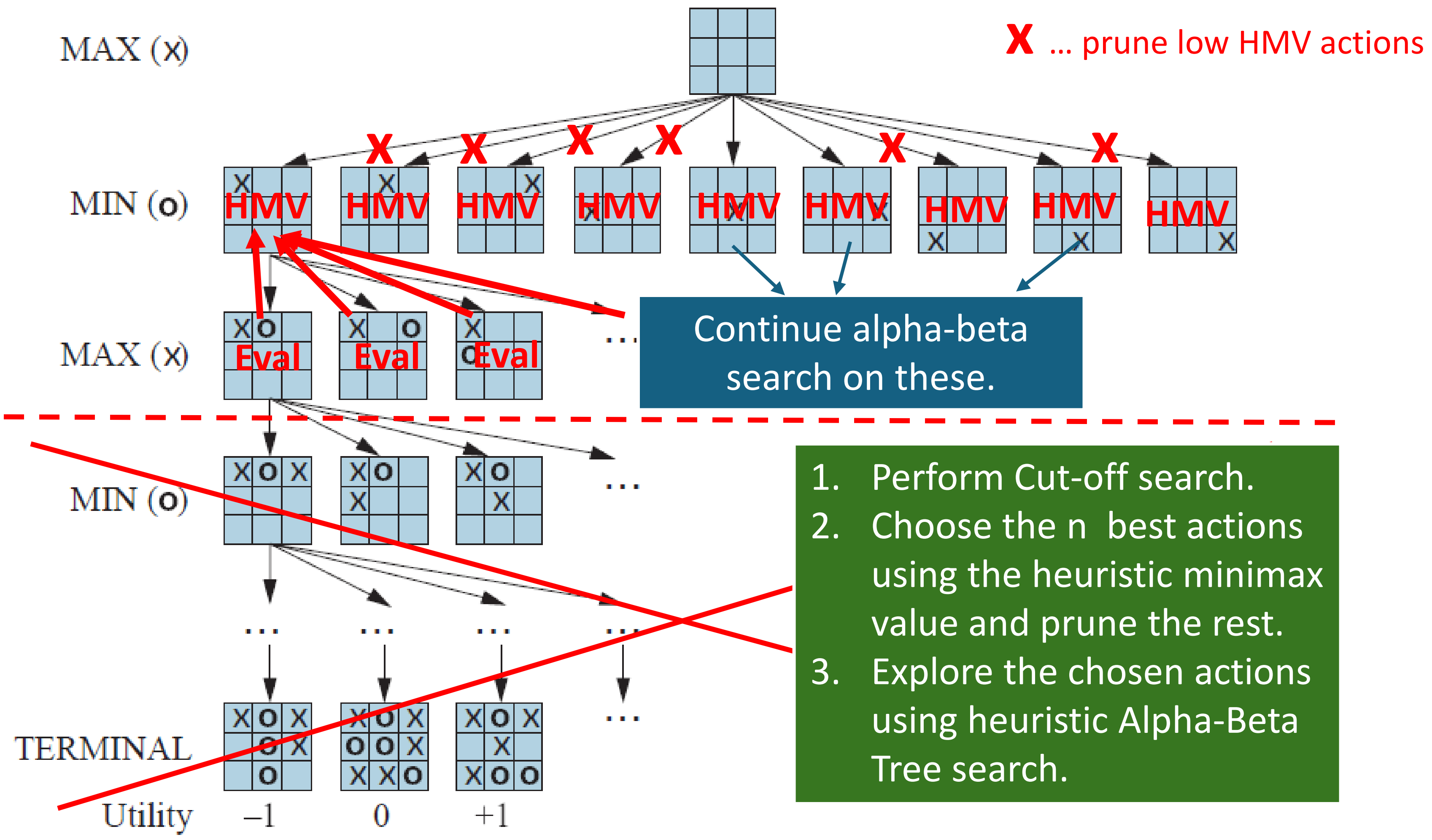
There are many ways in which quality can be evaluated:

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

Beam search: Focus on the n best moves at every layer in the game tree.

Issue: May prune important moves.

Heuristic Alpha-Beta Tree Search: Example for Forward Pruning


x ... prune low HMV actions
MAX (x)
MIN (o)
MAX (x)
MIN (o)
TERMINAL
Utility
HMV
Eval
X
–1
0
+1
Continue alpha-beta search on these.

1. Perform Cut-off search.
2. Choose the n best actions using the heuristic minimax value and prune the rest.
3. Explore the chosen actions using heuristic Alpha-Beta Tree search.

Important Considerations

- Designing a good evaluation heuristic can be difficult.
  - We need **expert knowledge**.
  - **Experimentation** may be needed to choose the best heuristic.

- The cutoff depth affects the runtime and the quality of the found move.
  - **Low cutoff**: Fast, but the approximation of the evaluation function will be poor.
  - **Intermediate cutoff**: Slower because a larger tree needs to be searched, but the evaluation function will work better.
  - **Infinity** (= no cutoff): The algorithm reverts to complete minimax search and optimal decisions.

Heuristic Methods

Monte Carlo Tree Search (MCTS)

Idea of Monte Carlo Search

*"Monte Carlo simulation is a computational technique that uses repeated random sampling to obtain **numerical results**, often used to **model uncertain events** or systems where outcomes are **difficult to predict deterministically**."* *[Wikipedia]*

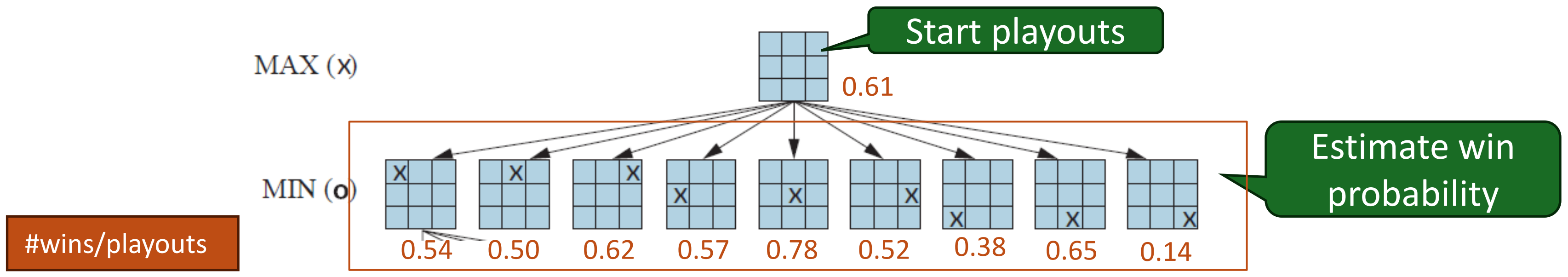
- **Approximate** $\boldsymbol{Eval(s)}$ as the **average utility** of several playouts (= simulated games).

- **Playout policy**: How to choose moves during the simulation runs?
  Example playout policies:
  - Random.
  - Heuristics for good moves developed by experts.
  - Learn a good playout policy from self-play (e.g., with deep neural networks).
    We will discuss this further when we cover "Learning from Examples."

- Typically used for problems with
  - High branching factor (many possible moves make the tree very wide).
  - Unknown or hard to define evaluation functions.

Pure Monte Carlo Search

- **Goal**: Find the best next move.
- **Method**
    1. Simulate N playouts from the **current state** using a random playout policy.
    2. Track which move has the highest win percentage (or largest expected utility) in its subtree.



- **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.

Monte Carlo Tree Search (MCTS)

Pure Monte Carlo search always starts playouts from a given state (or randomly from its children).

Issue: Many playouts are performed for very bad moves.

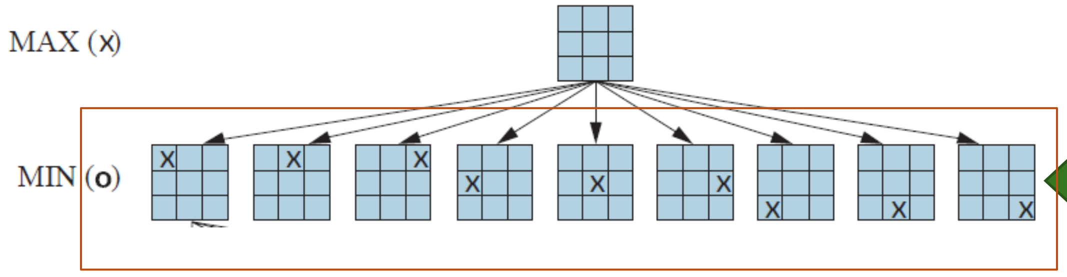
Idea: Focus on **sequences of good moves**.

We will introduce

a) **UCB1** to select a good move to play out next.

b) A **tree** to deal with short sequences of moves.

Playout Selection Strategy

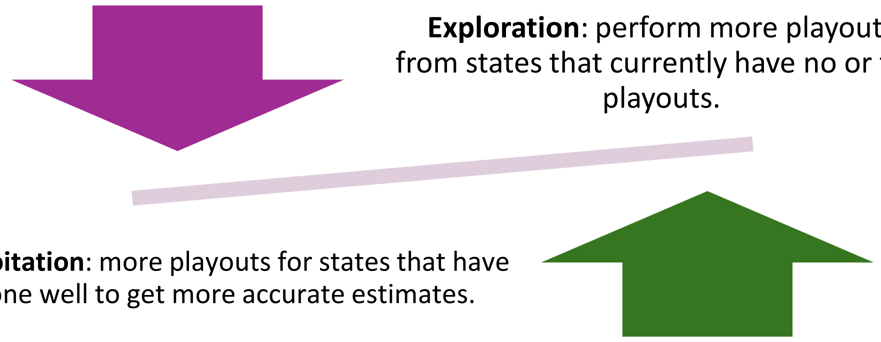


For the empty board, Max can start a playout at any of these states. Which one should it choose?

Issue: Pure Monte Carlo Search with a random playout policy 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:



Exploration: perform more playouts from states that currently have no or few playouts.

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

Upper Confidence Bound 1 (UCB1) 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 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.

Trees in MCTS

Monte Carlo Tree Search builds a **partial game tree** representing short sequences of the next few moves.

Playouts can start from any state (leaf node) in that tree. This means the algorithm can focus on a good sequence of moves.

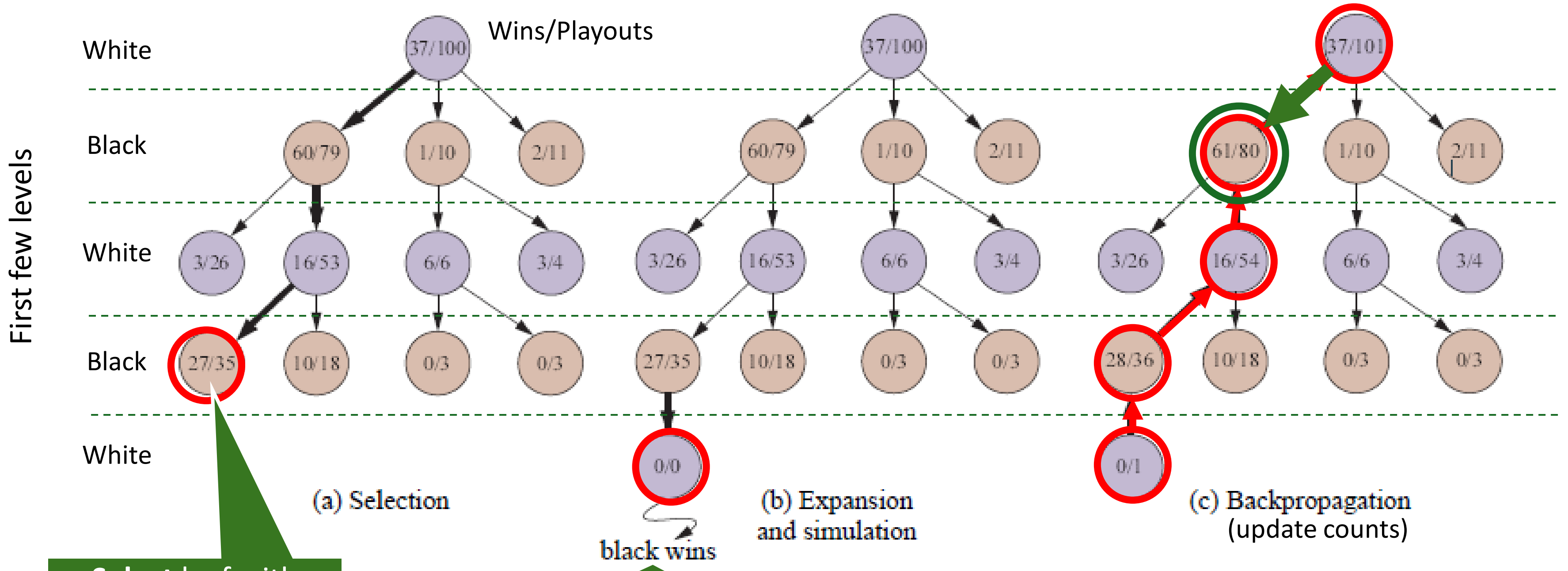
Important considerations:

- We typically can only store a **small part of the game tree**, so we do not store the complete playout runs.
- 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.

```
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 (update counts)

Select leaf with highest UCB1 score

Expand and Simulate: the simulation path is not recorded to preserve memory!

Advantage over pure MCS: Selection strategically focuses search

Some Considerations

- Estimating the value of a position using simple playouts is **very effective** and typically beats many other methods.
- Playouts can be **done in parallel** (multi-core or on multiple machines).
- **MCTS selects playouts strategically** by using UCB1 playout selection and looking several moves ahead.

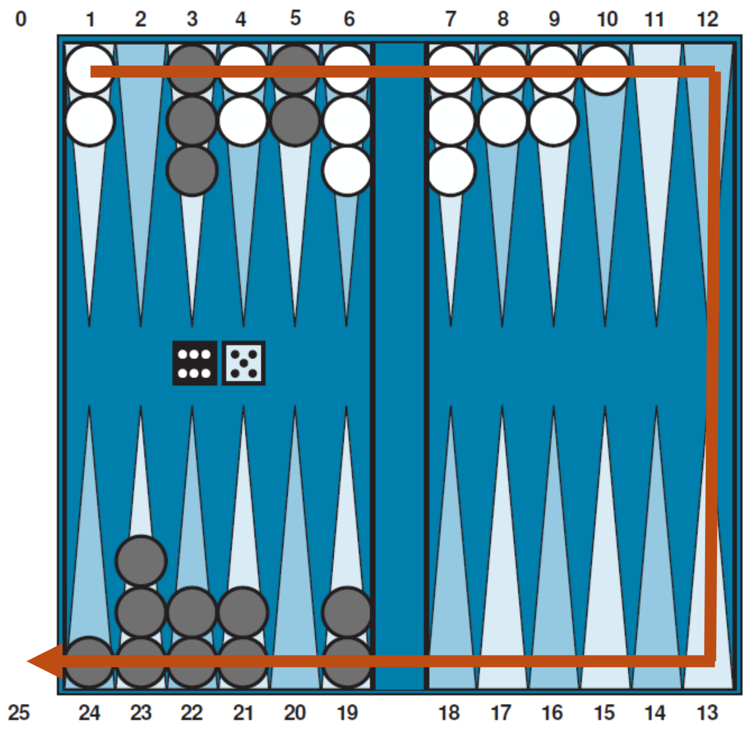
- **Note**: Random playouts may not work well, and a **better playout policy** can help.
  - **Slow Convergence**. Playouts may be wasted on playing very bad (random) moves that nobody would ever play.
  - Random play makes discovering **long-term strategies** very unlikely.

Stochastic Games

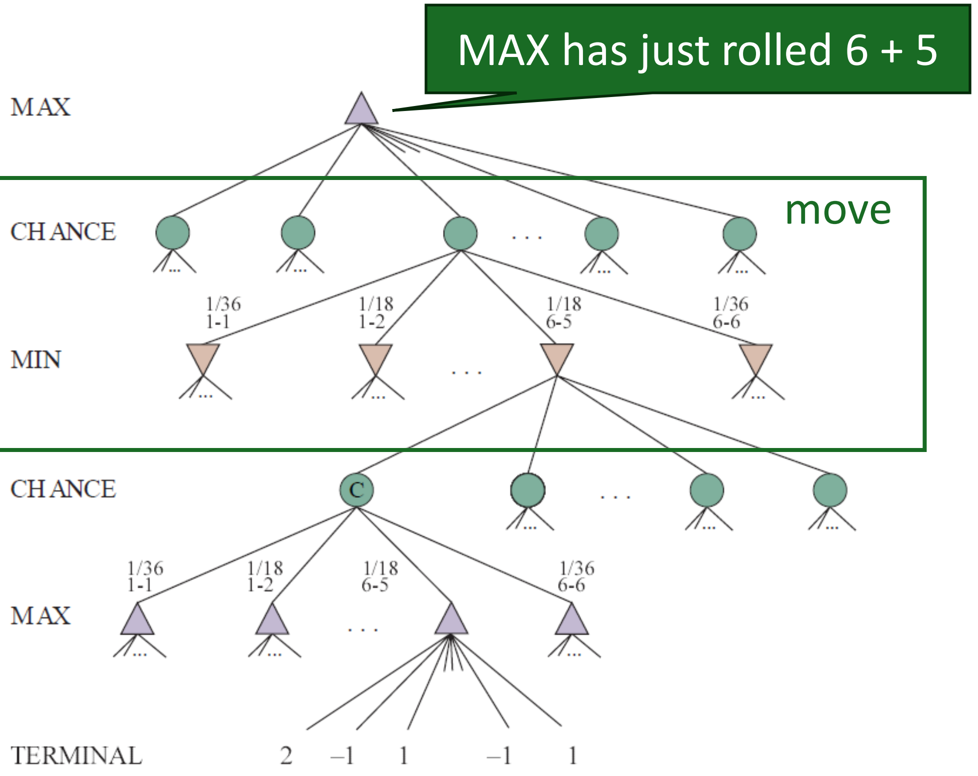
Games With Random Events

Stochastic Games

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



Backgammon: Move checker pieces according to the dice.



Expectiminimax

- Game includes a “random action” r (e.g., dice, dealt cards).
- For **chance nodes** we calculate the expected minimax value.

$$Expectiminimax(s) =$$

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

- Options:
  - Use Minimax algorithm. Issue: Search tree size explodes if the number of “random actions” is large. Think of drawing cards for poker!
  - Heuristic Expectiminimax Search: Cut-off search and with an evaluation function.

MCTS for Stochastic Games

Monte Carlo Tree Search can be directly applied to stochastic games: Random actions can be easily added to playouts.

Issue: Random actions result in a significantly larger game tree.

- Requires a much **larger number of playouts** to converge to good solutions.
- The **tree cannot be very deep** because the random actions make it very wide.

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.

Scale only for tiny problems!

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.

State of the Art