Discussion

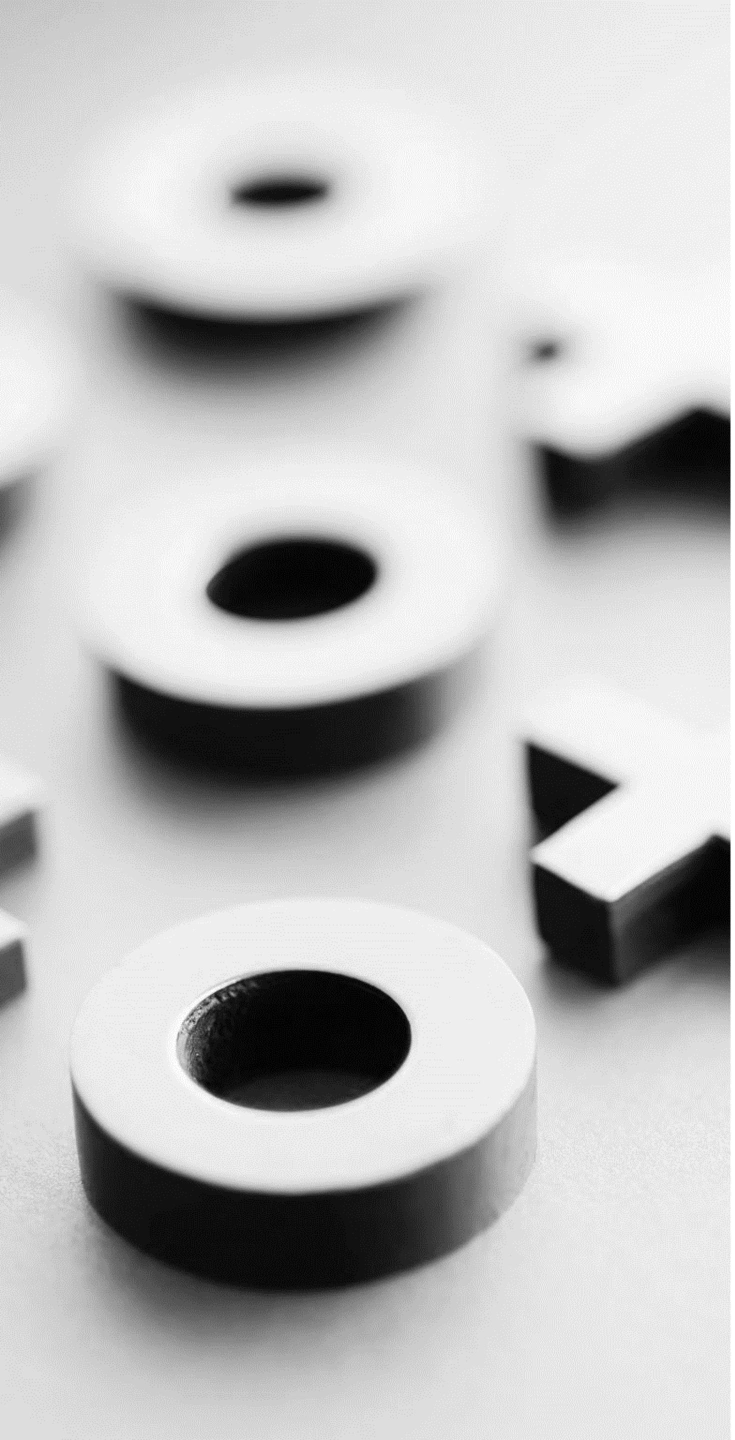
CS 5/7320
Artificial Intelligence

Adversarial Search and Games
AIMA Chapter 5

Slides by Michael Hahsler
with figures from the AIMA textbook

Image: "Reflected Chess pieces" by Adrian Askew

Online Material



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**).

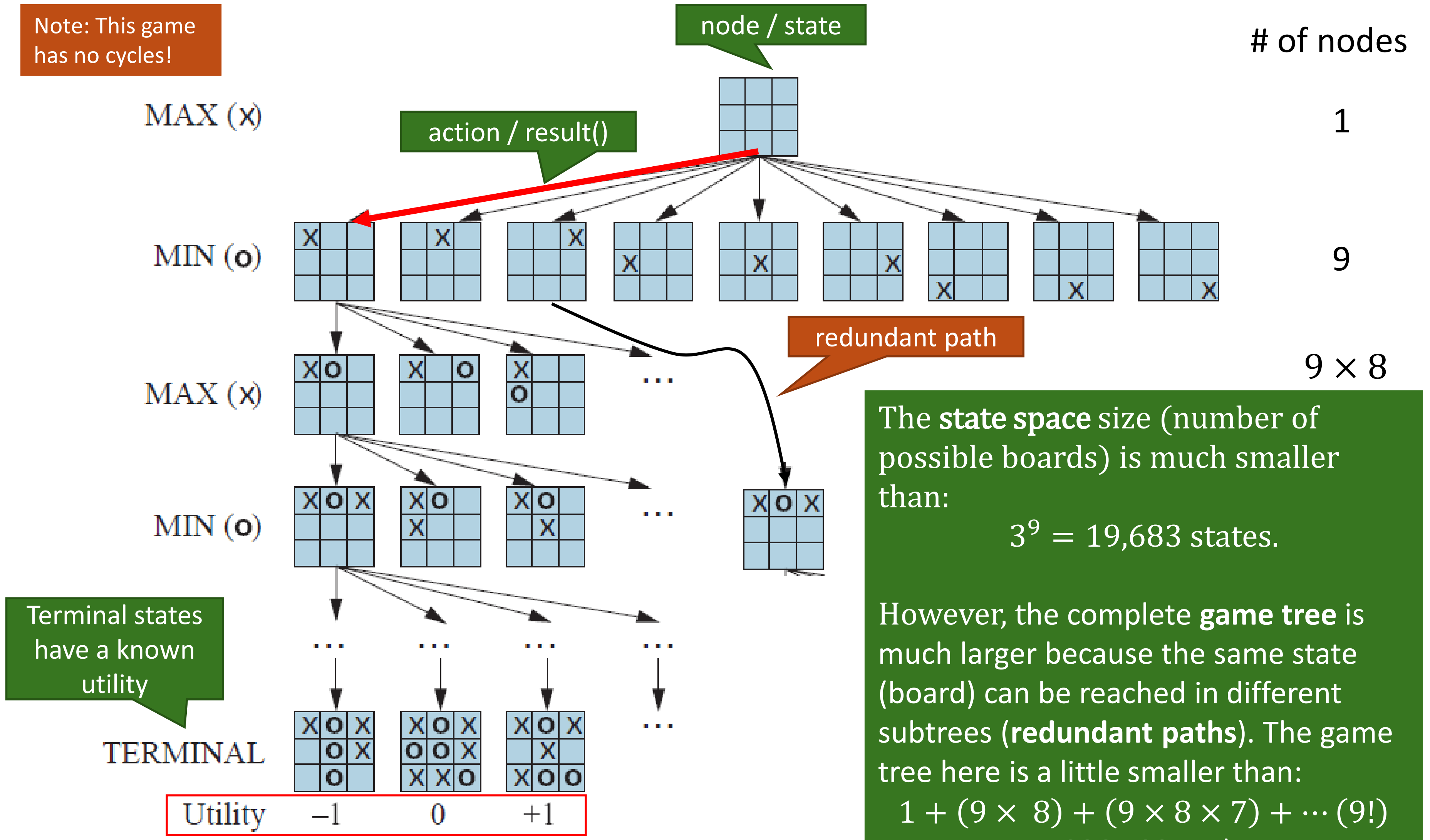
- We call the two players:
  1) **Max** tries to maximize its utility.
  2) **Min** tries to minimize Max's utility (zero-sum game).

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.

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**: opponent is seen as part of an environment with nondete actions. Non-determinism the unknown moves by the **consider all possible moves** opponent.
- **Find optimal decisions**: Minima and Alpha-Beta pruning, wh **player plays optimally** to game.

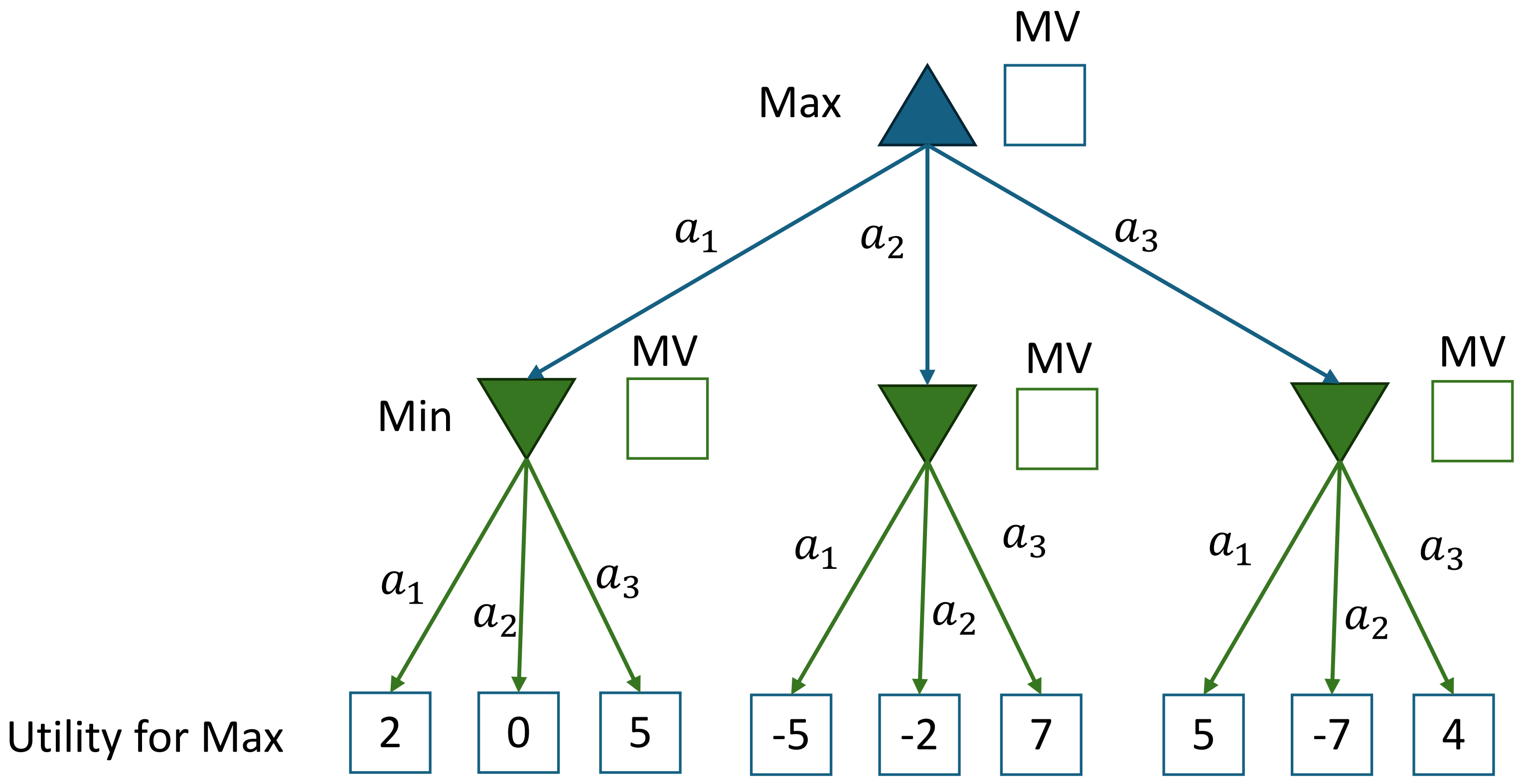
$results()$ function
+ AND-OR Search tree
= conditional plan

DFS to calculate the minmax value of each state
+ pick action that leads to the best state

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.

Minmax 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

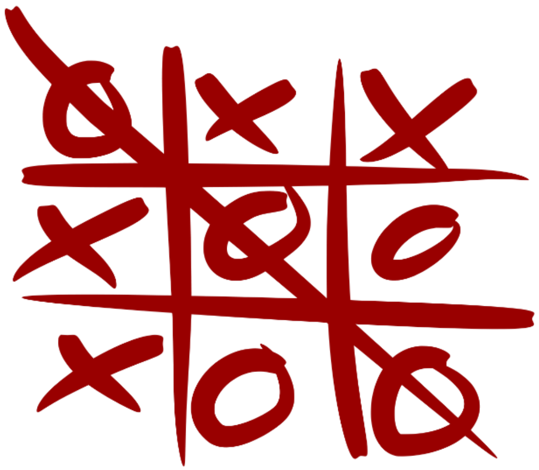
$$b = 9, m = 9 \rightarrow O(9^9) = O(387{,}420{,}489)$$

  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!
- **Game tree pruning using alpha-beta pruning + move ordering**

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.

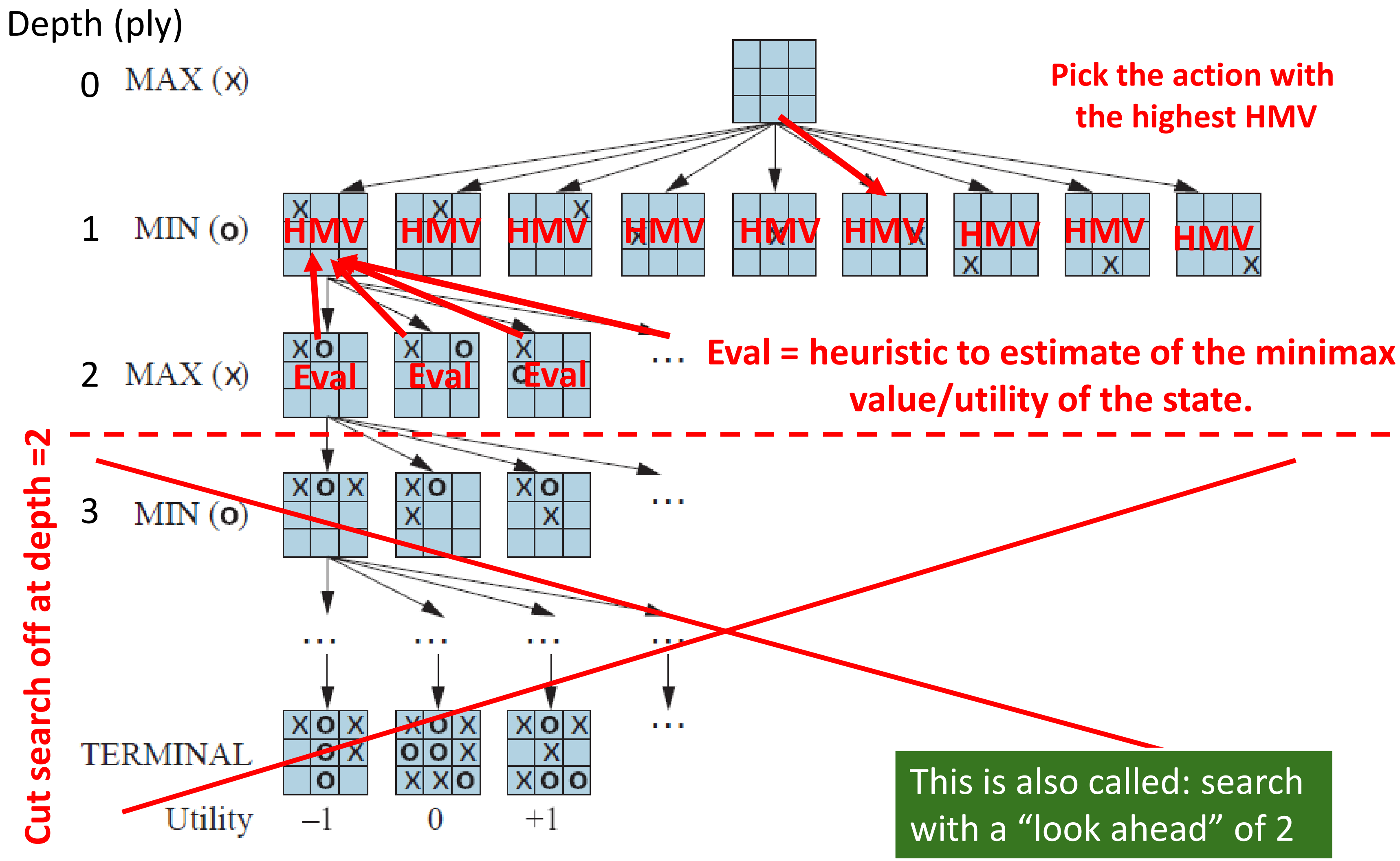
- **Fast approximate methods are needed**.
  We may lose the optimality guarantee, but we can work with larger problems.

Heuristic Methods

Heuristic Alpha-Beta Tree Search

Heuristic Alpha-Beta Tree Search: Cut Off Search

HMV = heuristic minimax value


Depth (ply)
0 MAX (x)
1 MIN (o)
2 MAX (x)
3 MIN (o)
TERMINAL
Utility
–1
0
+1
HMV
Eval
Pick the action with the highest HMV
Eval = heuristic to estimate of the minimax value/utility of the state.
Cut search off at depth =2

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

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]*

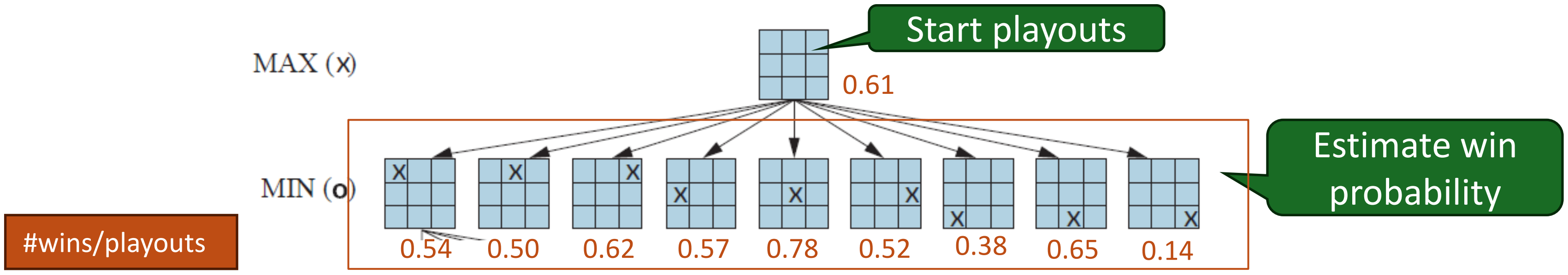
- **Approximate** $\boldsymbol{Eval(s)}$ as the **average utility** of several playouts (= simulated games).

- **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.

Playout Selection Strategy: 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.

Monte Carlo Tree Search (MCTS)

Pure Monte Carlo search always starts playouts from a given state (or its children). **Issue**: We have to start the simulation for each move from scratch.

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

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.

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).

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

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