TECHNISCHE UNIVERSITÄT DRESDEN

Computational Logic Group

Hannes Strass

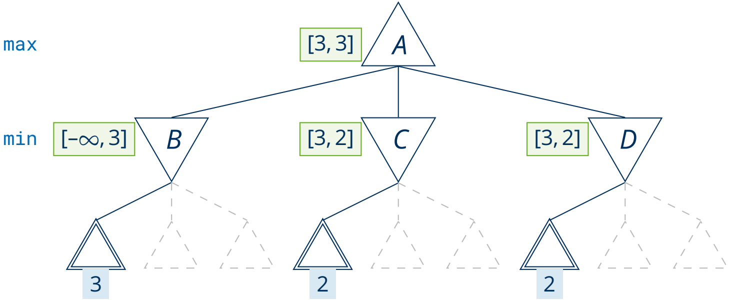
Faculty of Computer Science, Institute of Artificial Intelligence, Computational Logic Group

Playing Games: Monte Carlo Tree Search

Lecture 5, 13th May 2024 // Algorithmic Game Theory, SS 2024

Previously ...

- Game trees can be succinctly represented by **state-based game models**.
- **Minimax Tree Search** can be used to solve sequential (two-player zero-sum) games with perfect information.
- **Alpha-Beta Pruning** allows to reduce the search space without sacrificing solutions.
- **Heuristic Evaluation** of states can be used to reduce search depth.
- Further heuristics may reduce the search space (typically with sacrifices).



Overview

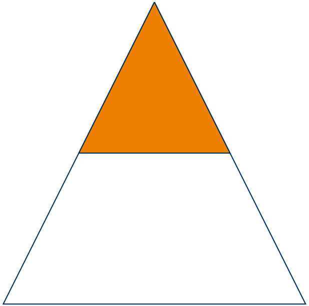
Monte Carlo Tree Search

Selection Policy: UCT

Tree Search: Shannon's Type A and Type B

In Claude Shannon's 1950 paper *Programming a Computer for Playing Chess*, he suggests two types of tree search strategies:

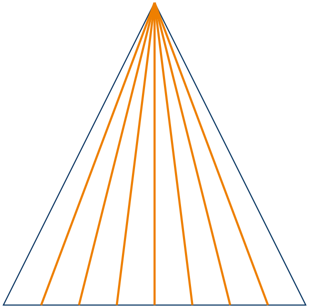
wide, but shallow



Type A

Alpha-Beta Tree Search

narrow, but deep



Type B

Monte Carlo Tree Search

Monte Carlo Tree Search

Monte Carlo Tree Search

Terminology

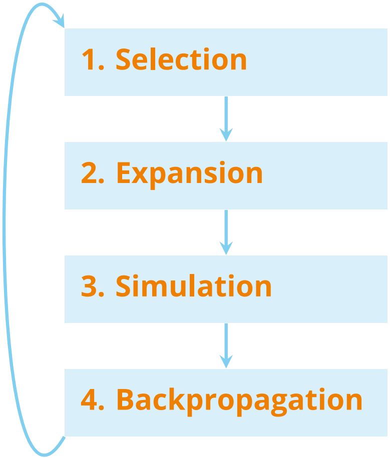
A **Monte Carlo algorithm** is a randomised algorithm whose output may be incorrect with a certain (typically small) probability.

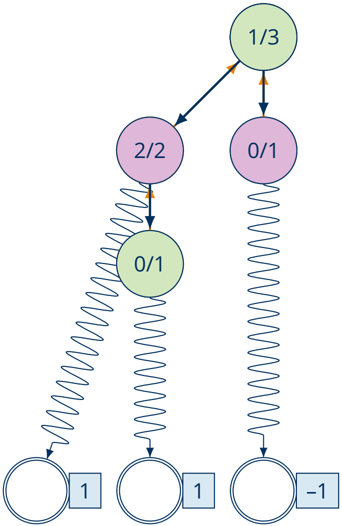
Main Idea of Monte Carlo Tree Search: Simulate random move sequences from current to terminal states and do statistics on moves leading to wins.

Some relevant notions:

- Playout: Complete move sequence from a state to a terminal state.
- Random move sequences only inform about random play, so a playout policy is needed to bias simulation towards optimal play.
- In pure Monte Carlo search, we do N simulations starting in the current state and record average payoffs for all moves.
- Selection policy: Determines from which nodes to start simulations; faces the fundamental issue to balance exploitation and exploration.

Monte Carlo Tree Search: Example (1)





`max`'s choice

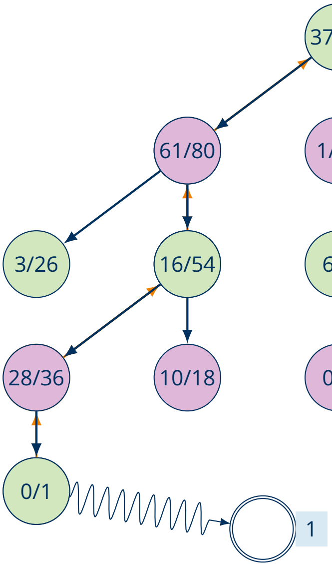
`max`'s statistics

`min`'s choice

`min`'s statisctics

Monte Carlo Tree Search: Example (2)

1. Selection
2. Expansion
3. Simulation
4. Backpropagation



Monte Carlo Tree Search: Algorithm

function **monte-carlo-tree-search**(s : *state*) {
  tree := **get-tree-below**(s)
  while **is-time-remaining**() **do** {
    leaf := **select**(*tree*)
    child := **expand**(*leaf*)
    result := **simulate**(*child*)
    back-propagate(*tree*, *child*, *result*) }
  return **move-to-node-with-most-playouts**(*tree*) }

- **get-tree-below** returns the search tree below the node for the state
- **is-time-remaining** checks whether we are still within the time limit
- **select** uses the selection policy to find a node to expand next
- **expand** adds a new child to the given node (makes a move)
- **simulate** does a full playout, returning only the result (utility value)
- **back-propagate** propagates the result value up the search tree

Selection Policy: UCT

Selection Policy: UCT

An effective policy: UCT – "upper confidence bounds applied to trees".

UCT ranks moves according to their "upper confidence bound" value.

Definition

The **upper confidence bound** value for a node n is obtained thus:

$$\mathrm{UCB1}(n) := \frac{U(n)}{N(n)} + c \cdot \sqrt{\frac{\ln N(n')}{N(n)}}$$

where

- n' is the unique parent of n in the search tree,
- $U(n)$ is the total utility of node n (summed up over all playouts),
- $N(m)$ is the total number of playouts through node m,
- c is a constant that is typically chosen empirically (theoretically $c = \sqrt{2}$).

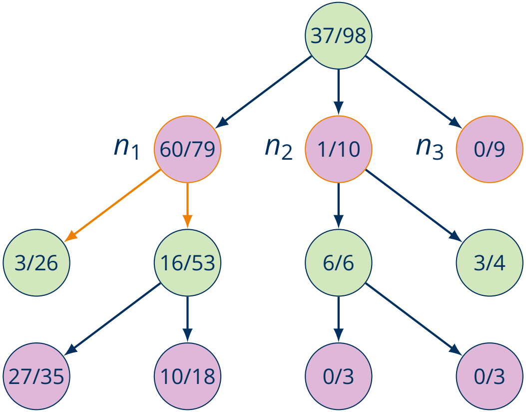
Constant c balances exploitation (first fraction) and exploration (square root).

UCT: Example

$$\mathrm{UCB1}(n_1) = \frac{60}{79} + \sqrt{\frac{2 \cdot \ln 98}{79}}$$
$$\approx 0.76 + 0.34 = 1.1$$

$$\mathrm{UCB1}(n_2) = \frac{1}{10} + \sqrt{\frac{2 \cdot \ln 98}{10}}$$
$$\approx 0.1 + 0.96 = 1.06$$

$$\mathrm{UCB1}(n_3) = \frac{0}{9} + \sqrt{\frac{2 \cdot \ln 98}{9}}$$
$$\approx 1$$



We thus continue with n_1 and apply UCB1 to its children.

Digression: Multi-Armed Bandits (1)

- A K-**armed bandit problem** is given by random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$, where each i is the index of a gambling machine (the "arm" of a bandit).
- Successive plays of machine i yield rewards $X_{i,1}, X_{i,2}, \ldots$ which are independent and identically distributed according to an unknown law with unknown expectation μ_i.
- Rewards across machines are also independent (and not identically distributed): $X_{i,s}$ and $X_{j,t}$ are independent for $1 \leq i < j \leq K$ and $s, t \geq 1$.
- A **policy** is a function mapping past plays and rewards to the next arm to play.
- The **regret** of a policy is the difference between the maximally possible payoff and the actually obtained payoff.

Digression: Multi-Armed Bandits (2)

- UCB1 is a specific policy of "playing" multi-armed bandits that achieves logarithmic regret (in the number n of plays; known to be optimal):

deterministic policy ucb1:

---

initialisation: play each machine once

loop:

play machine j that maximises $\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}}$ where

- $\bar{x}_j$ is the average reward obtained from machine j,
- n_j is the number of times machine j has been played so far,
- n is the overall number of plays done so far.

UCB1 for Multi-Armed Bandits: Example

Consider the following multi-armed bandit with arms 1, 2, and 3. After playing each arm once ($n = 1, 2, 3$), UCB1 determines the next arm to play.

n	u_1	n_1	u_2	n_2	u_3	n_3
$1, 2, 3$	0.3	1	0.0	1	1.0	1
UCB1	1.78		1.48		2.48	
4					1.0 + 0.0	2
UCB1	1.97		1.67		1.68	
5	0.3 + 0.3	2				
UCB1	1.57		1.79		1.77	
6			0.0 + 0.5	2		
UCB1	1.64		1.59		1.84	
7					1.0 + 1.0	3
UCB1	1.69		1.64		1.81	
8					2.0 + 0.0	4
UCB1	1.74		1.69		1.52	
9	0.6 + 0.3	3				

MCTS with UCT: Remarks

- Thus MCTS treats the selection of a new leaf as a sequence of multi-armed bandit problems, using UCB1 for each of them (separately).
- Typically, all successors of a selected leaf are created after expansion; then every unvisited child gets one playout until all have been visited.
- To avoid memory overflow, at some point expansion is omitted (but selection, simulation, and backpropagation are continued).
- The definition of the UCB1 values guarantees that the node with the highest number of playouts is also the one with the highest average utility (and highest confidence in that average utility).
- MCTS does not require hand-crafted heuristic evaluation functions.
- Computing the result of a playout takes linear time in the height of the game tree.
- We still need a playout policy to achieve "realistic" playout values.
- AlphaZero [Silver et al., 2018] learns a playout policy from self-play using neural networks (interleaving learning and MCTS).

Advances in Computer Go

- Go is estimated to have 10^{172} states and a branching factor of at least 361
- Heuristic evaluation of states is not very effective because material value is not very important and most positions are in flux until the endgame

$\rightsquigarrow$ Alpha-Beta Tree Search is not well-suited for Go playing

- Go-playing AIs were weak (beaten by humans) until the late 2000s
- Monte Carlo Tree Search [Coulom, 2006] improved computer Go playing
- UCT algorithm incorporated UCB1 into MCTS [Kocsis & Szepesvári, 2006]
- Adaptive Multistage Sampling (AMS) algorithm incorporated UCB1 into Monte Carlo sampling [Chang et al., 2005]
- Deep reinforcement learning to obtain a playout policy [Silver et al., 2018]
- Computer victory (AlphaGo) over human champions (2015 Fan Hui, 2016 Lee Sedol, 2017 Ke Jie)

The End?

Example: Attackability

Gleave et al. [2023] recently presented an attack on a super-human Go AI:

- Using reinforcement learning against a fixed victim (KataGo), they are able to discover systematic weaknesses in KataGo's gameplay.
- They use AlphaZero-style training, but where AZ plays against itself, they train an attacker to play against KataGo.
- The trained attacker achieves significant win rates against the victim, with and without search.
- The discovered exploit is interpretable and can be learnt by (expert) human players, who can then in turn reliably win against KataGo.

$\leadsto$ If there are single moves that can turn the game, MCTS might fail to consider those moves due to its stochastic mode of operation.

Conclusion

Summary

- **Monte Carlo Tree Search** uses random playouts to evaluate moves and keeps statistics on which moves led to which payoffs how many times.
- A **selection policy** balances exploitation and exploration.
- **UCT** is an effective selection policy that applies UCB1 to trees.
- A **playout policy** steers playout simulations towards realistic play.
- MCTS and deep reinforcement learning led to expert-level Go programs.

Action Points

- Implement a MCTS-based program for playing Tic-Tac-Toe.