

# MONTÉ CARLO TREE SEARCH

This is the first of three (or four) labs. Its purpose is to 1) allow you to get hands-on experience with game trees, and 2) to ensure that your computer setup is up and running. Having installed `uv`<sup>1</sup> (python installer and dependency manager), and cloned our repo<sup>2</sup>, run `uv sync` to install all dependencies. I recommend using Zed<sup>3</sup> if you are on a Unix system and Notepad++, VSCode or Sublime if you are not (but anything will do). We will explore this lab in the coming sessions.

## 1 | Connect four

`aigs/games.py` contains an implementation of Tic-Tac-Toe. You have to:

- ▶ Look at the code and understand it
- ▶ Discuss with neighbor how connect four is different from tic tac toe
- ▶ Create a child class of `Env` called `ConnectFour`
- ▶ Implement an `init()` → State method
- ▶ Implement a `step(state, action)` → State method.

## 2 | Minimax

The minimax function has the signature `minimax(state: State, maxim: bool) → int`. In plain English: it takes a state and a boolean (indicating if we are maximizing or minimizing) and returns an integer (representing value). When we want to take an action, then, we call `minimax` for each of our potential actions, and take the action with the highest value. You have to:

- ▶ Implement the minimax function
- ▶ Call it for every potential action
- ▶ Then take the action with the highest value

---

<sup>1</sup>[docs.astral.sh/uv](https://docs.astral.sh/uv/)

<sup>2</sup>[github.com/syrkis/aigs](https://github.com/syrkis/aigs)

<sup>3</sup>[zed.dev](https://zed.dev)

### 3 | $\alpha - \beta$ pruning

One common sense modification to minimax is to break early when a particular branch allows the oponent something better than what we are already guaranteed to have. The function signature then becomes `alpha_beta(state: State, maxim: bool, alpha: int, beta: int) → int`. You have to:

- ▶ Copy your minimax function
- ▶ Add input parameters  $\alpha$  and  $\beta$
- ▶ Modify the function to break early when appropriate

### 4 | Heuristic variations

Notice how we never actually *looked* at the game board to gague its value, but instead we fully complete games, exhaustively exploring the game tree. Already for connect-four, the combinatoric explosion leaves exhaustive search intractable. One solution is heuristics (rules of thumb). Shannon's 1950s chess program used heuristics [1]. Rather than always fully finishing every simulated game, he'd maximally look  $n$  steps into the future, and then evaluate the board using a *heuristic*—a linear combination of manually crafted features (e.g.,  $0.3 \times \text{number of pawns in the center} + 0.1 \times \text{mean distance from ally pawn to enemy king}$ , or whatever.) You have to:

- ▶ Create a heuristic function that given a board returns a value
- ▶ Copy your minimax function, and add a depth parameter
- ▶ Modify the function to return winner if terminated or heuristic value of depth is 0

### 5 | Monte Carlo tree search (famously hard for games students)

Even with  $\alpha - \beta$  pruning, we are still exhaustively searching through the game tree (in what we know for sure not to be dead ends). Why not just sample? When faced with a choice of making move  $a$  or  $b$  we could simulate  $n$  potential futures for each possibility and count outcomes. There are many variations and smart tricks to MCTS [2]. You have to

- ▶ Implement a rollout function
- ▶ Implement a backpropagation function
- ▶ Implement a selection function
- ▶ Implement an expansion function
- ▶ Combine the above into MCTS

## Index of Sources

- [1] C. E. Shannon, “Programming a Computer for Playing Chess,” *Computer Chess Compendium*, pp. 2–13, 1950, doi: 10.1007/978-1-4757-1968-0\_1.
- [2] C. B. Browne *et al.*, “A Survey of Monte Carlo Tree Search Methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012, doi: 10.1109/TCIAIG.2012.2186810.