95-891 Introduction to AI

Homework 1: Search

Fall 2023

Due 11:59 PM September 14, 2023

Overview

In this, homework you will implement Monte Carlo Tree Search for the game of Tic-Tac-Toe (sometimes called “naughts-and-crosses”), but extended to be played on a 4 by 4 board, instead of the usual 3 by 3. That is, the board has four rows and four columns and player 1 and player 2 take turns marking one square with “X” and “O” respectively. The game continues until a row, column or diagonal has been filled with four X’s or 4 O’s, in which case that player wins, or until there are no more squares to fill in in which case the game is a draw.

We give you skeleton Python code where the program plays against itself, selecting moves at random, and shows the results in terms of the win rate for player one (or more precisely, getting full credit of 1 for a win and .5 for a draw). Your task is to modify the skeleton code so that the first player uses Monte Carlo Tree Search to pick its moves and run experiments to answer some questions. We have provided enough of a skeleton for MCTS that you will only need to write 14 lines of Python code for this assignment. (You can choose not to use the skeleton code, and write the program yourself if you wish). See if you can implement a version of MCTS for the first player that, on average, will outperform the second player using randomly selected moves. For extra credit, you can implement a version of MCTS that remembers the search tree between moves, and may therefore improve its performance

The use of ChatGPT or similar generative AI tools is not permitted for this assignment.

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an advanced technique for selecting moves in games. Deepmind’s AlphaGo used MCTS in combination with a neural network to defeat human Go champions in 2015 and 2016. It is described on pages 161-164 in the Russell & Norvig text (2020 edition). The algorithm involves four major steps:

- **Selection**: Start from the root (current state) and select successive child nodes until a leaf node is reached.
- **Expansion**: Unless the leaf node ends the game, create one or more child nodes and select one.
- **Simulation**: Play a random game from the node chosen in the expansion phase until a result is achieved.
- **Backpropagation**: Update the current move sequence with the simulation result.

Tasks (10 points for full credit, plus 2 points extra credit)

1. (1 point) What would you expect the average win rate for the first player to be when selecting random moves against the second player also using random move selection? Keep in mind the three outcomes of win (counts 1 towards the win rate), lose, (counts as 0) or draw (counts as 0.5). Verify your expectation by running the skeleton code. The skeleton code produces a plot that shows the rolling average win rate for the last ten games played. This is a small window for a moving average, perhaps maybe not enough to show convergence on an average, so you may increase the size of the window if you wish.
2. (5 points) Change the move selection for Player 1 ("X") from random to use Monte Carlo Tree Search, again assigning values for outcomes based on +1 for wins, .5 for draws, and 0 for losses, and rerun the experiment. The skeleton code follows three steps:
    a. Implement the MCTS Node class, using the UCT ("upper confidence bounds applied to trees") selection policy described in class, and on p. 163 of the Russell and Norvig text. The skeleton code contains the code for this; you should not need to change it.
    b. Implement the MCTS algorithm. You will need to make some changes to the code here.
    c. Modify the game-playing loop to use MCTS for player X's moves. You will need to change the code here as well.
3. (2 point) How well does MCTS perform against random move selection? Was this what you expected? How do you explain the result?
4. (2 points) Perform some experiments to see if you can fine-tune the exploration weight C in the UCT child selection to achieve better results.
5. Extra credit (2 points) Assuming your code to this point does not save the search tree from move to move, modify the MCTS move choice, so that rather than discarding the search tree at the end of each game, it retains the part of the tree that corresponds to the moves actually played. This allows the search to continue from where it left off. Rerun the experiment to see the impact of this change. No skeleton code is provided for this part, but you may wish to implement three steps :
    a. Maintain a root node that will represent the current game state.
    b. After each move, update this root node to be the child node that corresponds to the move made.
    c. Discard the rest of the tree (the parts not taken), and the tree will continue to grow from the new root in subsequent simulations.

    Evaluate your solution – what is the impact on performance over the MCTS without memory?

Submission

Please submit a Jupyter Notebook to Gradescope showing the code and including markdown to answer the questions above.(or submit the code and a separate .pdf file with the answers to the questions. If you do not wish to use the skeleton code, you are welcome to develop your own program to solve the problems.