Programming Final Project: Connect 4 Bot

Manual

The game:

The game played here is the classic board game called Connect Four, a two-player game where the objective of the game is to place 4 tokens in a row, vertically, horizontally, or diagonally. The game is played on a 7 columns x 6 rows board. To play, the player must drop their coloured token from the top of the board. Since the game is played with the board upright, the pieces fall onto each other to occupy the bottom most place of the column.

Our recreation:

Using tkinter, we've recreated the visual aspect of the board in python, the blue grid along with the coloured tokens that can be placed by the player and the computer opponent. In our recreation, the starting player is randomized between the bot and the player, and they always start as the red pieces. The game ends once a winner is decided. However, if the board is filled without a four-in-a-row, a draw will be declared. The bot chooses its move based on a minimax algorithm which chooses the optimal move according to the scoring system mentioned below.

How to use

1. Run main.py
2. To place a token, click on the squares at the top of the board to drop a piece in that column.
3. Once a column is filled, no more pieces can be places there.
4. Wait for the bot to place its piece.
5. Repeat until a winner is decided or a draw is declared.

Design:

Backend:

The backend was written using the board class as a node, the game class as a linked list, and the tree class to store the data required for a minimax tree.

Dictionaries:

coord_win:

Maps every coordinate (y, x) with a list containing all the possible winning possibilities

wins:

Maps every winning possibility with the number of slots that are filled within it

Win_conditions file:

Used to compute every possible win condition and the mappings for the coord_win and wins dictionaries. Not needed anymore, since the dictionaries are saved inside a separate file.
Win condition names follow the pattern:

H1_1

H for horizontal, 1_1 is the coordinate of one of the slots it contains

Every winning possibility is unique (H for horizontal, V for vertical, DD for diagonal down, DU for diagonal up).

Minimax Scoring system:

Sum of the filled slots for every winning position for each player (in wins) (positive for player 1, negative for player 2).
Result: A positive score means player 1 is winning, negative means player 2 is winning.

Board Class:

__init__:

The board class acts like a node, which must be given the state of the current board, and the state of the previous board as inputs.
The state is represented by a 6x7 matrix containing 3 types of data. (None for empty or 1/2 depending on the player.
Each board also stores data such as:
Who's turn it is to play,
The remaining set of winning combinations,
The win possibilities left for each player,
Is the game won, and
The minimax score of the board state.

__repr__:
Represents the board as a list containing 6 nested lists of length 7.

__gt__:
Always returns True. (it is needed in the case of comparisons of tuples, but a clear definition isn't necessary)

check_player:
Returns the opposite player to the one given.
Ex: Returns 1 if the player given is 2

check_win:
Given a coordinate (x, y), it

1. updates the amount of slots filled for the affiliated winning possibilities
2. removes the same winning possibilities for the opponent
3. Check if the board state is won
4. Updates the score (Adds new points and subtracts points lost by opponent)

Refresh:
Refresh the state of the board by running check_player() and check_win()

Move_b:
Snip bit of code that is useful and often reused in the move() function in the game class.

Tree Class:

__init__:
Represents a tree containg a board, if this board should be maximised/minimised, and the last column played. It also contains a list of its children.

Add_children:
Adds a tree to the list of children

Add_score:
Updates the maximum/minimum scoring board depending on if minimax is True.

Game class:

Enables the editing and storing of multiple boards

__init__:
Contains the current board, the coord_win dictionary, and the number of turns played

__repr__:
Returns the representation of the current board

Move:
Changes the board state according to the column played by the player (input is a click of the button)

Move_bot:
Determines the optimal move for the bot to make with the input (i) being the depth of the minimax tree.
In chronological order, the function:

1. Creates a tree containing all the possible moves for the i next turns, exiting early if an immediate win or an edge case (ways to beat the ai easily at the start of a game. Ex: Starting in column 6 followed by 4 and 5 to force a win) is found.
2. Making a tree traversal with minimax algorithm returning the maximum point earning move for the bot.
3. Updating the current board of the game class to the one done after the bot has moved.
4. Returns the number representing the column played by the bot (0 to 6)

Visual element:

The visual aspect of the game was created using python's built-in "tkinter" library. The game class inherits from the Tk class.
Starts with defining some variables used in the program such as the colours and the size of the board.

__init__:

The board of the game was created by creating a blue canvas and filling it with white circles in a grid-like pattern. Those circles are appended to a dictionary "self.grid_d" to be able to change the colour of certain places.
Then some rectangles are created at the top of the board to act as the buttons to click on
It then uses randint to randomly determine whether the player or the bot starts first. Finally, it binds left click to be able to drop pieces.

place:

given an inputted row, column, and colour, it places a circle at the location specified.

drop_piece:

Drops a piece at the inputted column and updates the game state. Does the player move then the bot move in the same function. Called when buttons on top of the board are clicked.

on_click:

Handles click events on the canvas. Only drops if it is in the upper part of the board and separates it into seven sections to determine which column to drop it in.

drop_animation:

Changes colours of the pieces from top to bottom with a slight delay to simulate a dropping piece, calls update every time. Purely aesthetic.

update:

updates the visual state of the board to match the game state defined by the game class. Checks the positions of the pieces on the board class. Uses canvas.update(). Also updates the turn number and checks for win.