CS 120 (Spring 22): Introduction to Computer Programming II

Long Project #6 - Connect N

due at 5pm, Tue 1 Mar 2022

REMEMBER: The `itertools` and `copy` libraries in Python are **banned.**

1 Overview

Classes are a great way to organize your code inside a single program. But sometimes, they can be even more: a way to share code across a number of similar programs.

In this Project, you will write code that implements a "game state" object for Connect N - a generalization of Connect 4 (`https://en.wikipedia.org/wiki/Connect_Four`). This class will be used by several programs, all of which I will provide:

- A graphical interface, which simply allows you to play the game.
- A text-based interface, which plays the game without graphics (and thus I can use for testcases).
- A variety of one-shot testcases, each of which creates a board, does one or two calls, and then terminates.

Thus, as you implement the methods of the `Connect_N_State` class, you will be able to use the tools to test out how the class is working.

1.1 Why Connect N Instead of Connect 4?

In the original Connect 4 game, you play on a 7x6 board, you need to get 4 tokens in a row to win, and there are only two players, Red and Yellow.

Most of the testcases in this project will make these assumptions, as well - so if you want to hard-code them into your program, you can still earn most of the points.

However, to earn **all** of the points, you must be flexible. The constructor for your class will take 4 arguments (whether or not you use them): the width, height, target length, and a list of players. To earn all of the points, you must support flexibility in all of these parameters.

2 Requirement: Private Variables

Every variable used by this class (of course, this does *not* include methods) must be **private.**

3 Required Class: `Connect_N_State`

Implement the class `Connect_N_State`, inside the file `connect_n_state.py` . You must implement the following methods:

- `__init__(self, wid,hei, target, players)`

  Even if your code only supports the standard game (7x6, target=4, 2 players), you must accept all of the parameters to the constructor.

  Note that `players` is an array of strings, giving the name of the players.

  You may assume that the paramters have reasonable values; you do not have to do error checking on them.

- `get_size(self)`
  `get_target(self)`
  `get_player_list(self)`

  These simply query what values were used in the constructor. For `size`, return it as a tuple: `(wid,hei)`. For the player list, you may return the original array that you were given in the class constructor, or a duplicate.

- `is_game_over(self)`

  Return `True` if, after the most recent move, the game has ended - either because one of the players made the target length, or because the board is completely full.

- `get_winner(self)`

  Return the string for the player that won. You may assume that this will only be called if the user has already called `is_game_over()` and you returned `True`.

  If the game was a tie because the board filled up, return `None`.

- `is_board_full(self)`

  Return `True` if the board is full. Perhaps a player won the game; or perhaps not.

- `is_column_full(self, col)`

  Return `True` if the column in question is full (that is, not able to accept any new tokens.

  NOTE: Column numbering starts at 0.

- `get_cell(self, x,y)`

  Queries the state of a single cell in the game. As in the previous method, column values (`x`) start counting at 0, which is the left edge of the board.

  `y` values also start at 0, which is the **bottom** of the board.

  If a player has played a token into that cell, then return the string that represents the player. Otherwise, return `None`.

- `get_cur_player(self)`

  Return the string name of the player that will move next (if any moves are allowed).

  When the game begins, the first player to move will be `[0]` of the player list. Each time that a successful `move()` occurs, move the cur player to the next in the list.

  If a `move()` is attempted but rejected, do **not** change the cur player.

- `move(self, col)`

  The current player attempts to drop a token into the column in question.

  Return `True` if you succeed in adding it to he column, and `False` if it fails for any reason.

- `print(self)`

  Prints out the current state of the board, as a grid of characters. Empty spaces in the board should be represented by period (`.`) characters; spaces that contain tokesn should be represented by the first letter of the player's name. (See the testcase output for examples.)

4 Turning in Your Solution

You must turn in your code using GradeScope.