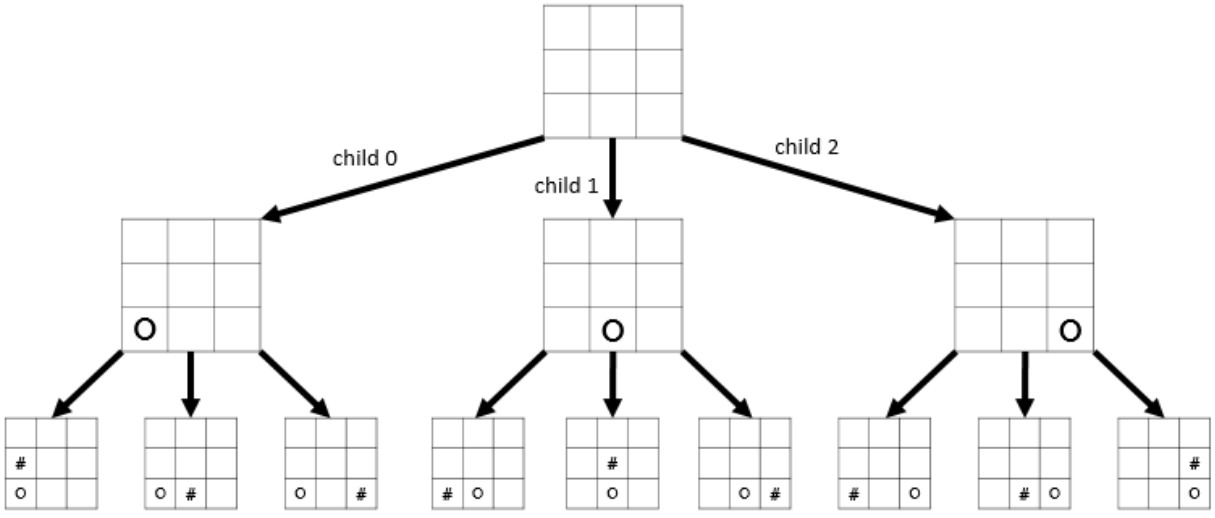
CP2410 Assignment 2 – Game Trees and Connect 3

Introduction

Tree data structures are useful for representing hierarchies and branching data. One application of trees is to represent the possible states of a two-player game as it progresses with different possible moves. Such a representation, a **game tree**, provides a starting point for creating an AI (artificial intelligence) to play the game effectively. Game trees are a major part of the way sophisticated Chess and Go playing programs can beat even the best human players. For simple games, we can construct a full game tree of all possible positions and determine the best moves for each player. More complicated games cannot represent a full game tree – for example, the game tree for Chess is estimated to have 10^{120} nodes – far larger than the number of atoms in the observable universe.

For this assignment, you'll create a game tree for Connect 3 on a small board. Connect 3 plays like its inspiration, Connect 4® (if you've never seen Connect 4, there are lots of videos on YouTube, e.g. https://www.youtube.com/watch?v=d-7eiD2DNGw). Two players start with an empty vertical board, and take turns dropping tokens into a particular column. The first player to have three tokens in a row, horizontal, vertically, or diagonally, wins. The game ends in a draw if the board fills up with tokens.

If we have a game tree for Connect 3, we can work out how good a move is for either player 1 or player 2 by evaluating how good all the subsequent moves are, and so on recursively until we reach the leaves of the tree. At the external nodes (leaves) of a game tree are the games that have finished – either a win for player 1 or player 2, or a draw.



Three incomplete Python files are provided: **connect3board.py**, **playgame.py, and gametree.py**. Your submission should consist of completed versions of these files to satisfy the requirements given in this document, and a Word Doc containing your planning and analysis.

Requirements

Part 1 – Two-player mode on a variable size board

For the first part of the assignment you'll create a two-player mode for the game, where the choice for each turn, O or #, is entered by the users, so they can play a friend, or against themselves.

A starting point is available in **playgame.py**. For the two-player mode, users should be able to enter their preferred number of rows and columns (at least three, and up to a maximum of seven).

Starting code is provided for **connect3board.py**, which contains the class Connect3Board, representing a current game of Connect 3. In the version provided, wins are only detected correctly on 3×3 boards. **It is your task to add the ability to detect wins for boards with more rows and columns.**

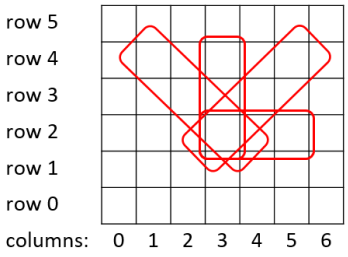
The provided Connect3Board class provides several methods to facilitate playing a Connect 3 game:

- **`get_whose_turn()`** – returns the token of the player whose turn it is (O or #).
- **`add_token(column)`** – adds a token to the given column based on whose turn it is, and advances to the next turn.
- **`can_add_token_to_column(column)`** – returns true if adding a token to column is valid, returns false if the column is full, or an invalid number.
- **`get_winner()`** – returns the token of the player who has won (O or #), or "DRAW" in case of a draw, or None if the game is incomplete – **you are to add code to this method so that it can determine who the winner is for boards larger than 3×3.**
- **`__str__()`** – returns a string representation of the game board.

For this part of the assignment, you are to complete the function `run_two_player_mode()` in **play_game.py**.

The provided Connect3Board class represents the board internally as a list of lists. Each list represents a row of the board, and consists of `None` elements, or "O" or "#" elements. The elements can be accessed within the class as `self._board[row][column]`.

To compute the winner for large board, you will need to loop through the rows and columns. See the figure below for a hint on which neighbours to check for a given cell at row, column.



For sample output of the two-player mode, see the appendix at the end of this document.

Requirements for part 1:

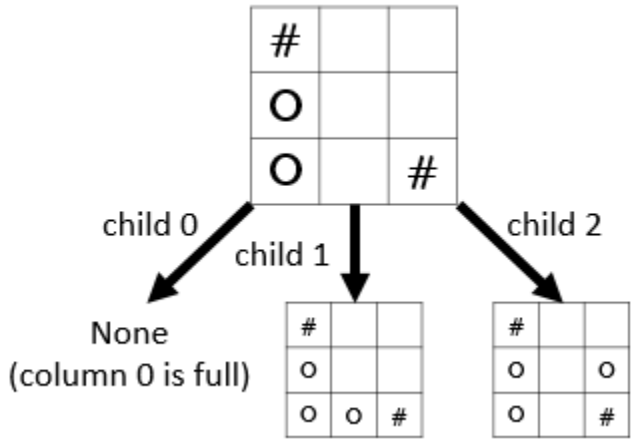
- **Planning – pseudocode or high level overview, and Python implementation for**
  - **Two-player mode for the Connect 3 game**
  - **Playable on a user-selected size board**
  - **Program correctly detects wins (3-in-a-row vertically, horizontally, or diagonally)**

Part 2 – Game tree and minimax for AI

For the second part of the assignment, you'll implement a mode where users can play against an AI on a 3×3 board.

gametree.py contains starter code for building a game tree for effective AI. The **GameTree** class represents an entire game tree, and includes two nested classes, **_Node** and **_Position**.

_Node represents a single node of the game tree and contains a Connect3Board representing a single game state, and references to three (3) child nodes, one for each column that a token could be placed in. Each of the child nodes contains a Connect3Board in a subsequent state where another move has been played. If a move cannot be played, then child will be None (e.g. below, column 0 is unplayable, so there is no child 0).



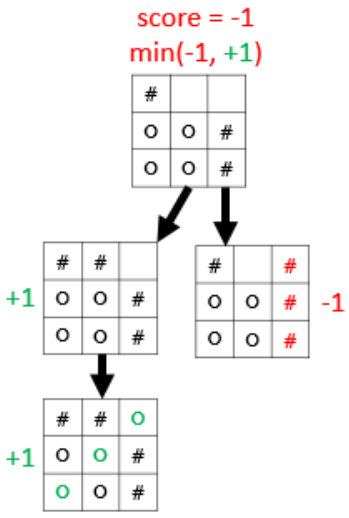
_Position provides a way of navigating down the tree. Calling **`get_root_position()`** on the GameTree object will give the Position of the root node, which can then be used to access

information about the node and its children. _Position is useful for using the game tree once it has been constructed; it's not necessary for constructing the game tree itself.

- **`get_child(column)`** – `return the Position of the child represented`
- **`get_children_scores()`** – `returns a list of scores of the children.`

You must provide planning and implementation for building the full game tree for the 3×3 case for Connect 3. There are multiple ways to do this, but we suggest building the tree **recursively**. Each Node creates its children, which each create their own children, and so on, until the end of the game tree is reached. If a Node contains that board that has a winner (O or #) or is drawn (board is full, and no one has won), it will not have any children, and so the recursion ends. You can make use of the Connect3Board method **`get_winner()`** here. To construct the children, you will need to make **copies** of the Connect3Board in the current node, and then use **`add_token()`** to advance the game. We've provided a Connect3Board method **`make_copy()`** for this purpose. (If you don't make copies, all the nodes will store the same Connect3Board, and the game tree won't make sense.)

To create an effective AI, you are to implement the minimax decision rule to calculate scores for all the nodes in your game tree. Consider two players, Max and Min. Given several possible moves, Max will always choose the option that **maximises** the score, and Min will always choose the option that **minimises** the score. Suppose we give scores to the leaves of our game tree as follows: +1 if the node is a win for Max (O), -1 if the node is a win for Min (#), and 0 if the node is a draw. We can then calculate scores recursively for the internal nodes as follows. If it's Max's turn (O), then the score is the maximum of the scores of the children. So if the node's children are +1, 0, and 0, then the node itself is +1. If it's Min's turn (#), then the score is the minimum of the scores of the children. For example, in the figure below, it is #'s turn at the top node. There are two available moves – one leads to a win for # and scores -1, the other leads to a win for O and scores +1. The score for the top is the minimum of these scores, and so it's -1.



Basically, minimax asks the question *"What is the best that I can do in this situation, if I assume my opponent will do the best they can on the next turn (and we each make the best moves after that, and so on)?"* Because we are building the entire game tree, we are guaranteed to get an effective move selection strategy this way, because, at each stage, we can maximise and minimise over all possible paths to the end of the game. You can read more about minimax at https://en.wikipedia.org/wiki/Minimax.

Once you have your game tree construction working, you can plan and implement the minimax scoring. After you construct a node's children, you can calculate the score at that node. Again, this can work recursively. In the base case, if a node represents a win for O, give it a score of +1, if a node represents a win for #, give it a score of -1, and if it represents a draw, give it a score of 0. In the recursive case, if it's O's turn, then give the node the maximum of the scores of its children. If it's #'s turn, then give the node the minimum of the scores of its children.

After all this, to implement your AI, you can construct your game tree, get the root node position as `position = gametree.get_root_position()`. On the computer's turn, ask the position object for the scores of the children, and find the index of the child which has the maximum (if the computer is playing O) or minimum (if the computer is playing #). That will be the column to play for the best move. After each move selection by the computer or player, you can navigate down the game tree by setting

```
position = position.get_child(column_choice).
```

The AI mode should allow the user to choose whether to play as O or # at the start (O always goes first). See the appendix for sample output of how this mode should work.

Requirements for part 2:

- **Planning – pseudocode or high level overview, and Python implementation for**
  - **Constructing the Connect 3 game tree for a 3×3 board**
  - **Minimax scoring of the nodes of the game tree**
- **Python implementation for AI opponent mode of Connect 3 on a 3×3 board**
  - **Offers the user the choice of token (O and #)**
  - **Correctly alternates between user input and AI control**
  - **Never loses, and always take a win if available (if your game tree and minimax implementations are correct, this is guaranteed)**

Part 3 – Analysis

A. Perform a theoretical (big O) analysis of the running time of your `get_winner()` algorithm from Part 1 in terms of the number of rows (r) and columns (c).

B. Calculate an upper bound on the number of nodes for the game tree for Connect 3 on an $r \times c$ board. Hint: at each stage the tree branches out c more nodes, and the longest a game can last is $r \times c$ moves. What is the biggest size board for which you could feasibly store a Connect 3 game tree? Assume each node requires about $r \times c$ bytes of storage.

Three incomplete Python files are provided: **connect3board.py**, **playgame.py, and gametree.py**. Your submission should consist of completed versions of these files to satisfy the requirements given in this document, and a Word Doc containing your planning and analysis.

Marking Rubric

Criteria	Excellent	Good	Marginal	Unacceptable
Two-player game	40 Clear and correct high level overview or pseudocode for the two-player game mode and get_winner() method. Two-player game runs correctly on a user-selected board size and correctly detects winners.	30 Clear high level overview or pseudocode for the two-player game mode and get_winner() method. Two-player game generally runs correctly on a user-selected board size and correctly detects winners.	20 Generally clear high level overview or pseudocode for the two-player game mode and get_winner() method. Two-player game runs on a user-selected board size and detects winners.	0 Missing overview or pseudocode. Two-player game does not run correctly or correctly identify winners.
Game tree construction	20 Clear and correct high level overview or pseudocode for game tree construction. Correct implementation of game tree construction in Python.	15 Clear high level overview or pseudocode for game tree construction. Generally correct implementation of game tree construction in Python.	10 Generally clear high level overview or pseudocode for game tree construction. Reasonable attempt at implementing game tree construction in Python.	0 Missing overview or pseudocode. Implementation of game tree construction in Python not reasonably attempted.
Scoring nodes	20 Clear and correct high level overview or pseudocode for minimax scoring of game tree nodes. Correct implementation of minimax for scoring nodes in Python.	15 Clear high level overview or pseudocode for minimax scoring of game tree nodes. Generally correct implementation of minimax for scoring nodes in Python.	10 Generally clear high level overview or pseudocode for scoring game tree nodes. Reasonable attempt at implementing minimax for scoring nodes in Python.	0 Missing overview or pseudocode. Implementation of minimax for scoring game tree nodes in Python not reasonably attempted.
AI controlled game	10 AI controlled game correctly covers all the following: • offers user choice of token • alternates between user input and computer controlled turns • never loses, or fails to win when able	7.5 AI controlled game correctly covers all the following, with minor lapses: • offers user choice of token • alternates between user input and computer controlled turns • never loses, or fails to win when able	5 AI controlled game correctly alternates between user input and computer control. The AI makes some moves that are better than random chance.	0 Not attempted, or fails to work correctly in all aspects.
Analysis	10 Clear, well-justified, and correct responses to the part 3 analysis questions.	7.5 Justified and generally correct responses to the part 3 analysis questions.	5 Partially correct responses to the part 3 analysis questions with some justification.	0 Nonsensical or unjustified responses to the part 3 analysis questions.

Appendix

Two-player mode sample output

```
Welcome to Connect 3 by Donald Knuth
A. Two-player mode
B. Play against AI
Q. Quit
>>> A
How many rows? (3 to 7) 5
How many columns? (3 to 7) 5
 01234
|     |
|     |
|     |
|     |
|     |
-------
 01234
Player O's turn. Choose column (0 to 4): 2
 01234
|     |
|     |
|     |
|     |
|  O  |
-------
 01234
Player #'s turn. Choose column (0 to 4): 1
 01234
|     |
|     |
|     |
|     |
| #O  |
-------
 01234
Player O's turn. Choose column (0 to 4): 2
 01234
|     |
|     |
|     |
|  O  |
| #O  |
-------
 01234
Player #'s turn. Choose column (0 to 4): 2
 01234
|     |
|     |
|  #  |
|  O  |
| #O  |
-------
 01234
Player O's turn. Choose column (0 to 4): 2
 01234
|     |
|  O  |
|  #  |
|  O  |
| #O  |
-------
 01234
Player #'s turn. Choose column (0 to 4): 2
 01234
|  #  |
|  O  |
|  #  |
|  O  |
| #O  |
```

```
-------
 01234
Player O's turn. Choose column (0 to 4): 2
That column is not available. Please choose again.
Player O's turn. Choose column (0 to 4): 1
 01234
|  #  |
|  O  |
|  #  |
| OO  |
| #O  |
-------
 01234
Player #'s turn. Choose column (0 to 4): 0
 01234
|  #  |
|  O  |
|  #  |
| OO  |
|##O  |
-------
 01234
Player O's turn. Choose column (0 to 4): 0
 01234
|  #  |
|  O  |
|  #  |
|OOO  |
|##O  |
-------
 01234
Player O wins!
```

Play against AI sample output

```
Welcome to Connect 3 by Donald Knuth
A. Two-player mode
B. Play against AI
Q. Quit
>>> B
Will you play as O or #? O
 012
|   |
|   |
|   |
-----
 012
Your turn. Choose column (0 to 2): 0
 012
|   |
|   |
|O  |
-----
 012
AI's turn
 012
|   |
|   |
|O #|
-----
 012
Your turn. Choose column (0 to 2): 0
 012
|   |
|O  |
|O #|
-----
 012
AI's turn
 012
```

```
|#  |
|O  |
|O #|
-----
 012
Your turn. Choose column (0 to 2): 2
 012
|#  |
|O O|
|O #|
-----
 012
AI's turn
 012
|# #|
|O O|
|O #|
-----
 012
Your turn. Choose column (0 to 2): 1
 012
|# #|
|O O|
|OO#|
-----
 012
AI's turn
 012
|# #|
|O#O|
|OO#|
-----
 012
Player # wins!
```