Project

Deadline: 16.11.2018 23:59 SGT

The project is an assignment that has to be completed in groups of 3 persons maximum (the persons **have to be** in the same lab group). This project will test your ability to use Python in a more concrete and complex setting than what you were used to during the lab sessions. Also, it will improve your ability to work as a team when having a project to complete. The goal of the project is to program the game **Connect4** in Python (the pop out version), with a simple user interface and game display, and where a human can play against either a human or a computer player.

Very Important!!! Please follow the instructions below. Projects that will not **strictly follow** these instructions will risk getting big grade penalty.

- The Python script to execute the program must be in your project repertory and named exactly `connect4.py` (no uppercase letter). For submission of your project on NTU Learn, please create a zip archive from your project repertory, name the archive according to your matric number (i.e. `<YourMatricNumber>.zip`), and upload it on NTU Learn. If two or three people are in your group, simply separate the matric numbers with an underscore character.
- The submission system automatically closes exactly at the deadline. Hence, after that, if you didn't submit, you will get 0 point for your project. You can update your submission as many times as you want on NTU Learn, only the last submission will be taken into account. Thus, I advise you to submit an earlier version much before the deadline, to be sure you won't end up with no submission at all.
- Only one person of your group submits the project on NTU Learn. Do not submit the same project for all your group members.
- Make sure your project works properly (i.e. the program doesn't output errors). If errors are output during the test of a functionality of your project, this functionality will be considered as not working at all.
- Do not copy any code (or part of) from other groups or from implementations available online. Special software will be run to check for such cheating cases. If you are caught copying code, you will risk serious consequences (see course guidelines). Also, do not let other groups copy your code, as both groups with similar code will get 0 mark for the project.
- Your Python script must implement the functions `check_victory`, `apply_move`, `check_move`, `computer_move`, `display_board` and `menu` (see below). Make sure you don't miswrite the functions names, or make an error in the input/output that are expected for these functions. The grading will be mostly be based on these functions.
- You can find on NTU Learn a file `test.py` to pre-test your functions (simply copy the file in your repertory and run it). It should output OK for all tests. You can also find on NTU Learn a skeleton of the `connect4.py` file.
- You should not have any Python code that does not belong to a function (i.e. all your Python code must be inside a function), except the `Game` class definition, the Numpy module import, and except the call to the `menu` function, which will start your program. When submitting (or when testing with `test.py` file), please remove the call to the `menu` function (or put it as comment), so that it is not executed when your file is imported.

1 Connect 4 game

Connect4 is a two-player game in which the players first choose a color and then take turns dropping colored discs from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the next available space within the column. The object of the game is to connect four of one's own discs of the same color next to each other vertically, horizontally, or diagonally before your opponent (description taken from wikipedia: `http://en.wikipedia.org/wiki/Connect_Four`). In case the board is full and no player managed to connect 4 discs, then it is a draw.

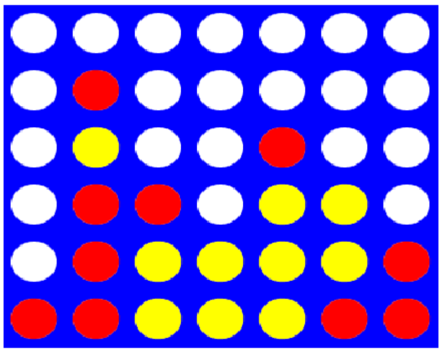

Figure 1: Example of a **Connect4** game. The yellow player wins with 4 consecutive yellow discs in the second row.

For those who don't know the game, I would advise you spend ten minutes to try it on this website: `http://www.mathsisfun.com/games/connect4.html`

You will have to implement the **pop out** version of this game: during each turn, instead of adding disks, a player can otherwise choose to "pop out" one of **its own** disk on the bottom row (i.e. only a single column of the board is impacted). Popping a disc out from the bottom drops every disc above it down one space, changing their relationship with the rest of the board and changing the possibilities for a connection.

2 Objectives

The objective of the project is to write in Python a **working Connect4** (pop out version) program. The program must allow the user to configure:

- the size of the board (i.e. the number of rows r and the number of columns c of the board, the default values must be $r = 6$ and $c = 7$). You can set minimum and maximum values if you want.
- the number N of consecutive same-color discs required for a player to win (by default the value must be $N = 4$). Again, you can set minimum and maximum values if you want.
- the type of the two players (human or computer), and the difficulty level in case of a computer player (two levels).

You have to implement this **Connect4** game following the three steps given below. Moreover, note that everytime you add a feature to your program, you should test it thoroughly before continuing. Testing your program only at the very end is the best way to render the bug hunting close to impossible !

3 Three steps to complete the project

3.1 1st step: implementing the skeleton of the project and the user interface

The first step in a programming project is perhaps the most important one: before writing any Python code, you should think about the functions you will need to implement, their input/output, their goal, the overall structure of the entire program. Once this done, you should write a skeleton of the project that only handles the interface with the user (i.e. how the user will choose which column to play), the display of the board, and the initialization of the variables.

3.1.1 Data Structure for the game.

In order to represent the board in Python, you can use a simple data structure: a $r \times c$ matrix of integers, where 0 represents no disc (i.e. empty), 1 stands for a disc belonging to player 1, and 2 for a disc belonging to player 2 (row 0 being the bottom row). In order to represent that matrix, use a Numpy two-dimensional array: first import the Numpy module using:

```
import numpy as np
```

You can create an empty board (all elements filled to 0) of r rows and c columns using:

```
game_board = np.zeros((r, c))
```

Finally, the element of the matrix located at row r and column c can be accessed using `game_board[r,c]`. We will use as convention that the row 0 is the bottom row (the one in which will end up the very first disk inserted).

You must create a `Game` object that will represent a **Connect4** game. For that, simply copy/paste the following code in the beginning of your script:

```
class Game:
    mat = None # this represents the board matrix
    rows = 0 # this represents the number of rows of the board
    cols = 0 # this represents the number of columns of the board
    turn = 0 # this represents whose turn it is (1 for player 1, 2 for player 2)
    wins = 0 # this represents the number of consecutive disks you need to force in order to win
```

This will allow you to create a game instance like this:

```
my_game = Game()
```

You can later change the values of the attributes of that game instance by using the dot operator. For example, if you would like to access the rows attribute of the game, simply write `my_game.rows`

3.1.2 Skeleton of the project.

You will have to implement the following functions for your project.

- `check_victory(game)`. This function's role is to check if a victory situation has been reached. It will take a game as input and will return:
  - 0 is no winning/draw situation is present for this game
  - 1 if player 1 wins
  - 2 if player 2 wins
  - 3 if it is a draw
- `apply_move(game,col,pop)`. This function's role is to apply a certain move to a game. It will take a game as input, as well a column value `col` and a boolean value `pop` that will represent the move. It returns a game with an updated board according to that move.
- `check_move(game,col,pop)`. This function's role is to check if a certain move is valid for a certain game. It will take a game as input, as well a column value `col` and a boolean value `pop` that will represent the move. It returns a boolean value False if the move is impossible, and it return True if the move is possible.
- `computer_move(game,level)`. This function's role is to ask for the computer to make a move for a certain game. It will take a game as input, as well a level that will represent the level of the computer opponent. It will return a column and a boolean value (for the pop) that represent the move chosen by the computer.
- `display_board(game)`. This function's role is to display the board in the console. It takes a game as input and does not return anything.

- `menu()`. This function's role is to handle the menu interface with the user via the console. It is basically the director function that will interact with the user and distribute the work to all functions. It should be the main function that is called in your Python script. It takes no input and doesn't output anything.

Note that these functions must be implemented to work with different board size, different number of consecutive same-color disks to win, etc.

3.2 2nd step: implementing the rules of the Connect 4 game

Once the skeleton ready, you can start writing the internals of the functions that will implement the game.

Display. The display should be handled in a separate function `display_board`, that can be called every time a new move was made by a player. A simple way to implement that function is to simply print the board matrix (optionally, you can create a more advanced display function, using a graphical display, or using the SenseHat module of a Raspberry Pi).

Making a move. Note that a move from a player can be described by a column index (in order to specify which column will be played) and a boolean value (in order to specify if the player would like to insert a disk or pop out a disk). Your menu should error-check that the user did not try to enter a column beyond board limits. Once the move chosen, your program must check if the move is valid and apply it only if it is indeed a valid move. Then, if the move is valid, it must check if a victory situation is reached. This entire sequence repeats until the game is over, or the user would like to quit the game.

Checking victory. In order to check if a victory situation is reached, you must check if N consecutive discs of the same color are present in the board. Be careful, it might happen a situation where more than N consecutive discs of the same color are present, which obviously also leads to a victory of this player. Besides, if a pop out move was done, it might happen that both players get N consecutive discs of the same color aligned. In that case, the player who did the move will win. Finally, if the board is full, the game is a draw (even though a pop out move could be done in theory).

Once this entire second step is fully implemented, you should be able to play human versus human with your program. Do not start the third step before this second step is fully functional (test several games, try unusual situations to make sure there is no bug in your program).

3.3 3rd step: implementing the computer player

The last step of this project is to implement a computer player with two levels of quality.

Level 1: random computer player. To start, program a trivial strategy: each time he will have to play, the computer player will randomly choose a move among all the possible random moves (column and pop out choices). You can test that this player is easy to beat.

Level 2: medium computer player. Program a computer player that will necessarily play a move that leads to a direct win for him if such a move exists. If no such move exist, it will avoid to play a move that leads to a direct win for his adversary in the next round (if such a move exist for the adversary). If again no such move exist, the computer player will simply pick a random valid move.

Level 3 (optional): hard computer player. You can help the computer player to take smarter decisions, whenever there is not an obvious move to do (like a direct win or avoiding direct loss). For that, instead of letting him choose a random move, you can assign a grade to each possible move, for example by giving points depending on the disks aligned (100 points for each $N-1$ consecutive same color disks, 10 points for each $N-2$ consecutive same color disks, etc.). You can use negative values for the opponents disks. Eventually, make the move that maximizes the grade. This will make sure your computer player will try to get his disks aligned as much as possible, and avoid having the opponents disks aligned.

Level 4 (optional): ultra hard computer player. You can implement a Min-Max player, see Section below.

4 If you want to go further ...

This section proposes some possible extensions for your project. Note that these extensions are here if you would like to get extra challenges for your project, but they will not participate to the final grade.

4.1 Min-Max computer player

A much better computer player can be implemented by using the so-called *min-max algorithm*. The goal of the min-max algorithm is to **minimize the possible loss for a worst case scenario**, or in other words it will look for the move which leaves the opponent capable of doing the least damage.

It is an algorithm, usually implemented with a **recursive function**, that will basically go through all possible moves of the two players (like a search tree), up to a certain maximal number of moves (called the maximal depth d_{max}). More precisely, in order to decide what move to make, the computer considers **all** of its possible moves (depth $d = 1$), then for all these moves he will considers **all** of the possible moves from the opponent (depth $d = 2$), and for all these moves it will consider **all** of its possible moves (depth $d = 3$), etc ...up to a certain maximal depth d_{max} (no move will be searched at a depth $d > d_{max}$). This search is like a tree, as depicted in Figure 2: the root node of the tree is the current state of the game, and each branch from this node represents a potential move from the player, creating new nodes (called subnodes, aka new game states). New branches are then starting from these nodes to represent the possible moves of the opponent, and so on and so forth.

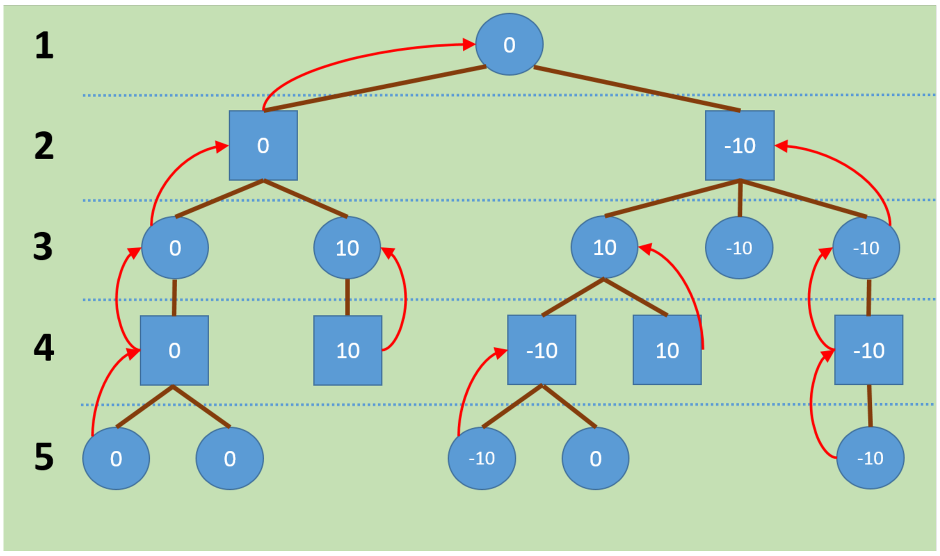

Figure 2: Example of a min-max search tree. The black numbers on the left represent the current depth in the search tree (depth 1, 3 and 5 are MAX steps, while depth 2 and 4 are MIN steps). The number in the node represents the value of that node, and the red arrows represent which value was passed to higher nodes, thanks to the min-max algorithm.

Now, everytime a move has been considered at depth d, the computer will evaluate the quality of the current game: if he wins (there are N consecutive discs of his color), then he gives 10 points to the value of this move. In contrary, if he looses (the adversary has N consecutive disks of the same color), then he gives -10 points to the the value of this move. Of course, once a victory situation has been reached at a depth d, there is no need to continue considering further moves at depth $d + 1$ since the game is over. If no victory situation is attained, then the algorithm continues to search at depth $d + 1$, or gives the value 0 to the current move if depth d_{max} is already reached.

At depth 1, the value of a node is the **maximal** value of his subnodes ($\max\{0, -10\} = 0$). At depth 2, the value of a node is the **minimal** value of his subnodes ($\min\{0, 10\} = 0$ and $\min\{10, 10, -10\} = -10$). At depth 3, the value of a node is again the **maximal** value of his subnodes ($\max\{0\} = 0$, $\max\{10\} = 10$, $\max\{-10, 10\} = 10$ and $\max\{-10\} = -10$), etc. Alternating the **max** and **min** phases, the computer eventually chooses the move that gave him the maximal value at depth 1 (in our example from Figure 2, the computer would choose the move on the left branch in depth 1, since it is the move that provides the maximal value 0).

You can find more information and even some pseudo-code for the min-max algorithm on Wikipedia: `http://en.wikipedia.org/wiki/Minimax` , and especially watch the animation that shows an execution example of the algorithm for a depth of 4 `http://en.wikipedia.org/wiki/File:Plminmax.gif`

The difficulty level of the computer player is directly correlated to the maximal depth d_{max}: the deeper you check the moves, the better will play the computer. A computer player with a depth of 4 should take a few seconds to play, and is generally good enough to beat a human player. Interestingly, the **Connect4** game has been solved (which means that the entire tree of all the possible moves can be searched) when the condition $r + c < 15$ is fulfilled. In that case, the computer will always win if he is the first to play.

4.2 Other features

Here are some potential extra features ideas that you can try to implement (again, this will not be graded):

- more players ! Patch your program so one can play with 3 or 4 players.
- add a coin toss in the beginning of the game to decide who will start
- add a timer to force slow users to play fast (if they don't respect the timer, then play a random column)
- allow the user to save/load a game
- allow the human player to go back one step (or more), so that he can continue his game even if he made an obvious error
- give hints to the human player (by first running a "transparent" computer player and checking what would have been the choice of the computer)
- a better artificial intelligence: implement a better computer player, for example by computing a heuristic function, as explained for example in `http://www.roadtolarissa.com/connect-4-ai-how-it-works/`. A simple and efficient way to do it, is to give 1 point for each pattern of $N - 1$ consecutive discs of the players color (and of course remove 1 point for such patterns of the opponent color).
- improve the graphical display of the board: use textures, backgrounds, show where is the winning discs line in case of a victory, add an animation of the discs falling inside the board when a player played, etc.
- user interface: let the user choose the option of the game in a menu (i.e. size of the board, number of players, human/computer players, level of the computer players, discs colors, etc.).