Project

Deadline: 13.11.2017 23:59 SGT

The project is an assignment that has to be completed in groups of 5/6 persons. The groups were chosen randomly and the list was sent to you by email. 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 a **QUIXO** board game, with a simple user interface and a computer player.

Important!!! Make sure your project works properly (i.e. the program doesn't output errors, even if the user inputs unexpected values). The script or function to execute the program must be in your project repertory and named `quixo.py`. For submission of your project on iNTU, please create a zip archive from your project repertory, name the archive according to your group number (i.e. <YourGroupNo>.zip), and upload it on iNTU. Not following these guidelines will lead to potential penalties on your project grade. The submission system closes at the deadline. Hence, after that deadline, if you didn't submit you will get 0 mark for your project.

The Quixo game

The **QUIXO** game is a simple board game that is played with 2 players (but it can handle up to 4 players). The board is composed of a $5 * 5$ square of small cubes. These cubes have a circle symbol on one side, a cross symbol on another, and blank faces on the other four, see Figure 1. We start a game by placing the 25 cubes with blanks face-up.

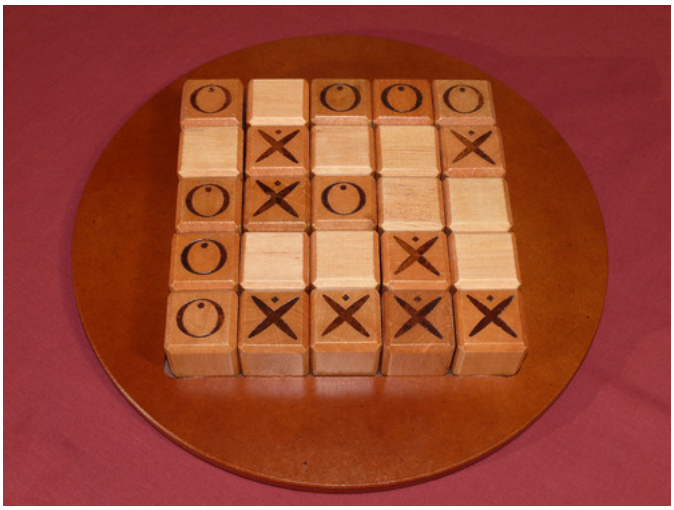

Figure 1: Example board of a **QUIXO** game (image taken from www.boardgamegeek.com
.

One player uses cross symbols, while the other uses circle symbols, and the players goal is to be the first to form a line of their own symbol (similar to Tic-Tac-Toe). Each turn, the active player takes a cube that is blank or bearing his own symbol from the **outer ring** of the grid (see left figure of Figure 2), rotates it so that it shows his symbol (in case it was blank), then adds it back to the grid by pushing it into one of the rows from which it was removed (note that you are not allowed to place the cube back in the position from which it was taken originally, see right figure of Figure 2). Thus, a few pieces of the grid change places each turn, and the cubes slowly go from blank to crosses and circles. Play continues until someone forms an horizontal, vertical or diagonal line of five cubes bearing his symbol, with this person winning the game. Note that there is no draw. Moreover if both players form a line at the very same time, the player who played that move looses the game.

You can for example view a video of the game rules here: `https://www.youtube.com/watch?v=cZT5N6hIFYM`

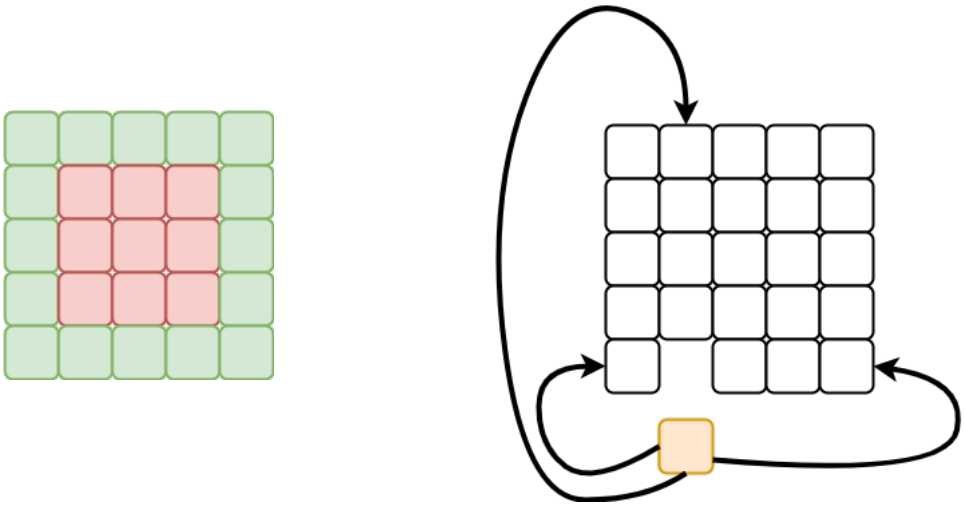

Figure 2: Rules of the **QUIXO** game: the left-hand side figure represents in green the cubes that can be played (outer ring), the right-hand side figure represents the pushes that can be played (it is forbidden to put back the cube to its original place).

Objectives

The objective of the project is to write in `Python` a **working QUIXO** program. Be VERY careful to implement EXACTLY the game rules (test your program thoroughly and try to think of all the special cases). 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 being $r = 5$ and $c = 5$). The board must always be square, thus $r = c$. One always have that the number N of consecutive same-symbol cubes required for a player to win is equal to r and c.
- the type of the two players (human or computer), and the difficulty level in case of a computer player.

You are free to program this **QUIXO** game with the method you prefer. However, in order to ease the implementation of this project, we strongly advise you to follow the three steps given below. Moreover, note that every time 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 !

Three steps to complete the project

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

The first step is perhaps the most important one: before writing any `Python` code, you should think about the functions you will need to implement, their input(s)/output(s), 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 menu will be organised, etc.), the display of the board, and the initialization of the game (the rest should be functions stubs).

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 a cube with blank face up, 1 stands for a cube with cross symbol face up, 2 stands for a cube with circle symbol face up, etc.

Interface with the user and display of the board. A simple menu interface can be used to navigate through the different functionalities of the program, and the keyboard can be used by the user to type in which moves he would like to make.

The game display should be handled in a separate function, that can be called every time a new move was made by a player. A simple option can be to simply print the board matrix. Yet, to make it easier for a human player to see the game, you can also use a more graphical display.

2nd step: implementing the rules of the Quixo game

Once the skeleton ready (and tested !), you can start writing the functions that will implement the rules of the game. Namely, you should have a function that applies a move to the current game (whether the move originates from a human player or a computer player) and a function that checks if a victory state is reached (and of course returns the identity of the winner).

Making a move. Note that a move from a player can be described by a cube index, and a row/column index for the pushing phase. Indeed, when a player has to play, he first selects a cube to play (the cube must be in the outer ring of the board, it must be blank or of the player own symbol, so some error-check must be conducted), then he selects from which of the 3 possible directions he would like to insert the cube back in the board and push.

Checking victory. In order to check if a victory situation is reached, you must check if N consecutive cube of the same symbols are present in the board. Be careful, depending on the game parameters, it might happen a situation where more than N consecutive cubes of the same symbol are present, which obviously also leads to a victory of this player.

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).

3rd step: implementing the artificial intelligence

The last step of this project is to implement a computer player.

Random computer player. To start, you can program a trivial strategy: for each move, the computer player will randomly choose which cube to play (excluding the impossible ones of course). You can test that this player is easy to beat.

Simple computer player. A simple computer player can be created by checking at each of its turn if any of the possible moves he can make would lead to a direct victory. If that is the case, you would of course play that move. Also, he would avoid the moves that lead to a direct victory of the opponent. If none of these cases occur, he can just play a random move.

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 3: 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.

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. You can also give some points if for example the game is at your advantage (say there are more cubes with your symbols, or you manage to already place a few cubes with your symbols in a row). 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 ($\max\{0\} = 0$, $\max\{10\} = 10$, $\max\{-10, 10\} = 10$

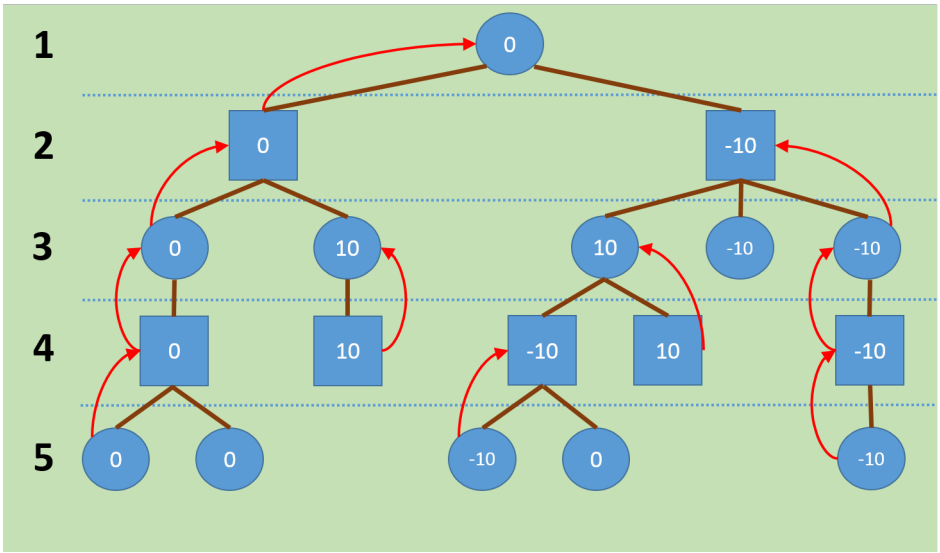

Figure 3: Example of a min-max search tree. The black numbers on the left represent the current depth in the search tree (depth 1, 3 and 5 are MAX steps, while depth 2 and 4 are MIN steps). The number in the node represents the value of that node, and the red arrows represent which value was passed to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

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 3, 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 3 should take a few seconds to play (on a $5 * 5$ board), and is generally good enough to beat a human player.

Grading of the project

We will grade the functionalities of the program, as well as the quality of the code produced (program style guidelines from the lectures, comments in the code, etc.). Note that if we discover that you used a part of the code from an existing **QUIXO** implementation, you will get 0 mark for your project (see also the NTU Academic Integrity Policy `http://www.ntu.edu.sg/ai/ForEveryone/Pages/NTUAcademicIntegrityPolicy.aspx`). Be careful, automatic tools are very efficient to find such cheating !