Project

Deadline: 15.11.2019 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 **PENTAGO** in Python, 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 `pentago.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. Note that every year several groups get 0 mark (or worse: academic misconduct) because of plagiarism.
- 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 `pentago.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 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 PENTAGO game

PENTAGO is a two-player game invented by Tomas Flodén that won several awards when first commercialised (Wikipedia link: `https://en.wikipedia.org/wiki/Pentago`).

The game is played on a 6×6 board divided into four 3×3 sub-boards (or quadrants), see Figure 2. Taking turns, the two players place a marble of their color onto an unoccupied space on the board, and then rotate one of the sub-boards by 90 degrees either clockwise or anti-clockwise. A player wins by getting five of their marbles in a vertical, horizontal or diagonal row (either before or after the sub-board rotation in their move). If all 36 spaces on the board are occupied without a row of five being formed then the game is a draw. If a situation arises where both players have five of their marbles in a vertical, horizontal or diagonal row at the same time (whatever the number of 5-in-a-row lines they separately have), then the game is also a draw.

There is also a 3 or 4 player version called **PENTAGO** XL, where the board is instead a 9×9 board made of nine 3×3 sub-boards.

Interestingly, the 6×6 version of **PENTAGO** has been completely been solved with the help of a huge supercomputer. Using the 4 TeraBytes (!) of data produced by this huge computation, a player can be guaranteed to win provided that he starts the game. Yet, without using such enormous precomputed data, the game is quite difficult because of its important branching (i.e. how many moves can be done at each turn, in the worst case $36 \times 8 = 288$).

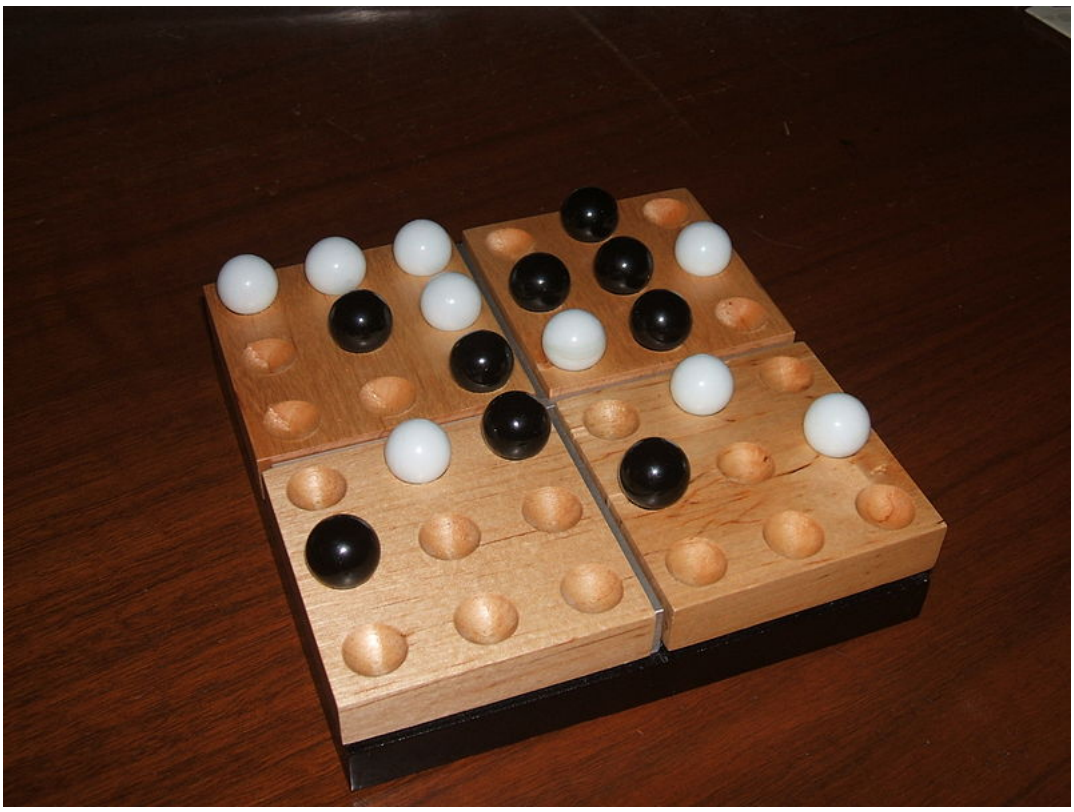

Figure 1: Example of a 6×6 **PENTAGO** board (image taken from Wikipedia). White wins with 5 consecutive marbles in a diagonal.

2 Objectives

The objective of the project is to write in Python a **working PENTAGO** program. The program must allow the user to configure 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 **PENTAGO** 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 position/rotation 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 6×6 matrix of integers, where 0 represents no marble (i.e. empty), 1 stands for a marble belonging to player 1, and 2 for a marble belonging to player 2 (row 0 being the top 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 top row.

3.1.2 Skeleton of the project.

You will have to implement the following functions for your project. Note that *board* will denote a 6×6 matrix of integers representing the board, *turn* will denote an integer representing who's turn it is to play (thus equal to either 1 or 2),

- `check_victory(board, turn, rot)`. This function's role is to check if a victory situation has been reached. It will take a matrix *board* and an integer *turn* as inputs, as well as the rotation integer *rot* that has just been played (we consider this rotation action has already been applied to *board*) and will return:
  - 0 if 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(board, turn, row, col, rot)`. This function's role is to apply a certain move to a game. It will take a matrix *board*, an integer *turn* as inputs, as well as three integers *row*, *col* and *rot* that will represent the move made by the player. It returns an updated board according to that move.
- `check_move(board, row, col)`. This function's role is to check if a certain move is valid for a certain board. It will take a matrix *board* as well as two integers *row* and *col* (that will represent the position played) as inputs. It returns a boolean value False if the move is impossible, and it return True if the move is possible.
- `computer_move(board, turn, level)`. This function's role is to ask for the computer to make a move for a certain board. It will take a matrix *board* and an integer *turn* as inputs, as well as an integer *level* that will represent the level of the computer opponent. It will return three integers: a row, a column and a rotation value, representing the move played by the computer.
- `display_board(board)`. This function's role is to display the board in the console. It takes a matrix *board* 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.

3.2 2nd step: implementing the rules of the PENTAGO 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 row index (integer between 0 and 5), a column index (integer between 0 and 5) and a rotation index (integer between 1 and 8, see Figure 2). Your menu should error-check that the user did not try to enter a row/column beyond board limits, or non-existing rotation index. 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.

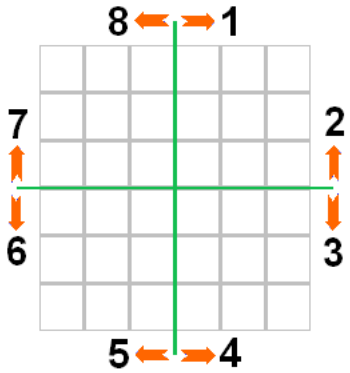

Figure 2: Correspondence between the rotation ID (between 1 and 8) and the actual board rotations.

Checking victory. In order to check if a victory situation is reached, you must check if 5 marbles (or more) of the same color are present in one of the rows, columns or diagonals of the board. Be careful, victory situation must be checked before the rotation was applied, and then after the rotation was applied. It might happen that both players get at least 5 marbles of their color in different rows, columns or diagonals. In that case, the game is a draw. Finally, if the board is full and no win situation occurred, the game is also a draw.

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 (row, column and rotation 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 number of marbles in the same row, column or diagonal (say 100 points for each 4 marble in the same row, column or diagonal, 10 points for each 3 marbles in the same row, column or diagonal, etc.). You

can use negative values for the opponents marbles. Eventually, make the move that maximizes the grade. This will make sure your computer player will try to get his marbles aligned as much as possible, and avoid having the opponents marbles 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 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.

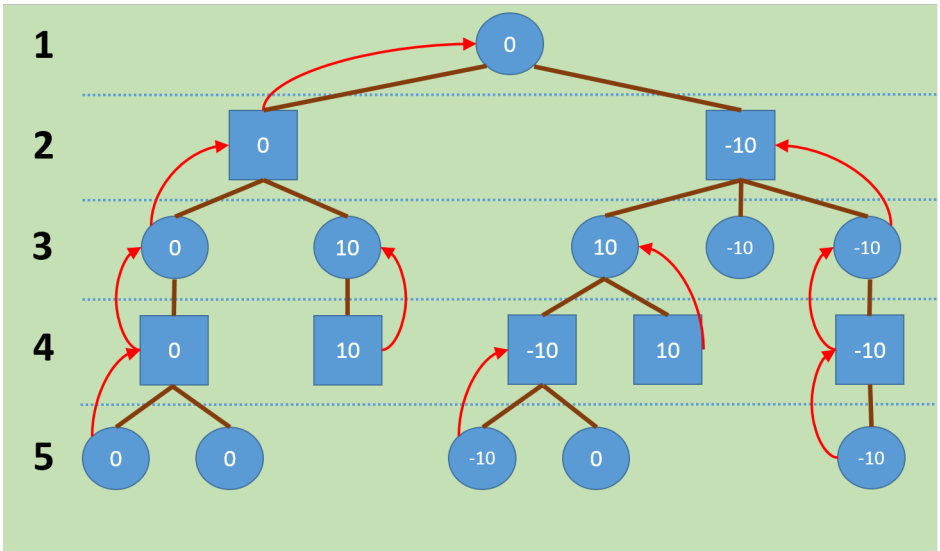

Figure 3: Example of a min-max search tree. The black numbers on the left represent the current depth in the search tree (depth 1, 3 and 5 are MAX steps, while depth 2 and 4 are MIN steps). The number in the node represents the value of that node, and the red arrows represent which value was passed to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

Now, everytime a move has been considered at depth d, the computer will evaluate the quality of the current game: if he wins (5 of its own marbles in the same row, column or diagonal), then he gives 10 points to the value of this move. In contrary, if he looses (5 of the opponent's marbles in the same row, column or diagonal), 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 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, and is generally good enough to beat a lazy human player. Interestingly, the **PENTAGO** 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)
- improve the graphical display of the board: use textures, backgrounds, show where are the winning marbles in case of a victory, etc.
- user interface: let the user choose the option of the game in a menu (i.e. number of players, human/computer players, level of the computer players, marbles colors, etc.).