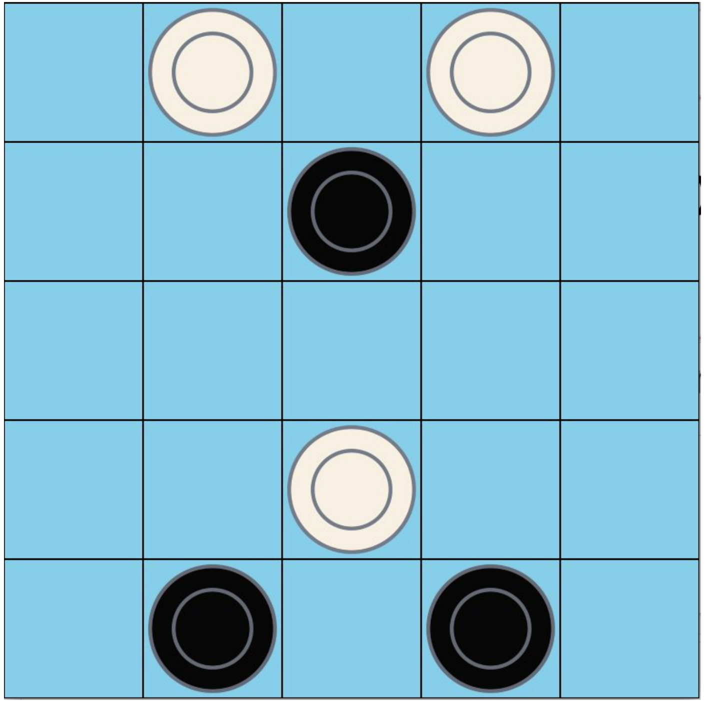
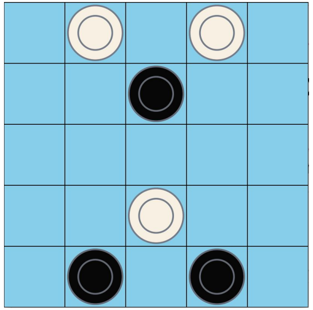
Neutreeko

Turma 3 - Grupo 22

Catarina Fernandes - up201806610
Diogo Almeida - up201806630
Pedro Queirós - up201806329



Game Description



The name is a blend of Neutron and Teeko, two games on which it is based.

Initial setup: Neutreeko is a simple game played on a board with 5×5 squares. The players have three pieces each, as shown in the figure.

Rules: A piece slides orthogonally or diagonally until stopped by an occupied square or the border of the board. Black always moves first. A match is declared a draw if the same position occurs three times.

Goal: To get three in a row, orthogonally or diagonally. The row must be connected.

Strategy: This is a highly tactical game with little room for long-term strategic planning. But if neither side has an attack coming up, it can often be a good idea to immobilize your opponent by forcing him into a corner.

Related work

- https://www.neutreeko.net/neutreeko.htm
- https://github.com/Abi1024/Neutreeko
- https://github.com/awkwardbunny/Neutreeko
- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/
- https://moodle.up.pt/pluginfile.php/185349/mod_resource/content/0/IART_Lecture2c_SearchGames.pdf
- https://moodle.up.pt/pluginfile.php/197382/mod_resource/content/0/Exercises_AI3_AdversarialSearch_SolutionTopics.pdf

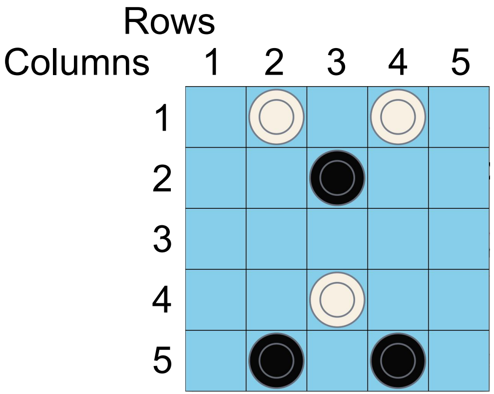
Formulation of the problem as a search problem

State Representation:

The Board will be represented by a matrix 5x5 (B[5, 5], or in the general case B[Rows,Columns]) with the following values: 0 for empty spaces; 1 for the player with black pieces; 2 for the player with white pieces.

Additionally, there will be a state (rowLast, columnLast) representing the last move made so that it will be easier to verify if there is a winner. There will also be a state to store the different positions across the game play to verify if a position occurs three times, representing a draw.

Finally, we will represent the player to move (Player).



Initial State:

B[5, 5] = {0} except B[1,2] = 2, B[1,4] = 2, B[2,3] = 1, B[4,3] = 2, B[5,2] = 1, B[5,4] = 1;
Player = 1;

Objective Test:

```
//returns 0- draw, 1-Win for player 1, 2-Win for player 2, -1 – game not finished
int objectiveTest(Board board, int player, int rowLast, int columnLast) {
        //Verify every direction from the last move (rowLast,columnLast)
}
```

Operators:

<table>
<tr><th>Operator Name</th><th>Preconditions</th><th>Effects</th></tr>
<tr><td>movePieceUp(Piece piece)</td><td>piece.row > 1;
B[piece.row-1, piece.column] = 0</td><td>State movePieceUp(Board B, Player player, int row, int column) {
i = row; while(B[i,column]!=0) i--; B[i,column] = player;player = 3-player;lastRow = i; lastColumn = column; return B|player|lastColumn|lastRow;
}</td></tr>
<tr><td>movePieceDown(Piece piece)</td><td>piece.row < 5;
B[piece.row+1, piece.column] = 0</td><td>State movePieceDown(Board B, Player player, int row, int column) {
i = row; while(B[i,column]!=0) i++; B[i,column] = player;player = 3-player;lastRow = i; lastColumn = column; return B|player|lastColumn|lastRow;
}</td></tr>
<tr><td>movePieceLeft(Piece piece)</td><td>piece.column > 1;
B[piece.row, piece.column-1] = 0</td><td>State movePieceDown(Board B, Player player, int row, int column) {
i =column; while(B[row,i]!=0) i--; B[row,i] = player;player = 3-player;lastRow = row; lastColumn =i; return B|player|lastColumn|lastRow;
}</td></tr>
<tr><td>movePieceRight(Piece piece)</td><td>piece.column < 5;
B[piece.row, piece.column+1] = 0</td><td>State movePieceDown(Board B, Player player, int row, int column) {
i =column; while(B[row,i]!=0) i++; B[row,i] = player;player = 3-player;lastRow = row; lastColumn =i; return B|player|lastColumn|lastRow;
}</td></tr>
</table>

<table>
<tr><th>Operator Name</th><th>Preconditions</th><th>Effects</th></tr>
<tr><td>movePieceRight(Piece piece)</td><td>piece.column < 5;
B[piece.row, piece.column+1] = 0</td><td>State movePieceDown(Board B, Player player, int row, int column) {
i =column; while(B[row,i]!=0) i++; B[row,i] = player;player = 3-player;lastRow = row; lastColumn =i; return B|player|lastColumn|lastRow;
}</td></tr>
<tr><td>movePieceLeftUp(Piece piece)</td><td>piece.column > 1;
piece.row > 1;
B[piece.row-1, piece.column-1] = 0</td><td>State movePieceDown(Board B, Player player, int row, int column) {
i1 =row; i2 = column; while(B[i1,i2]!=0) {i1--;i2--;} B[i1,i2] = player;player = 3-player;lastRow = i1; lastColumn =i2; return B|player|lastColumn|lastRow;
}</td></tr>
<tr><td>movePieceRightUp(Piece piece)</td><td>piece.row > 1;
piece.column < 5;
B[piece.row-1, piece.column+1] = 0</td><td>State movePieceDown(Board B, Player player, int row, int column) {
i1 =row; i2 = column; while(B[i1,i2]!=0) {i1--;i2++;} B[i1,i2] = player;player = 3-player;lastRow = i1; lastColumn =i2; return B|player|lastColumn|lastRow;
}</td></tr>
<tr><td>movePieceLeftDown(Piece piece)</td><td>piece.row < 5;
piece.column > 1;
B[piece.row+1, piece.column-1] = 0</td><td>State movePieceDown(Board B, Player player, int row, int column) {
i1 =row; i2 = column; while(B[i1,i2]!=0) {i1++;i2--;} B[i1,i2] = player;player = 3-player;lastRow = i1; lastColumn =i2; return B|player|lastColumn|lastRow;
}</td></tr>
</table>

Heuristics/evaluation function:

Jogadas	Pontos
Make a move that wins the game.	1600
Make a move that prevents the opposing player from winning.	800
Make a move that enables a victory in the next turn.	400
Make a move that prevents the opposing player of enabling a victory in their next turn.	200
Make a move that results in a 2-in-a-row.	100
Make a move that prevents the opposing player from making a 2-in-a-row.	50

Implementation work already carried out

Programming language: The programming language we chose for the game was Python. This is a flexible language that we're comfortable with and contains libraries like pygame that facilitate the creation of a GUI for the game.

Development Environment: All of the group's members are using Visual Studio Code for the code's creation/edition and a Python 3.9.2 interpreter.

Data Structures: The data structures used so far are Classes, used to represent the Game, Board and its Pieces. The Board class also contains a list of lists that stores all the pieces' positions, representing, therefore, the board itself.

File Structure: The current file structure has a "classes" module which contains all the classes (Game, Board and Piece) .py files, there is also a constants.py file which contains several constants used throughout the rest of the code (interface's window size, color rgb codes, etc.).