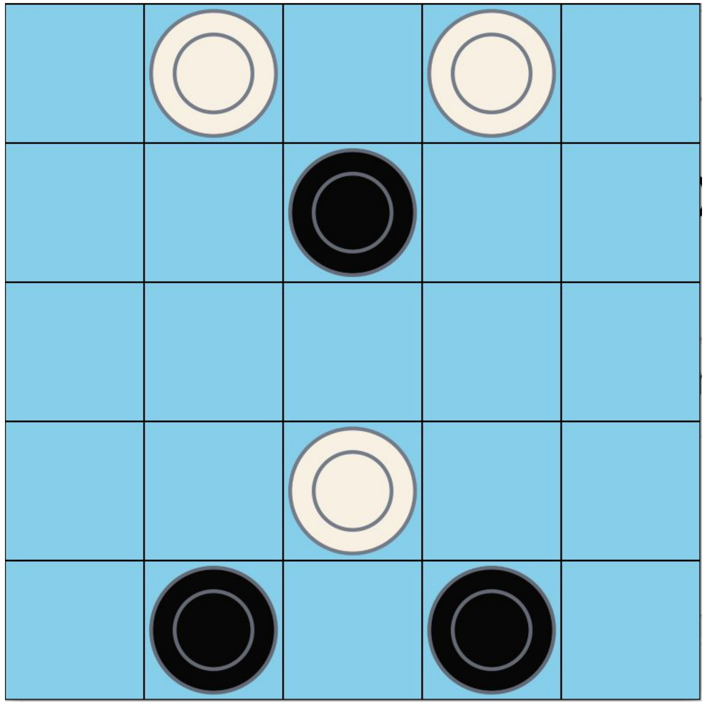
Neutreeko

Class 3 - Group 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.

Formulation of the problem as a search problem

State Representation:

The Board is 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 is a state (lastRow, lastColumn) representing the last move made so that it is easier to verify if there is a winner. There is also a state to store the different positions across the gameplay to verify if a position occurs three times, representing a draw.

Finally, it is represent the player to move (Player).

Initial State:

B[5, 5] = {0} except B[0,1] = 2, B[0,3] = 2, B[1,2] = 1, B[3,2] = 2, B[,1] = 1, B[4,3] = 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)
}
```

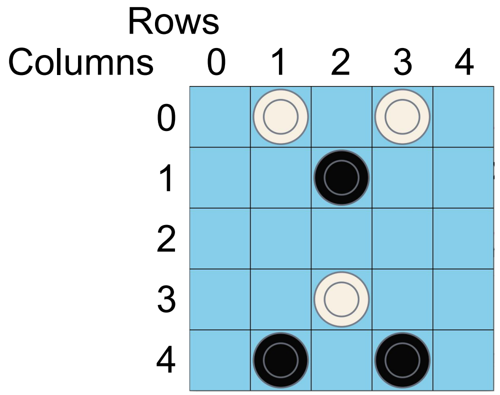


Figure 1 - Initial Board

The approach

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 are Classes, which represent the Game and Board. The Game class draws the board, controls the game flow and has the implementations of the chosen algorithms. The Board class contains a list of lists that stores the board state, representing, therefore, the board itself and, besides that, contains the board evaluation functions.

File Structure: The current file structure has a "classes" module which contains all the classes (Game, Board and Button) .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.).

The approach - heuristics, evaluation functions

We have 3 levels of difficulty, each level uses a different heuristic achieved with a combination of evaluation functions.

Each level has a maximum depth:

- Easy - maximum depth of 1
- Medium - maximum depth of 3
- Hard - maximum depth of 5 (minimax) / 6 (minimax with alpha-beta cuts)

Board	Points
3 player pieces in a row (wins the game)	1000
Other	0

Table 1 - Heuristic for Easy Level

Board	Points
3 player pieces in a row (wins the game)	1000
2 player pieces in a row	200
Other	0

Table 2 - Heuristic for Medium Level

Board	Points
3 player pieces in a row (wins the game)	1000
2 player pieces in a row	200
2 player pieces close with an empty space in the middle	100
Other	0

Table 3 - Heuristic for Hard Level

The approach - operators

For the operators, two methods were implemented in the *Board* class: *get_valid_moves* e *move_piece*. The first one calculates the valid positions that a given piece can go to. The second one moves the chosen piece to the given position. This position is one of the positions calculated by the first function.

The *get_valid_moves* function, to calculate the valid positions of one piece, takes the row and column of that piece as parameters. To get the valid positions, the method verifies every direction that the piece can go to and the respective last empty square.

The *move_piece* function receives the row and column of the piece to move and the row and the column of the square where the piece will be moved to as parameters. The new position will be occupied by that piece and the old position will be empty.

Algorithms implemented

- **Minimax**
  - Implemented in game.py
  - Customizable depth level and node ordering system
  -
- **Minimax with alpha-beta pruning**
  - Implemented in game.py
  - Customizable depth level and node ordering system
  - Optimization of the previous algorithm, cuts branches in the game tree which need not be searched because there already exists a better move available.

Experimental results

Several trials were carried out in order to study the efficiency of the implemented algorithms by evaluating execution time and the number of nodes visited. The influence of ordering in these algorithms was also studied: best ordering (descending order of the heuristic score), worst ordering (descending order of the heuristic score) and without any specific ordering. Other statistics can be found in the attachments document.

Level \ Algorithm	Minimax	Minimax With Alpha-Beta Cuts
Easy	0,00016	0,00013
Medium	0,01725	0,00632
Hard	3,28174	2,48941

Table 4 - Execution time of each algorithm in each level without ordering

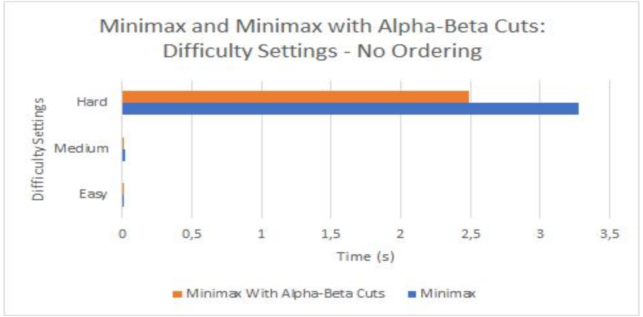


Figure 2 - Graphic representation of the execution time of each algorithm in each level without ordering

Level \ Algorithm	Minimax	Minimax With Alpha-Beta Cuts
Easy	14	14
Medium	2605	776
Hard	422795	290870

Table 5 - Nodes visited in each algorithm in each level without ordering

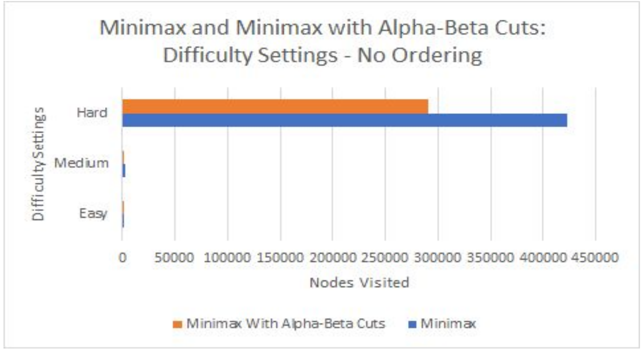


Figure 3 - Graphic representation of the nodes visited in each algorithm in each level without ordering

Experimental results

Ordering / Algorithm	Best	Worst	None
Minimax	5,12755	5,15526	3,27071
Minimax With Alpha-Beta Cuts	3,92950	3,6901	2,36907

Table 5 - Execution time of each algorithm with each ordering in hard level

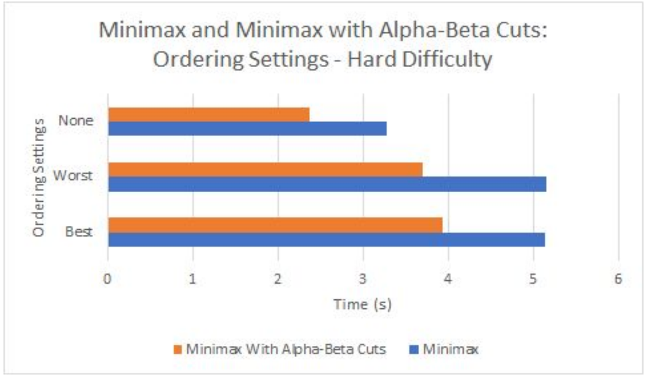


Figure 4 - Graphic representation of the execution time of each algorithm with each ordering in hard level

Ordering / Algorithm	Best	Worst	None
Minimax	422795	422795	422795
Minimax With Alpha-Beta Cuts	258369	253225	266444

Table 6 - Nodes visited in each algorithm in each ordering in hard level

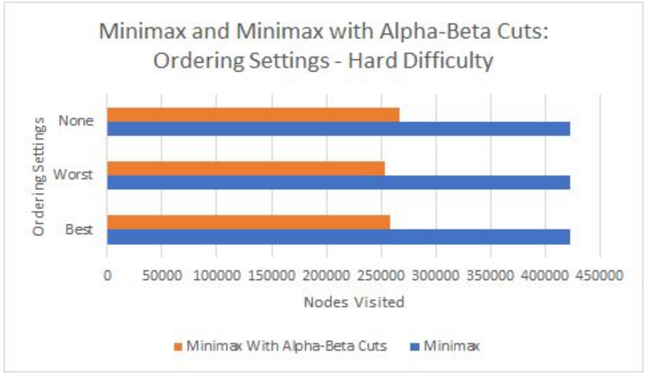


Figure 5 - Graphic representation of the nodes visited in each algorithm in each ordering in hard level

Conclusions

By modelling the Neutreeko game as a search problem we were able to develop an AI that is able to win in most situations, depending on the chosen difficulty.

The optimal solution to an adversarial search problem can be approximately found by performing a depth-limited Minimax adversarial search. It is by varying this depth that we get the different difficulty options.

Although the choice of heuristics is important, it doesn’t affect the quality of the solutions as much as search depth does (in most cases, simply verifying wins grants optimal solutions).

Pruning the search tree with alpha-beta cuts to remove nonviable branches results in significant performance gains, especially in higher depth searches (harder difficulties).

Finally, ordering the different tree nodes by heuristics’ score resulted in a significant performance hit, with no improvement to the quality of the solutions obtained.

References

- Neutreeko page: https://www.neutreeko.net/neutreeko.htm
- Neutreeko version made in Python by Abi1024: https://github.com/Abi1024/Neutreeko
- Neutreeko version made in Java by Gadon: https://gitlab.com/g-dv/neutreeko/-/tree/master/java/src
- Minimax Algorithm in Game Theory: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/
- Lecture slides on Search Games: https://moodle.up.pt/pluginfile.php/185349/mod_resource/content/0/IART_Lecture2c_SearchGames.pdf
- Class exercises about Adversarial Search: https://moodle.up.pt/pluginfile.php/197382/mod_resource/content/0/Exercises_AI3_AdversarialSearch_SolutionTopics.pdf
- Pygame documentation: https://www.pygame.org/docs/
- Pygame-menu documentation: https://pygame-menu.readthedocs.io/en/4.0.1/