Report: Computational Intelligence Project

Lorenzo Fezza s312689

January 31, 2024

1 Introduction

This report contains a detailed explanation of the executed work carried out for the implementation of the Computational Intelligence Project 2023/2024 (PoliTo).

In this project there is the implementation of the Quixo game and a possible player strategy used to achieve the best performances during the game. The game is played on a 5x5 board, where the two players fight for the victory. Each player is represented by a different symbol, for example one player can use Xs and the other Os, respectively represented on the board with 0s and 1s. Neutral cells, instead, are represented with -1s. The players take turns to move and the winner is the first player that has 5 elements aligned, either horizontally, vertically or diagonally. If a player places 5 aligned elements of his symbol and contemporary of the other player, he loses, making the other player win.

Note : In this report you can find personal reflections, changes and variations applied in order to better see the process to build the final code on the repository.

2 Implementation

For the implementation of the optimal player, the method used is the Adversarial Search, where the player focuses on finding the optimal moves by going through a search tree used to evaluate possible situations. In particular, the algorithm used in this project is the MinMax algorithm, where the player tries to maximize its score and contemporary minimize the opponent's score.

The algorithm is used in the `MinMaxPlayer` class, which makes a move by calling the implemented `make_move()` method, which calls the `minimax()` method in order to find the best move. The `minimax()` method is a recursive method that evaluates the best move for the current player, by evaluating the best move for the opponent. This evaluation is performed using the `evaluate()` method that (in the first implementations) simply attributed 1 in case of `MinMaxPlayer` victory, -inf in case of lost, and 0 in all the other cases.

2.1 Problems

At the beginning, a basic minmax version was implemented, in order to test and evaluate the algorithm performance. Unfortunately, the basic version was not able to find the best move in a reasonable time, due to several reasons:

- The decisional tree is too big to be explored, considering that each move takes the board in a different configuration with a variable (often large) number of different possible moves to perform, so the tree grows almost exponentially in terms of depth and breadth.
- The basic algorithm is not able to prune the tree, so it explores all the possible paths, even if they are not useful.
- The choice of a big fixed depth can be costly because the action space is still very big to be explored. It is not necessary to explore all the possible moves when the board is almost empty. In the other case, instead, when the player is losing, it is necessary to explore more in depth in order to find the best move to defeat the opponent and then to win.

So the problems are found in the excessive depth and breadth of the tree.

2.2 Possible solutions

Some possible approaches to mitigate these problems can be adopted :

Alpha-Beta pruning :

The first one is to use the so-called `alpha beta pruning`, a method to discard some paths of the tree that are not useful to find the best move, in order to decrease the computational cost to explore all the nodes.

It works as follows: two values, `alpha` and `beta`, respectively represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of. Whenever the maximum score of the minimizing player (`beta`) becomes less than the minimum score of the maximizing player (`alpha`), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.

Deep Pruning :

Another solution can be to sort the possible moves based on how likely they are going to be good. It's better to explore these moves first because they will more likely improve the alpha beta pruning optimization, cutting all the useless paths. The sort is performed directly in the `get_possible_moves()` method where the moves are sorted according to the maximum number of player's symbols aligned using the `max_inline_pieces()` function.

Variable depth (when to vary) [Part 1.] :

Another possible solution can be the use of a different depth values. The depth of the generated tree can vary in some particular conditions. For example if the `MinMaxPlayer` is losing, it is necessary to explore more moves, so the depth can be increased. In other cases, if the `MinMaxPlayer` is winning, it is not necessary to predict so many steps, so the depth can be decreased. At the beginning, it's not necessary to go more in-depth because the board is empty, and each move can give the same results, so the `minmax()`

method is not used. The `is_losing()` function allows to understand if a player is losing and to quantify how much is losing by an input parameter.

The following piece of code describes how the function makes the evaluation:

```
def is_losing(player_id, len, game: 'Game') -> bool:
    # if the adversary has a line of len+1 pieces, I am losing
    for i in range(game._board.shape[0]):
        if sum(game._board[i, :] == (player_id+1)%2) >= len:
            return True
        if sum(game._board[:, i] == (player_id+1)%2) >= len:
            return True
    if sum(game._board.diagonal() == (player_id+1)%2) >= len:
        return True
    if sum(game._board[::-1].diagonal() == (player_id+1)%2) >=
       len:
        return True
```

This function instead in not used in the final version of the project because other different criteria made it possible to evaluate the state of the game more quickly and accurately.

The termination of the minmax recursion is determined by the following condition:

```
winner = game.check_winner()
if depth == 0 or winner != -1 or is_terminal(node):
```

where the `is_terminal()` function allows you to evaluate whether there are still possible moves.

If the minmax recursion terminates, the state is evaluated by the following method:

```
def evaluate(winner, pid, game : 'Game') -> float:
    if winner == pid%2:
        return float('inf')
    elif winner == (pid+1)%2:
        return float('-inf')
    else:
        max_p1 = max_inline_pieces(pid, game)
        max_p2 = max_inline_pieces((pid+1)%2, game)
        return max_p1 - max_p2
```

Note: At the beginning, it's better to fill the neutral cells with the player's symbol because it is more likely to have more possible moves in the future. More choices mean more possibilities to win or defeat.

2.3 Consequent Problem encountered :

In the first implementations, if two `MinMaxPlayer` players compete, they can loop forever or lose. This problem is due to the fact that when the board is in a symmetric state, the `get_possible_moves()` method returns the same moves sorted in the same way, and the alpha-beta minmax implemented with always the same varying depths criteria, selects the first optimal move, which makes a loop.

2.4 Possible solution :

A possible solution can be the introduction of some randomness in the sorted list returned by the `get_possible_moves()` method, shuffling the elements with the same piece symbol (neutral and not neutral) and the same number of elements in line, in order to avoid to select always the same moves first in the alpha-beta pruning minmax.

2.5 Other improvements (how much vary) [Part 2.] :

The depth can be changed in different ways, providing always different results. At the beginning, the choice is irrelevant; a trick can be to begin from angles or borders because they can be used to block adversary's moves or to protect your winning moves so the minmax method is not called and the depth is put to 1 for the next moves.

Many attempts have been done in order to provide the best solution in the next turns, trying to increase the depth to 3 if the player is winning or losing (if it's winning, it can win in 3 turns or avoid to lose in 3 turns). But the fixed choice of depth=3 is not sufficient.

The optimal variation of the depth, but also the slowest one (reaching a maximum depth of 4, because going more in depth takes too much time) is to choose the depth to be proportional to the `max_inline_pieces()` method when one of the two players is winning or losing. When the agent is winning or losing (evaluated using the `is_losing()` method), the depth can be increased or decreased for the following reasons:

- If the opponent is tricking the `MinMaxPlayer` making it gain more scores (so faking to lose), it will be able to find recognize the trick finding the moves that can make the opponent win.
- If the opponent is winning, more depth is needed to find the best move to defeat and then to win.

Then, if there is a move that makes the player win, it can be performed without going through the decision tree.

Another possible approach, which can be faster but not always optimal is based on the concept of advantage. Let's look the following code :

```
1 max_p1 = max_inline_pieces(pid, game)
2 max_p2 = max_inline_pieces((pid+1)%2, game)
3 depth = min(abs(max_p1 - max_p2) + 1, max_p2)
```

where:

- `max_p1` and `max_p2` : represent the maximum number of aligned elements for each player.
- `abs(max_p1 - max_p2)` : represents the advantage of a player over the other one

If the opponent is not aligning pieces on the board (losing the game), the advantage will increase, increasing the depth value, but it is useless, because the opponent is losing, and it's not necessary to search more in depth, so `max_p2` will be lower and it can be chosen as depth value.

If the opponent is aligning a lot of pieces on the board (winning the game), `max_p2` will be more than the `advantage + 1`, but going that deep (for example a value of `max_p2`

= 3 or 4) is computationally expensive, so it's preferred to choose the depth using the advantage value.

Note : the advantage is increased by 1 in order to anticipate one more turn in addition to the advantage.

3 Final Implementation

The final version of the code consists of an evolution of the old version, remodularized in order to have a library that can be imported for use of the implemented player.

Also, a tree a tree can be plotted in order to evaluate how big is the search space by setting the `self.plot_trees` variable equal to True. In this way, for each turn, the tree explored is shown using the NetworkX library.

It was useful to introduce different parameters in the main in order to run tests in an easier way:

- –mode: def('MyGame') MyGame is a class derived from Game, which modifies only some methods for a simpler display through the use of colors and debug prints. If this parameter is modified, the standard Game version is used.
- –p1: def('RandomPlayer') To select the player 1 (symbol 0, green in MyGame)
- –p2: def('MinMaxPlayer') To select the player 2 (symbol 1, red in MyGame)
- –n_tests: def(100) To select the number of matches the two players will play.
- –my_player: def(1) To select how to evaluate the victories. By default its possible to know the percentage of wins of the MinMaxPlayer who plays with symbol 1.
- –human: def(False) To play a match between a HumanPlayer, who takes moves by input, and the MinMaxPlayer.

Note : Some examples of the usage of these parameters is present on the provided `quixo_project.ipynb` notebook, which can be easily opened and tested on Google Colab to evaluate the project.

`make_move() :`

It acts as a wrapper function. It selects the best moves at the beginning without requiring the minmax algorithm. If the going gets tough it resorts to a minmax of variable depth.

```
def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move
    ]:
    pid = game.get_current_player()
    possible_moves = self.get_possible_moves(game, pid%2)

    alpha = float('-inf')
    beta = float('inf')
    b_val = float('-inf')
    # Begin from one corner
    if possible_moves[0][1] <= 1:
        from_pos, move = random.choice(list(filter(lambda pm : pm
            [0][0] == (0, 0) or
```

```
            pm[0][0] == (0, game._board.shape[1]-1) or
            pm[0][0] == (game._board.shape[0]-1, 0) or
            pm[0][0] == (game._board.shape[0]-1, game._board.
               shape[1]-1)
        , possible_moves)))[0]
        return from_pos, move
    elif possible_moves[0][1] == 5 :
        return possible_moves[0][0]
    else:
        from_pos, move = possible_moves[0][0]

    #actual depth variation
    max_p1 = max_inline_pieces(pid, game)
    max_p2 = max_inline_pieces((pid+1)%2, game)
    depth = min(abs(max_p1 - max_p2) + 1, max_p2)

    TreeNode.node_count = 0
    tree = TreeNode('root', pid%2)

    for child in possible_moves:
        next_game = deepcopy(game)
        next_game._Game__move(child[0][0], child[0][1], pid%2)
        p_moves = self.get_possible_moves(next_game, (pid+1)%2)
        treechild = TreeNode('child', (pid+1)%2)
        val = self.minmax(p_moves, depth - 1, next_game, alpha,
           beta, (pid+1)%2, pid, treechild)
        treechild.val = val
        tree.add_child(treechild)
        if val > b_val:
            tree.val = val
            b_val = val
            from_pos = child[0][0]
            move = child[0][1]

    if self.plot_trees and (max_p1 >= 3 or max_p2 >= 3):
        print(f"depth: {depth}")
        graph = nx.Graph()
        tree.add_to_graph(graph)
        # colors is a sequence
        colors = [graph.nodes[n]['color'] for n in graph.nodes]
        # labels is a dictionary
        labels = nx.get_node_attributes(graph, 'value')

        # pos = graphviz_layout(graph, prog="dot")
        pos = graphviz_layout(graph, prog="twopi")
        plt.figure(figsize=(12, 8))
        nx.draw_networkx_labels(graph, pos, labels=labels,
           font_size=10, font_color="black")
        nx.draw(graph, pos, node_color=colors)
        plt.show()
        print("best value : ", b_val)
```

```
59      return from_pos, move
```

get_possible_moves() :

Selects the possible moves a player can perform. It orders the moves based on how many pieces of its sign it puts in a row and based on the symbol of the selected square (own or neutral) and it performs the shuffle discussed above.

```
def get_possible_moves(self, game: 'Game', player) -> list[tuple[
    tuple[int, int], Move]]:
        possible_moves = []
        # three possible moves for the cornice without the
            corners
        for i in range(1, 4, 1):
            # if I select a piece from the top row, I can slide
                it in any other direction, but not in the same
            if game._board[0][i] == -1 or game._board[0][i] ==
                player:
                possible_moves.append(((i, 0), Move.BOTTOM))
                possible_moves.append(((i, 0), Move.LEFT))
                possible_moves.append(((i, 0), Move.RIGHT))
            if game._board[4][i] == -1 or game._board[4][i] ==
                player:
                possible_moves.append(((i, 4), Move.TOP))
                possible_moves.append(((i, 4), Move.LEFT))
                possible_moves.append(((i, 4), Move.RIGHT))
            if game._board[i][0] == -1 or game._board[i][0] ==
                player:
                possible_moves.append(((0, i), Move.TOP))
                possible_moves.append(((0, i), Move.BOTTOM))
                possible_moves.append(((0, i), Move.RIGHT))
            if game._board[i][4] == -1 or game._board[i][4] ==
                player:
                possible_moves.append(((4, i), Move.TOP))
                possible_moves.append(((4, i), Move.BOTTOM))
                possible_moves.append(((4, i), Move.LEFT))
        # two possible moves for the corners
        if game._board[0][0] == -1 or game._board[0][0] == player
            :
            possible_moves.append(((0, 0), Move.RIGHT))
            possible_moves.append(((0, 0), Move.BOTTOM))
        if game._board[0][4] == -1 or game._board[0][4] == player
            :
            possible_moves.append(((4, 0), Move.BOTTOM))
            possible_moves.append(((4, 0), Move.LEFT))
        if game._board[4][0] == -1 or game._board[4][0] == player
            :
            possible_moves.append(((0, 4), Move.RIGHT))
            possible_moves.append(((0, 4), Move.TOP))
        if game._board[4][4] == -1 or game._board[4][4] == player
            :
```

```
            possible_moves.append(((4, 4), Move.LEFT))
            possible_moves.append(((4, 4), Move.TOP))

        sorted_pm = []
        for pm in possible_moves:
            new_game = deepcopy(game)
            new_game._Game__move(pm[0], pm[1], player)
            if new_game.check_winner() == player:
                return [(pm, 5, game._board[pm[0][1]][pm[0][0]])]
            # count the number of pieces in line the move creates
                and indicate if the piece is neutral or not in
               the original board
            sorted_pm.append((pm, max_inline_pieces(player,
               new_game), game._board[pm[0][1]][pm[0][0]]))
        # sort the possible moves in descending order based on
           the number of pieces in line
        # putting firstly the moves with neutral pieces
        sorted_pm = sorted(sorted_pm, key=lambda x: (x[1], -x[2])
           , reverse=True)
        # shuffle
        grouped_pm = {}
        for pm in sorted_pm:
            if (pm[1], pm[2]) not in grouped_pm:
                grouped_pm[(pm[1], pm[2])] = []
            grouped_pm[(pm[1], pm[2])].append(pm)

        for group in grouped_pm:
            random.shuffle(grouped_pm[group])
        shuffled_sorted_pm = [item for group in grouped_pm.values
           () for item in group]
        return shuffled_sorted_pm
```

minmax() :

```
def minmax(self, node, depth, game: 'Game', alpha, beta,
   maximizing_player, pid, tree):
        # a node is terminal if there are no more moves to make
        winner = game.check_winner()
        if depth == 0 or winner != -1 or is_terminal(node):
            value = evaluate(winner, pid, game)
            tree.val = value
            return value
        if maximizing_player == pid:
            b_val = float('-inf')
            for child in node:
                next_game = deepcopy(game)
                next_game._Game__move(child[0][0], child[0][1],
                   pid%2)
                # next_game.print()
```

```
14                  p_moves = self.get_possible_moves(next_game, (
                        maximizing_player+1)%2)
15                  treechild = TreeNode('child', (maximizing_player
                        +1)%2)
16                  val = self.minmax(p_moves, depth - 1, next_game,
                        alpha, beta, (maximizing_player+1)%2, pid,
                        treechild)
17                  treechild.val = val
18                  tree.add_child(treechild)
19                  b_val = max(b_val, val)
20                  alpha = max(alpha, b_val)
21                  if beta <= alpha:
22                      break
23              return alpha
24          else:
25              b_val = float('inf')
26              for child in node:
27                  next_game = deepcopy(game)
28                  next_game._Game__move(child[0][0], child[0][1], (
                        pid+1)%2)
29                  # next_game.print()
30                  p_moves = self.get_possible_moves(next_game, (
                        maximizing_player+1)%2)
31                  treechild = TreeNode('child', (maximizing_player
                        +1)%2)
32                  val = self.minmax(p_moves, depth - 1, next_game,
                        alpha, beta, (maximizing_player+1)%2, pid,
                        treechild)
33                  treechild.val = val
34                  tree.add_child(treechild)
35                  b_val = min(b_val, val)
36                  beta = min(beta, b_val)
37                  if beta <= alpha:
38                      break
39              return beta
```

4 Results

This work highlights the trade-off between high and low depth and the power that some optimizations have to lower the computational cost of an algorithm which, if performed in the standard and complete way, would not be able to provide any results, even if optimal.

The last version of the algorithm takes about 20/25 minutes to perform 100 games with a random player (about 15 seconds per game) and it takes much more time to play against itself. This time is variable according to the depth of the tree that the minmax algorithm has to explore.

The implemented player is capable of winning always against a random player (100% of win percentage), but is not able to always win against itself. This is due to the fact that the algorithm is not able to find the best move in a reasonable time if the depth is increased too much and, on the contrary, if the depth is decreased, the algorithm is not

optimal because it is not complete.

Last considerations (Some Math) :

Consider :

- b number of items selectable on the border
- l is the number of free borders
- $mb = 3$ is the number of possible moves if the item is on the border
- a is the number of selectable angles
- $ma = 2$ is the number of possible moves which can be performed on the angle

At the beginning, when the board is empty, the possible moves the minmax player can choose are $((b = 3) \times (l = 4)) \times (mb = 3) + (a = 4) \times (ma = 2) = 44$. Moving through the minmax tree, the opponent will then have $(3 \times 4) \times 3 + 3 \times 2 = 42$ possible moves to choose if the board has a symbol in the corner and $(3 \times 3) \times 3 + 2 \times 3 + 4 \times 2 = 41$ possible moves if the board has a symbol in the border. So, the number of possible moves decreases as the game goes on.

However, despite the minmax strategy is not needed for the initial moves, the number of possible moves is still high during the game, because it is not possible to finish selectable pieces without losing, so the minmax can't easily reach the leaves in a reasonable time. In fact, with a depth of 2, it can predict the outcome of $44 \times 42 = 1848$ possible moves in an empty board state and $(3 \times 2) \times 3 + 2 \times 2 = 20 \times 20 = 400$ possible moves if the player can only use one half of the board. It's difficult for a human player to predict the outcome of 400 possible moves, so it's difficult to beat a minmax player that reaches a depth of 2 (or, with the strongest version of the algorithm 3, or 4) in the tree (the number increases almost exponentially).

The following images (Fig. 1, 2, 3, 4, 5, 6, 7, 8) show some minmax trees (in two different flavours) obtained with alpha beta pruning using the advantage depth variable strategy and the shuffled sorted version of deep pruning, with the corresponding configuration of the board, where the player is playing.

Disclaimer : The objective of these figures is not to show the possible fitness values obtained in the search, but to observe the quantity of possible paths evaluated each time by the algorithm in order to choose the best move in the long (quantified by depth) term.

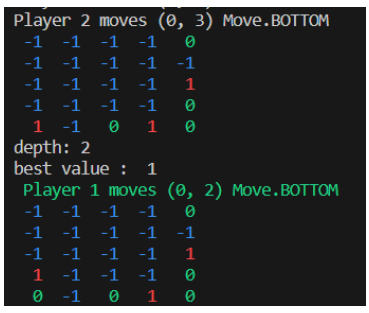

Figure 1: Board state before the minmax tree (MinMax vs Random)

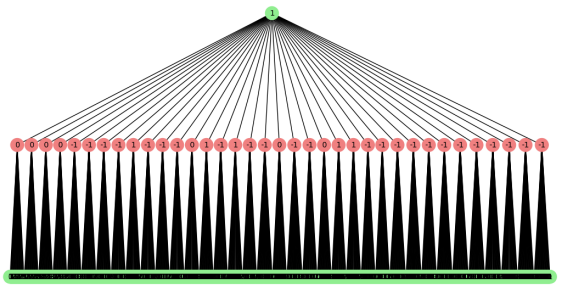

Figure 2: Corresponding MinMax tree generated

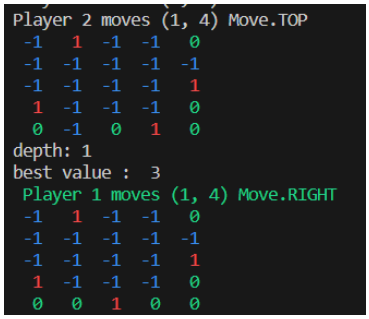

Figure 3: Board state before the minmax tree (MinMax vs Random)

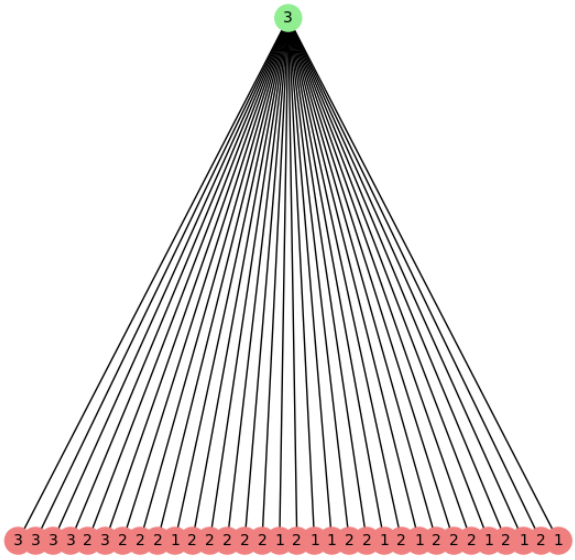

Figure 4: Corresponding MinMax tree generated

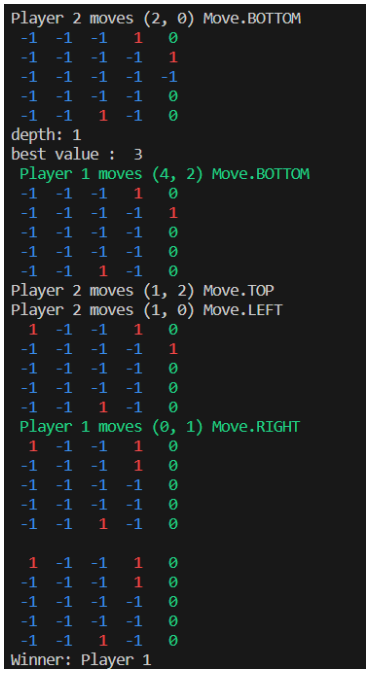

Figure 5: Board state before the minmax tree (MinMax vs Random)

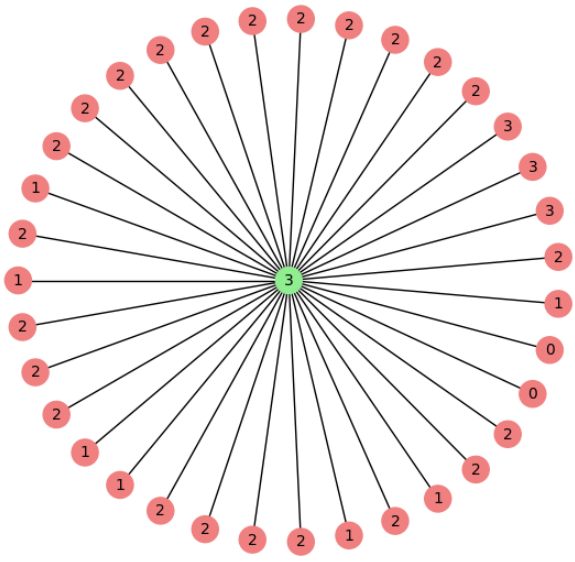

Figure 6: Corresponding MinMax tree generated

```
Player 2 moves (1, 4) Move.LEFT
  1  -1  -1  -1   0
 -1  -1  -1  -1  -1
 -1  -1  -1  -1  -1
 -1  -1  -1  -1   0
  1   1  -1  -1   0
depth: 2
best value :  0
 Player 1 moves (4, 1) Move.TOP
  1  -1  -1  -1   0
 -1  -1  -1  -1   0
 -1  -1  -1  -1  -1
 -1  -1  -1  -1   0
  1   1  -1  -1   0
Player 2 moves (3, 0) Move.LEFT
  1   1  -1  -1   0
 -1  -1  -1  -1   0
 -1  -1  -1  -1  -1
 -1  -1  -1  -1   0
  1   1  -1  -1   0
 Player 1 moves (0, 2) Move.RIGHT
  1   1  -1  -1   0
 -1  -1  -1  -1   0
 -1  -1  -1  -1   0
 -1  -1  -1  -1   0
  1   1  -1  -1   0

  1   1  -1  -1   0
 -1  -1  -1  -1   0
 -1  -1  -1  -1   0
 -1  -1  -1  -1   0
  1   1  -1  -1   0
Winner: Player 1
```

Figure 7: Board state before the minmax tree (MinMax vs Random)

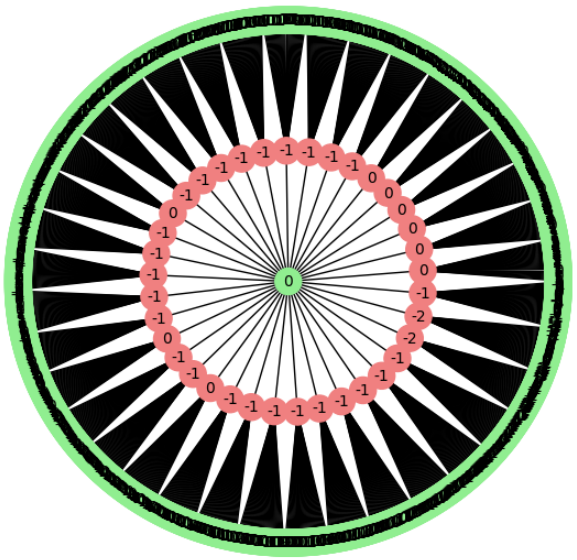

Figure 8: Corresponding MinMax tree generated

In order to evaluate the performance of the actual version of MinMax player with other players, it was useful to import other players from other repositories.

In particular, it was possible to compare the performance of the MinMax player against other valuable strategies, explore other solutions, and understand the vulnerabilities of the MinMax strategy implemented.

For example, importing the Montecarlo Tree Search strategy taken from one of my colleague's repository (`https://github.com/fgiacome/compint/blob/main/quixo/mcts.py`), the MinMax player is able to win about 95% of the times beginning first or second.

Figures 9, 10, 11, 12, 13 show some trees explored by the alpha beta pruning minmax implemented using the following depth variation rule:

```
if is_losing(pid, 3, game) or is_losing((pid+1)%2, 3, game):
    depth = 3
else:
    depth = 1
```

```
Player 1 moves (4, 3) Move.BOTTOM
-1  -1   0   1   0
-1  -1  -1  -1   0
-1  -1  -1  -1  -1
-1  -1  -1  -1   1
 1  -1  -1  -1   0
depth: 4
best value :  -1
Player 2 moves (1, 0) Move.RIGHT
-1   0   1   0   1
-1  -1  -1  -1   0
-1  -1  -1  -1  -1
-1  -1  -1  -1   1
 1  -1  -1  -1   0
Player 1 moves (2, 4) Move.TOP
-1   0   0   0   1
-1  -1   1  -1   0
-1  -1  -1  -1  -1
-1  -1  -1  -1   1
 1  -1  -1  -1   0
```

Figure 9: Board state before the minmax tree (MinMax vs MinMax)

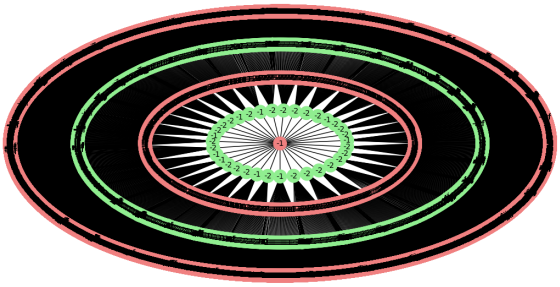

Figure 10: Corresponding MinMax tree generated (depth = 4)

```
Player 1 moves (0, 1) Move.TOP
 0  -1  -1  -1   0
-1  -1  -1  -1  -1
-1  -1  -1  -1  -1
-1  -1  -1   0   1
 1  -1  -1   1   0
depth: 3
best value :  1
Player 2 moves (1, 4) Move.LEFT
 0  -1  -1  -1   0
-1  -1  -1  -1  -1
-1  -1  -1  -1  -1
-1  -1  -1   0   1
 1   1  -1   1   0
```

Figure 11: Board state before the minmax tree (MinMax vs MinMax)

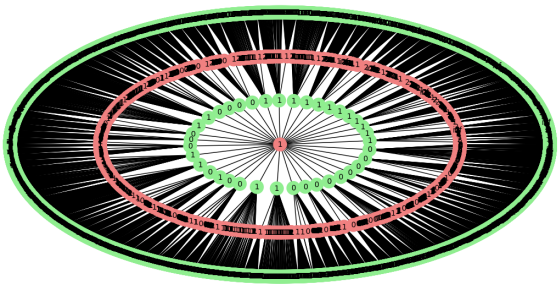

Figure 12: Corresponding MinMax tree generated (depth = 3)

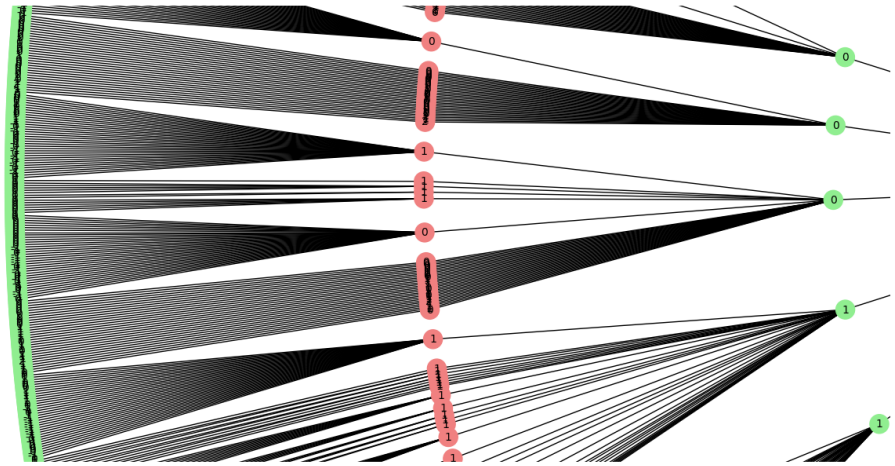

Figure 13: Zoom in on the (depth=3) tree of the Figure 12