

Artificial Intelligence Report

PROJECT 01 – Othello Bot

19127102 – Võ Hoàng Gia Bảo

19127406 – Ngô Huy Hoàng

19127457 – Nguyễn Tuấn Kiệt



Bộ môn Cơ sở trí tuệ nhân tạo
Khoa Công nghệ thông tin
Đại học Khoa học tự nhiên TP. HCM

1. Theory of Adversarial Search:

- _ Adversarial Search is a search when there is an "enemy" or "opponent" changing the state of the problem every step in a direction you do not want.
- _ Planning used to play a game such as chess or checkers – algorithms are similar to graph search except that we plan under the assumption that our opponent will maximize his own advantage...
- _ A good example is in board games.
- _ Adversarial games, while much studied in AI, are a small part of game theory in economics.

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

Complexity: games are too hard to be solved

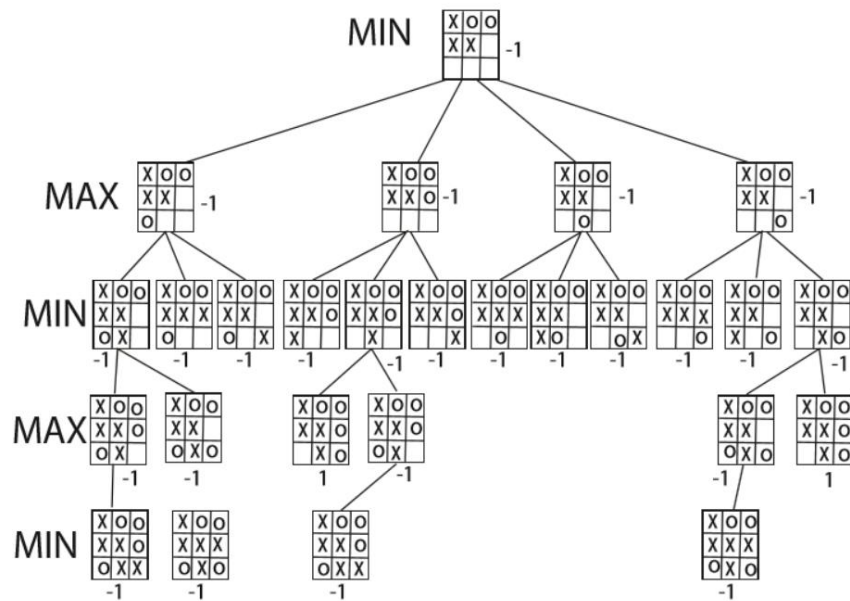
Time limits: make some decision even when calculating the optimal decision is infeasible

Efficiency: penalize inefficiency severely

A game can be defined as a type of search in AI which can be formalized of the following elements:

- Initial state: It specifies how the game is set up at the start.
- Player (s): It specifies which player has moved in the state space.
- Action (s): It returns the set of legal moves in state space.
- Result (s, a): It is the transition model, which specifies the result of moves in the state space.
- Terminal-Test (s): Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- Utility (s, p): A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For tic-tac-toe, utility values are +1, -1, and 0, which stand for win, loss, and draw

*Minimax



Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.

The values of the board are calculated by some heuristics which are unique for every type of game.

Pseudocode:

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := -∞
        for each child of node do
            value := max(value, minimax(child, depth - 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth - 1, TRUE))
        return value
```

* Alpha-Beta Pruning

_ Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.

_ Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

_ The two-parameter can be defined as:

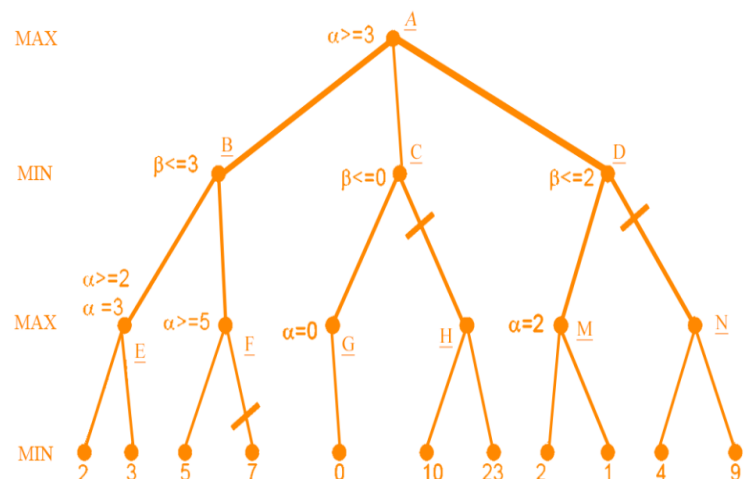
+ Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is $-\infty$.

+ Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is $+\infty$.

_ Alpha-beta pruning seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games.

_ Worst-case performance: $O(b^d)$

_ Best-case performance: $O(\sqrt{b^d})$



*Heuristic

A heuristic is a mental shortcut that allows people to solve problems and make judgments quickly and efficiently. These rule-of-thumb strategies shorten decision-making time and allow people to function without constantly stopping to think about their next course of action.

Heuristics are the strategies derived from previous experiences with similar problems. These strategies depend on using readily accessible, though loosely applicable, information to control problem solving in human beings, machines and abstract issues.

A heuristic can be used in artificial intelligence systems while searching a solution space. The heuristic is derived by using some function that is put into the system by the designer, or by adjusting the weight of branches based on how likely each branch is to lead to a goal node.

2. User Manual

In this project we use Pycharm Community, which is an IDE used in computer programming, specifically for the Python language.

Download link for Pycharm Community:

[Download PyCharm: Python IDE for Professional Developers by JetBrains](#)

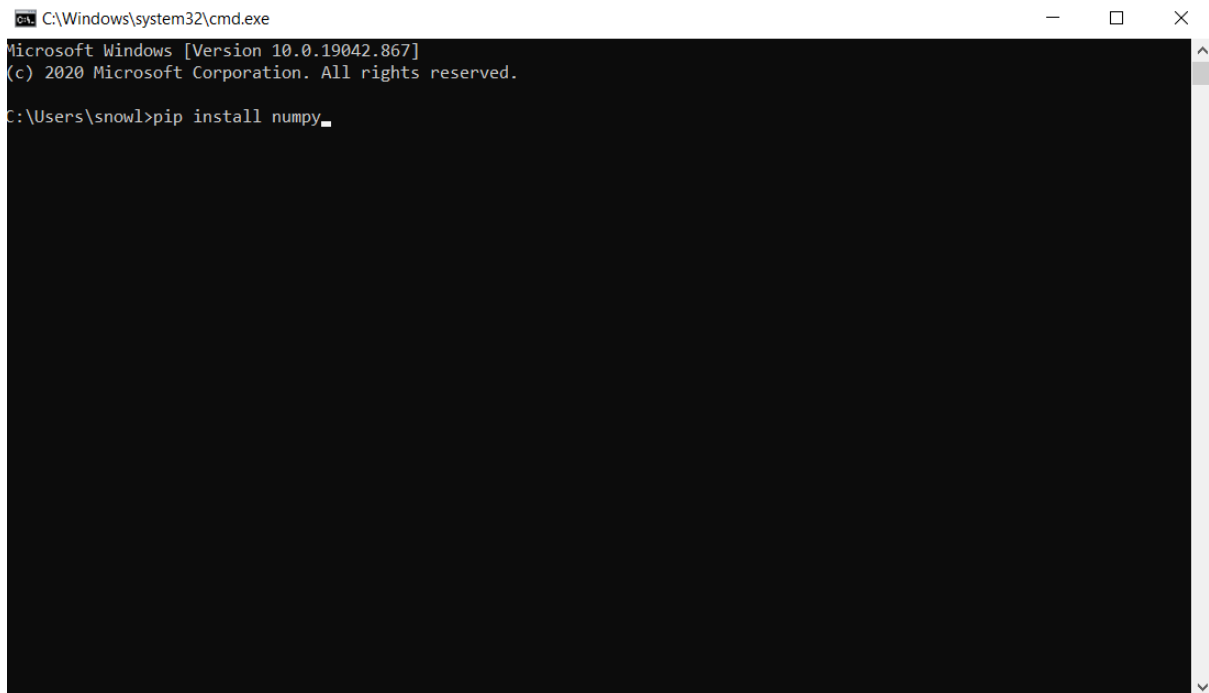
It also requires numpy to run the code:

Download link for Python 3.9.2: [Download Python | Python.org](#)

To install numpy, there are 2 ways:

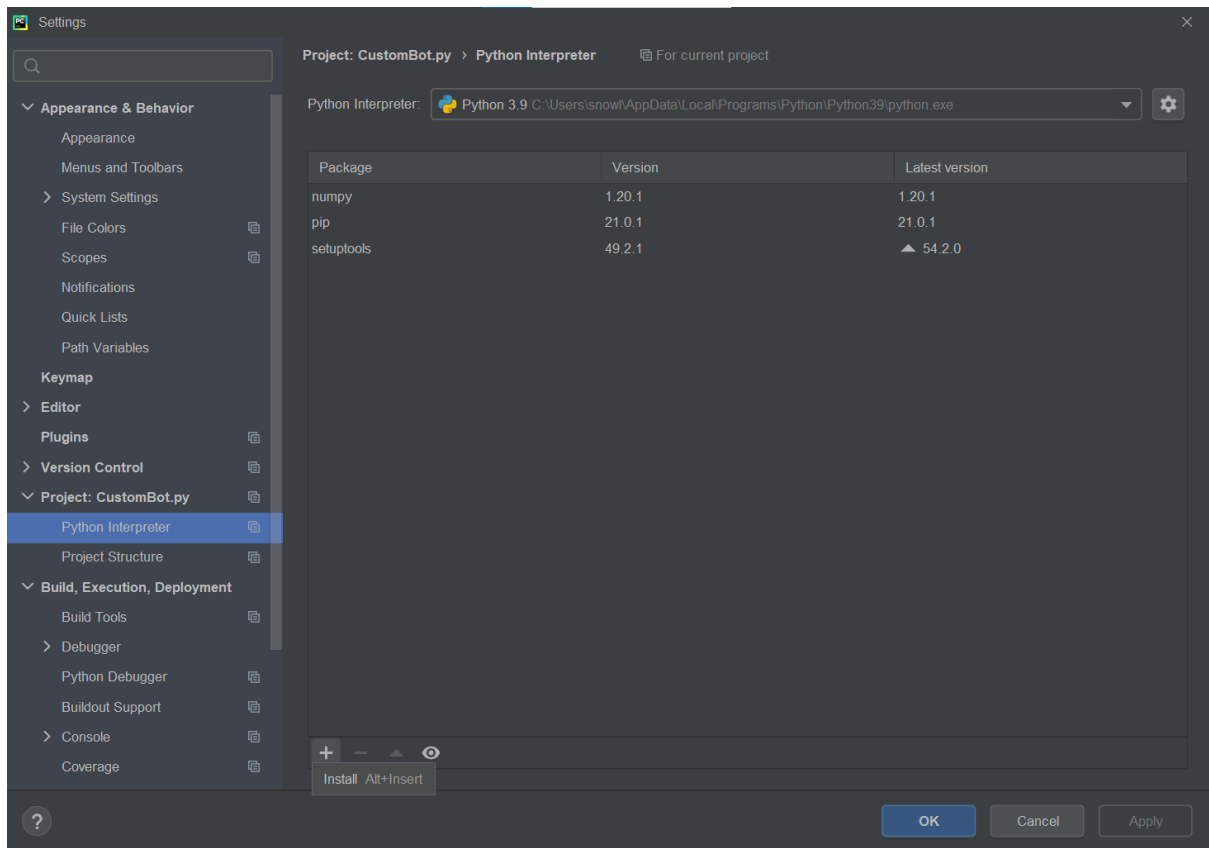
1. Open command prompt then type the following command:

pip install numpy

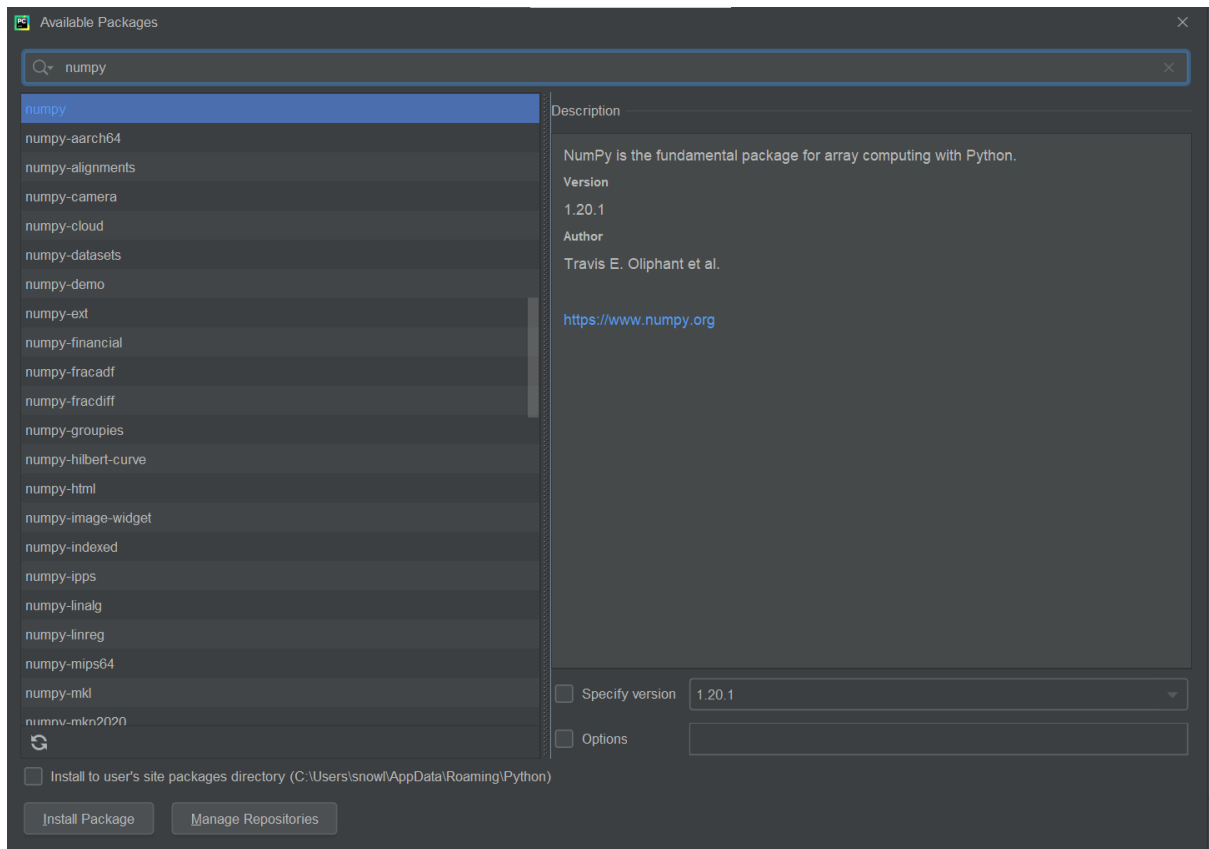
A screenshot of a Windows Command Prompt window. The title bar at the top reads 'C:\Windows\system32\cmd.exe'. The window content shows the following text: 'Microsoft Windows [Version 10.0.19042.867]', '(c) 2020 Microsoft Corporation. All rights reserved.', and the command prompt 'C:\Users\snowl>'. The command 'pip install numpy' has been typed into the prompt, followed by a cursor. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

2. Open Pycharm Community -> File -> Settings -> Project -> Python Interpreter

Click on the “+” icon



Search for “numpy” then press “install package”

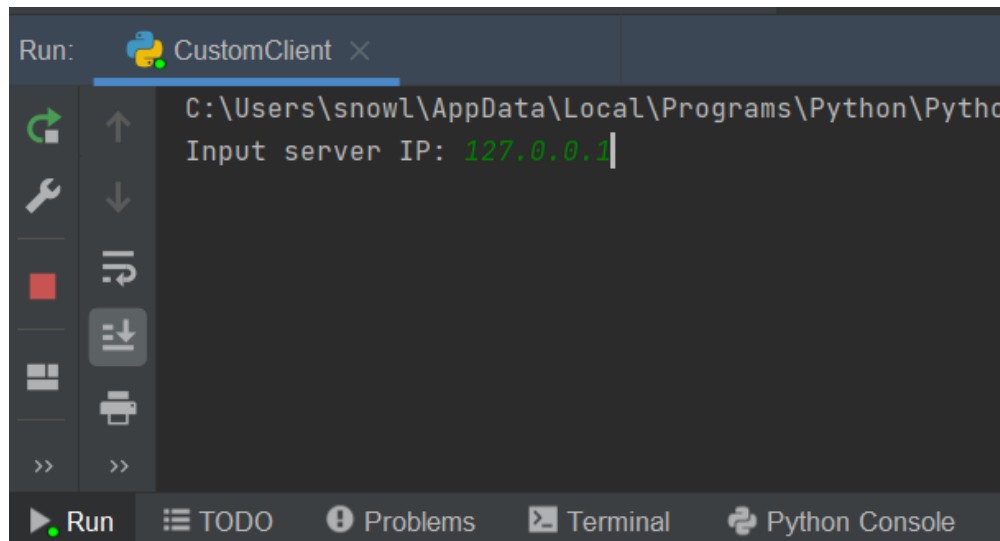


First the user must run the server code given by the instructor. The server code will run a bot with random choice among the possible positions which the player can make on the board. To run the code, double click on the **server.py** file to open it in Pycharm, then right-click on the **server.py** tab, click **Run**.

Then, double click on the **CustomClient.py** file and run it.

The user will be prompted to input the server IP address.

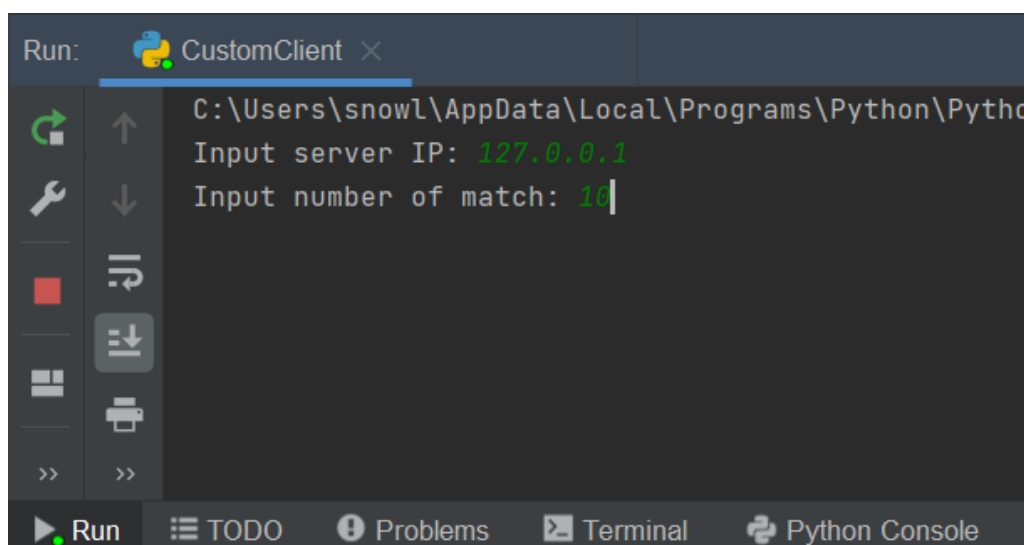
If the user want to play offline, please input “**127.0.0.1**”

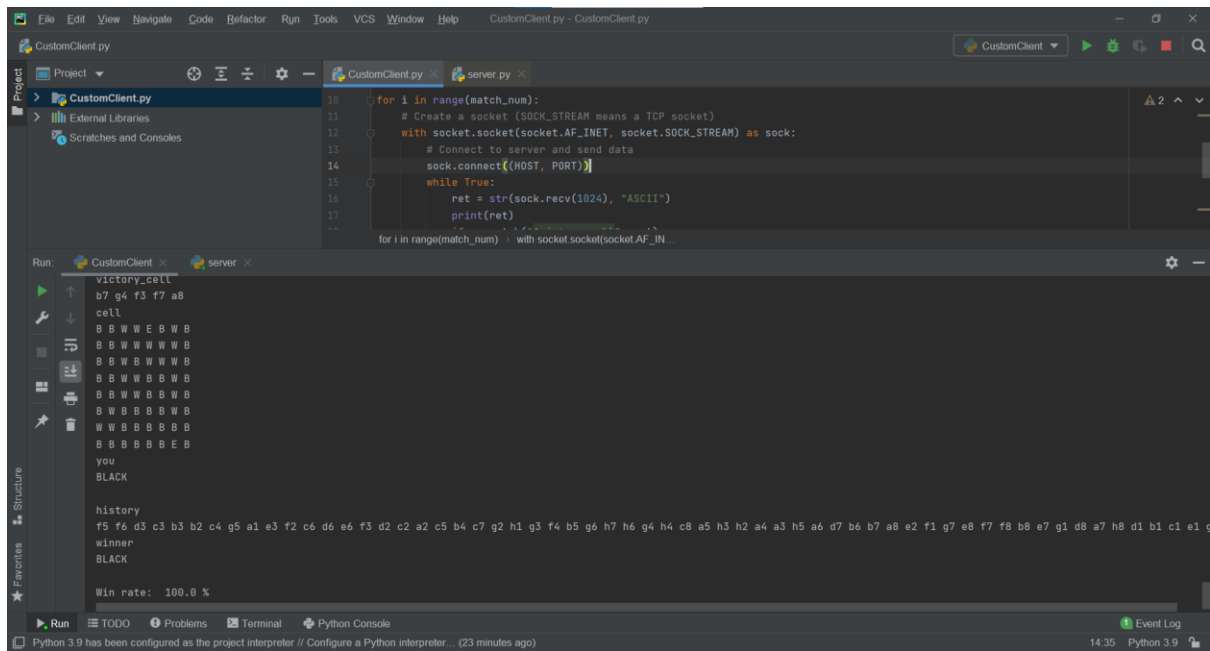


If the user want to play online with the server on another PC, please input his/her IP address while you are on the same **LAN**. To get server IP address, open cmd on server's PC then type the following command and look for the IPv4 Address:

ipconfig

After inputting the server IP address, the user will be prompt to enter the number of matches which the user want to play with the server.





Pseudocode:

```

minimax(victory_cell, cell, you)
    return minimax_max(victory_cell, cell, you, depth)

minimax_max(victory_cell, cell, you, depth):
    move_states = {move: play_move(new_board, you, ColumnID, RowID for move in possible_positions)

    best_move = None
    best_value = None

    if len(possible_positions) > 0:
        if depth == 1:
            for move, state in move_states:
                if best_move == None or minimax_min(victory_cell, state, you, depth + 1) > best_value:
                    best_move = move
                    best_value = minimax_min(victory_cell, state, you, depth + 1)

            return best_move

        else:
            for move, state in move_states:
                if best_move == None or minimax_min(victory_cell, state, you, depth + 1) > best_value:
                    best_value = minimax_min(victory_cell, state, you, depth + 1)
            return best_value

    return compute_heuristic_score(victory_cell, cell, you)

```



```

minimax_min(victory_cell, cell, you, depth):

    opponent = 'W' if you == 'B' else 'B'

    move_states = {move: play_move(new_board, opponent, ColumnID, RowID for move in possible_positions}

    best_move = None
    best_value = None

    if len(possible_positions) > 0:
        if depth <= 3:
            for move, state in move_states:

                if best_move == None or minimax_max(victory_cell, state, you, depth + 1) < best_value:
                    best_move = move
                    best_value = minimax_max(victory_cell, state, you, depth + 1)

            return best_value

        else:
            for move, state in move_states:

                if best_value == None or compute_heuristic_score(victory_cell, state, you) < best_value:
                    best_value = compute_heuristic_score(victory_cell, state, you)

            return best_value
    return compute_heuristic_score(victory_cell, cell, you)

```

3. Advantages and Disadvantages:

3.1. Advantages:

- _ Find the best move in shorter time using minimax function
- _ Don't use recursion in minimax function, makes it easier to keep tracks while debugging.
- _ Have depth of 3 so the bot can calculate more opponent's moves, then make the right decision on choosing the best move based on the best score calculated by the heuristic function.

3.2. Disadvantages:

- _ The bot cannot connect to each other if server and client are not on the same LAN.
- _ Don't use the Alpha-Beta pruning, the code takes a few more seconds to make the move.
- _ Calculating heuristic score function is not yet optimized for the final state of the game.

3.3. Improvement:

- _ Use Alpha-Beta pruning to reduce the time finding the best move.
- _ Optimize the calculating heuristic score function in the final state of the game by returning the most pieces instead of the best score.

3.4. Applicable methods the world currently uses:

* Negamax

In the above Minimax procedure, we alternately have to take the minimum or the maximum of the values returned for successor positions. This makes programming a Minimax procedure a little cumbersome: we have to distinguish the two cases explicitly.

For programming a Minimax search, the Negamax approach is more comfortable. Instead of always evaluating the position from the computer's point of view (as we do in Minimax), we now evaluate it always from the point of view of the player who is to move next! The opponent's value for the same position then becomes simply the negative of that player's value.

The main advantage of this approach is that we can get rid of the two different cases. We don't have to distinguish between MIN and MAX levels anymore, instead, we always take the maximum, but we negate the position evaluation on each level. The two variants are equivalent since taking the minimum of positive values yields the same as taking the maximum of the same values, negated.

* NegaScout

NegaScout is a negamax algorithm that can be faster than alpha-beta pruning. Like alpha-beta pruning, NegaScout is a directional search algorithm for computing the minimax value of a node in a tree. It dominates alpha-beta pruning in the sense that it will never examine a node that can be pruned by alpha-beta; however, it relies on accurate node ordering to capitalize on this advantage.

NegaScout works best when there is a good move ordering. In practice, the move ordering is often determined by previous shallower searches. It produces more cutoffs than alpha-beta by assuming that the first explored node is the best. In other words, it supposes the first node is in the principal variation. Then, it can check whether that is true by searching the remaining nodes with a null window (also known as a scout window; when alpha and beta are equal), which is faster than searching with the regular alpha-beta window. If the proof fails, then the first

node was not in the principal variation, and the search continues as normal alpha-beta. Hence, NegaScout works best when the move ordering is good. With a random move ordering, NegaScout will take more time than regular alpha-beta; although it will not explore any nodes alpha-beta did not, it will have to re-search many nodes.