Warsaw University of Technology
Faculty of Electronics and Information Technology
Specialization: Computer Science
Course: Introduction to Artificial Intelligence

PROJECT DOCUMENTATION

Min-Max algorithm with alpha-beta prunning

Author:
Monika Jung

Student ID:
331384

1 Task description and interpretation

The goal of this assignment was to implement the Min-Max algorithm with alpha-beta pruning and apply it to a text-based implementation of Tic-Tac-Toe. Then, the performance of the implemented AI player (MinMax) was to be compared against both a human player and a random player.

Additionally, in the next part of the task, the influence of different search tree depths on the quality of MinMax's gameplay was to be analyzed using different board sizes.

2 Min-Max algorithm with alpha-beta pruning

The core of this project was the implementation of the Min-Max algorithm enhanced with alpha-beta pruning to optimize decision-making in the Tic-Tac-Toe game. The algorithm was implemented in the class `MinMaxPlayer`, based on the following structure:

- The method `make_move()` is responsible for selecting the best move by simulating all possible actions from the current board state. It calls the recursive function `minimax()`.
- The recursive function `minimax()` explores the game tree up to the depth defined by `depth_limit`. It evaluates each possible game state, alternating between maximizing and minimizing players.
- The function implements alpha-beta pruning by maintaining two variables: `alpha` (best score for maximizer so far) and `beta` (best score for minimizer). If at any point `alpha >= beta`, further exploration of a branch is stopped.
- The function `evaluate()` checks the current board for a winner. If the AI player wins, it returns 1; if the opponent wins, it returns -1; and 0 otherwise.
- To ensure that the original board is not modified during recursive calls, every move is applied on a cloned copy of the board using `board.clone()`.

3 Experiment: MinMax vs Random Player

To evaluate the strength of the implemented MinMax algorithm, 50 games were played against a random player. The depth limit was set to 3.

Table 1: Results of 25 games: MinMax (depth 9) vs Random - MinMax as 'O' player.

Outcome	Count
MinMax wins	25
Random wins	0
Draws	0

Table 2: Results of 25 games: MinMax (depth 9) vs Random - MinMax as 'X' player

Outcome	Count
MinMax wins	18
Random wins	0
Draws	7

Outcomes snapshots - 3x3 board

These are snapshots of outcomes from 4 games.

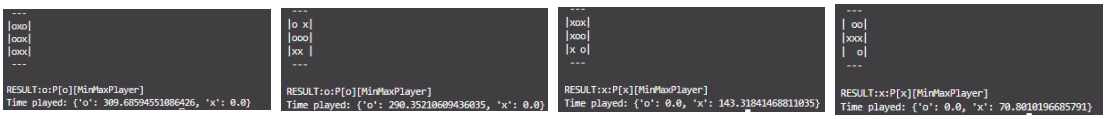


Figure: Final board states in MinMax vs Random games.

Game snapshots - 5x5 board

Snapshots from one game random vx minmax.

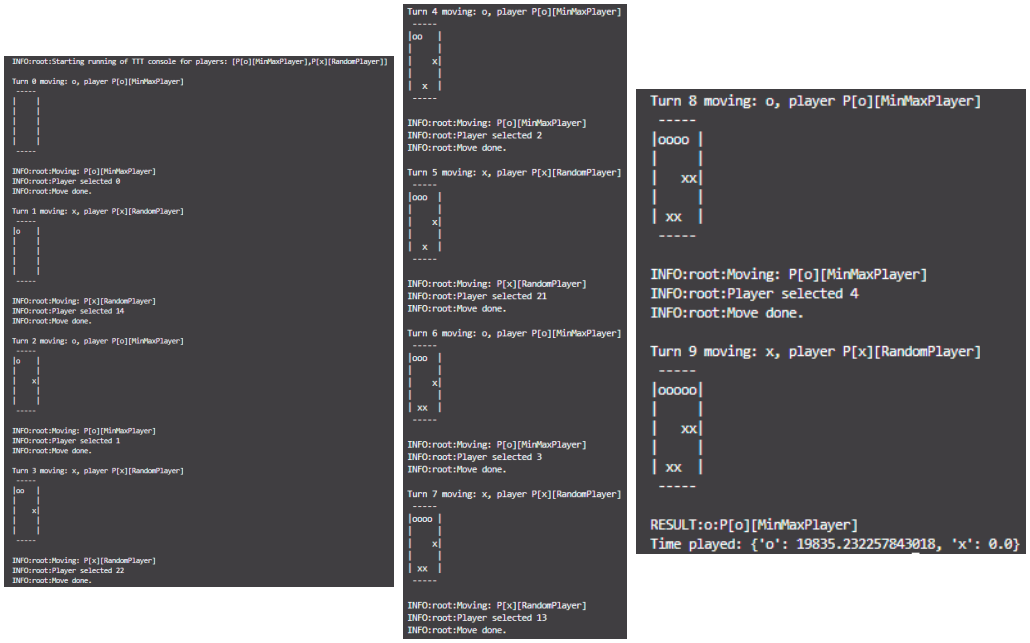


Figure: Sample end-game states from MinMax games.

- For board size 5x5, the game took significantly longer to compute due to the increased number of possible moves.
- In matches against the Random player, many games ended in a draw. This is expected, as on larger boards it is generally more difficult to achieve a win – especially for the second player ('x').

4 Experiment: MinMax vs Human Player

To examine how the MinMax algorithm performs against a human on larger boards, a few test games were conducted with board sizes 3x3 and 5x5.

Board size 3x3

- The human player occasionally managed to win, especially when MinMax was a 'X' player
- When MinMax was an 'O' player, it was unbeatable

Outcomes snapshots - 3x3 board

These are snapshots of outcomes from 2 games.

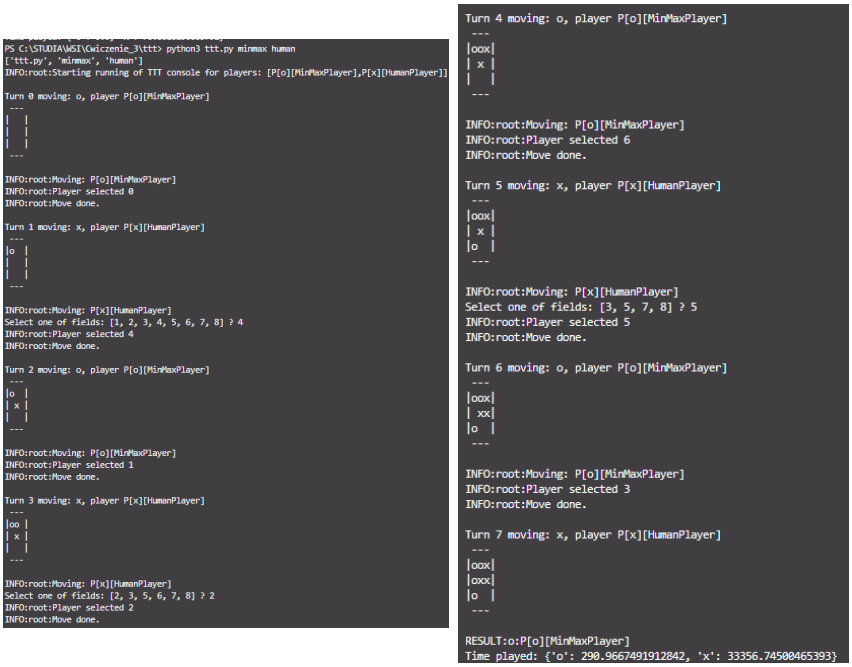


Figure: Game Human vs MinMax example 1

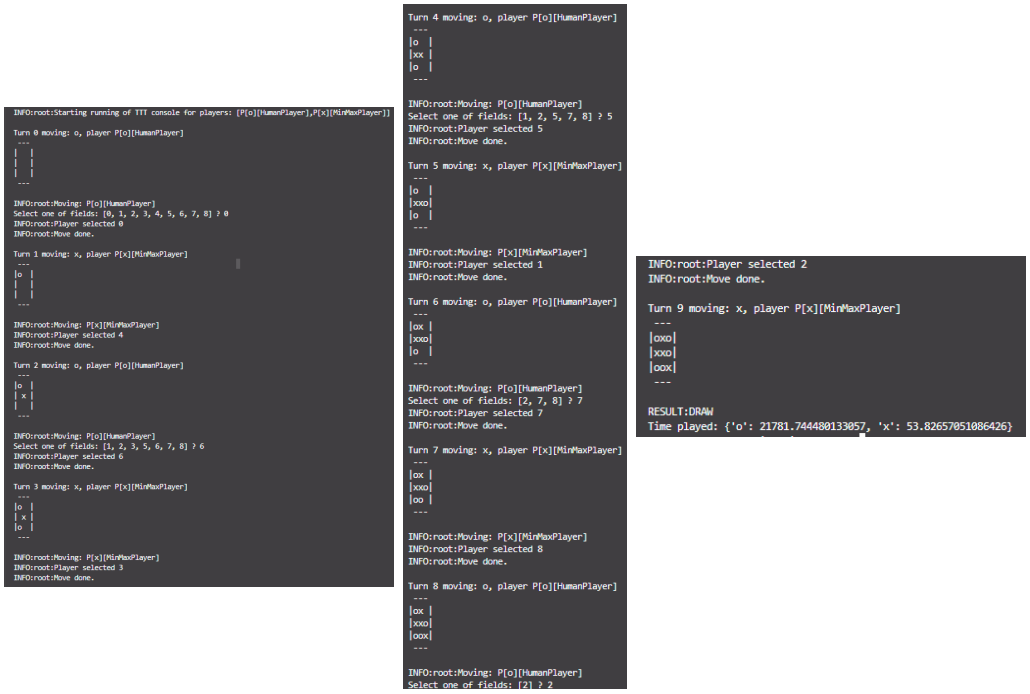


Figure: Game Human vs MinMax example 2

Board size 5x5

- The computation time per move increased significantly due to the large number of legal positions.
- Many games ended in a draw, especially when the human played as 'x'. This is likely due to the difficulty of building a winning line on a larger board.

Outcomes snapshots - 5x5 board

These are snapshots of outcomes from one game.

Figure: Game Human vs MinMax example 1

5 Experiment: MinMax vs MinMax with different depth limits

To analyze how the search tree depth influences MinMax performance, multiple games were played between two MinMax players with varying depth configurations.

The following code modification was made in the `main.py` file to support custom depth levels for both players via command line arguments:

```
depth_limit = []
depth_limit.append(int(sys.argv[3]) if len(sys.argv) > 3 else 9)
depth_limit.append(int(sys.argv[4]) if len(sys.argv) > 4 else 9)
player1 = build_player(player1_type, "o", depth_limit[0])
player2 = build_player(player2_type, "x", depth_limit[1])
```

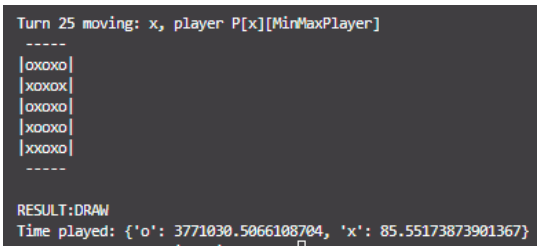
This enabled launching games with different search depths using:
`python3 main.py minmax minmax 3 9`

Board size: 3x3

- depth0 = 3, depth1 = 6: draw
- depth0 = 3, depth1 = 9: draw
- depth0 = 9, depth1 = 3: **MinMax 'o' wins**
- depth0 = 6, depth1 = 6: draw
- depth0 = 9, depth1 = 9: draw
- depth0 = 3, depth1 = 3: **MinMax 'o' wins**
- depth0 = 9, depth1 = 6: **MinMax 'o' wins**

Board size: 5x5

- All combinations resulted in **draws**.
- In every game, the move sequences were identical, regardless of depth differences.
- Games took much longer to compute, especially with depths like 9 or 10.
- Even for 10 vs 3 or 9 vs 3, the outcomes did not improve for the deeper player.



Conclusion

- On the 3x3 board, increasing the search depth improved performance. The player with the higher or equal depth limit (MinMax 'o') consistently outperformed the shallower opponent.
- On the 5x5 board, increasing the search depth up to 10 did not improve the quality of play. All matches resulted in a draw and the move sequences were identical, regardless of the depth settings.
- It is likely that a higher depth could eventually lead to a win on a larger board. However, the computational cost would be extremely high, resulting in impractically long waiting times for the opponent.