Artificial Intelligence: Minimax algorithm implementation in Othello

Konstantin Unruh

February 10, 2025

1 Background: Minimax Algorithm

The Minimax algorithm is a decision-making algorithm used in artificial intelligence, particularly in game theory and two-player games such as chess, tic-tac-toe, and checkers. The algorithm aims to minimize the possible loss for a worst-case scenario. The Minimax algorithm operates by simulating ideally all possible moves in a game and their subsequent outcomes, assuming that both players play optimally. The algorithm consists of the following steps:

1. **Generate the Game Tree**: Create a tree structure where each node represents a game state, and each edge represents a possible move by a player.
2. **Evaluate Terminal States**: Assign a value to each terminal state (leaf node) based on the outcome of the game (win, lose, draw).
3. **Backpropagate Values**: Starting from the terminal states, propagate the values back up the tree. For each non-terminal node:
   - If it is the maximizing player's turn, assign the maximum value of its children.
   - If it is the minimizing player's turn, assign the minimum value of its children.
4. **Choose the Optimal Move**: At the root node, the maximizing player chooses the move that leads to the child node with the highest value.

[1], [2], [3]

2 Code description

2.1 General

The game is implemented in C++ using the SFML graphics library[4] and is built using CMake[5].

[1] Elin Anna Topp et al. Lecture: Artificial Intelligence, Lund university, 2025
[2] https://www.geeksforgeeks.org/mini-max-algorithm-in-artificial-intelligence/
[3] https://www.freecodecamp.org/news/minimax-algorithm-guide-how-to-create-an-unbeatable-ai/
[4] https://www.sfml-dev.org/
[5] https://cmake.org/

2.2 File description

- **src/main.cpp**: This file contains the `main` function, which is the entry point of the application. It initializes the game by creating an instance of the `Game` class and starts the main game loop. It handles the setup and teardown of the game environment.
- **src/Game.cpp**: Implements the main game logic. This includes initializing game components, handling player input, updating the game state, and rendering the game. It manages different game states such as the start menu, playing, and end screen. It also includes methods for starting and stopping the game, processing user commands, and updating the game board.
- **src/AI.cpp**: Contains the implementation of the AI logic for the game. This includes decision-making algorithms for the AI player, such as evaluating possible moves and selecting the best move based on a given strategy. It uses the minimax algorithm, alpha-beta pruning, or other heuristic methods to make decisions.
- **src/Board.cpp**: Manages the game board, including initializing the board, drawing it, and handling moves. It includes methods for resetting the board, checking and highlighting valid moves (e.g., win, lose, draw). It also handles the graphical representation of the board and the placement of pieces or markers.

2.3 Minimax algorithm

The minimax algorithm is implemented in the **src/AI.cpp** file. As arguments the minimax algorithm takes the following values:

- `Board &board`: A reference to the current state of the game board.
- `int depth`: The current depth of the search tree.
- `bool isMaximizing`: A boolean indicating whether the current player is the maximizing player. Here we did in contrary to the example in the lecture only use one method and then use this variable to distinguish between maximizing and minimizing player.
- `int alpha`: The alpha value for alpha-beta pruning, representing the best value that the maximizer currently can guarantee.
- `int beta`: The beta value for alpha-beta pruning, representing the best value that the minimizer currently can guarantee.
- `std::chrono::high_resolution_clock::time_point startTime`: The start time of the minimax function, used to enforce a time limit.
- `int timeLimitMs`: The time limit in milliseconds for the minimax function to run.

We then check whether the time limit has been reached. There we use the `td:chrono` library to get the elapsed between the first call of the function and the current state. If the elapsed time first surpasses the time limit the evaluate function is immediately called.

```
// Check if the time limit has been reached
auto currentTime = std::chrono::high_resolution_clock::now();
int elapsedTimeMs = std::chrono::duration_cast<std::chrono::milliseconds>
                    (currentTime - startTime).count();
if (elapsedTimeMs >= timeLimitMs) {
    return evaluateBoard(board);
}
```

Then we check whether the maximum depth has been reached. For a reasonable time limit this will only occur for small boards or near the end of the game.

```
// If the maximum depth is reached, evaluate the board
if (depth == 0) {
    return evaluateBoard(board);
}
```

The minimax function recursively evaluates possible moves to determine the best move for the maximizing player, considering a time limit and depth limit. The getPossibleMoves function generates all potential moves for a given player by checking empty cells on the board. The applyMove function applies a move to a temporary board to simulate the game state. Alpha-beta pruning is used to cut off branches in the search tree that won't affect the final decision, improving efficiency. The equivalent is done for the minimizing player.

```
if (isMaximizing) {
    int maxEval = std::numeric_limits<int>::min();
    // Iterate through all possible moves for the maximizing player
    for (const auto &move : getPossibleMoves(board, -1)) {
        if (board.isValidMove(move.first, move.second, -1)) {
            Board tempBoard = board;
            applyMove(tempBoard, move.first, move.second, -1);
            // Recursively call minimax for the minimizing player
            int eval = minimax(tempBoard, depth - 1, false, alpha, beta, startTim
            maxEval = std::max(maxEval, eval);
            alpha = std::max(alpha, eval);
            if (beta <= alpha) {
                break; // Beta cut-off
            }
        }
    }
    return maxEval;
} else {
    int minEval = std::numeric_limits<int>::max();
    // Iterate through all possible moves for the minimizing player
    for (const auto &move : getPossibleMoves(board, 1)) {
        if (board.isValidMove(move.first, move.second, 1)) {
            Board tempBoard = board;
            applyMove(tempBoard, move.first, move.second, 1);
            // Recursively call minimax for the maximizing player
```

```
                int eval = minimax(tempBoard, depth - 1, true, alpha, beta, startTime
                minEval = std::min(minEval, eval);
                beta = std::min(beta, eval);
                if (beta <= alpha) {
                    break; // Alpha cut-off
                }
            }
        }
        return minEval;
    }
}
```

Now for the evaluate function different heuristic techniques can be applied. For most board sizes we simply sum over all cells and for an 8x8 we each cell has a weight taken from a heuristic matrix. This matrix is taken from the open source game Reversi [6].

```
int AI::evaluateBoard(const Board &board) {
    // Heuristic values for an 8x8 board
    static const int heuristic_8[8][8] = {
        { 410,  23,  13,   8,   8,  13,  23, 410 },
        {  23, -75, -22, -51, -51, -22, -75,  23 },
        {  13, -22,  41,   3,   3,  41, -22,  13 },
        {   8, -51,   3, -87, -87,   3, -51,   8 },
        {   8, -51,   3, -87, -87,   3, -51,   8 },
        {  13, -22,  41,   3,   3,  41, -22,  13 },
        {  23, -75, -22, -51, -51, -22, -75,  23 },
        { 410,  23,  13,   8,   8,  13,  23, 410 }
    };

    int score = 0;
    int width = board.getWidth();
    int height = board.getHeight();

    // Evaluate the board based on the heuristic values if it's an 8x8 board
    if (width == 8 && height == 8) {
        for (int x = 0; x < width; ++x) {
            for (int y = 0; y < height; ++y) {
                score += board.getCell(x, y) * heuristic_8[x][y];
            }
        }
    } else {
        // Otherwise, simply sum up the cell values
        for (int x = 0; x < width; ++x) {
            for (int y = 0; y < height; ++y) {
                score += board.getCell(x, y);
            }
        }
```

[6] https://gitlab.gnome.org/GNOME/iagno

```
    }

    return score;
}
```