Project 2 - Connect 4 Using MinMax-AB With Independent Evaluation Functions

CS 4346 Dr.Moonis Ali

By Robert Jones

Group: John Courtrigth, Brittany Hale, Robert Jones

Problem Description

This project focuses on implementing the MINMAX-A-B algorithm in C++ to create an intelligent program capable of playing the Connect Four game. Connect Four is a two-player strategy game where players take turns occupying a valid space in a 6x7 tabular board. The objective is to connect four of one's own indicators in a row—either horizontally, vertically, or diagonally—before the opponent does. The nature of the game makes it a classic candidate for AI algorithms that simulate decision-making in competitive environments.

To simulate competitive gameplay between intelligent agents, the team has implemented the MINMAX-A-B algorithm, which enhances the basic minimax algorithm by pruning branches that cannot possibly affect the final decision. This significantly reduces the number of nodes that need to be evaluated, optimizing the algorithm's performance. Each AI player evaluates possible moves using a custom evaluation function, which attempts to score board positions based on their favorability. These evaluation functions are key to guiding the AI's decision-making and differ from one team member to another to allow for performance comparison.

The game is played by having one AI agent act as the maximizing player and another as the minimizing player. Each agent uses a different evaluation function and operates at a specified cutoff depth in the game tree. For our team of three students, nine separate runs are performed to test every pairing of evaluation functions at various cutoff depths—specifically 2, 4, and 8. These runs allow for the analysis of AI behavior under different configurations, simulating matches like Max (EV#1, depth 2) versus Min (EV#2, depth 2), and so on. By comparing each outcome, we can assess how well each evaluation function performs relative to others across different search depths.

The final analysis includes a detailed comparison of key performance metrics: the number of nodes generated and expanded, execution time, and win/loss statistics. These metrics provide insight into which evaluation functions are most efficient and effective at various depths. Ultimately, this project not only reinforces algorithmic thinking and performance analysis but also deepens understanding of how heuristic evaluation and search strategies affect AI gameplay in competitive settings.

Domain

The core task is to simulate AI-versus-AI gameplay using unique evaluation functions designed by individual team members. Each evaluation function is used in multiple matchups with varying search cutoff depths (2, 4, and 8), resulting in nine combinations to be tested per team of three.

Beyond coding, the project also emphasizes analytical and reporting skills. Each team member is responsible for implementing their own evaluation function, running the full battery of test matches, compiling performance metrics, and analyzing the results to draw meaningful conclusions about the efficacy of different evaluation strategies. The final deliverable aims to demonstrate a deep understanding of adversarial search techniques in AI, the impact of evaluation heuristics, and the trade-offs involved in computational.

Methodologies

This recursive algorithm simulates all possible future game states up to a specified depth and chooses the optimal move based on an evaluation function. It begins by checking if the current node is in a terminal state (a win, loss, or draw) or if the depth cutoff has been reached. In such cases, it returns a static evaluation score of the board. If not, the function iterates over all valid columns where a piece can be dropped and recursively evaluates each possible board state resulting from a move. The algorithm uses alpha and beta thresholds to prune branches of the game tree that cannot influence the final decision, significantly reducing the number of nodes evaluated and thereby improving efficiency.

In the context of Connect Four, this implementation ensures that the AI evaluates and chooses its moves intelligently by simulating both its own and its opponent's future actions. The recursive structure mimics the alternating turns of the game. The algorithm evaluates how favorable a move is based on the provided evaluation function, which can be customized by each team member. By comparing scores and pruning suboptimal branches, the AI efficiently identifies the best column to play in, simulating a competitive and strategic decision-making process. This implementation directly supports the goals of the project by enabling comparative testing of different evaluation functions and cutoff depths through a controlled and optimized game-playing routine.

Source Code Implementation

Player Representation and Turn Order

In the game, players are represented using constant integers: `PLAYER_1 = 1` and `PLAYER_2 = 2`. To determine who goes first, the program generates a random integer. If the number is even (`%2 == 0`), Player 1 starts; otherwise, Player 2 begins the game. For each move, the program verifies whether a chosen column is valid by checking if it is not already full. If valid, it identifies the next available row within that column where a piece can be placed. After each turn, the board is evaluated to check for a winning condition based on the current player's pieces. Additionally, the game checks for a draw—this occurs when no valid moves remain and no player has achieved a winning configuration.

Minimax Algorithm with Alpha-Beta Pruning

This program employs the Minimax algorithm, a common decision-making strategy used in two-player adversarial games like Connect 4. The goal of Minimax is to simulate all possible moves and counter-moves in order to select the optimal move for the current player, under the assumption that the opponent also plays optimally. Due to the high branching factor (b) and depth (d) of the game tree, the time complexity of Minimax is exponential—O(b^d). As the search depth increases, the algorithm becomes significantly more computationally expensive.

To address this, the program incorporates alpha-beta pruning, an optimization that eliminates branches in the search tree that do not need to be evaluated because they cannot affect the final decision. Alpha represents the best score that the maximizing player is assured, while beta represents the best score the minimizing player can secure. The Minimax function alternates between maximizing and minimizing roles, simulating moves for each player in turn.

The recursion stops under two conditions: (1) the maximum search depth is reached, or (2) the game has entered a terminal state, such as a win, loss, or draw. If neither condition is met, the algorithm explores all valid columns. For each move, it simulates the player's action, updates the board, and recursively evaluates the resulting game state.

- When it's the maximizing player's turn, the algorithm initializes the best score to negative infinity. For each move, it calculates a score using a chosen evaluation function, updates the best score if necessary, and adjusts the alpha value. If alpha becomes greater than or equal to beta, the branch is pruned.

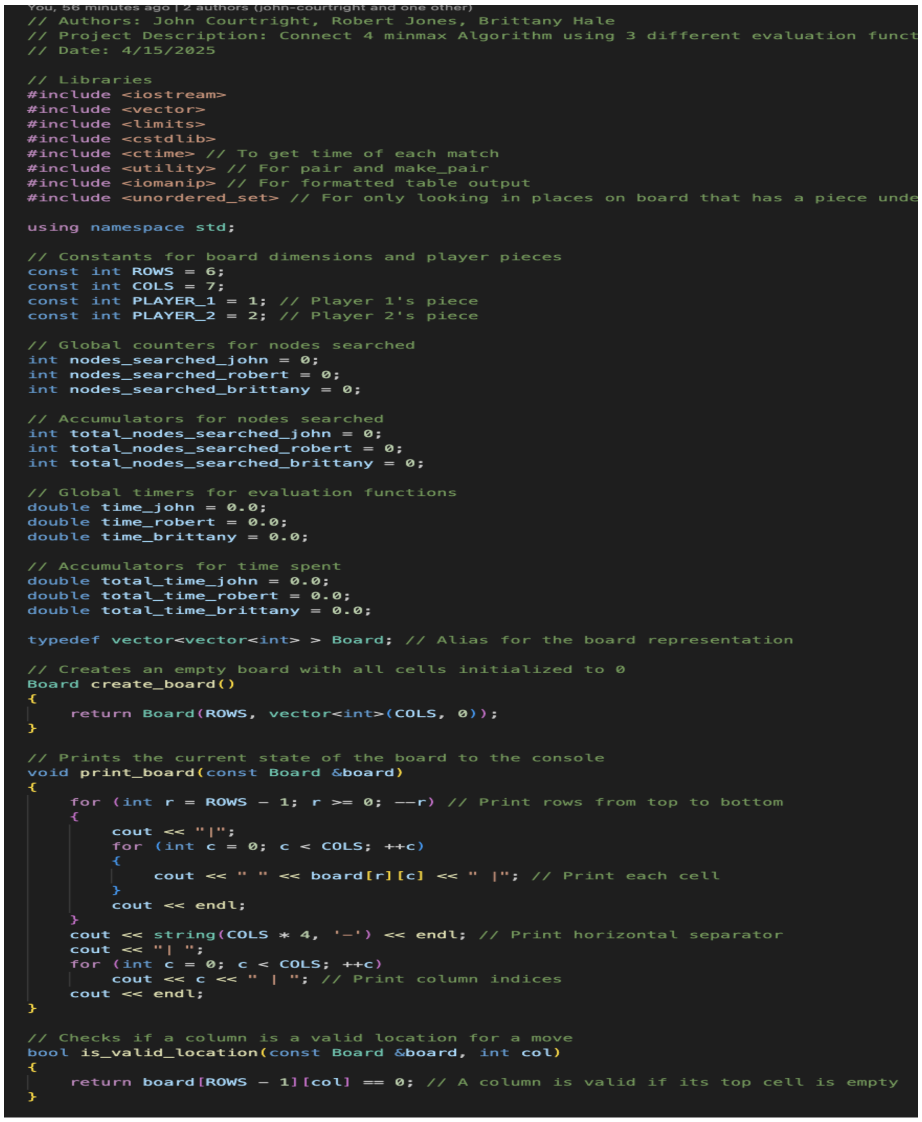
- When it's the minimizing player's turn, the best score is initialized to positive infinity. The same process is followed, but with beta being updated and pruning occurring when alpha ≥ beta.

After evaluating all potential moves (or pruning where appropriate), the algorithm returns the best column and its associated score.

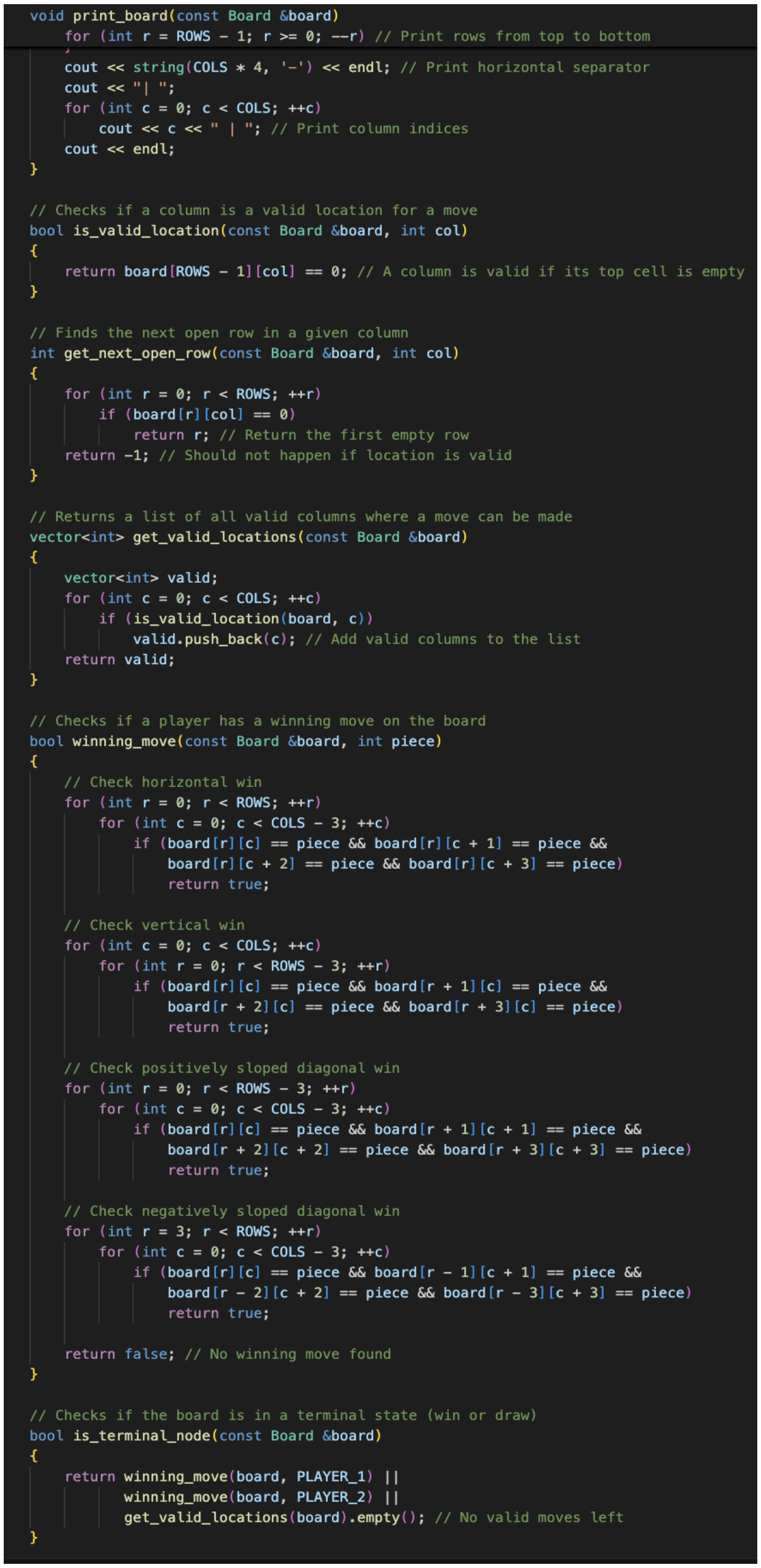
Evaluation Functions

At the heart of the Minimax decision process are the evaluation functions, which assess the desirability of a given board state. These functions encapsulate strategic heuristics that assign a numerical score to each board. The scoring takes into account factors such as the number of aligned pieces, the potential to form future winning moves, and control of key positions (like the center columns). Each evaluation function represents a distinct strategy and influences how the AI prioritizes its moves. During the recursive search, these functions are called when the base case is reached, and their output is used to inform the Minimax algorithm's decision-making.

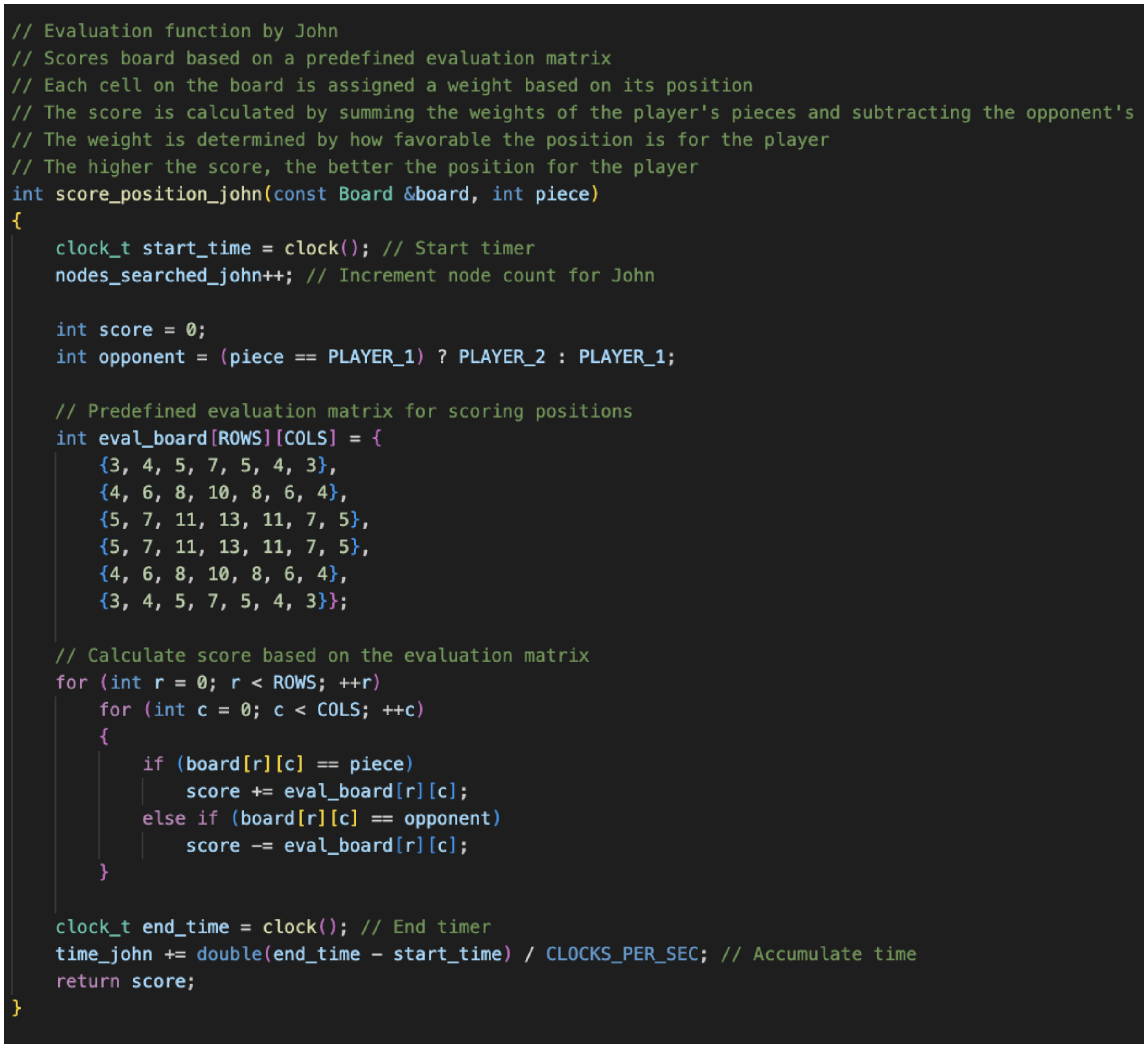
Source Code



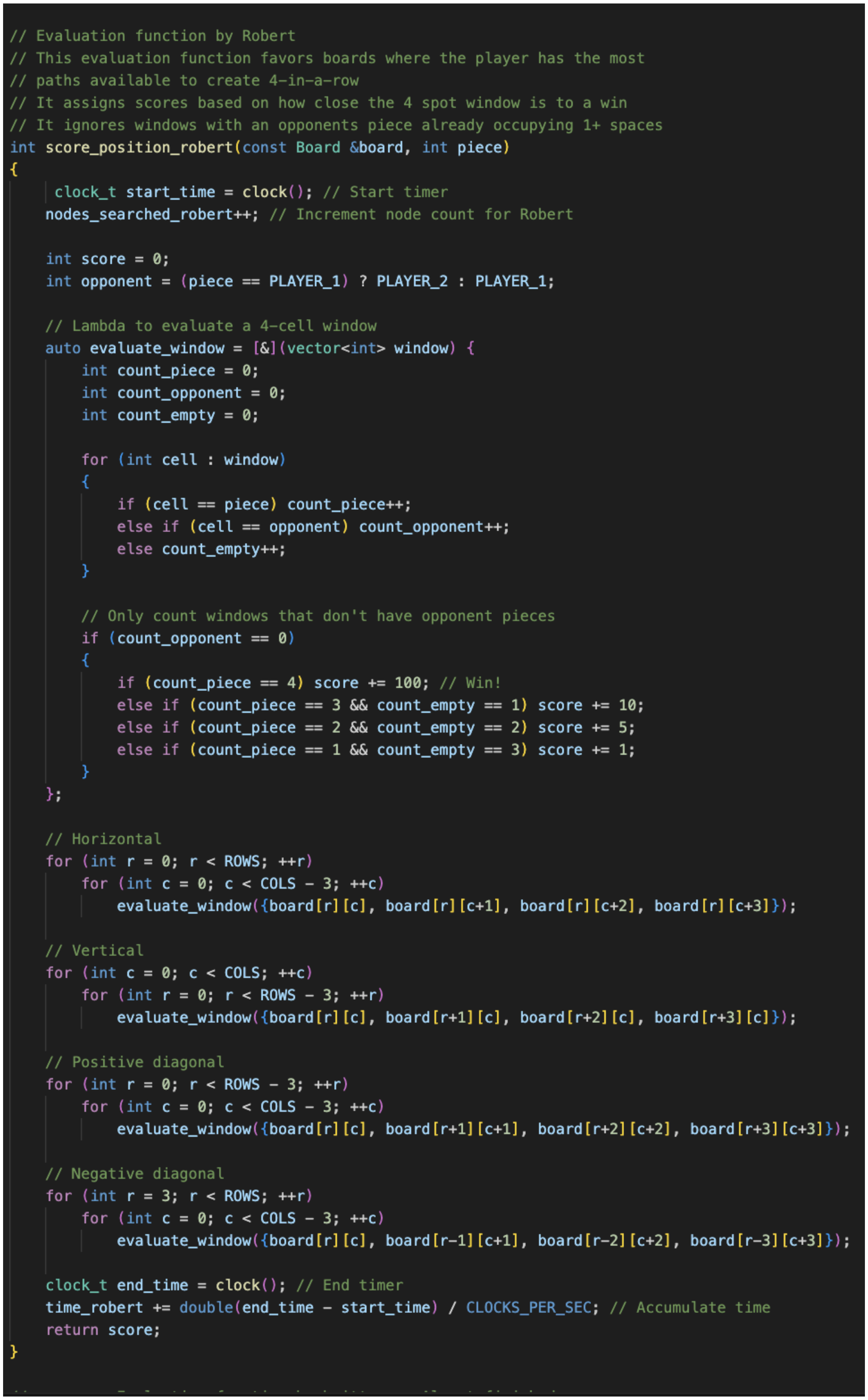
```
You, 56 minutes ago | 2 authors (john-courtright and one other)
// Authors: John Courtright, Robert Jones, Brittany Hale
// Project Description: Connect 4 minmax Algorithm using 3 different evaluation funct
// Date: 4/15/2025

// Libraries
#include <iostream>
#include <vector>
#include <limits>
#include <cstdlib>
#include <ctime> // To get time of each match
#include <utility> // For pair and make_pair
#include <iomanip> // For formatted table output
#include <unordered_set> // For only looking in places on board that has a piece unde

using namespace std;

// Constants for board dimensions and player pieces
const int ROWS = 6;
const int COLS = 7;
const int PLAYER_1 = 1; // Player 1's piece
const int PLAYER_2 = 2; // Player 2's piece

// Global counters for nodes searched
int nodes_searched_john = 0;
int nodes_searched_robert = 0;
int nodes_searched_brittany = 0;

// Accumulators for nodes searched
int total_nodes_searched_john = 0;
int total_nodes_searched_robert = 0;
int total_nodes_searched_brittany = 0;

// Global timers for evaluation functions
double time_john = 0.0;
double time_robert = 0.0;
double time_brittany = 0.0;

// Accumulators for time spent
double total_time_john = 0.0;
double total_time_robert = 0.0;
double total_time_brittany = 0.0;

typedef vector<vector<int> > Board; // Alias for the board representation

// Creates an empty board with all cells initialized to 0
Board create_board()
{
    return Board(ROWS, vector<int>(COLS, 0));
}

// Prints the current state of the board to the console
void print_board(const Board &board)
{
    for (int r = ROWS - 1; r >= 0; --r) // Print rows from top to bottom
    {
        cout << "|";
        for (int c = 0; c < COLS; ++c)
        {
            cout << " " << board[r][c] << " |"; // Print each cell
        }
        cout << endl;
    }
    cout << string(COLS * 4, '-') << endl; // Print horizontal separator
    cout << "| ";
    for (int c = 0; c < COLS; ++c)
        cout << c << " | "; // Print column indices
    cout << endl;
}

// Checks if a column is a valid location for a move
bool is_valid_location(const Board &board, int col)
{
    return board[ROWS - 1][col] == 0; // A column is valid if its top cell is empty
}
```



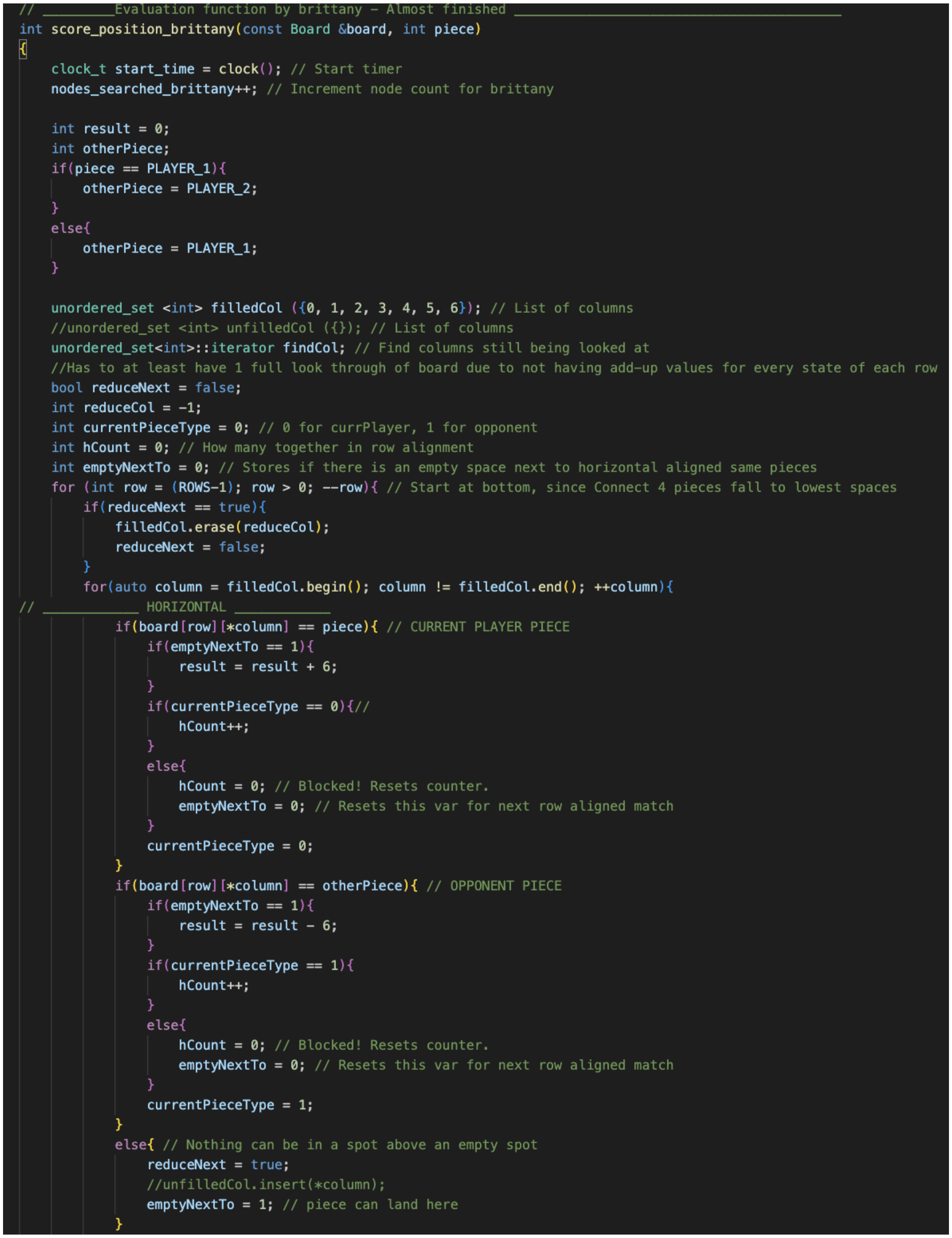
```cpp
void print_board(const Board &board)
    for (int r = ROWS - 1; r >= 0; --r) // Print rows from top to bottom
    cout << string(COLS * 4, '-') << endl; // Print horizontal separator
    cout << "| ";
    for (int c = 0; c < COLS; ++c)
        cout << c << " | "; // Print column indices
    cout << endl;
}

// Checks if a column is a valid location for a move
bool is_valid_location(const Board &board, int col)
{
    return board[ROWS - 1][col] == 0; // A column is valid if its top cell is empty
}

// Finds the next open row in a given column
int get_next_open_row(const Board &board, int col)
{
    for (int r = 0; r < ROWS; ++r)
        if (board[r][col] == 0)
            return r; // Return the first empty row
    return -1; // Should not happen if location is valid
}

// Returns a list of all valid columns where a move can be made
vector<int> get_valid_locations(const Board &board)
{
    vector<int> valid;
    for (int c = 0; c < COLS; ++c)
        if (is_valid_location(board, c))
            valid.push_back(c); // Add valid columns to the list
    return valid;
}

// Checks if a player has a winning move on the board
bool winning_move(const Board &board, int piece)
{
    // Check horizontal win
    for (int r = 0; r < ROWS; ++r)
        for (int c = 0; c < COLS - 3; ++c)
            if (board[r][c] == piece && board[r][c + 1] == piece &&
                board[r][c + 2] == piece && board[r][c + 3] == piece)
                return true;

    // Check vertical win
    for (int c = 0; c < COLS; ++c)
        for (int r = 0; r < ROWS - 3; ++r)
            if (board[r][c] == piece && board[r + 1][c] == piece &&
                board[r + 2][c] == piece && board[r + 3][c] == piece)
                return true;

    // Check positively sloped diagonal win
    for (int r = 0; r < ROWS - 3; ++r)
        for (int c = 0; c < COLS - 3; ++c)
            if (board[r][c] == piece && board[r + 1][c + 1] == piece &&
                board[r + 2][c + 2] == piece && board[r + 3][c + 3] == piece)
                return true;

    // Check negatively sloped diagonal win
    for (int r = 3; r < ROWS; ++r)
        for (int c = 0; c < COLS - 3; ++c)
            if (board[r][c] == piece && board[r - 1][c + 1] == piece &&
                board[r - 2][c + 2] == piece && board[r - 3][c + 3] == piece)
                return true;

    return false; // No winning move found
}

// Checks if the board is in a terminal state (win or draw)
bool is_terminal_node(const Board &board)
{
    return winning_move(board, PLAYER_1) ||
           winning_move(board, PLAYER_2) ||
           get_valid_locations(board).empty(); // No valid moves left
}
```



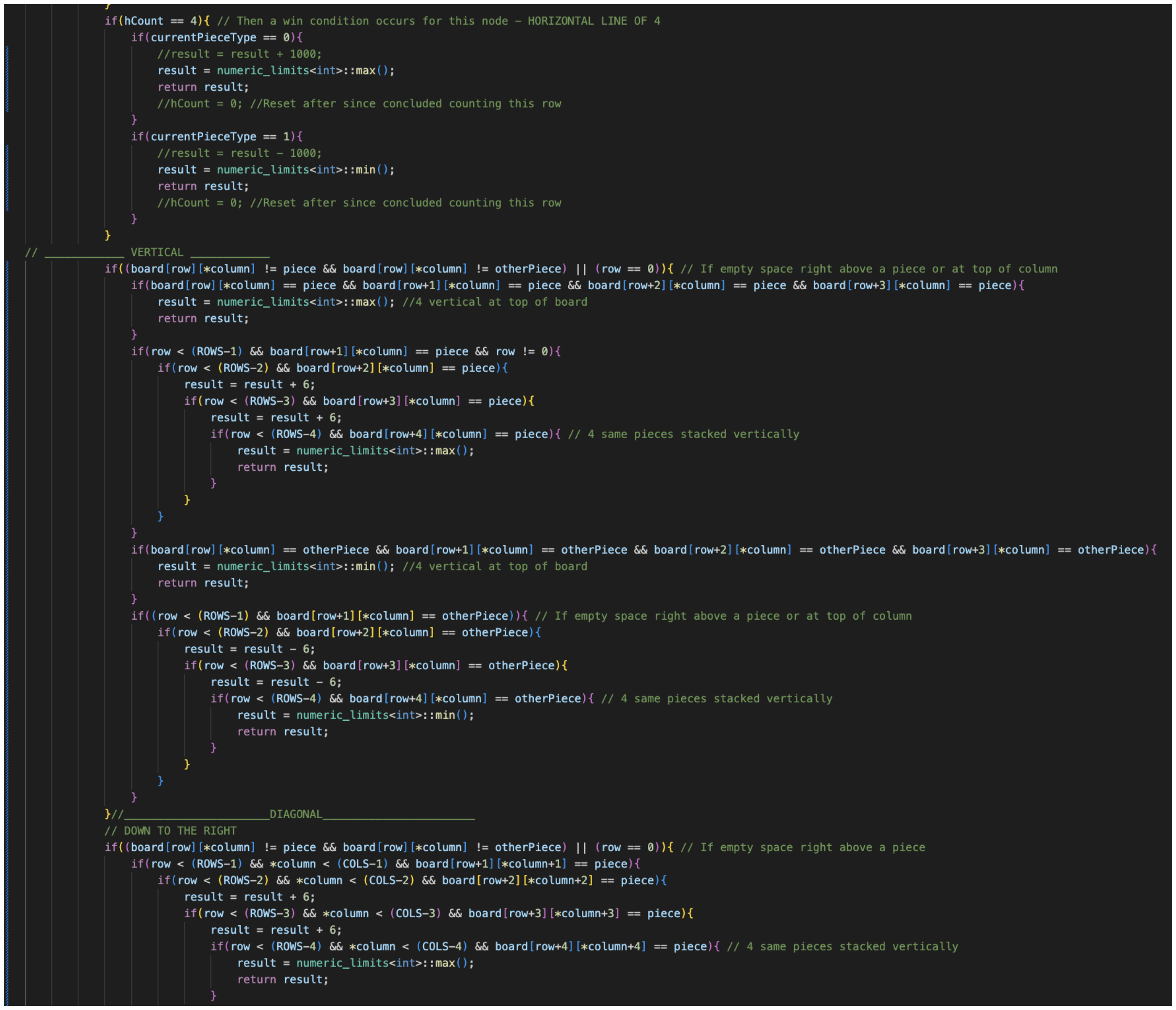
```cpp
// Evaluation function by John
// Scores board based on a predefined evaluation matrix
// Each cell on the board is assigned a weight based on its position
// The score is calculated by summing the weights of the player's pieces and subtracting the opponent's
// The weight is determined by how favorable the position is for the player
// The higher the score, the better the position for the player
int score_position_john(const Board &board, int piece)
{
    clock_t start_time = clock(); // Start timer
    nodes_searched_john++; // Increment node count for John

    int score = 0;
    int opponent = (piece == PLAYER_1) ? PLAYER_2 : PLAYER_1;

    // Predefined evaluation matrix for scoring positions
    int eval_board[ROWS][COLS] = {
        {3, 4, 5, 7, 5, 4, 3},
        {4, 6, 8, 10, 8, 6, 4},
        {5, 7, 11, 13, 11, 7, 5},
        {5, 7, 11, 13, 11, 7, 5},
        {4, 6, 8, 10, 8, 6, 4},
        {3, 4, 5, 7, 5, 4, 3}};

    // Calculate score based on the evaluation matrix
    for (int r = 0; r < ROWS; ++r)
        for (int c = 0; c < COLS; ++c)
        {
            if (board[r][c] == piece)
                score += eval_board[r][c];
            else if (board[r][c] == opponent)
                score -= eval_board[r][c];
        }

    clock_t end_time = clock(); // End timer
    time_john += double(end_time - start_time) / CLOCKS_PER_SEC; // Accumulate time
    return score;
}
```



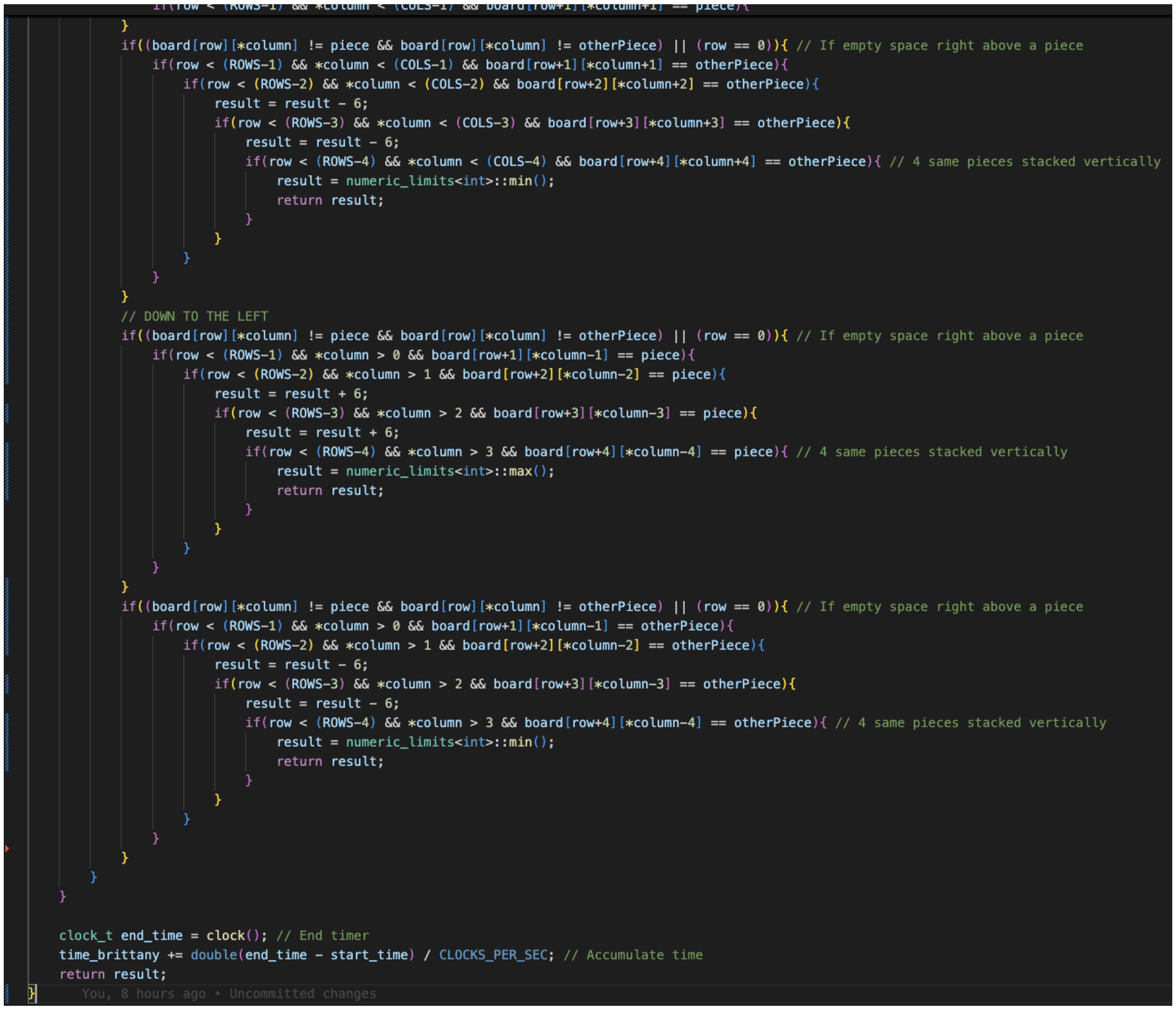
```cpp
// Evaluation function by Robert
// This evaluation function favors boards where the player has the most
// paths available to create 4-in-a-row
// It assigns scores based on how close the 4 spot window is to a win
// It ignores windows with an opponents piece already occupying 1+ spaces
int score_position_robert(const Board &board, int piece)
{
    clock_t start_time = clock(); // Start timer
    nodes_searched_robert++; // Increment node count for Robert

    int score = 0;
    int opponent = (piece == PLAYER_1) ? PLAYER_2 : PLAYER_1;

    // Lambda to evaluate a 4-cell window
    auto evaluate_window = [&](vector<int> window) {
        int count_piece = 0;
        int count_opponent = 0;
        int count_empty = 0;

        for (int cell : window)
        {
            if (cell == piece) count_piece++;
            else if (cell == opponent) count_opponent++;
            else count_empty++;
        }

        // Only count windows that don't have opponent pieces
        if (count_opponent == 0)
        {
            if (count_piece == 4) score += 100; // Win!
            else if (count_piece == 3 && count_empty == 1) score += 10;
            else if (count_piece == 2 && count_empty == 2) score += 5;
            else if (count_piece == 1 && count_empty == 3) score += 1;
        }
    };

    // Horizontal
    for (int r = 0; r < ROWS; ++r)
        for (int c = 0; c < COLS - 3; ++c)
            evaluate_window({board[r][c], board[r][c+1], board[r][c+2], board[r][c+3]});

    // Vertical
    for (int c = 0; c < COLS; ++c)
        for (int r = 0; r < ROWS - 3; ++r)
            evaluate_window({board[r][c], board[r+1][c], board[r+2][c], board[r+3][c]});

    // Positive diagonal
    for (int r = 0; r < ROWS - 3; ++r)
        for (int c = 0; c < COLS - 3; ++c)
            evaluate_window({board[r][c], board[r+1][c+1], board[r+2][c+2], board[r+3][c+3]});

    // Negative diagonal
    for (int r = 3; r < ROWS; ++r)
        for (int c = 0; c < COLS - 3; ++c)
            evaluate_window({board[r][c], board[r-1][c+1], board[r-2][c+2], board[r-3][c+3]});

    clock_t end_time = clock(); // End timer
    time_robert += double(end_time - start_time) / CLOCKS_PER_SEC; // Accumulate time
    return score;
}
```



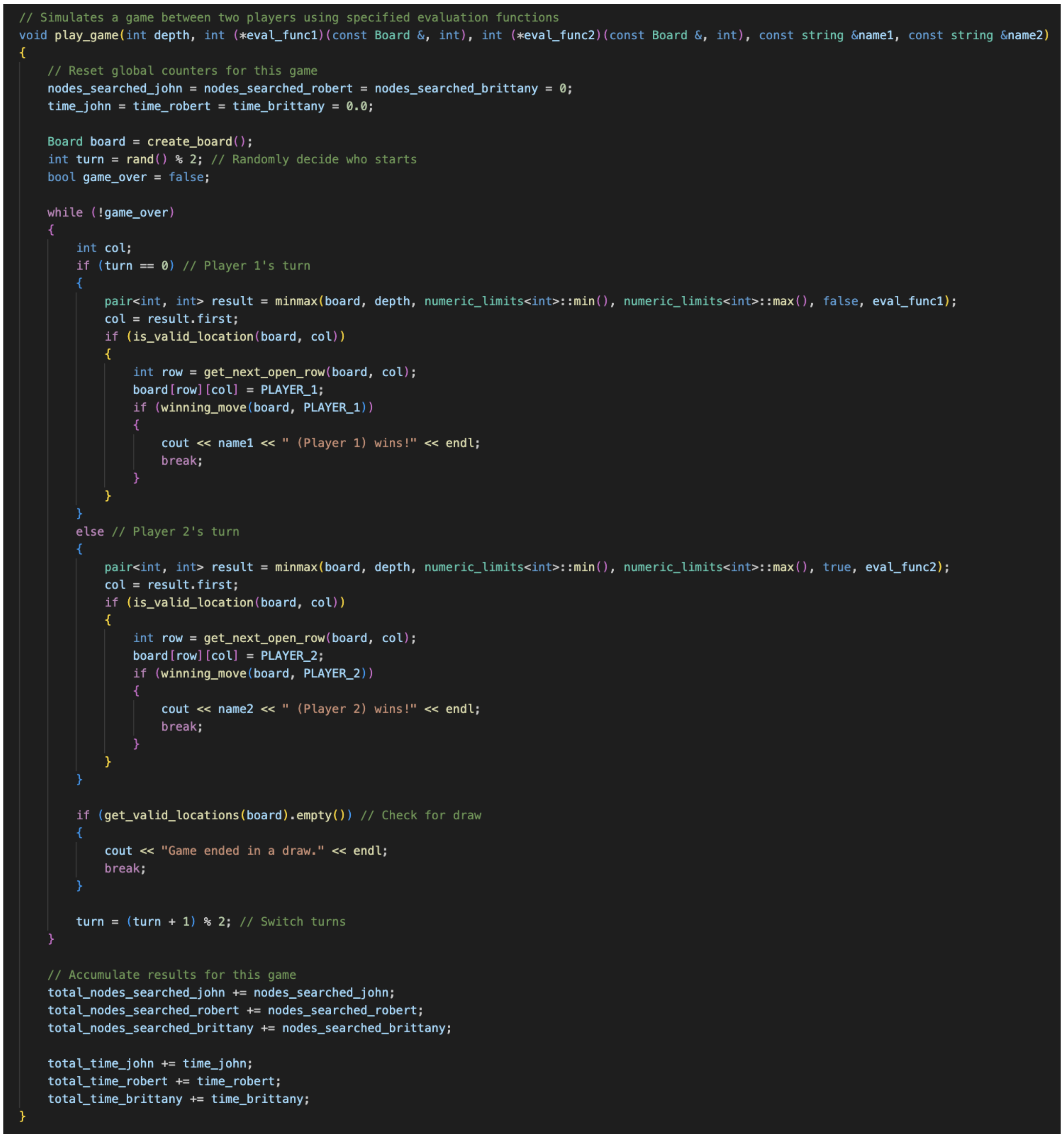
```
// _________Evaluation function by brittany - Almost finished ________________________________________
int score_position_brittany(const Board &board, int piece)
{
    clock_t start_time = clock(); // Start timer
    nodes_searched_brittany++; // Increment node count for brittany

    int result = 0;
    int otherPiece;
    if(piece == PLAYER_1){
        otherPiece = PLAYER_2;
    }
    else{
        otherPiece = PLAYER_1;
    }

    unordered_set <int> filledCol ({0, 1, 2, 3, 4, 5, 6}); // List of columns
    //unordered_set <int> unfilledCol ({}); // List of columns
    unordered_set<int>::iterator findCol; // Find columns still being looked at
    //Has to at least have 1 full look through of board due to not having add-up values for every state of each row
    bool reduceNext = false;
    int reduceCol = -1;
    int currentPieceType = 0; // 0 for currPlayer, 1 for opponent
    int hCount = 0; // How many together in row alignment
    int emptyNextTo = 0; // Stores if there is an empty space next to horizontal aligned same pieces
    for (int row = (ROWS-1); row > 0; --row){ // Start at bottom, since Connect 4 pieces fall to lowest spaces
        if(reduceNext == true){
            filledCol.erase(reduceCol);
            reduceNext = false;
        }
        for(auto column = filledCol.begin(); column != filledCol.end(); ++column){
// ____________ HORIZONTAL ____________
            if(board[row][*column] == piece){ // CURRENT PLAYER PIECE
                if(emptyNextTo == 1){
                    result = result + 6;
                }
                if(currentPieceType == 0){//
                    hCount++;
                }
                else{
                    hCount = 0; // Blocked! Resets counter.
                    emptyNextTo = 0; // Resets this var for next row aligned match
                }
                currentPieceType = 0;
            }
            if(board[row][*column] == otherPiece){ // OPPONENT PIECE
                if(emptyNextTo == 1){
                    result = result - 6;
                }
                if(currentPieceType == 1){
                    hCount++;
                }
                else{
                    hCount = 0; // Blocked! Resets counter.
                    emptyNextTo = 0; // Resets this var for next row aligned match
                }
                currentPieceType = 1;
            }
            else{ // Nothing can be in a spot above an empty spot
                reduceNext = true;
                //unfilledCol.insert(*column);
                emptyNextTo = 1; // piece can land here
            }
```



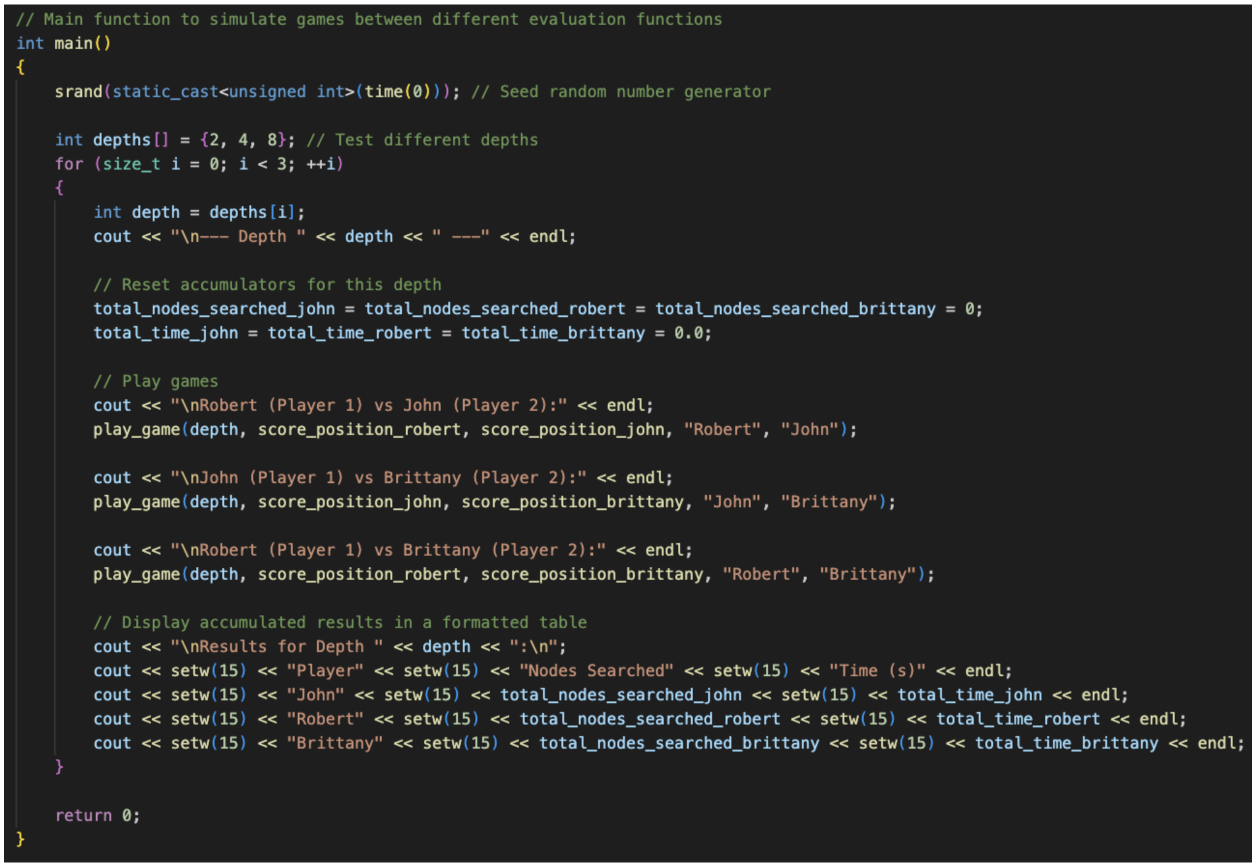
```
            if(hCount == 4){ // Then a win condition occurs for this node - HORIZONTAL LINE OF 4
                if(currentPieceType == 0){
                    //result = result + 1000;
                    result = numeric_limits<int>::max();
                    return result;
                    //hCount = 0; //Reset after since concluded counting this row
                }
                if(currentPieceType == 1){
                    //result = result - 1000;
                    result = numeric_limits<int>::min();
                    return result;
                    //hCount = 0; //Reset after since concluded counting this row
                }
            }
//  ____________ VERTICAL ____________
            if((board[row][*column] != piece && board[row][*column] != otherPiece) || (row == 0)){ // If empty space right above a piece or at top of column
                if(board[row][*column] == piece && board[row+1][*column] == piece && board[row+2][*column] == piece && board[row+3][*column] == piece){
                    result = numeric_limits<int>::max(); //4 vertical at top of board
                    return result;
                }
                if(row < (ROWS-1) && board[row+1][*column] == piece && row != 0){
                    if(row < (ROWS-2) && board[row+2][*column] == piece){
                        result = result + 6;
                        if(row < (ROWS-3) && board[row+3][*column] == piece){
                            result = result + 6;
                            if(row < (ROWS-4) && board[row+4][*column] == piece){ // 4 same pieces stacked vertically
                                result = numeric_limits<int>::max();
                                return result;
                            }
                        }
                    }
                }
                if(board[row][*column] == otherPiece && board[row+1][*column] == otherPiece && board[row+2][*column] == otherPiece && board[row+3][*column] == otherPiece){
                    result = numeric_limits<int>::min(); //4 vertical at top of board
                    return result;
                }
                if((row < (ROWS-1) && board[row+1][*column] == otherPiece)){ // If empty space right above a piece or at top of column
                    if(row < (ROWS-2) && board[row+2][*column] == otherPiece){
                        result = result - 6;
                        if(row < (ROWS-3) && board[row+3][*column] == otherPiece){
                            result = result - 6;
                            if(row < (ROWS-4) && board[row+4][*column] == otherPiece){ // 4 same pieces stacked vertically
                                result = numeric_limits<int>::min();
                                return result;
                            }
                        }
                    }
                }
            }//______________________DIAGONAL_______________________
            // DOWN TO THE RIGHT
            if((board[row][*column] != piece && board[row][*column] != otherPiece) || (row == 0)){ // If empty space right above a piece
                if(row < (ROWS-1) && *column < (COLS-1) && board[row+1][*column+1] == piece){
                    if(row < (ROWS-2) && *column < (COLS-2) && board[row+2][*column+2] == piece){
                        result = result + 6;
                        if(row < (ROWS-3) && *column < (COLS-3) && board[row+3][*column+3] == piece){
                            result = result + 6;
                            if(row < (ROWS-4) && *column < (COLS-4) && board[row+4][*column+4] == piece){ // 4 same pieces stacked vertically
                                result = numeric_limits<int>::max();
                                return result;
                            }
```



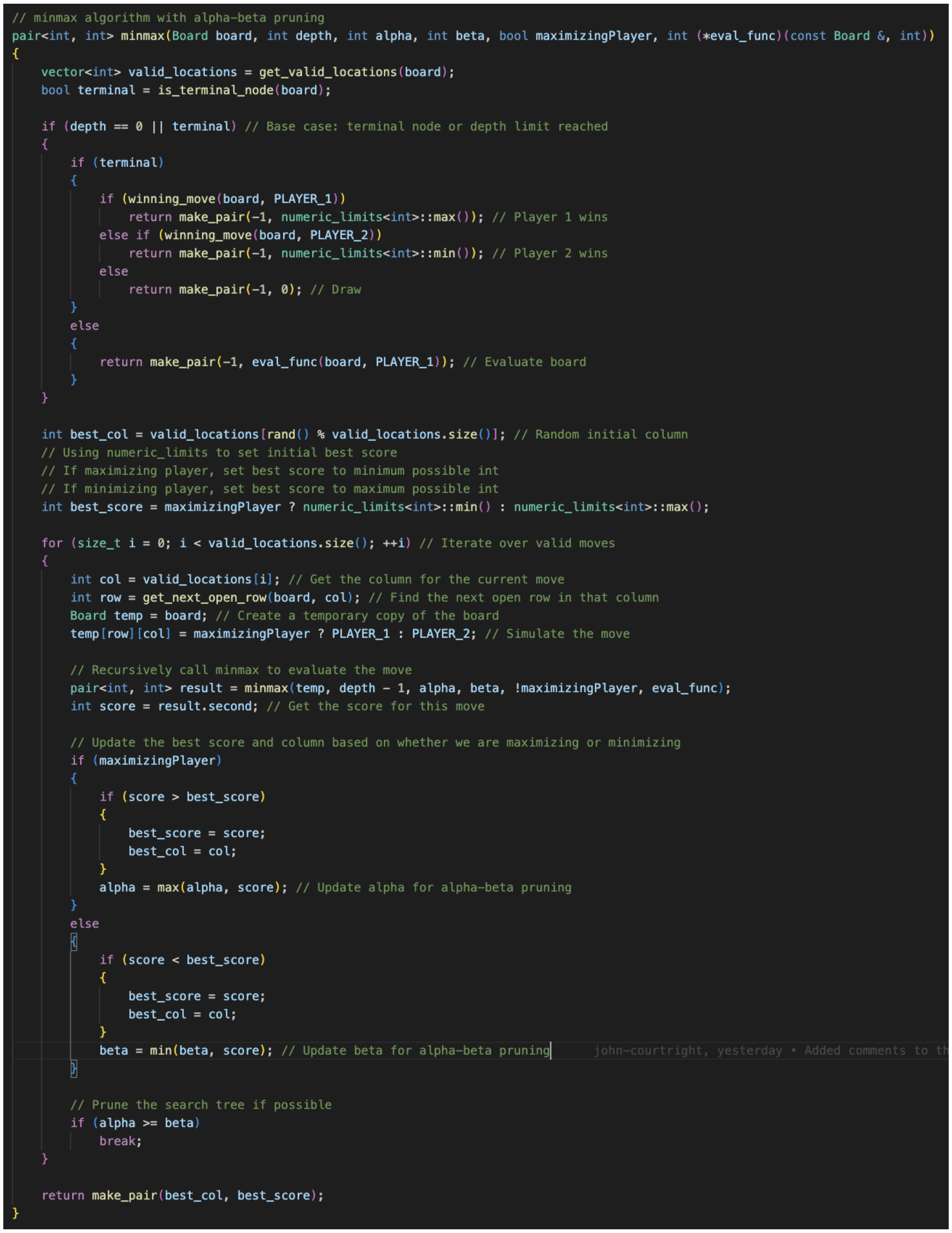
```
                if(row < (ROWS-1) && *column < (COLS-1) && board[row+1][*column+1] == piece){
            }
            if((board[row][*column] != piece && board[row][*column] != otherPiece) || (row == 0)){ // If empty space right above a piece
                if(row < (ROWS-1) && *column < (COLS-1) && board[row+1][*column+1] == otherPiece){
                    if(row < (ROWS-2) && *column < (COLS-2) && board[row+2][*column+2] == otherPiece){
                        result = result - 6;
                        if(row < (ROWS-3) && *column < (COLS-3) && board[row+3][*column+3] == otherPiece){
                            result = result - 6;
                            if(row < (ROWS-4) && *column < (COLS-4) && board[row+4][*column+4] == otherPiece){ // 4 same pieces stacked vertically
                                result = numeric_limits<int>::min();
                                return result;
                            }
                        }
                    }
                }
            }
            // DOWN TO THE LEFT
            if((board[row][*column] != piece && board[row][*column] != otherPiece) || (row == 0)){ // If empty space right above a piece
                if(row < (ROWS-1) && *column > 0 && board[row+1][*column-1] == piece){
                    if(row < (ROWS-2) && *column > 1 && board[row+2][*column-2] == piece){
                        result = result + 6;
                        if(row < (ROWS-3) && *column > 2 && board[row+3][*column-3] == piece){
                            result = result + 6;
                            if(row < (ROWS-4) && *column > 3 && board[row+4][*column-4] == piece){ // 4 same pieces stacked vertically
                                result = numeric_limits<int>::max();
                                return result;
                            }
                        }
                    }
                }
            }
            if((board[row][*column] != piece && board[row][*column] != otherPiece) || (row == 0)){ // If empty space right above a piece
                if(row < (ROWS-1) && *column > 0 && board[row+1][*column-1] == otherPiece){
                    if(row < (ROWS-2) && *column > 1 && board[row+2][*column-2] == otherPiece){
                        result = result - 6;
                        if(row < (ROWS-3) && *column > 2 && board[row+3][*column-3] == otherPiece){
                            result = result - 6;
                            if(row < (ROWS-4) && *column > 3 && board[row+4][*column-4] == otherPiece){ // 4 same pieces stacked vertically
                                result = numeric_limits<int>::min();
                                return result;
                            }
                        }
                    }
                }
            }
        }
    }

    clock_t end_time = clock(); // End timer
    time_brittany += double(end_time - start_time) / CLOCKS_PER_SEC; // Accumulate time
    return result;
}    You, 8 hours ago • Uncommitted changes
```



```cpp
// Simulates a game between two players using specified evaluation functions
void play_game(int depth, int (*eval_func1)(const Board &, int), int (*eval_func2)(const Board &, int), const string &name1, const string &name2)
{
    // Reset global counters for this game
    nodes_searched_john = nodes_searched_robert = nodes_searched_brittany = 0;
    time_john = time_robert = time_brittany = 0.0;

    Board board = create_board();
    int turn = rand() % 2; // Randomly decide who starts
    bool game_over = false;

    while (!game_over)
    {
        int col;
        if (turn == 0) // Player 1's turn
        {
            pair<int, int> result = minmax(board, depth, numeric_limits<int>::min(), numeric_limits<int>::max(), false, eval_func1);
            col = result.first;
            if (is_valid_location(board, col))
            {
                int row = get_next_open_row(board, col);
                board[row][col] = PLAYER_1;
                if (winning_move(board, PLAYER_1))
                {
                    cout << name1 << " (Player 1) wins!" << endl;
                    break;
                }
            }
        }
        else // Player 2's turn
        {
            pair<int, int> result = minmax(board, depth, numeric_limits<int>::min(), numeric_limits<int>::max(), true, eval_func2);
            col = result.first;
            if (is_valid_location(board, col))
            {
                int row = get_next_open_row(board, col);
                board[row][col] = PLAYER_2;
                if (winning_move(board, PLAYER_2))
                {
                    cout << name2 << " (Player 2) wins!" << endl;
                    break;
                }
            }
        }

        if (get_valid_locations(board).empty()) // Check for draw
        {
            cout << "Game ended in a draw." << endl;
            break;
        }

        turn = (turn + 1) % 2; // Switch turns
    }

    // Accumulate results for this game
    total_nodes_searched_john += nodes_searched_john;
    total_nodes_searched_robert += nodes_searched_robert;
    total_nodes_searched_brittany += nodes_searched_brittany;

    total_time_john += time_john;
    total_time_robert += time_robert;
    total_time_brittany += time_brittany;
}
```

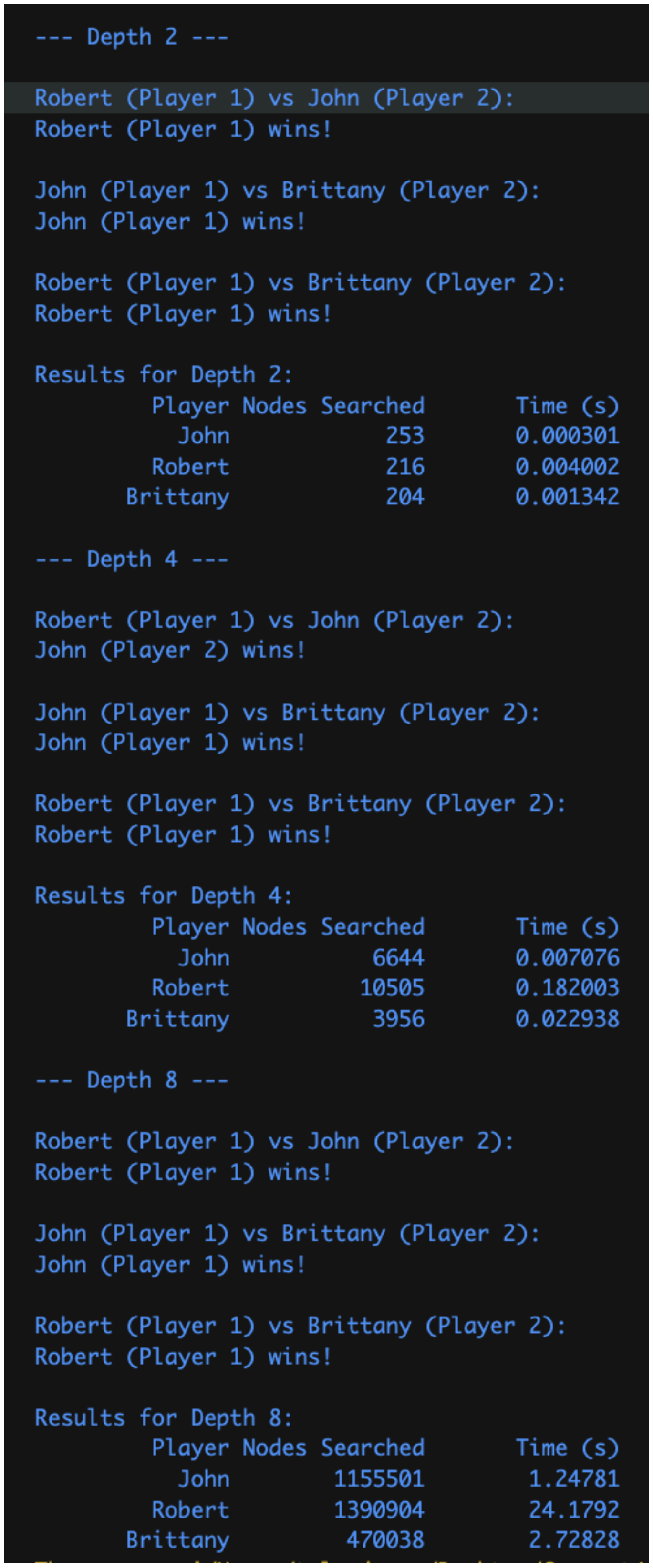


```
// Main function to simulate games between different evaluation functions
int main()
{
    srand(static_cast<unsigned int>(time(0))); // Seed random number generator

    int depths[] = {2, 4, 8}; // Test different depths
    for (size_t i = 0; i < 3; ++i)
    {
        int depth = depths[i];
        cout << "\n--- Depth " << depth << " ---" << endl;

        // Reset accumulators for this depth
        total_nodes_searched_john = total_nodes_searched_robert = total_nodes_searched_brittany = 0;
        total_time_john = total_time_robert = total_time_brittany = 0.0;

        // Play games
        cout << "\nRobert (Player 1) vs John (Player 2):" << endl;
        play_game(depth, score_position_robert, score_position_john, "Robert", "John");

        cout << "\nJohn (Player 1) vs Brittany (Player 2):" << endl;
        play_game(depth, score_position_john, score_position_brittany, "John", "Brittany");

        cout << "\nRobert (Player 1) vs Brittany (Player 2):" << endl;
        play_game(depth, score_position_robert, score_position_brittany, "Robert", "Brittany");

        // Display accumulated results in a formatted table
        cout << "\nResults for Depth " << depth << ":\n";
        cout << setw(15) << "Player" << setw(15) << "Nodes Searched" << setw(15) << "Time (s)" << endl;
        cout << setw(15) << "John" << setw(15) << total_nodes_searched_john << setw(15) << total_time_john << endl;
        cout << setw(15) << "Robert" << setw(15) << total_nodes_searched_robert << setw(15) << total_time_robert << endl;
        cout << setw(15) << "Brittany" << setw(15) << total_nodes_searched_brittany << setw(15) << total_time_brittany << endl;
    }

    return 0;
}
```



```cpp
// minmax algorithm with alpha-beta pruning
pair<int, int> minmax(Board board, int depth, int alpha, int beta, bool maximizingPlayer, int (*eval_func)(const Board &, int))
{
    vector<int> valid_locations = get_valid_locations(board);
    bool terminal = is_terminal_node(board);

    if (depth == 0 || terminal) // Base case: terminal node or depth limit reached
    {
        if (terminal)
        {
            if (winning_move(board, PLAYER_1))
                return make_pair(-1, numeric_limits<int>::max()); // Player 1 wins
            else if (winning_move(board, PLAYER_2))
                return make_pair(-1, numeric_limits<int>::min()); // Player 2 wins
            else
                return make_pair(-1, 0); // Draw
        }
        else
        {
            return make_pair(-1, eval_func(board, PLAYER_1)); // Evaluate board
        }
    }

    int best_col = valid_locations[rand() % valid_locations.size()]; // Random initial column
    // Using numeric_limits to set initial best score
    // If maximizing player, set best score to minimum possible int
    // If minimizing player, set best score to maximum possible int
    int best_score = maximizingPlayer ? numeric_limits<int>::min() : numeric_limits<int>::max();

    for (size_t i = 0; i < valid_locations.size(); ++i) // Iterate over valid moves
    {
        int col = valid_locations[i]; // Get the column for the current move
        int row = get_next_open_row(board, col); // Find the next open row in that column
        Board temp = board; // Create a temporary copy of the board
        temp[row][col] = maximizingPlayer ? PLAYER_1 : PLAYER_2; // Simulate the move

        // Recursively call minmax to evaluate the move
        pair<int, int> result = minmax(temp, depth - 1, alpha, beta, !maximizingPlayer, eval_func);
        int score = result.second; // Get the score for this move

        // Update the best score and column based on whether we are maximizing or minimizing
        if (maximizingPlayer)
        {
            if (score > best_score)
            {
                best_score = score;
                best_col = col;
            }
            alpha = max(alpha, score); // Update alpha for alpha-beta pruning
        }
        else
        {
            if (score < best_score)
            {
                best_score = score;
                best_col = col;
            }
            beta = min(beta, score); // Update beta for alpha-beta pruning      john-courtright, yesterday • Added comments to th
        }

        // Prune the search tree if possible
        if (alpha >= beta)
            break;
    }

    return make_pair(best_col, best_score);
}
```

Program Run



```
--- Depth 2 ---

Robert (Player 1) vs John (Player 2):
Robert (Player 1) wins!

John (Player 1) vs Brittany (Player 2):
John (Player 1) wins!

Robert (Player 1) vs Brittany (Player 2):
Robert (Player 1) wins!

Results for Depth 2:
         Player Nodes Searched       Time (s)
           John            253       0.000301
         Robert            216       0.004002
       Brittany            204       0.001342

--- Depth 4 ---

Robert (Player 1) vs John (Player 2):
John (Player 2) wins!

John (Player 1) vs Brittany (Player 2):
John (Player 1) wins!

Robert (Player 1) vs Brittany (Player 2):
Robert (Player 1) wins!

Results for Depth 4:
         Player Nodes Searched       Time (s)
           John           6644       0.007076
         Robert          10505       0.182003
       Brittany           3956       0.022938

--- Depth 8 ---

Robert (Player 1) vs John (Player 2):
Robert (Player 1) wins!

John (Player 1) vs Brittany (Player 2):
John (Player 1) wins!

Robert (Player 1) vs Brittany (Player 2):
Robert (Player 1) wins!

Results for Depth 8:
         Player Nodes Searched       Time (s)
           John        1155501        1.24781
         Robert        1390904        24.1792
       Brittany         470038        2.72828
```

Analysis of score_position_robert

This code defines an evaluation function called `score_position_robert` for use in a Minimax algorithm with alpha-beta pruning in a Connect 4 game. The function estimates the favorability of a given board state for a particular player, identified by the `piece` parameter, by scanning for potential winning opportunities. It begins by recording the start time and incrementing a counter for the number of nodes searched, which is useful for performance analysis. The function calculates a score based on sets of four consecutive cells—called "windows"—in all possible winning directions: horizontal, vertical, and both diagonal orientations.

A lambda function named `evaluate_window` is used to encapsulate the logic for scoring each 4-cell window. This lambda is defined inline for convenience and reusability, allowing the same evaluation logic to be applied in all four scanning directions without duplicating code. Within each window, the function counts how many cells belong to the player, the opponent, or are empty. Crucially, it only evaluates windows that do not contain any opponent pieces, under the assumption that these windows represent potential opportunities for the player. It then assigns a weighted score depending on how close the player is to winning in that window—ranging from 1 point for one piece and three empty cells, up to 100 points for four pieces in a row (a win). Once all windows have been evaluated, the function records the time taken and returns the total score. This score helps guide the Minimax algorithm toward board states that offer greater chances of winning.

Tabulation

Depth	**Player1**	**Nodes**	**Time**	**Player2**	**Nodes**	**Time**	**Winner**
2	**Robert**	**216**	**0.004002**	**John**	**253**	**0.000301**	**P1**
2	**John**	**253**	**0.00301**	**Brittany**	**204**	**0.001342**	**P1**
2	**Robert**	**216**	**0.004002**	**Brittany**	**204**	**0.001342**	**P1**
4	**Robert**	**10505**	**0.182003**	**John**	**6644**	**0.007076**	**P2**
4	**John**	**6644**	**0.007076**	**Brittany**	**3956**	**0.022938**	**P1**
4	**Robert**	**10505**	**0.182003**	**Brittany**	**3956**	**0.022938**	**P1**
8	**Robert**	**1390904**	**24.1792**	**John**	**1155501**	**1.24781**	**P1**
8	**John**	**1155501**	**1.24781**	**Brittany**	**470038**	**2.72828**	**P1**
8	**Robert**	**1390904**	**1.24781**	**Brittany**	**470038**	**2.72828**	**P1**

	Win	**Loss**
John	**4**	**2**
Robert	**5**	**1**
Brittany	**0**	**6**

Analysis of Results

The results from testing the three evaluation functions—Robert, John, and Brittany—across different depths reveal interesting trends in both effectiveness and computational performance. At depth 2, where the AI has limited foresight, Robert's evaluation function shows strong performance, winning both of his matches against John and Brittany. John also secures a win over Brittany, indicating that Brittany's function may be less effective in shallow searches. In terms of nodes searched and execution time, all three functions perform efficiently, though Robert's function uses fewer nodes than John's while still achieving better win rates, which suggests that Robert's evaluation function may be more efficient or better tuned for shallow play.

As the search depth increases to 4, the dynamics shift. John wins two out of three games, including a key victory over Robert, which suggests his evaluation function scales better with deeper searches. Interestingly, Robert still manages to beat Brittany, maintaining a competitive edge. Notably, Robert’s function searches more nodes than John’s and takes significantly more time, indicating that while it can still perform well, it is computationally more expensive. Brittany’s performance again falls short, and her evaluation function expands far fewer nodes than the others. This could imply a more simplistic or less exploratory evaluation, potentially limiting its competitiveness at this depth.

At depth 8, the early result shows Robert defeating John again, reclaiming his dominance from depth 2. This indicates that Robert’s evaluation function, despite its higher computational cost, may be particularly effective when given more room to analyze future moves. His previous loss at depth 4 might have been an anomaly due to cutoff limitations or pruning behavior. From the perspective of nodes and time, Robert's function appears to be the most aggressive and exhaustive, possibly making it ideal for end-game strategy where deeper foresight pays off.

Overall, these results highlight the trade-offs between evaluation function complexity, search depth, and performance. Robert’s function performs strongly across all depths but demands more computational resources. John’s function is efficient and performs particularly well at mid-depth, while Brittany’s function appears to lag behind in both win rate and node expansion, possibly due to a more conservative or less effective evaluation strategy. It was also noticed that in all but 1 game, player 1 was the winner. These insights provide a foundation for optimizing future evaluation functions and tailoring them to specific game stages or depth constraints.

Conclusion

From my perspective, this project was a clear success on all fronts. The implementation of the MINMAX-A-B algorithm provided a solid foundation for simulating intelligent decision-making in a competitive environment, and our AI agents performed reliably across a range of scenarios. My evaluation function demonstrated consistent strength throughout the testing process, particularly in shallow and deep searches, securing multiple wins and showing strong strategic value—even when facing more efficient but less thorough opponents. While it required more computational time and searched more nodes, the results affirmed its effectiveness in navigating complex game states, especially at higher depths where deeper foresight can be critical. Working alongside John and Brittany, I found it both rewarding and insightful to see how different evaluation strategies played out in practice. Each function brought a unique perspective to gameplay, enriching the analysis and comparative discussion. Beyond the technical implementation, the process of collecting, analyzing, and interpreting performance metrics gave me a deeper appreciation for the trade-offs in AI game strategy. Overall, this project not only reinforced my understanding of adversarial search and heuristic design but also proved to be a valuable exercise in collaborative problem-solving and algorithmic optimization.

Contributions

Though we all researched connect 4 games John had the most luck with completing a working game and became the majority contributor to the source code. Through lecture slides and teamwork John completed a rough draft for the minmax algorithm that Brittany and I added details to that were inline with the algorithm given in the project. After the completion of the game and minmax algorithm each of us independently came up with and implemented our own evaluation functions. Occasional running and debugging help was given throughout the entire process with a lot of team communication and brainstorming.