# Minimax Algorithm for Game Playing (Tic-Tac-Toe) - Detailed Explanation

This document explains a practical implementation of the Minimax algorithm (with Alpha-Beta pruning) applied to the game Tic-Tac-Toe (3x3). The PDF covers: what is Minimax, why it is used, algorithm details, evaluation function, alpha-beta pruning, complexity, optimizations, and how the provided C++ code works.

**1. What is Minimax?**
Minimax is a recursive algorithm used in two-player zero-sum games to determine the optimal move. It assumes both players play optimally. - The **maximizer** tries to maximize the utility value. - The **minimizer** tries to minimize the utility value.

**2. Why use Minimax?**
- Provides optimal moves for turn-based, perfect information games (like Tic-Tac-Toe, Chess at small depth, Connect-Four, etc.). - Simple to reason about and easy to implement. - Can be enhanced with heuristics, alpha-beta pruning, move ordering, iterative deepening, transposition tables to scale further.

**3. Algorithm Description:**
- Recursively explore all possible moves down to terminal states (win/loss/draw) or a depth limit. - Assign utility values to terminal states (+10 for win, -10 for loss, 0 for draw in this example). - Propagate those values up the game tree: maximizer chooses the move with the highest value; minimizer chooses the move with the lowest value. - Alpha-Beta pruning maintains two values (alpha = best already explored option for maximizer, beta = best for minimizer). If at any point alpha >= beta, further exploration of that branch can be pruned since it won't affect the final decision.

**4. Evaluation Function:**
In Tic-Tac-Toe we use a simple evaluation: - +10 if 'X' (AI) has a winning line. - -10 if 'O' (human) has a winning line. - 0 otherwise (including draw or non-terminal). We slightly modify the returned score by depth to prefer faster wins and slower losses: - return score - depth for maximizing wins (prefer quicker wins) - return score + depth for minimizing losses (prefer to delay losing)

**5. Alpha-Beta Pruning:**
Alpha-beta reduces the number of nodes explored without changing the result of minimax. It keeps two values: - alpha: best (highest) value that the maximizer currently can guarantee at that level or above. - beta: best (lowest) value that the minimizer currently can guarantee at that level or above. When beta <= alpha, we can prune the remaining siblings of the node. This implementation includes alpha-beta bounds passed into recursive calls.

**6. Complexity:**
- Minimax without pruning has time complexity O(b^d) where b is branching factor and d is depth. - With alpha-beta, in the best case, complexity can approach O(b^(d/2)). For Tic-Tac-Toe (branching ~9), full search is feasible. - Memory complexity is O(d) due to recursion stack (plus board storage if copied).

**7. How the provided C++ code works (overview):**
- Board representation: 3x3 vector of chars ('X', 'O', '_').
- evaluate(): checks rows, columns, diagonals for terminal win states and returns +10/-10/0.
- isMovesLeft(): checks if there are empty squares left.
- minimax(): recursive function with parameters (board, depth, isMax, alpha, beta). It returns the best achievable score from that state.
- If isMax is true, try all moves for 'X', update alpha and prune when possible.
- If isMax is false, try all moves for 'O', update beta and prune when possible.
- findBestMove(): tries each possible move for 'X', uses minimax to score them and picks the highest-value move.
- main(): simple interactive loop where AI (X) plays first and human (O) inputs moves via console.

**8. Practical notes and tips:**
- For larger games (Chess, Go), full minimax is infeasible; use heuristics, depth limits, iterative deepening, transposition tables, and Monte Carlo/learning methods.
- Move ordering (trying captures or center moves first) improves pruning.
- Use transposition tables (hash of board states) to cache evaluated positions.
- For non-deterministic or imperfect-information games, other algorithms are used (expectimax, MCTS, RL).

**9. Compile and run:**
Save the C++ file (minimax_tictactoe.cpp) and compile with:
g++ -std=c++17 -O2 -o minimax_ttt minimax_tictactoe.cpp Run:
./minimax_ttt The program runs a console-based game where AI plays as 'X' and human plays as 'O'.

**10. Full C++ Source (excerpt):**

```
#include <bits/stdc++.h>
using namespace std;

/*
 Minimax implementation for Tic-Tac-Toe (3x3)
 Two players: 'X' (maximizer) and 'O' (minimizer)
 This example includes alpha-beta pruning and simple board evaluation.
*/

const char PLAYER = 'X'; // Maximizer
const char OPPONENT = 'O'; // Minimizer
const char EMPTY = '_';

// Print board
void printBoard(const vector<vector<char>>& board) {
    for (auto &row : board) {
        for (char c : row) cout << c << ' ';
        cout << '\n';
    }
    cout << '\n';
}

// Check for win state
int evaluate(const vector<vector<char>>& b) {
    // Rows
    for (int row = 0; row < 3; ++row) {
        if (b[row][0] == b[row][1] && b[row][1] == b[row][2]) {
            if (b[row][0] == PLAYER) return +10;
            else if (b[row][0] == OPPONENT) return -10;
        }
    }
    // Columns
    for (int col = 0; col < 3; ++col) {
        if (b[0][col] == b[1][col] && b[1][col] == b[2][col]) {
            if (b[0][col] == PLAYER) return +10;
            else if (b[0][col] == OPPONENT) return -10;
        }
    }
    // Diagonals
    if (b[0][0] == b[1][1] && b[1][1] == b[2][2]) {
        if (b[0][0] == PLAYER) return +10;
        else if (b[0][0] == OPPONENT) return -10;
    }
    if (b[0][2] == b[1][1] && b[1][1] == b[2][0]) {
        if (b[0][2] == PLAYER) return +10;
        else if (b[0][2] == OPPONENT) return -10;
    }
    // No winner
    return 0;
}

// Check if moves left
bool isMovesLeft(const vector<vector<char>>& board) {
```

```
    for (auto &row : board)
        for (char c : row)
            if (c == EMPTY) return true;
    return false;
}

// Minimax with alpha-beta pruning
int minimax(vector<vector<char>>& board, int depth, bool isMax, int alpha, int beta) {
    int score = evaluate(board);

    // If Maximizer has won the game return evaluated score
    if (score == 10) return score - depth; // subtract depth to prefer faster wins
    // If Minimizer has won the game return evaluated score
    if (score == -10) return score + depth; // add depth to prefer slower losses
    // If no moves left and no winner then it is a tie
    if (!isMovesLeft(board)) return 0;

    if (isMax) {
        int best = INT_MIN;
        // Traverse all cells
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                if (board[i][j] == EMPTY) {
                    board[i][j] = PLAYER;
                    int val = minimax(board, depth + 1, false, alpha, beta);
                    board[i][j] = EMPTY;
                    best = max(best, val);
                    alpha = max(alpha, best);
                    if (beta <= alpha) return best; // Beta cut-off
                }
            }
        }
        return best;
    } else {
        int best = INT_MAX;
        // Traverse all cells
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                if (board[i][j] == EMPTY) {
                    board[i][j] = OPPONENT;
                    int val = minimax(board, depth + 1, true, alpha, beta);
                    board[i][j] = EMPTY;
                    best = min(best, val);
                    beta = min(beta, best);
                    if (beta <= alpha) return best; // Alpha cut-off
                }
            }
        }
        return best;
    }
}

// Find the best move for the current player (PLAYER - 'X')
pair<int,int> findBestMove(vector<vector<char>>& board) {
    int bestVal = INT_MIN;
    pair<int,int> bestMove = {-1, -1};

    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
```

```
            if (board[i][j] == EMPTY) {
                board[i][j] = PLAYER;
                int moveVal = minimax(board, 0, false, INT_MIN, INT_MAX);
                board[i][j] = EMPTY;

                if (moveVal > bestVal) {
                    bestMove = {i, j};
                    bestVal = moveVal;
```

**11. Conclusion:**

Minimax with alpha-beta pruning is a foundational algorithm for perfect-information, two-player games. The provided C++ implementation is simple, clear, and suitable for learning. For production-level game engines, add optimizations like move ordering, caching, iterative deepening, and domain-specific heuristics.