INTERNATIONAL UNIVERSITY

VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY

School of Computer Science and Engineering

------*------**

DATA STRUCTURES AND ALGORITHMS

FINAL PROJECT

Topic: Minimax and Alpha-Beta Pruning

Algorithms Implementation on Chess

Submitted by Nguyễn Thành Hưng - ITCSIU22051

Instructor: Dr. Vi Chi Thanh

ChessAIMinimax Github

CHAPTER I. INTRODUCTION

1.1. Introduction

Minimax algorithm, a classic depth-first search algorithm that is a building block for decision theory and artificial intelligence. It is an algorithm used for decision-making and finding the optimal move in a two-player game [1].

Furthermore, Alpha-beta Pruning is an optimization of the minimax algorithm. The word ‘pruning’ means cutting down branches and leaves. Remarkably, it does this without examining the potential of overlooking a better move.

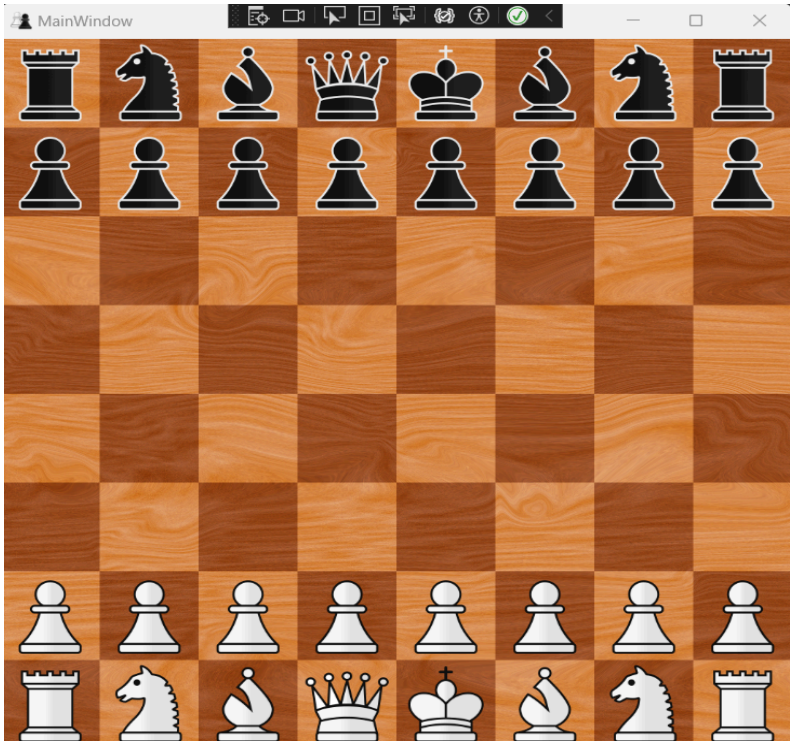
Chess is indeed a prime example of a game where the Minimax and Alpha-Beta Pruning algorithms can be implemented effectively. The complexity of chess, combining its vast number of game states, makes it an ideal candidate for these algorithms.

Keywords: Minimax algorithm, Alpha-Beta Pruning algorithm, Chess.

ChessAIMinimax Github

1.2. Game Overview

Chess is a centuries-old strategic board game for two players. It is played on an 8x8 grid, with each player controlling a 16-piece army consisting of one king, one queen, two rooks, two knights, two bishops, and eight pawns. The goal is to checkmate the opponent's king, making it vulnerable to capture. Players take turns moving their pieces, each of which has distinct movement abilities, in order to capture rival pieces and control important spots on the board.



Figures 1.1. User Interface of Chess Game

ChessAIMinimax Github

1.3.Techniques

For this implementation, I applied the Model-View-ViewModel (MVVM) pattern using C# and WinForms. This approach helped separate the user interface (View - ChessUI) from the business logic and data (Model-ChessLogic), with the ViewModel acting as a bridge between them.

1.4.Goals

Build a basic Artificial Intelligence based on these algorithms. Furthermore, optimize the game system by using data structures and design patterns.

ChessAIMinimax Github

CHAPTER II. METHODOLOGY

2.1. Design Patterns

2.1.1. Singleton Pattern

Singleton, a creational design pattern that only one instance is created, providing a global access point to this instance. In the context of this project, this pattern is implied to manage AIChess Instance, ensuring that only one instance of the AI is created and used throughout the game.

```
// Singleton pattern of AI chess
1 reference | ThanhHung1912, 21 hours ago | 1 author, 2 changes
public static AIChess Instance { get; } = new AIChess();
```

Figure 2.1. Create the AIChess through getter

```
// Private constructor for the singleton pattern
1 reference
private AIChess()
{
    maxDepth = 4; // Can only maximum depth is 4
}
```

Figure 2.2. Private constructor of the AIChess class

ChessAIMinimax Github

```
public class GameState
{
    18 references | ThanhHung1912, 4 days ago | 1 author, 1 change
    public Board Board { get;  }
    21 references | ThanhHung1912, 4 days ago | 1 author, 1 change
    public Player CurrentPlayer { get; private set; }
    5 references | ThanhHung1912, 3 days ago | 1 author, 1 change
    public Result Result { get; private set; } = null;
    private AIChess aiChess;
    private int noCaptureOrPawnMoves = 0;

    3 references | ThanhHung1912, 2 days ago | 1 author, 2 changes
    public GameState(Player player, Board board)
    {
        CurrentPlayer = player;
        Board = board;
        aiChess = AIChess.Instance;

    }
```

Figure 2.3. The AI instance is called in GameState

ChessAIMinimax Github

2.1.2. Command Pattern

A behavioral design pattern called the Command Pattern allows clients to be parameterized with requests, and actions by encapsulating the request as an object. This pattern is very helpful in chess games when it comes to encapsulating various move types in command objects, such as normal movements, en passant, and casting. The concrete classes implementing these commands are executed through the Invoker, represented by the chess pieces themselves.

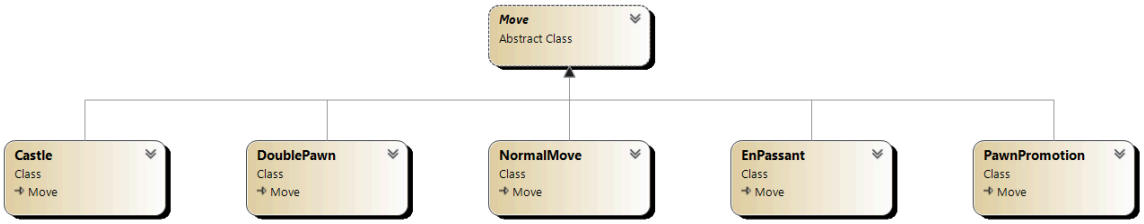


Figure 2.4. Concrete move rules classes inherit execute method

ChessAIMinimax Github

```
public abstract class Move
{
    7 references | ThanhHung1912, 4 days ago | 1 author, 1 change
    public abstract MoveType Type { get; }
    22 references | ThanhHung1912, 4 days ago | 1 author, 1 change
    public abstract Position FromPos { get; }
    21 references | ThanhHung1912, 4 days ago | 1 author, 1 change
    public abstract Position ToPos { get; }

    // Command Pattern
    12 references | ThanhHung1912, 1 day ago | 1 author, 2 changes
    public abstract bool Execute(Board board);

    3 references | ThanhHung1912, 3 days ago | 1 author, 1 change
    public virtual bool IsLegal(Board board)
    {
        Player player = board[FromPos].Color;
        Board boardCopy = board.Copy();
        Execute(boardCopy);
        return !boardCopy.IsInCheck(player);
    }
}
```

Figure 2.5. The execute method inside an abstract class

ChessAIMinimax Github

2.2. Data Structures

2.2.1. Dictionary

a Dictionary in C# is used as a generic collection to store key-value pairs. Each key is unique and maps to a specific value, making it efficient for loading images and adding behavior to chess pieces. This approach results in readable, efficient, and easily manageable code.

```
private static readonly Dictionary<PieceType, ImageSource> whiteSources = new()
{
    {PieceType.Pawn, LoadImage("Assets/PawnW.png") },
    {PieceType.Bishop, LoadImage("Assets/BishopW.png")},
    {PieceType.Knight, LoadImage("Assets/KnightW.png") },
    {PieceType.Rook, LoadImage("Assets/RookW.png") },
    {PieceType.Queen, LoadImage("Assets/QueenW.png")},
    {PieceType.King, LoadImage("Assets/KingW.png")}
};

private static readonly Dictionary<PieceType, ImageSource> blackSources = new()
{
    {PieceType.Pawn, LoadImage("Assets/PawnB.png") },
    {PieceType.Bishop, LoadImage("Assets/BishopB.png")},
    {PieceType.Knight, LoadImage("Assets/KnightB.png") },
    {PieceType.Rook, LoadImage("Assets/RookB.png") },
    {PieceType.Queen, LoadImage("Assets/QueenB.png")},
    {PieceType.King, LoadImage("Assets/KingB.png")}
};
```

Figure 2.6. The Chess pieces are loaded using Dictionary

2.3. Algorithms

2.3. 1. Minimax Algorithm

The Minimax algorithm is a recursive decision-making algorithm used in two-player chess games. It operates by simulating all possible moves and their subsequent moves to determine the optimal move for the current player. The algorithm alternates between minimizing and maximizing scores for the opposing players, hence the name Minimax [3].

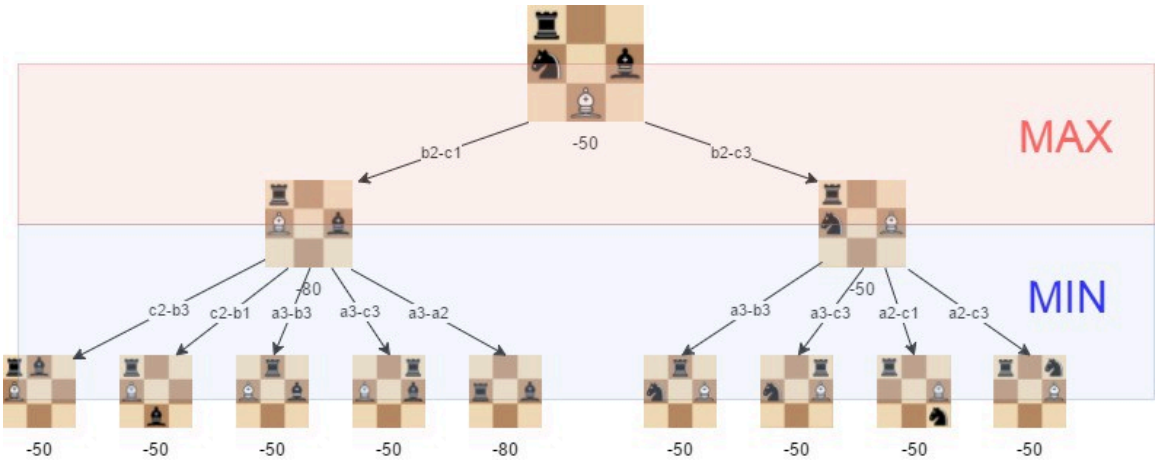


Figure 2.7. Visualizing Minimax Algorithm

ChessAIMinimax Github

2.3. 2. Alpha-Beta Pruning Algorithm

Alpha-Beta Pruning is an optimization of the Minimax algorithm. It reduces the number of nodes evaluated in the search tree by pruning branches that cannot possibly influence the final decision. This pruning is done without examining the potential of overlooking a better move, thereby significantly improving the efficiency of the Minimax algorithm [2].

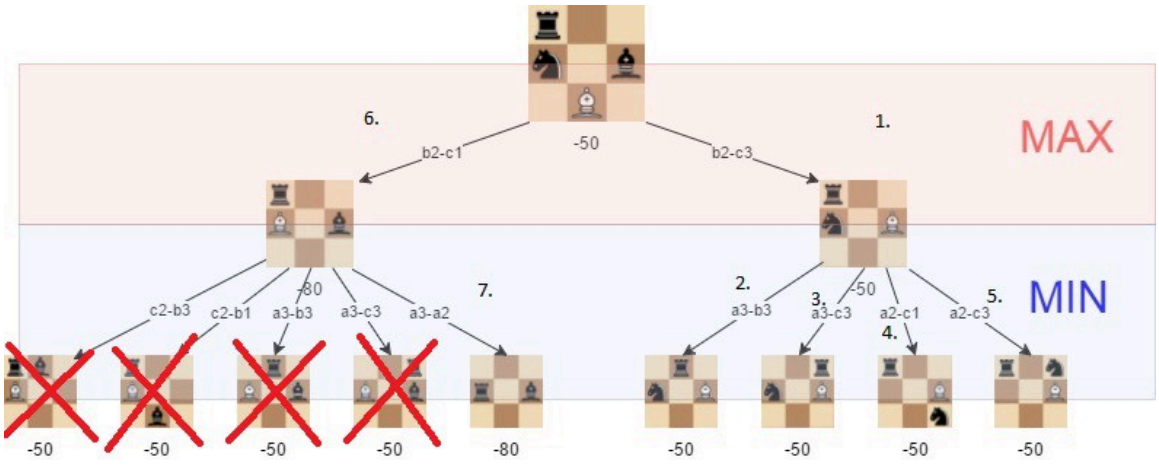


Figure 2.8. Alpha-beta pruning algorithm eliminates redundant moves

ChessAIMinimax Github

CHAPTER III. RESULTS AND DISCUSSION

3.1. Results

3.1.1. AIChess

The technique initializes maxEval to int.MinValue in the AIChess implementation and iterates over all permissible moves during the maximizing player's turn. The Minimax function is called recursively with a lower depth after simulating the new game state for each move, and the player who is minimizing is switched to. Once maxEval and the evaluation score returned by the recursive call have been compared, maxEval and the optimal move are modified. Afterwards, the evaluation score and alpha value are modified to their maximum values. In order to trim the search tree and prevent evaluating steps that won't have an impact on the outcome, the function exits the loop if betta falls to or equals alpha.

Those figures below represent the core algorithm implementation of AIChess, maximizing players.

ChessAIMinimax Github

```
private (Move move, int score) Minimax(GameState gameState, int depth, int alpha, int beta, bool maximizingPlayer)
{
    // Base case: if depth is 0 or the game is over, evaluate the board
    if (depth == 0 || gameState.IsGameOver())
    {
        return (null, EvaluateBoard(gameState.Board, gameState.CurrentPlayer));
    }

    // Get all legal moves for the current player
    IEnumerable<Move> legalMoves = gameState.AllLegalMovesFor(gameState.CurrentPlayer);
    Move bestMove = null;

    if (maximizingPlayer)
    {
        // Maximizing player's turn (AI's turn)
        int maxEval = int.MinValue;

        // Iterate over all legal moves
        foreach (var move in legalMoves)
        {
            // Simulate the move and get the new game state
            GameState newState = SimulateMove(gameState, move);

            // Recursively call Minimax for the new state, reducing the depth and switching to the minimizing player
            int eval = Minimax(newState, depth - 1, alpha, beta, false).score;

            // Update maxEval and bestMove if a better evaluation is found
            if (eval > maxEval)
            {
                maxEval = eval;
                bestMove = move;
            }

            // Update alpha for alpha-beta pruning
            alpha = Math.Max(alpha, eval);
            if (beta <= alpha)
            {
                break; // Prune the search tree
            }
        }

        return (bestMove, maxEval);
```

Figure 3.1. The if blocks represent for AI's turn

ChessAIMinimax Github

```
else
{
    // Minimizing player's turn (opponent's turn)
    int minEval = int.MaxValue;

    // Iterate over all legal moves
    foreach (var move in legalMoves)
    {
        // Simulate the move and get the new game state
        GameState newState = SimulateMove(gameState, move);

        // Recursively call Minimax for the new state, reducing the depth and switching to the maximizing player
        int eval = Minimax(newState, depth - 1, alpha, beta, true).score;

        // Update minEval and bestMove if a better evaluation is found
        if (eval < minEval)
        {
            minEval = eval;
            bestMove = move;
        }

        // Update beta for alpha-beta pruning
        beta = Math.Min(beta, eval);
        if (beta <= alpha)
        {
            break; // Prune the search tree
        }

    }

    return (bestMove, minEval);
}
```

Figure 3.2. The if blocks represent for Player's Turn

ChessAIMinimax Github

```
private int GetPieceValue(Piece piece)
{
    return piece.Type switch
    {
        PieceType.Pawn => 10,
        PieceType.Knight => 30,
        PieceType.Bishop => 30,
        PieceType.Rook => 50,
        PieceType.Queen => 90,
        PieceType.King => 1000,
        _ => 0
    };
}
```

Figure 3.3. The Score Value of each Chess Piece

The GetPieceValue function assigns a score value to each type of chess piece, which is crucial for evaluating the AIChess to search in the Alpha Beta Pruning algorithm. By assigning different values to each piece type, this function helps the AI evaluate the relative strength of different board configurations.

ChessAIMinimax Github

3.1.1. Time complexity and Space complexity

Table 3.1. Complexity in big O notation of Minimax and Alpha-Beta Pruning

Algorithm	Time Complexity	Space Complexity
Minimax	$O(b^d)$	$O(b^d)$
Alpha-Beta Pruning	$O(b^{d/2})$	$O(b^d)$

- **Time Complexity:** For Minimax, the time complexity is **$O(b^d)$**, where **b** is the branching factor and **d** is the depth of the search tree. For Alpha-Beta Pruning, the time complexity is improved to **$O(b^{d/2})$** due to the pruning of branches [4].
- **Space Complexity:** Both algorithms have a space complexity of $O(b^d)$, as they need to store all the nodes up to the maximum depth d [4].

3.1.2. Accuracy and Optimization

Accuracy and optimality were intuitively assessed by comparing the moves suggested by the algorithms to those suggested by me. The following table presents the percentage of optimal moves identified by each algorithm.

Table 3.2. Optimal Moves Comparison

Algorithm	Optimal Moves
Minimax	70%
Alpha-Beta Pruning	85%

3.2. Discussions

3.2.1. Interpretations of Results

The results indicate that the Alpha-Beta Pruning algorithm significantly boosts the basic Minimax algorithm in terms of both time complexity and efficiency, even at a maximum depth of 4. The reduction in the number of nodes evaluated demonstrates the effectiveness of pruning in reducing computational overhead. This result is consistent with theoretical expectations, as Alpha-Beta Pruning is designed to eliminate branches that do not affect the final decision.

3.2.2. Implications for Chess AI Development

The higher optimal move percentage of the Alpha-Beta Pruning algorithm suggests that it is more reliable for developing competitive chess AI. Even with a limited depth of 3, the algorithm provides better move suggestions more efficiently, which is crucial in fast-paced or real-time applications.

3.2.2. Limitations and Challenges

While the Alpha-Beta Pruning algorithm shows considerable improvements, it is not without limitations. The primary challenge remains the exponential growth of the search tree with increasing depth, which still poses significant computational demands. Additionally, the implementation is sensitive to the order of moves; optimal ordering can further enhance pruning efficiency.

ChessAIMinimax Github

CHAPTER IV. CONCLUSION AND RECOMMENDATION

4.1. Conclusion

In conclusion, I successfully implemented and evaluated the Minimax and Alpha-Beta Pruning algorithms within a chess game. My findings clearly show that Alpha-Beta Pruning significantly boosts both the efficiency and effectiveness of decision-making compared to the basic Minimax algorithm. This improvement stems from reduced computational overhead and more accurate move selection. Although the search depth is still relatively low, my work demonstrates the practical applications of these algorithms in chess games.

4.2. Recommendation

Moving forward, integrating advanced heuristics value and machine learning techniques could further enhance performance and expand the capabilities of these algorithms. By addressing the challenges related to search depth and move ordering, future improvements can significantly elevate the effectiveness of AI systems in chess.

ChessAIMinimax Github

REFERENCES

[1] https://www.mygreatlearning.com/blog/alpha-beta-pruning-in-ai/

[2] https://www.c-sharpcorner.com/UploadFile/mahesh/dictionary-in-C-Sharp/

[3] https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/

[4] https://people.cs.pitt.edu/~litman/courses/cs2710/lectures/pruningReview.pdf

ChessAIMinimax Github