A Typesetting System to Untangle the Scientific Writing Process

Jia Khot
Computer Science and Engineering
Shiv Nadar IoE
jk637@snu.edu.in

Lalit Maurya
Computer Science and Engineering
Shiv Nadar IoE
lm742@snu.edu.in

Rachit Kumar
Computer Science and Engineering
Shiv Nadar IoE
rk551@snu.edu.in

***Abstract*—The process of scientific writing is often tangled up with the intricacies of typesetting, leading to frustration and wasted time for researchers. In this paper, we introduce Typst, a new typesetting system designed specifically for scientific writing. Typst untangles the typesetting process, allowing researchers to compose papers faster. In a series of experiments we demonstrate that Typst offers several advantages, including faster document creation, simplified syntax, and increased ease-of-use. Our implementation can be found at [1]**

I. Introduction

Chess has long been a benchmark for AI research, testing algorithms for decision-making, search, and heuristic evaluation. Smaller variants like the Silverman 4x4 chess board (Fig. 1) reduce complexity while keeping the game strategically rich, making them ideal for experimentation.

The Silverman 4x4 board has 4 Pawns, 2 Rooks, a Queen, and a King on each side. Even on this compact board, every move matters, requiring careful evaluation of trade-offs and tactical opportunities. The smaller size accelerates gameplay and magnifies the impact of each decision.

In this work, we explored the use of minimax combined with alpha-beta pruning to develop an AI for this variant. This approach allowed efficient exploration of possible moves and evaluation of board positions, providing insight into search strategies and decision-making in a constrained but strategically rich environment.

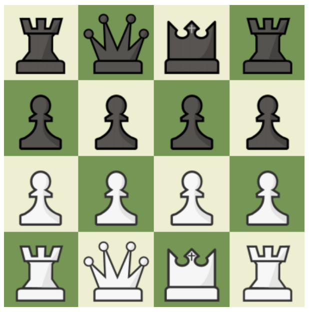
Fig. 1. Silverman 4x4 Chess Board

II. Literature Review

A. The Minimax algorithm

The Minimax algorithm is a fundamental method for decision-making in two-player, zero-sum games like chess. It models gameplay as a tree of possible moves, where each node represents a board state. The algorithm recursively evaluates these states, assuming that one player (Max) aims to maximize the score while the other player (Min) aims to minimize it.

On the 4x4 Silverman board, Minimax allows the AI to anticipate the consequences of moves several turns ahead. Each possible move is scored using an evaluation function that considers material, piece positioning, and potential threats. By traversing the game tree, the algorithm identifies the move that optimizes the AI's outcome while accounting for the opponent's best responses.

However, as the board complexity grows, the number of possible states expands exponentially. To manage this, we integrated alpha-beta pruning.

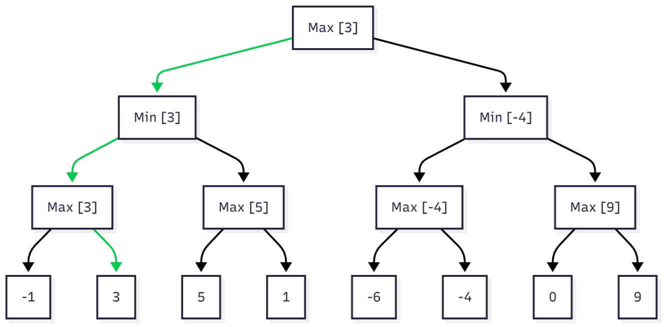

Fig. 2. Minimax algorithm. The green path represents the flow of the game

B. Alpha-Beta Pruning

Alpha-beta pruning is an optimization of the Minimax algorithm that reduces the number of nodes the AI must evaluate in the game tree. By keeping track of the best already explored options for both players, the algorithm can prune branches that cannot influence the final decision, thereby saving computation time.

The algorithm maintains two values during traversal:

1) α: the best value that the maximizing player can guarantee
2) β: the best value the minimizing player can guarantee

Whenever the algorithm encounters a node where the current value cannot improve upon alpha or beta, it stops exploring that branch. This pruning does not change the outcome of the Minimax decision but significantly reduces the number of states that need evaluation.

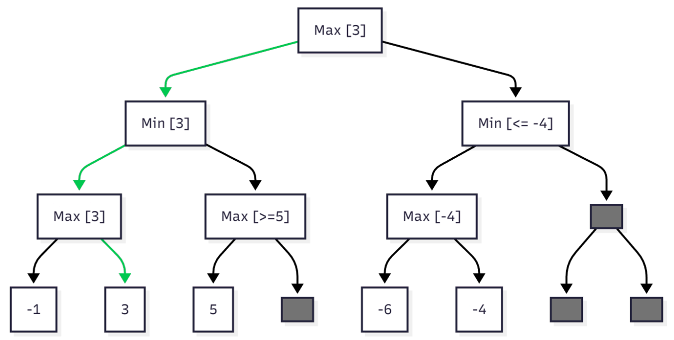

Fig. 3. Application of Alpha-Beta Pruning on the tree in Fig. 2, the grayed out nodes have been pruned since they couldn't possibly affect the outcome of the game

III. Our Implementation

A. Architecture Overview

Our implementation follows a modular design with three core components: board representation, move generation, and the AI agent. The system is built in Python using object-oriented principles for clarity and extensibility.

B. Board Representation

The board state is stored as a 4×4 NumPy array of `Piece` objects. Each piece is defined by two enums:

```
class PieceColor(Enum):
    BLACK = 0
    WHITE = 1
    EMPTY = None

class PieceType(Enum):
    PAWN = 0
    ROOK = 1
    QUEEN = 2
    KING = 3
    EMPTY = None
```

The `Piece` class encapsulates piece behavior, including:

- **Movement patterns**: Each piece type has predefined displacement vectors
- **Material values**: King=1000, Queen=9, Rook=5, Pawn=1 (Silverman values)
- **Display symbols**: Unicode characters for visualization

C. Move Generation and Validation

The `Board` class implements a two-phase move generation system:

1) **Pseudo-legal move generation**: Generate all geometrically valid moves based on piece movement rules
2) **Legal move filtering**: Remove moves that would leave the king in check

Key features include:

- Path obstruction checking for sliding pieces (Rook, Queen)
- Special pawn capture logic (diagonal attacks only)
- King safety verification after each move

The board maintains several precomputed game states:

- Valid moves for each color
- Check/checkmate/stalemate status
- Threatened pieces and threat scores

D. Evaluation Function

Our heuristic evaluation considers multiple factors:

1) **Material balance**: Sum of piece values for each side
2) **Positional advantage**: Bonus for controlling center squares (b2, b3, c2, c3)
3) **Threat assessment**: Value of opponent pieces under attack
4) **Material advantage**: Opponent's captured pieces

The evaluation score is computed as:

```
score = (material_black + absence_white +
center_black + threats_black) −
(material_white + absence_black +
center_white + threats_white)
```

where positive scores favor Black (the AI) and negative scores favor White.

E. Minimax with Alpha-Beta Pruning

The `MinimaxAgent` implements depth-limited search with several optimizations:

Move Ordering: Captures are prioritized using the MVV-LVA heuristic (Most Valuable Victim - Least Valuable Attacker):

Immediate Checkmate Detection: Before recursing, the agent checks if a move delivers checkmate, assigning infinite value to such moves.

Terminal Condition Handling:

- Checkmate: $\pm\infty$ score
- Stalemate: 0 score
- No legal moves: Game over evaluation

F. User Interface

The system provides two interaction modes:

1) **CLI**: Text-based interface for debugging and testing
2) **GUI**: Pygame-based visual interface with move animations and sound effects

The GUI features smooth piece animations, move hints, and a move history panel, making the game accessible to non-technical users

IV. Results and Evaluation

A. Test Suite Design

We developed a comprehensive test suite to evaluate the AI's tactical capabilities across various scenarios. The suite includes predefined board positions testing:

- **Mate-in-one detection**: Forcing immediate checkmate
- **Free material capture**: Taking undefended pieces
- **Threat response**: Defending against opponent attacks
- **Best move selection**: Choosing optimal moves in complex positions

Each test case validates whether the AI selects the tactically correct move within a reasonable search depth.

B. Performance Analysis

The AI demonstrates strong tactical awareness in most scenarios:

Strengths:

- Consistent one-move checkmate delivery across test cases
- Reliable capture of hanging pieces
- Effective use of alpha-beta pruning to explore deeper positions
- Fast move generation (< 1 second per move at depth 3-4)

Limitations:

- Tendency toward stalemate rather than pursuing long-term checkmate strategies
- Evaluation function occasionally undervalues positional advantages
- Horizon effect: tactical combinations beyond search depth remain invisible

C. Areas for Improvement

Based on our evaluation, we identify three key enhancement opportunities:

Move Ordering Refinement: Current implementation uses simple MVV-LVA ordering. Incorporating:

- Killer move heuristic
- History heuristic
- Hash move ordering

could dramatically improve pruning efficiency, potentially allowing depth 5-6 searches.

Enhanced Evaluation Function: The current heuristic could be strengthened by:

- King safety considerations (pawn shield, open files)
- Piece mobility scoring
- Checkmate pattern recognition
- Endgame-specific evaluation tables

References

[1] [Online]. Available: https://github.com/lalitm1004/mini-chess