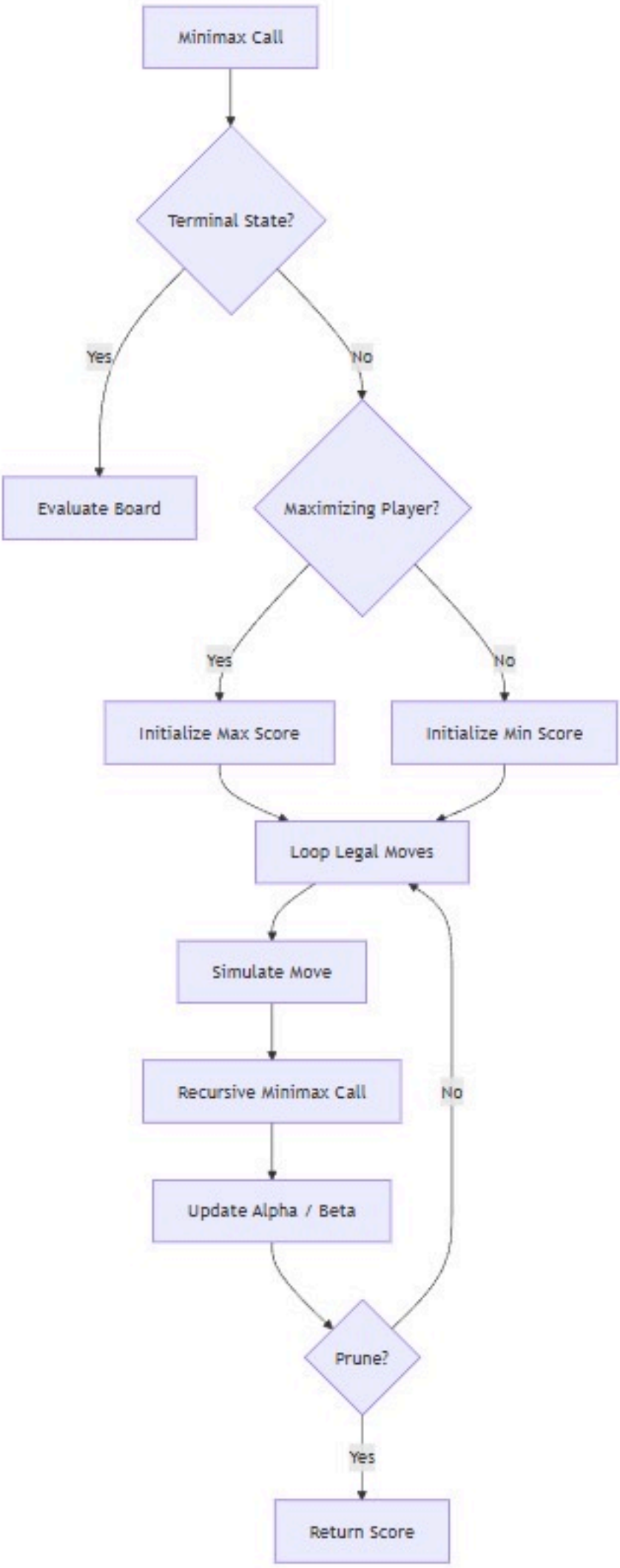
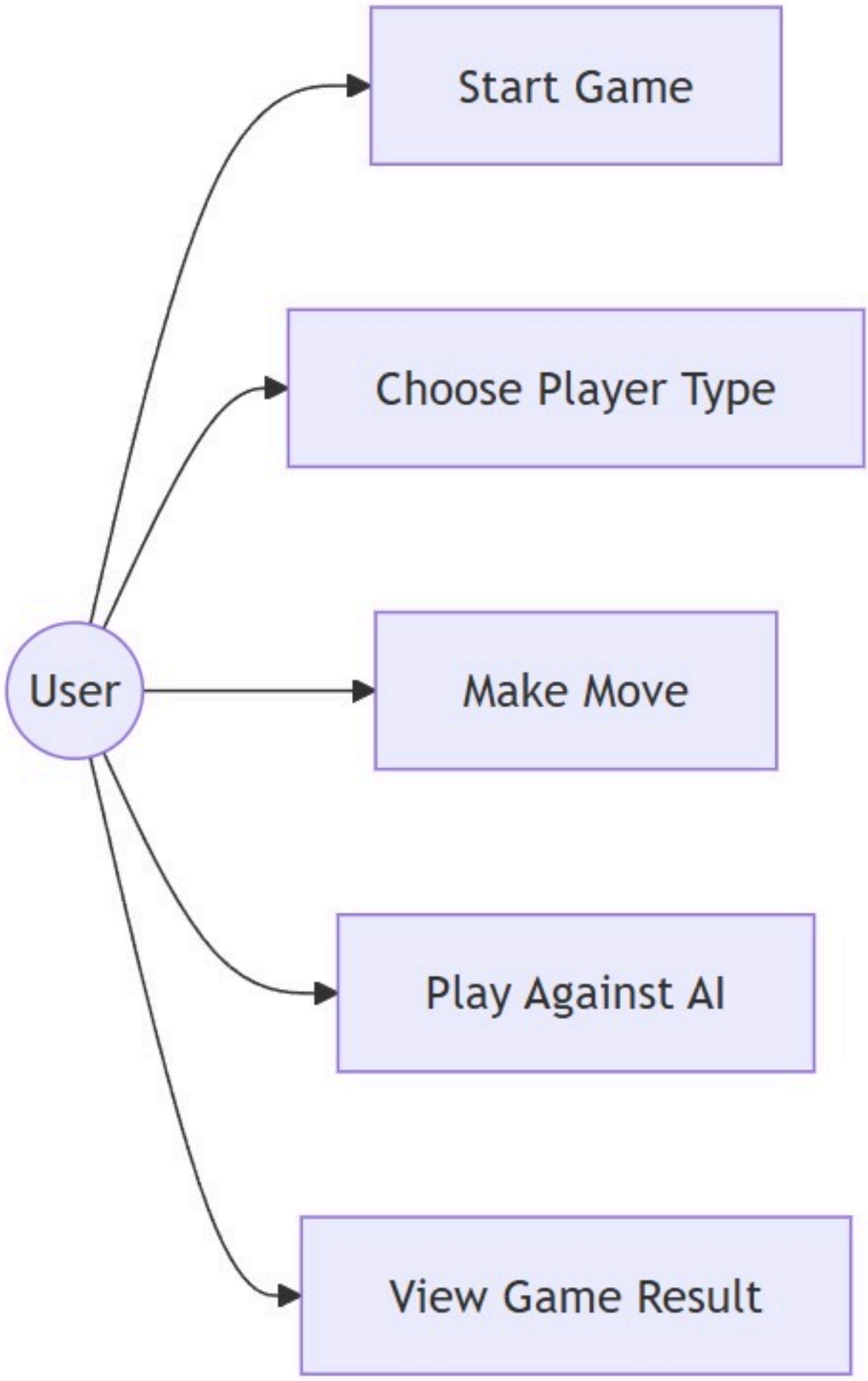
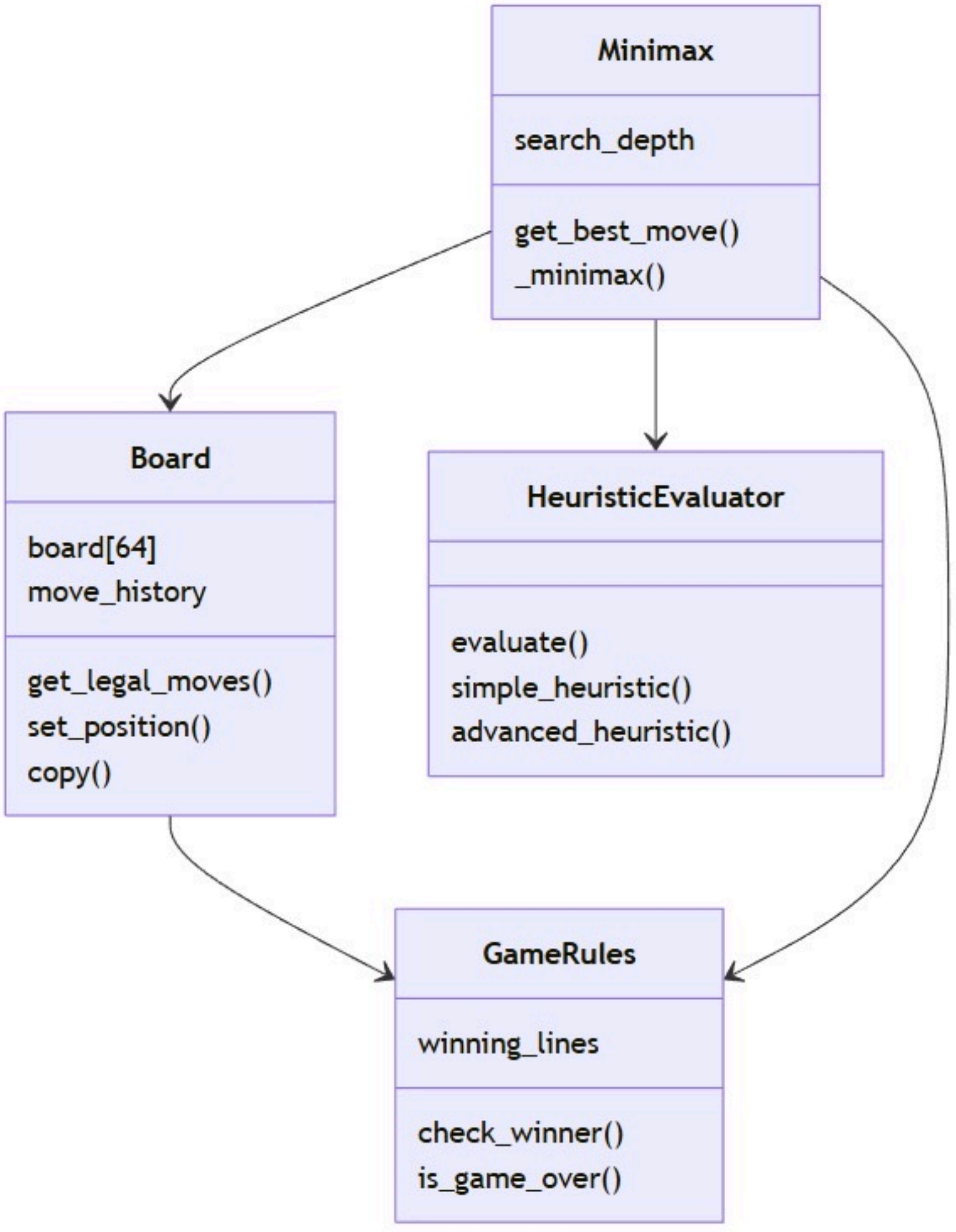

User
GUI
Board
GameRules
Minimax
HeuristicEvaluator


Minimax Call
Terminal State?
Yes
No
Evaluate Board
Maximizing Player?
Yes
No
Initialize Max Score
Initialize Min Score
Loop Legal Moves
Simulate Move
Recursive Minimax Call
Update Alpha / Beta
Prune?
No
Yes
Return Score


Start Game
Choose Player Type
User
Make Move
Play Against AI
View Game Result


Minimax
search_depth
get_best_move()
_minimax()
Board
board[64]
move_history
get_legal_moves()
set_position()
copy()
HeuristicEvaluator
evaluate()
simple_heuristic()
advanced_heuristic()
GameRules
winning_lines
check_winner()
is_game_over()

# 3D Tic-Tac-Toe Game

Implementation with Minimax Algorithm and AI

## Project Report

Generated: December 14, 2025

# 1. Executive Summary

This project implements a complete 3D Tic-Tac-Toe game (4×4×4 cubic board) with an intelligent AI opponent using the Minimax algorithm enhanced with Alpha-Beta pruning. The game features a user-friendly graphical interface, multiple heuristic evaluation functions, and comprehensive algorithm optimization techniques.

## 1.1 Project Overview

The 3D Tic-Tac-Toe game is an extension of the classic 2D Tic-Tac-Toe game into three dimensions. The game is played on a 4×4×4 cubic board, providing 64 positions for players to place their marks. The objective remains the same: be the first player to form a line of four marks in any direction (horizontal, vertical, diagonal, or through 3D space).

## 1.2 Key Features

• 4×4×4 cubic board with 64 playable positions

• 76 possible winning lines (16 X-axis, 16 Y-axis, 16 Z-axis, 24 plane diagonals, 4 space diagonals)

• Minimax algorithm for optimal move selection

• Alpha-Beta pruning for efficient search space reduction

• Two heuristic evaluation functions (Simple and Advanced)

• Symmetry reduction for move space optimization

• Heuristic reduction for faster search

• Interactive GUI built with Tkinter

• Real-time AI decision analysis display

• Comprehensive performance analysis tools

# 2. System Architecture

The project is organized into a modular architecture with clear separation of concerns. Each component handles a specific aspect of the game logic and user interface.

## 2.1 Project Structure

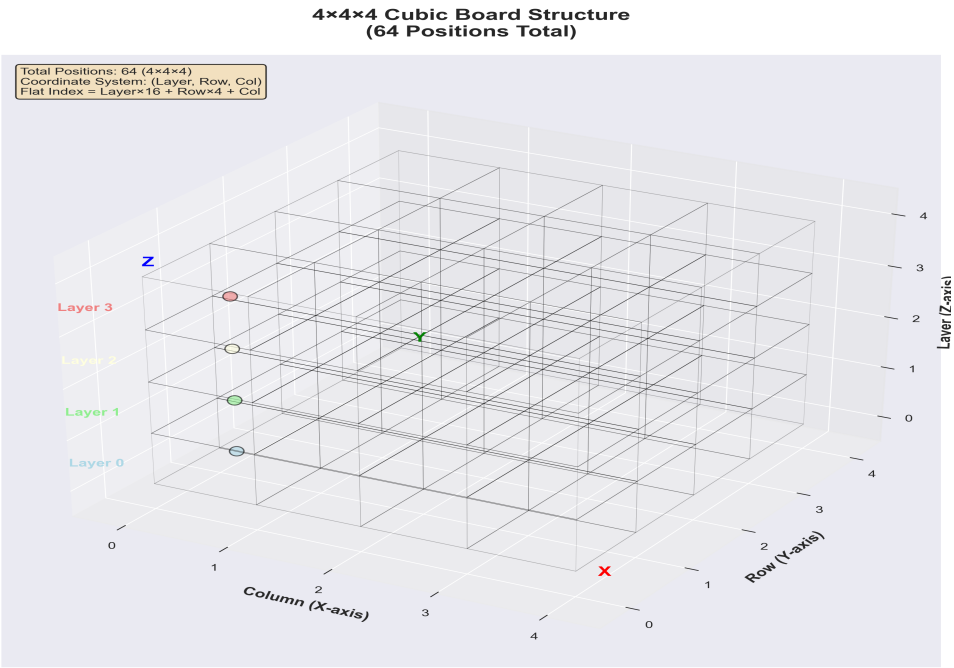
| Module | Description |
|---|---|
| cubic/board.py | Board representation and position management |
| cubic/rules.py | Game rules and win detection (76 winning lines) |
| cubic/players.py | Player classes (Human, AI, Random) |
| cubic/minimax.py | Minimax algorithm with Alpha-Beta pruning |
| cubic/heuristics.py | Heuristic evaluation functions |
| cubic/gui.py | Graphical user interface (Tkinter) |
| main.py | Application entry point |
| analysis.py | Performance analysis and benchmarking |

## 2.2 Board Representation

The 4×4×4 cubic board is represented as a flat array of 64 integers. Each position can be EMPTY (0), PLAYER_X (1), or PLAYER_O (2). The conversion between 3D coordinates (layer, row, col) and flat indices is handled using the formula:

```
flat_index = layer × 16 + row × 4 + col
```

## Board Structure Visualization:




## 2.3 Winning Lines

The game detects 76 possible winning lines across the 3D cube:

• 16 lines parallel to X-axis (fixed layer and row, varying column)

• 16 lines parallel to Y-axis (fixed layer and column, varying row)

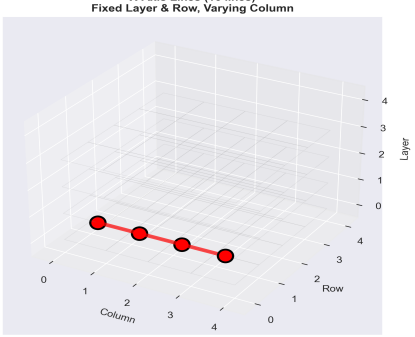
• 16 lines parallel to Z-axis (fixed row and column, varying layer)

• 24 diagonal lines in XY, XZ, and YZ planes

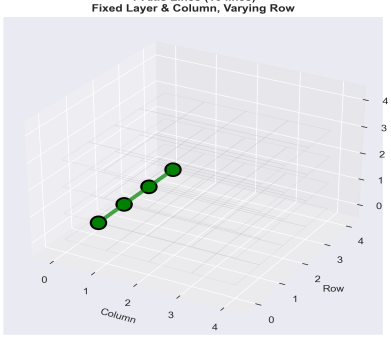
• 4 space diagonals through the 3D cube

## Winning Lines Visualization
## (76 Total Winning Lines)

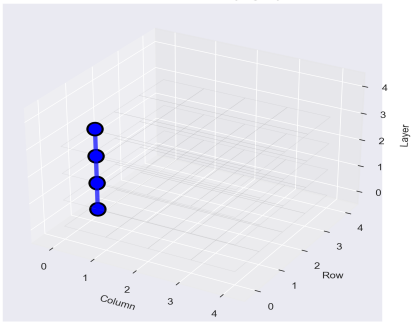
**X-Axis Lines (16 lines)**
**Fixed Layer & Row, Varying Column**



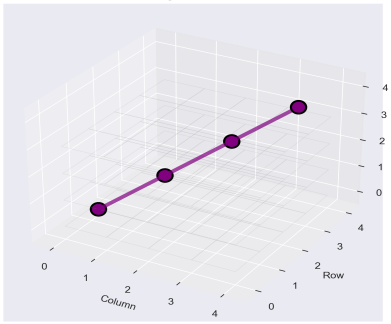

**Y-Axis Lines (16 lines)**
**Fixed Layer & Column, Varying Row**




**Z-Axis Lines (16 lines)**
**Fixed Row & Column, Varying Layer**




**Space Diagonals (4 lines)**
**Through the 3D Cube**



# 3. Algorithm Implementation

The core of the AI implementation is the Minimax algorithm, enhanced with various optimization techniques to make it computationally feasible for the large state space.

## 3.1 Minimax Algorithm

The Minimax algorithm is a decision-making algorithm used in game theory for minimizing the possible loss in a worst-case scenario. In this implementation:

• The algorithm explores the game tree to a specified depth

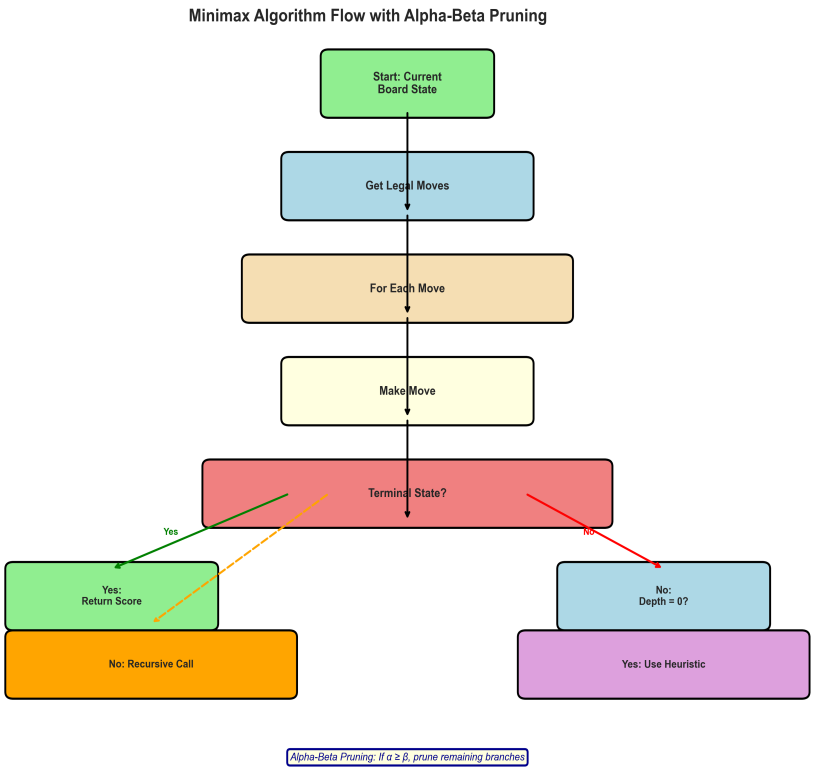
• At terminal nodes (win/loss/draw), it returns a score: +10000 for win, -10000 for loss, 0 for draw

• At non-terminal nodes, it recursively evaluates all possible moves

• The maximizing player (AI) chooses moves that maximize the score

• The minimizing player (opponent) chooses moves that minimize the score

## Minimax Algorithm Flow:




## 3.2 Alpha-Beta Pruning

Alpha-Beta pruning is an optimization technique that reduces the number of nodes evaluated by the Minimax algorithm. It works by eliminating branches that cannot possibly influence the final decision:

• Alpha ($\alpha$): The best value that the maximizing player can guarantee

• Beta ($\beta$): The best value that the minimizing player can guarantee

• If $\alpha \geq \beta$, the remaining branches can be pruned (cut off)

• This can reduce the search space by 50-90% depending on move ordering

## 3.3 Heuristic Evaluation Functions

Since exploring the entire game tree is computationally infeasible (3^64 possible states), heuristic functions are used to evaluate non-terminal positions:

### 3.3.1 Simple Heuristic

The simple heuristic counts open lines for each player. An open line is one that contains only one player's marks (no opponent marks). The score is weighted by the square of the number of marks in the line.

### 3.3.2 Advanced Heuristic

The advanced heuristic provides a more sophisticated evaluation considering:

• Threat detection: 3 in a row with 1 empty (immediate winning threat) = 500 points

• Two in a row patterns = 50 points

• Single piece in line = 5 points

• Center control bonus = 10 points per center position

• Opponent threat blocking: Penalties for opponent threats (must block)

## 3.4 Optimization Techniques

Additional optimization techniques are implemented to improve performance:

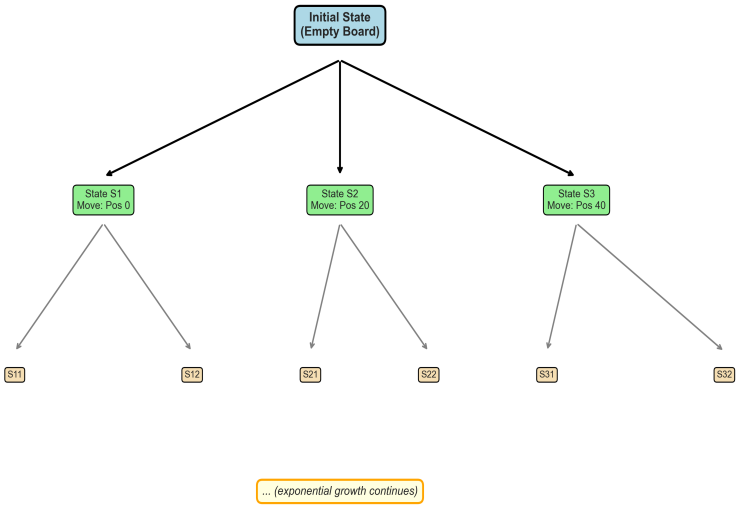
• Transposition Table: Caches previously evaluated board positions

• Symmetry Reduction: Exploits board symmetries to reduce move space

• Heuristic Reduction: Orders moves by heuristic value and explores only top N moves

## State Space Tree (First 3 Levels):

## State Space Tree Diagram
## (First 3 Levels of Game Tree)




*Note: Full state space has 3^64 ≈ 3.43 × 10^30 possible states*
*With 64 initial moves, branching factor is very high*

# 4. User Interface

The game features a comprehensive graphical user interface built with Tkinter, providing an intuitive and informative gaming experience.

## 4.1 Interface Components

• 4×4×4 board visualization displayed as 4 layers in a 2×2 grid

• Color-coded cells: Blue for X, Red for O, Green for winning positions

• Real-time turn indicator showing current player

• AI algorithm settings panel with configurable options

• AI decision analysis display showing:

• - Selected move position

• - Evaluation score

• - Nodes explored

• - Pruned nodes (if Alpha-Beta enabled)

• - Time taken

• - Algorithm configuration used

• Move counter tracking total moves

• New Game button to restart

## 4.2 AI Configuration Options

| Option | Description |
|---|---|
| Heuristic | None, Simple, or Advanced |
| Alpha-Beta Pruning | Enable/disable pruning optimization |
| Symmetry Reduction | Enable/disable symmetry-based move reduction |
| Heuristic Reduction | Enable/disable heuristic-based move ordering |
| Search Depth | Maximum depth for Minimax search (1-5) |

# 5. Performance Analysis

The project includes comprehensive performance analysis tools to evaluate different algorithm configurations and their effectiveness.

## 5.1 Analysis Metrics

• Nodes Explored: Total number of game states evaluated

• Pruned Nodes: Number of branches eliminated by Alpha-Beta pruning

• Pruning Ratio: Percentage of nodes pruned

• Time Elapsed: Total computation time for move selection

• Search Depth: Maximum depth reached in the game tree

### 5.2 Algorithm Comparison

The analysis.py script compares various algorithm configurations:

• Minimax without heuristics (terminal states only)

• Minimax with Simple Heuristic

• Minimax with Advanced Heuristic

• Minimax + Alpha-Beta with Simple Heuristic

• Minimax + Alpha-Beta with Advanced Heuristic

• With Symmetry Reduction

• With Heuristic Reduction

• All optimizations combined

## 6. Technical Specifications

### 6.1 Requirements

• Python 3.6 or higher

• Tkinter (usually included with Python)

• Matplotlib (for diagram generation)

• ReportLab (for PDF generation)

### 6.2 State Space Complexity

The 4×4×4 board has an enormous state space:

```
Total possible states: 3^64 ≈ 3.43 × 10^30
```

This makes exhaustive search impossible, necessitating the use of heuristics and optimization techniques.

## 7. Conclusion

This project successfully implements a complete 3D Tic-Tac-Toe game with an intelligent AI opponent. The combination of Minimax algorithm, Alpha-Beta pruning, and sophisticated heuristic functions creates a challenging and engaging gaming experience.

## 7.1 Key Achievements

• Successfully implemented Minimax algorithm with multiple optimization techniques

• Created an intuitive and informative graphical user interface

• Developed two heuristic evaluation functions with different complexity levels

• Implemented comprehensive performance analysis tools

• Achieved significant performance improvements through Alpha-Beta pruning

## 7.2 Future Enhancements

• Machine learning-based move evaluation

• Improved symmetry reduction algorithms

• Parallel processing for deeper search

• Tournament mode with multiple AI difficulty levels

• Network multiplayer support

--- End of Report ---