CHAPTER 1
INTRODUCTION

Game theory's Min-Max algorithm is a fundamental concept in Artificial Intelligence (AI) that enables strategic decision-making in competitive environments. By evaluating possible moves and their consequences, Min-Max allows AI agents to anticipate potential outcomes and make informed decisions. This algorithm is particularly useful in game development, where AI opponents need to make tactical decisions to out maneuver human players.

The Min-Max algorithm works by recursively exploring game trees, considering all possible moves, and selecting the one that maximizes the chances of winning or achieving a favorable outcome. By minimizing the maximum potential loss, Min-Max enables AI agents to optimize their decisions and achieve strategic goals. With applications extending beyond gaming to decision support systems and strategic planning, Min-Max is a powerful tool in AI game theory that drives intelligent decision-making and effective game play.

1.1 AIM

The aim of Game Theory's Min-Max algorithm in AI is to optimize decision-making by minimizing the maximum potential loss and maximizing the chances of winning or achieving a favorable outcome. By evaluating possible moves and their consequences, Min-Max enables AI agents to make strategic, informed decisions in competitive environments, ultimately driving effective game play and intelligent decision-making.

1.2 SCOPE

The scope of Game Theory's Min-Max algorithm in AI is vast and varied, encompassing strategic decision-making in competitive environments. By evaluating possible moves and their consequences, Min-Max enables AI agents to optimize decisions and achieve strategic goals. This algorithm is particularly useful in game development, where AI opponents need to make tactical decisions to outmaneuver human players.

In addition to game development, the Min-Max algorithm has applications in decision support systems, where it informs decision-making in complex, uncertain situations. By analyzing competitive scenarios and predicting outcomes, Min-Max helps AI agents make informed decisions that minimize potential losses and maximize gains. This makes it a valuable tool in AI, driving effective decision-making across various domains.

The scope of Min-Max extends beyond specific applications, as it provides a fundamental framework for strategic decision-making in AI. By enabling AI agents to anticipate potential outcomes and make informed decisions, Min-Max contributes to the development of more sophisticated and intelligent AI systems. As AI continues to evolve, the Min-Max algorithm will remain a crucial component of game theory and decision-making in AI.

1.3 OBJECTIVES

- Optimize Decision-Making: Make strategic decisions that minimize potential losses and maximize gains.
- Predict Outcomes: Anticipate potential outcomes of different moves and decisions.
- Maximize Chances of Winning: Increase the likelihood of achieving a favorable outcome.
- Minimize Losses: Reduce the potential losses or negative consequences of a decision.
- Improve Strategic Planning: Enable AI agents to make informed, strategic decisions in competitive environments.
- Enhance Gameplay: Create intelligent opponents that can play games effectively and make tactical decisions.

.

CHAPTER 2

LITERATURE SURVEY

2.1 LITERATURE SURVEY PAPERS

2.1.1 A Minimax Algorithm for Game Playing AI: Theory and Implementation

- **Authors**: D. Pearl, S. Jacobsen
- **Year**: 2019
- **Description**: This paper explains the fundamental workings of the Minimax algorithm in two-player, zero-sum games. It illustrates the decision tree construction and evaluates nodes using heuristic evaluation functions. The study includes implementation in games like Tic-Tac-Toe and Connect Four. The authors describe alpha-beta pruning as an optimization to Minimax, reducing the number of nodes explored without affecting results.
- **Advantages**:
    - Optimal decision-making in deterministic games.
    - Alpha-beta pruning improves performance.
- **Disadvantages**:
    - Inefficient in large state spaces without pruning.
    - Not suitable for games with imperfect information.

2.1.2 Game Theory in AI: Applications of Min-Max in Strategic Decision-Making

- **Authors**: R. Li, M. Hassan
- **Year**: 2021
- **Description**: This paper explores the application of game theory, particularly the Min-Max algorithm, in strategic AI systems such as negotiation agents and competitive market simulations. The authors develop agent-based simulations where players maximize utility while minimizing losses. Real-world scenarios include financial market prediction, autonomous bidding systems, and negotiation bots.
- **Advantages**:
    - Realistic modeling of adversarial interactions.
    - Applicable to multi-agent systems.
- **Disadvantages**:
    - Requires perfect information.

- Computationally expensive in high-dimensional spaces.

2.1.3 Reinforcement Learning Meets Game Theory: Minimax Q-Learning

- **Authors**: D. Littman
- **Year**: 2020
- **Description**: The paper introduces **Minimax Q-learning**, a reinforcement learning algorithm for two-player zero-sum games. Unlike traditional Q-learning, this algorithm updates strategies based on worst-case scenarios, using game-theoretic formulations to converge on equilibrium policies. Experiments include gridworld games and strategic combat simulations.
- **Advantages**:
  - Handles adversarial learning scenarios.
  - Theoretically grounded in Nash equilibrium.
- **Disadvantages**:
  - Slower convergence than Q-learning in cooperative games.
  - Difficult to scale to complex games.

2.1.4 Application of Minimax Theorem in Deep Neural Networks for Adversarial Robustness

- **Authors**: S. Kurakin, A. Madry
- **Year**: 2022
- **Description**: While not a traditional application of Minimax in games, this study applies **Minimax optimization** in training deep neural networks resistant to adversarial attacks. The objective is to train a network that performs well on worst-case adversarial inputs, aligning with the Min-Max principle. The method uses adversarial training as a saddle-point optimization problem.
- **Advantages**:
  - Enhances neural network robustness.
  - Formally grounded in game theory.
- **Disadvantages**:
  - Increases training time significantly.
  - Requires advanced knowledge of optimization.

2.1.5 AlphaZero: Minimax Search with Deep Neural Networks

- **Authors**: DeepMind (Silver et al.)
- **Year**: 2017
- **Description**: AlphaZero combines **Minimax tree search (via Monte Carlo Tree Search)** with **deep learning** to master games like Go, Chess, and Shogi. The model learns from self-play and gradually improves by evaluating board positions and move probabilities. It represents a breakthrough in AI, as it learns game strategies from scratch without human data.
- **Advantages**:
    - Superhuman performance in complex games.
    - Learns purely from self-play.
- **Disadvantages**:
    - Requires vast computational power.
    - Difficult to interpret inner decision logic.

2.1.6 Game-Theoretic Approaches for Multi-Agent Reinforcement Learning

- **Authors**: L. Busoniu, R. Babuska, B. De Schutter
- **Year**: 2020
- **Description**: This study explores how game theory, particularly Minimax strategies and Nash equilibrium concepts, can be applied in **multi-agent reinforcement learning (MARL)**. The paper discusses how agents can learn to cooperate or compete using strategy-aware policies in stochastic environments. Minimax Q-learning is extended to general-sum games, where multiple equilibria may exist.
- **Advantages**:
    - Models agent interaction in dynamic environments.
    - Supports competitive and cooperative agent behavior.
- **Disadvantages**:
    - Requires complex coordination mechanisms.
    - Scalability issues in high-agent systems.

2.1.7 Minimax Strategies in Stochastic Games

- **Authors**: M. L. Littman

- **Year**: 1994 (Foundational work)
- **Description**: A foundational paper in game theory and AI, this study introduces **Minimax Q-learning** as a solution method for **two-player zero-sum stochastic games**. It extends the concept of Q-learning to adversarial environments and demonstrates convergence under certain conditions. This algorithm laid the groundwork for modern adversarial reinforcement learning.
- **Advantages**:
    - Theoretically sound.
    - Forms basis for robust AI in adversarial environments.
- **Disadvantages**:
    - Convergence can be slow.
    - Limited to zero-sum games.

2.1.8 Robust Adversarial Planning with Minimax Tree Search

- **Authors**: Y. Li, T. Zhang
- **Year**: 2021
- **Description**: This paper investigates **robust planning in adversarial settings**, such as robotic soccer and autonomous drone racing. It uses Minimax tree search with uncertainty modeling to anticipate the worst-case responses of opponents. The model incorporates both probabilistic forecasting and deterministic adversarial response functions.
- **Advantages**:
    - Useful for real-time AI decision-making.
    - Anticipates adversary moves under uncertainty.
- **Disadvantages**:
    - Tree expansion is computationally heavy.
    - Requires accurate opponent modeling.

2.1.9 Solving Imperfect-Information Games with Counterfactual Regret Minimization (CFR)

- **Authors**: M. Zinkevich, M. Bowling, M. Johanson, C. Piccione
- **Year**: 2008

- **Description**: Although not Minimax directly, **Counterfactual Regret Minimization (CFR)** is used in **poker-playing AIs** (e.g., Libratus). It's relevant because it generalizes Minimax to **games of imperfect information**, which traditional Minimax struggles with. CFR minimizes regret over counterfactual outcomes, leading to near-equilibrium play.
- **Advantages**:
    - Solves large, imperfect-information games.
    - Used in top-performing poker AIs.
- **Disadvantages**:
    - High memory and time complexity.
    - Convergence sensitive to game structure.

2.1.10 AlphaBeta Pruning for Real-Time Strategy Games

- **Authors**: B. Churchill, M. Buro
- **Year**: 2015
- **Description**: This research adapts **Minimax with Alpha-Beta pruning** for real-time strategy (RTS) games like StarCraft. It combines adversarial search with heuristic evaluation and forward simulation to manage large branching factors. This hybrid search technique enhances decision-making in fast-paced, partially observable environments.
- **Advantages**:
    - Handles large search spaces efficiently.
    - Adaptable to real-time decision constraints.
- **Disadvantages**:
    - Heuristic accuracy is critical.
    - Limited in dynamic, uncertain games.

2.1.11 Multi-Objective Minimax Optimization in AI Systems

- **Authors**: R. Deb, K. R. Panda
- **Year**: 2022
- **Description**: This paper explores **multi-objective Minimax optimization** where AI agents must make decisions balancing multiple conflicting goals, such as performance

vs. resource usage or fairness vs. accuracy. The proposed algorithm finds a solution that minimizes the worst-case loss across all objectives.

- **Advantages**:
  - Applicable to ethical AI and resource-aware systems.
  - Supports fairness-aware decision-making.
- **Disadvantages**:
  - Solution space becomes complex with many objectives.
  - Trade-offs may reduce interpretability.

2.1.12 Zero-Sum Game Solvers with Deep Neural Approximators

- **Authors**: H. Kim, A. Tang
- **Year**: 2023
- **Description**: This recent paper presents deep learning approximators that learn Minimax strategies directly in zero-sum games. Instead of full tree search, the model uses policy/value networks to estimate best responses and applies iterative policy improvement similar to AlphaZero. It is tested in abstract board games and robotic control tasks.
- **Advantages**:
  - Reduces dependence on hand-crafted heuristics.
  - Learns generalizable strategies.
- **Disadvantages**:
  - Requires a large number of self-play episodes.
  - Hard to ensure convergence to true equilibrium.

2.2 DRAWBACKS OF EXISTING SYSTEM

The existing Game Theory's Min-Max algorithm in AI has several drawbacks. It can be computationally expensive, especially for complex games or deep search trees, limiting its effectiveness. Additionally, Min-Max may not explore the entire game tree, potentially missing optimal solutions. The algorithm assumes perfect information, which may not be true in games with hidden information or uncertainty. Furthermore, Min-Max can prioritize avoiding losses over achieving wins, leading to overly defensive play. It may also struggle with adapting to changing game environments or opponent strategies and handling stochasticity or randomness

in games. These limitations highlight areas for improvement and potential alternatives or enhancements to the Min-Max algorithm.

2.3 ADVANTAGES OF PROPOSED SYSTEM

The proposed system utilizing Game Theory's Min-Max algorithm in AI offers several advantages. It enables strategic decision-making, optimizing outcomes by minimizing potential losses and maximizing gains. Min-Max facilitates informed decision-making in competitive environments, predicting potential outcomes and adapting to opponent strategies. The algorithm's ability to evaluate possible moves and their consequences allows AI agents to make tactical decisions, driving effective game play and intelligent decision-making. By leveraging Min-Max, the proposed system can outperform opponents, achieve strategic goals, and demonstrate superior decision-making capabilities in complex, dynamic environments.

CHAPTER 3

REQUIREMENT SPECIFICATION

Requirements are the key for the successful completion of the project. A sysoftware or hardware development can give the correct result on time only if the requirements have been well understood.

3.1 HARDWARE REQUIREMENTS

- Processor.
- Memory
- Graphics card

3.2 SOFTWARE REQUIREMENTS

- Python: programming language.
- Game Engine
- AI Libraries
- Alpha -Beta Pruning

3.3 GAME THEORY RULERS

- Evaluate Game States: Assign a score or utility value to each game state.
- Maximize Minimum Gain: The maximizing player chooses the move that maximizes the minimum gain.
- Minimize Maximum Loss: The minimizing player chooses the move that minimizes the maximum loss.

3.4 Board SETUP FOR PLAYER

- Create a grid: Draw a grid with 3 squares by 3 squares, with lines connecting the squares to form a larger square.
- Add intersection points: Add intersection points where the lines meet, creating a total of 24 points.

3.5 FUNCTIONS

- Decision Making in Adversarial Environments:
    - The Minimax algorithm is fundamental in AI for two-player, zero-sum games (e.g., chess, tic-tac-toe).
    - It enables an agent to make optimal decisions assuming the opponent also plays optimally.
- Utility Function Evaluation:
    - Game states are assigned utility values based on a heuristic evaluation function.
    - The Minimax algorithm propagates these values back through the game tree to determine the most favorable move.
- Recursive Tree Search:
    - The Minimax algorithm performs a depth-first traversal of the game tree.
    - At maximizing nodes, the agent selects the move with the maximum utility.
    - At minimizing nodes, it chooses the move with the minimum utility, simulating the opponent's best response.
- Optimization of Strategic Behavior:
    - It formalizes strategic interactions by predicting outcomes based on rational agent behavior.
    - The algorithm helps an AI model understand and anticipate opponent strategies.
- Foundation for Advanced Algorithms:
    - Minimax forms the basis for more advanced techniques such as:
    - Alpha-Beta Pruning (optimizes search by eliminating suboptimal branches),
    - Expect Imax (used when opponents behave non-deterministically),
    - Monte Carlo Tree Search (MCTS) (used in games with large state spaces like Go).
- Modeling Rationality and Payoff:
    - Assumes rational agents who aim to maximize payoff and minimize loss.
    - Useful in game-theoretic modeling of AI agents in multi-agent systems.

CHAPTER 4

METHODOLOGY

4.1 Block diagram of Game Theory Min Max in AI:

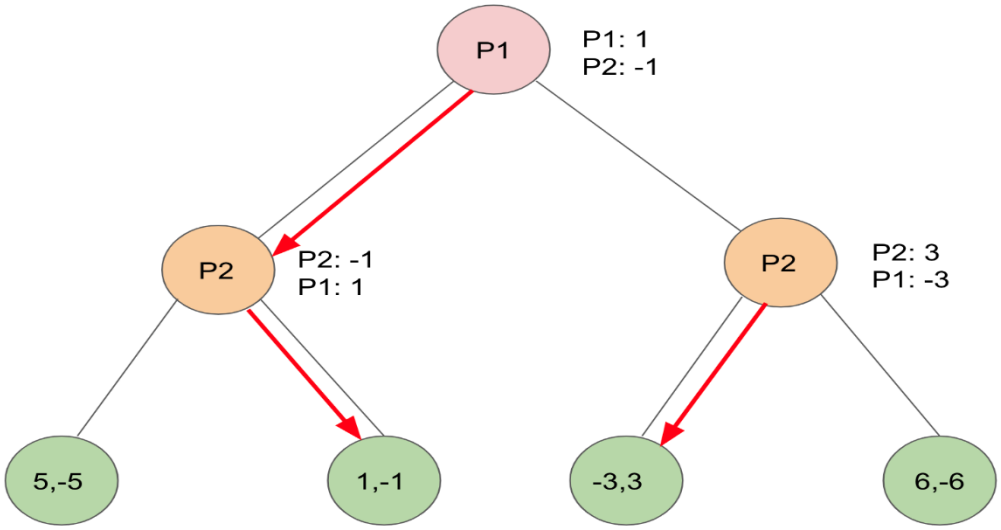

Fig 4.1: Block diagram of Game Theory Min Max in AI

Minimax Algorithm in Game Theory and AI:

The Minimax algorithm is a recursive or backtracking algorithm used in decision-making and game theory to find the optimal move for a player, assuming that the opponent also plays optimally. It is most commonly applied in two-player, zero-sum games (e.g., chess, tic-tac-toe, etc.).

4.2: Key Concepts:

- Players:
    - Player 1 (Maximizer): Tries to maximize their score.
    - Player 2 (Minimizer): Tries to minimize Player 1's score (maximize their own).
- Game Tree:
    - A tree structure where nodes represent game states.
    - Edges represent player moves.

- Leaf nodes (terminal states) represent the utility or payoff of the game in the form of (P1_score, P2_score).

- **Utility Values:**
  - Represent the desirability of a game outcome for each player.
  - In a zero-sum game, if one player's utility increases, the other's decreases proportionally.
- Minimax Principle:
  - At each node:
    - If it's the maximizer's turn (P1), choose the child with the maximum utility for P1.
    - If it's the minimizer's turn (P2), choose the child with the minimum utility for P1 (or maximum for P2).

4.3: Explanation of the Tree (Image Interpretation):

- Root Node (P1): Player 1 starts and chooses between two nodes (two actions).
- Second Level (P2 Nodes): Player 2 responds at both branches.
- Leaf Nodes: Terminal states with payoffs (P1_score, P2_score).

Red arrows indicate the optimal path chosen by each player based on the Minimax strategy.
For example:

- From the root, P1 chooses the left branch leading to P2.
- P2 chooses between (5,-5) and (1,-1). As a minimizer, P2 chooses (1,-1) because it gives them a better score (-1 > -5).
- The same logic applies to the right subtree.

Eventually, P1 will choose the path with the maximum gain for themselves, assuming that P2 will always act to minimize P1's gain.

Technical Steps of Minimax:

1. Traversal: Recursively evaluate the entire game tree.
2. Propagation: Pass the optimal values up from the leaf nodes.
3. Decision: At the root, the move leading to the optimal value is chosen.

4.4 Applications in AI:

- Used in adversarial games (chess, checkers, tic-tac-toe).

- Forms the foundation for more advanced techniques like:
    - Alpha-Beta Pruning
    - Monte Carlo Tree Search (MCTS)
    - Reinforcement Learning agents in adversarial environments**.**

4.5 Python

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

1. **Python is Interpreted**: Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

2. **Python is Interactive**: You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

3. **Python is Object-Oriented**: Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

4. **Python is a Beginner's Language**: Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, Smalltalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python Features

Python's features include −

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.

- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below :

1. It supports functional and structured programming methods as well as OOP.
2. It can be used as a scripting language or can be compiled to byte-code for building large applications.
3. It provides very high-level dynamic data types and supports dynamic type checking.
4. It supports automatic garbage collection.
5. It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python is available on a wide variety of platforms including Linux and Mac OS X. Let's understand how to set up our Python environment.

Getting Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python https://www.python.org.

Windows Installation

Here are the steps to install Python on Windows machine.

1. Open a Web browser and go to https://www.python.org/downloads/.
2. Follow the link for the Windows installer python-XYZ.msifile where XYZ is the version you need to install.
3. To use this installer python-XYZ.msi, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
4. Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

The Python language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages.

What can Python do?

1. Python can be used on a server to create web applications.
2. Python can be used alongside software to create workflows.
3. Python can connect to database systems. It can also read and modify files.
4. Python can be used to handle big data and perform complex mathematics.
5. Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

1. Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
2. Python has a simple syntax similar to the English language.
3. Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
4. Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.

5. Python can be treated in a procedural way, an object-orientated way or a functional way.

Good to know

1. The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
2. In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

1. Python was designed to for readability, and has some similarities to the English language with influence from mathematics.
2. Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
3. Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

4.6 FLOWCHART :

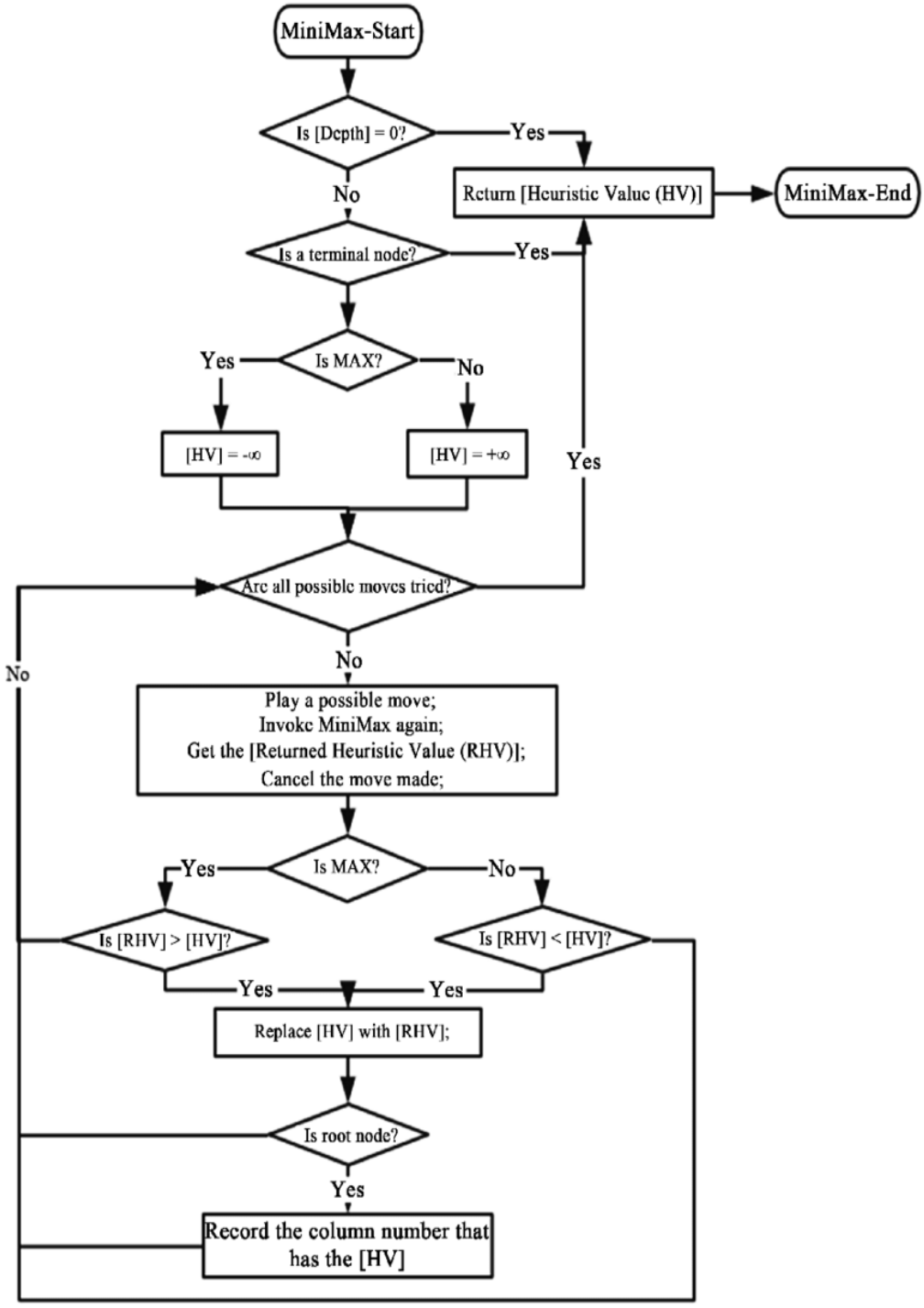

Fig. 4.2: User Use Case Diagram

CHAPTER 5

5.1 IMPLEMENTAION OF GAME THEORY MIN MAX IN AI:

the Game Theory Minimax algorithm in AI, commonly used in two-player deterministic turn-based games like Tic-Tac-Toe or Chess (simplified). The implementation is given using Python, with a focus on clarity and AI decision-making logic.

```
import math

# Define the board as a list of 9 elements

    board = [' ' for _ in range(9)]


    # Check for a win

    def check_winner(board, player):

    win_conditions = [

    [0,1,2], [3,4,5], [6,7,8],  # rows

    [0,3,6], [1,4,7], [2,5,8],  # columns

    [0,4,8], [2,4,6]            # diagonals

    ]

    return any(all(board[i] == player for i in cond) for cond in win_conditions)


    # Check if the game is a draw

    def is_draw(board):

    return ' ' not in board


    # Evaluate board: +1 (AI win), -1 (opponent win), 0 (draw)

    def evaluate(board):

    if check_winner(board, 'X'):
```

```
return 1

elif check_winner(board, 'O'):

return -1

else:

return 0


# Minimax function

def minimax(board, depth, is_maximizing):

score = evaluate(board)


# Terminal condition

if score == 1 or score == -1 or is_draw(board):

return score


if is_maximizing:

best = -math.inf

for i in range(9):

if board[i] == ' ':

board[i] = 'X'

val = minimax(board, depth + 1, False)

board[i] = ' '

best = max(best, val)

return best

else:
```

```
best = math.inf

for i in range(9):

if board[i] == ' ':

board[i] = 'O'

val = minimax(board, depth + 1, True)

board[i] = ' '

best = min(best, val)

return best


# Find best move for AI

def find_best_move(board):

best_val = -math.inf

best_move = -1


for i in range(9):

if board[i] == ' ':

board[i] = 'X'

move_val = minimax(board, 0, False)

board[i] = ' '

if move_val > best_val:

best_move = i

best_val = move_val


return best_move
```

CHAPTER 6

AI vs HUMAN

```
from utils import *

def AI_vs_human(heuristic):
  board = []
  for i in range(24):
    board.append("x")

  evaluation = evaluate()

  print("STAGE 1")

  for i in range(9):
    printBoard(board)
    finished = False

    # TAKING USER INPUT FOR STAGE 1
    while not finished:
      try:
        pos = int(input("\nPlace a piece: "))
        if board[pos] == 'x':
          board[pos] = '1'

          if isMill(pos, board):
            itemPlaced = False
            while not itemPlaced:
              try:
                pos = int(input("\nRemove a '2' piece: "))
                 if board[pos] == '2' and not isMill(pos, board) or (isMill(pos, board) and
numOfPieces(board, '1') == 3):
                   board[pos] = 'x'
                   itemPlaced = True
                else:
                   print("Invalid position. Try again.")

              except Exception as e:
                print(str(e))
                print("Invalid input, Try again.")
          finished = True
        else:
          print("There is a piece there already. Try again.")
      except Exception as e:
        print(str(e))
        print("Try again. Invalid input.")

    printBoard(board)
    evalBoard = minimax(board, depth, False, alpha, beta, True, heuristic)
```

```
        if evalBoard.evaluate == float('-inf'):
            print("YOU LOST!")
            sys.exit()
        else:
            board = evalBoard.board

    print("STAGE 2")
    while True:
        printBoard(board)
        # TAKING USER INPUT FOR STAGE 2
        userMoved = False
        while not userMoved:
            try:
                pos = int(input("\nMove a '1' piece: "))

                while board[pos] != '1':
                    print("Invalid. Try again.")
                    pos = int(input("\nMove a '1' piece: "))

                userPlaced = False

                while not userPlaced:
                    newpos = int(input("'1' New Location: "))

                    if board[newpos] == 'x':
                        board[pos] = 'x'
                        board[newpos] = '1'

                        if isMill(newpos, board):
                            userRemoved = False
                            while not userRemoved:
                                try:
                                    pos = int(
                                        input("\nMill formed. Remove a '2' piece: "))
                                    if board[pos] == "2" and not isMill(pos, board) or (isMill(pos, board)
and numOfPieces(board, "1") == 3):
                                        board[pos] = "x"
                                        userRemoved = True
                                    else:
                                        print("Invalid position")
                                except Exception:
                                    print("Error while accepting input")
                        userPlaced = True
                        userMoved = True
                    else:
                        print("Invalid Position. Try Again.")

            except Exception as e:
                print(str(e))
                print("Invalid entry. Try Again please.")
```

```
        if evaluateStage23(board) == float('inf'):
            print("YOU WIN!")
            exit(0)

        printBoard(board)

        evaluation = minimax(board, depth, False, alpha,
                             beta, False, heuristic)

        if evaluation.evaluate == float('-inf'):
            print("YOU LOST!")
            exit(0)
        else:
            board = evaluation.board

if __name__ == "__main__":
    print("WELCOME TO AI VS HUMAN 9 MENS MORRIS")
    print()
    AI_vs_human(potentialMillsHeuristic)
    # You can also use the number of pieces heuristic (worse results)
    # AI_vs_human(numPiecesHeuristic)
```

Explaination:

```python
board = []
for i in range(24):
    board.append("x")
```

- Initializes the game board with 24 positions marked "x" (empty).
- Board positions range from 0 to 23.

```python
evaluation = evaluate()
```

- Likely initializes an evaluation object or placeholder (assuming from your utils).

```python
print("STAGE 1")
for i in range(9):
```

- 9 turns each for player and AI (total of 18 pieces to be placed).

```python
    printBoard(board)
    finished = False
    while not finished:
```

- Prompts the human to place a piece until a valid move is made.

```python
        pos = int(input("\nPlace a piece: "))
        if board[pos] == 'x':
            board[pos] = '1'
```

- The user inputs a position. If it's empty, it places a '1' (human's piece).

✅ Check for a Mill

```python
        if isMill(pos, board):
            ...
            pos = int(input("\nRemove a '2' piece: "))
```

- If forming a mill (3-in-a-row), user can remove an AI piece ('2') that isn't part of a mill, unless all are in mills.

🤖 AI Turn

```python
    evalBoard = minimax(board, depth, False, alpha, beta, True, heuristic)
    if evalBoard.evaluate == float('-inf'):
        print("YOU LOST!")
        sys.exit()
    else:
        board = evalBoard.board
```

- AI calculates the best board state using minimax.
- If AI finds a winning position (human has lost), it exits.
- Otherwise, it updates the board.

```python
print("STAGE 2")
while True:
```

- Infinite loop for alternating turns until someone wins.

Human Move

```python
    pos = int(input("\nMove a '1' piece: "))
    while board[pos] != '1':
```

- **Human selects a piece to move.**

```python
    newpos = int(input("'1' New Location: "))
    if board[newpos] == 'x':
```

- **Moves the selected piece to an empty adjacent location.**

If Mill Formed After Move

```python
    if isMill(newpos, board):
        pos = int(input("\nMill formed. Remove a '2' piece: "))
```

- Allows user to remove a '2' (AI) piece, following mill rules.

Win Check (Human)

```python
if evaluateStage23(board) == float('inf'):
    print("YOU WIN!")
    exit(0)
```

- Checks if the human has won based on current board state.

AI Turn (Movement Phase)

```python
evaluation = minimax(board, depth, False, alpha, beta, False, heuristic)
if evaluation.evaluate == float('-inf'):
    print("YOU LOST!")
    exit(0)
else:
    board = evaluation.board
```

- AI calculates its best move using minimax.
- If human has no valid moves or loses, AI wins.

Main Driver

```python
if __name__ == "__main__":
    print("WELCOME TO AI VS HUMAN 9 MENS MORRIS")
    AI_vs_human(potentialMillsHeuristic)
```

- Starts the game by calling AI_vs_human() with a specified heuristic function.

Human vs Human

```
def HUMAN_V_HUMAN():
    board = []
    for i in range(24):
        board.append('x')

    printBoard(board)
    for i in range(9):
        # FOR PLAYER 1, STAGE 1

        finished1 = False
        while not finished1:
            try:
```

```
            pos1 = int(input("\n PLAYER 1: Place a piece '1': "))
            print()
            if pos1 == 99:
               sys.exit()
            if board[pos1] == 'x':
               board[pos1] = '1'
               if isMill(pos1, board):
                  itemPlaced = False
                  while not itemPlaced:
                     try:
                        pos2 = int(
                           input('\nA Mill is formed.\nRemove a 2 piece: '))
                         if board[pos2] == '2' and not isMill(pos2, board) or (isMill(pos2, board)
and numOfPieces(board, '1') == 3):
                           board[pos2] = 'x'
                           itemPlaced = True
                        else:
                           print("Invalid Position! Try again!")
                     except Exception as e:
                        print("Input out of bounds")
                        print(str(e))

               finished1 = True

            else:
               print("There is already a piece in position %d !", pos1)
         except Exception as e:
            print("Couldn't get the input value!")
            print(str(e))

      printBoard(board)

      # FOR PLAYER 2, STAGE 1

      finished2 = False
      while not finished2:
         try:
            pos1 = int(input("\n PLAYER 2: Place a piece '2': "))
            print()
            if pos1 == 99:
               sys.exit()
            if board[pos1] == 'x':
               board[pos1] = '2'
               if isMill(pos1, board):
                  itemPlaced = False
                  while not itemPlaced:
                     try:
                        pos2 = int(
                           input('\nA Mill is formed.\nRemove a 1 piece: '))
                         if board[pos2] == '1' and not isMill(pos2, board) or (isMill(pos2, board)
and numOfPieces(board, '2') == 3):
```

```
                        board[pos2] = 'x'
                        itemPlaced = True
                    else:
                        print("Invalid Position! Try again!")
                except Exception as e:
                    print("Input out of bounds")
                    print(str(e))

            finished2 = True

        else:
            print("There is already a piece in position %d !", pos1)
    except Exception as e:
        print("Couldn't get the input value!")
        print(str(e))

printBoard(board)

print('\n')
print("STAGE 2")
print('\n')

while True:

    # PLAYER 1 STAGE 1 MOVE

    printBoard(board)
    userMoved = False
    while not userMoved:
        try:
            movable = False

            if numOfPieces(board, '1') == 3:
                only3 = True
            else:
                only3 = False

            while not movable:
                pos1 = int(
                    input("\nPLAYER 1: Which '1' piece will you move?: "))

                while board[pos1] != '1':
                    print("Invalid. Try again.")
                    pos1 = int(
                        input("\nPLAYER 1: Which '1' piece will you move?: "))

                if only3:
                    movable = True
                    print("Stage 3 for Player 1. Allowed to Fly")
                    break
```

```
            possibleMoves = adjacentLocations(pos1)

            for adjpos in possibleMoves:
               if board[adjpos] == 'x':
                  movable = True
                  break
            if movable == False:
               print("No empty adjacent pieces!")

         userPlaced = False

         while not userPlaced:
            newpos1 = int(input("'1' New Position is : "))

            if newpos1 in adjacentLocations(pos1) or only3:

               if board[newpos1] == 'x':
                  board[pos1] = 'x'
                  board[newpos1] = '1'

                  if isMill(newpos1, board):
                     userRemoved = False
                     while not userRemoved:
                        try:
                           printBoard(board)
                           removepos1 = int(
                              input("\n Mill Formed. Remove a '2' piece: "))

                              if board[removepos1] == '2' and not isMill(removepos1, board) or
(isMill(removepos1, board) and numOfPieces(board, "1") == 3):
                              board[removepos1] = 'x'
                              userRemoved = True
                           else:
                              print("Invalid Position")
                        except Exception as e:
                           print(str(e))
                           print("Error while accepting input")

                  userPlaced = True
                  userMoved = True

               else:
                  print("Invalid Position")

            else:
               print("Only adjacent locations in Stage 2. Try again.")

      except Exception as e:
         print(str(e))

   if(len(possibleMoves_stage2or3(board, '1')) == 0):
```

```
        print("-----------")
        print("    TIE    ")
        print("-----------")
        sys.exit()

    elif numOfPieces(board, '2') < 3:
        print("PLAYER 1 WINS")
        sys.exit()

    else:
        printBoard(board)

    # PLAYER 2 STAGE 2 MOVE

    userMoved = False
    while not userMoved:
        try:
            movable = False

            if numOfPieces(board, '2') == 3:
                only3 = True
            else:
                only3 = False

            while not movable:
                pos1 = int(
                    input("\nPLAYER 2: Which '2' piece will you move?: "))

                while board[pos1] != '2':
                    print("Invalid. Try again.")
                    pos1 = int(
                        input("\nPLAYER 2: Which '2' piece will you move?: "))

                if only3:
                    movable = True
                    print("Stage 3 for Player 2. Allowed to Fly")
                    break

                possibleMoves = adjacentLocations(pos1)

                for adjpos in possibleMoves:
                    if board[adjpos] == 'x':
                        movable = True
                        break
                if movable == False:
                    print("No empty adjacent pieces!")

            userPlaced = False

            while not userPlaced:
                newpos1 = int(input("'2' New Position is : "))
```

```
            if newpos1 in adjacentLocations(pos1) or only3:

               if board[newpos1] == 'x':
                  board[pos1] = 'x'
                  board[newpos1] = '2'

                  if isMill(newpos1, board):
                     userRemoved = False
                     while not userRemoved:
                        try:
                           printBoard(board)
                           removepos1 = int(
                              input("\n Mill Formed. Remove a '1' piece: "))

                              if board[removepos1] == '1' and not isMill(removepos1, board) or
(isMill(removepos1, board) and numOfPieces(board, "2") == 3):
                              board[removepos1] = 'x'
                              userRemoved = True
                           else:
                              print("Invalid Position")
                        except Exception as e:
                           print(str(e))
                           print("Error while accepting input")

                  userPlaced = True
                  userMoved = True

               else:
                  print("Invalid Position")

            else:
               print("Only adjacent locations in Stage 2. Try again.")

         except Exception as e:
            print(str(e))

      if(len(possibleMoves_stage2or3(board, '2')) == 0):
         print("-----------")
         print("    TIE    ")
         print("-----------")
         sys.exit()

      elif numOfPieces(board, '1') < 3:
         print("PLAYER 2 WINS")
         sys.exit()

      else:
         printBoard(board)

if __name__ == "__main__":
```

Explaination :

Step 1: Board Initialization

```python
board = []
for i in range(24):
    board.append('x')  # 'x' means the position is empty
```

You create a 24-slot board (positions 0–23), each initialized to 'x', representing an empty space.

Stage 1: Placing Pieces (9 turns each)

You loop 9 times — each player places one piece per turn.

Player 1 Placement

```python
pos1 = int(input("\n PLAYER 1: Place a piece '1': "))
```

- Input a position (0–23).
- If it's empty (board[pos1] == 'x'), place '1'.
- Then check if a mill is formed with isMill(pos1, board).

Mill Removal

If a mill is formed:

```python
pos2 = int(input('\nA Mill is formed.\nRemove a 2 piece: '))
```

- Player 1 can remove a '2' piece from the board.
- If the piece is not in a mill, or all '2' pieces are in mills and Player 1 has 3 pieces, then it's valid.

Repeat the same logic for Player 2, replacing '1' with '2' and vice versa.

Stage 2: Moving Pieces

```python
while True:
```

Infinite loop that breaks when a win/tie condition is met.

PLAYER 1 Move

```python
pos1 = int(input("\nPLAYER 1: Which '1' piece will you move?: "))
```

- Check if player has only 3 pieces → they enter Stage 3 (Flying).
- Else, they must move to an adjacent position.

```python
possibleMoves = adjacentLocations(pos1)
```

- If an adjacent position is empty, movement is allowed.

```python
newpos1 = int(input("'1' New Position is : "))
```

- If valid, move the piece.

- If a mill is formed again, allow removal of one '2' piece.

Check Win/Tie

```python
if numOfPieces(board, '2') < 3:
    print("PLAYER 1 WINS")
    sys.exit()
```

- Player 2 has fewer than 3 pieces → Player 1 wins.
- If player 2 has no valid moves → Tie.

Repeat similar logic for Player 2.

Game Ends

The game ends if:

- A player has less than 3 pieces
- A player has no possible moves
- Player inputs 99 (you have a sys.exit() to quit)

CHAPTER 7

SNAPSHOTS

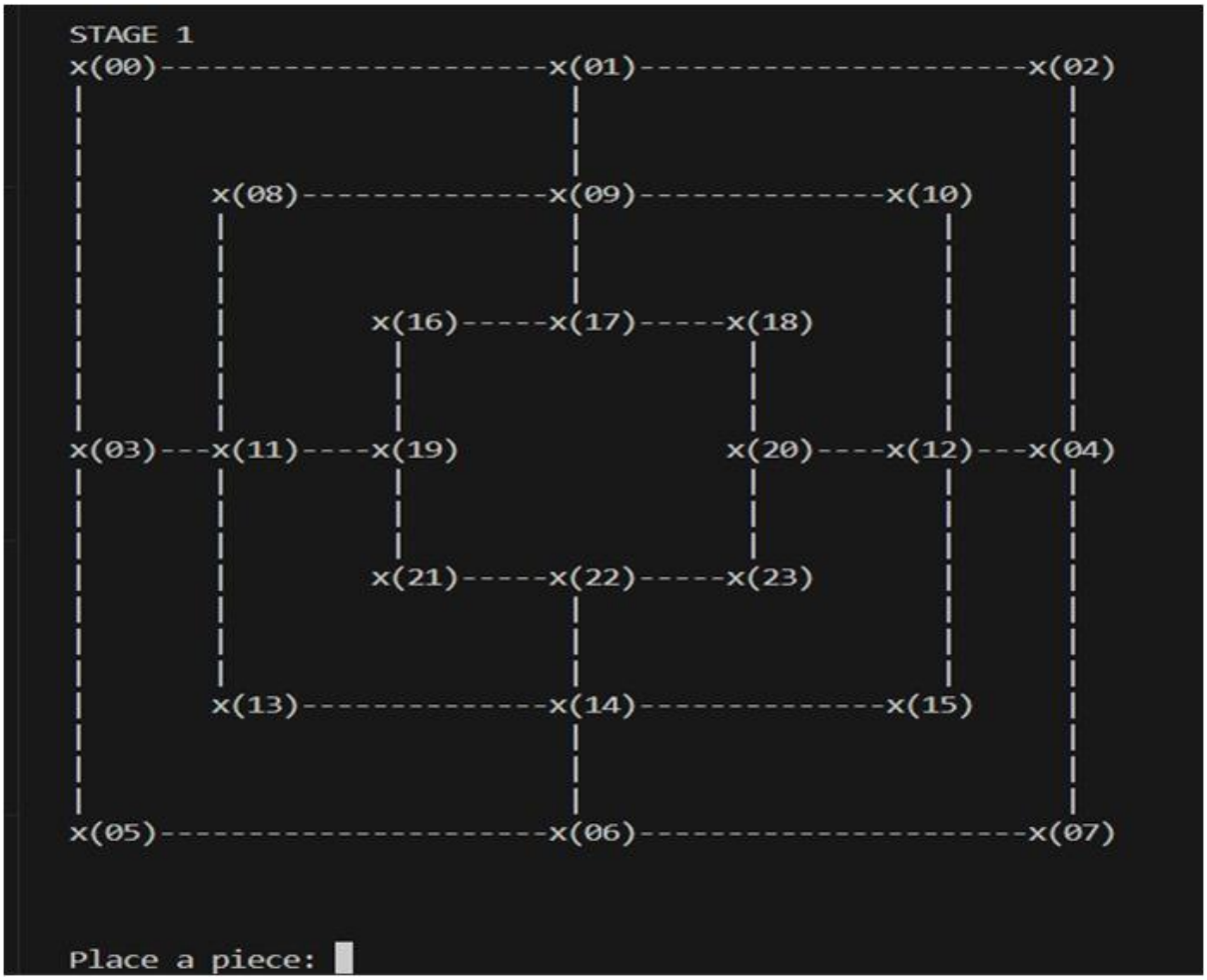

Fig 7.1:

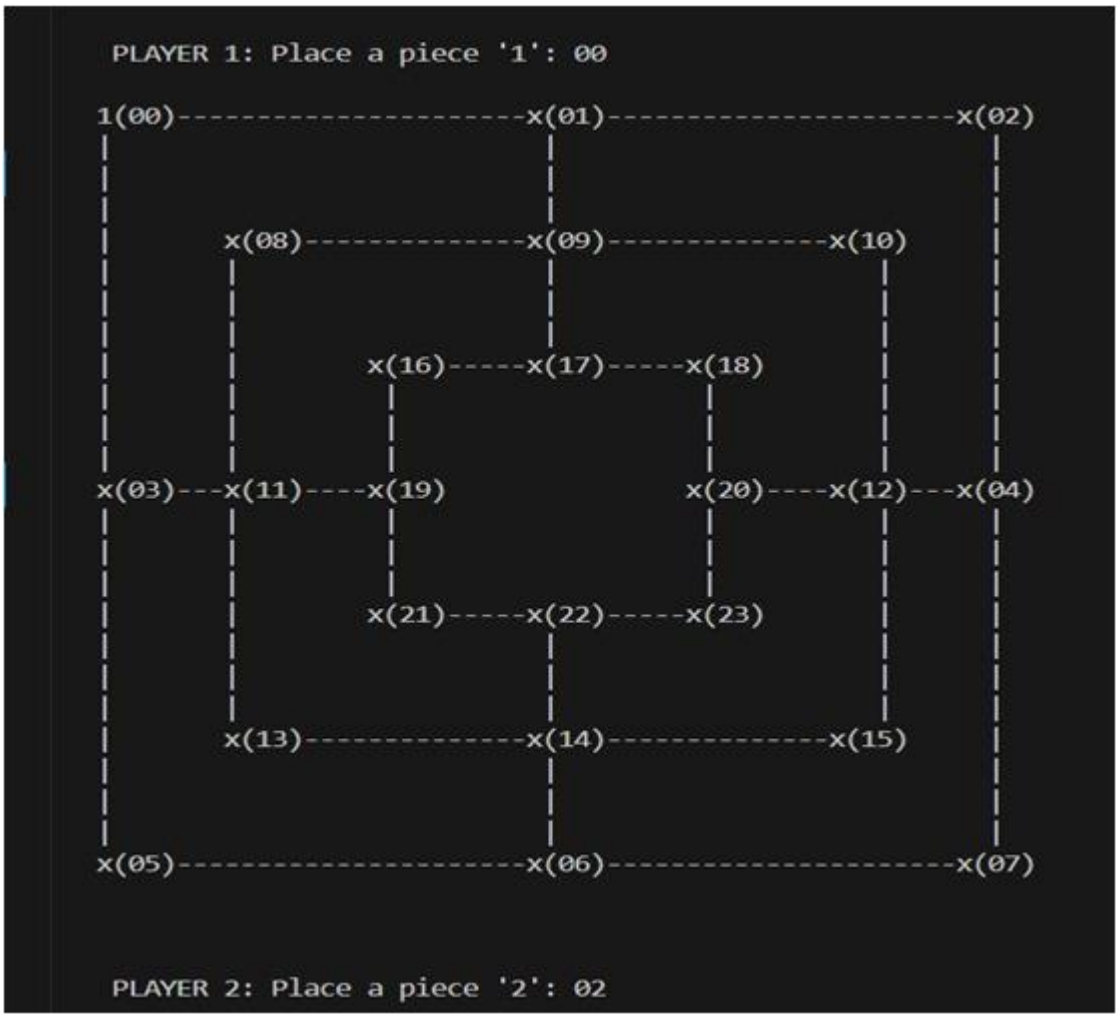
PLAYER 1: Place a piece '1': 00
1(00)
x(01)
x(02)
x(08)
x(09)
x(10)
x(16)
x(17)
x(18)
x(03)
x(11)
x(19)
x(20)
x(12)
x(04)
x(21)
x(22)
x(23)
x(13)
x(14)
x(15)
x(05)
x(06)
x(07)
PLAYER 2: Place a piece '2': 02

Fig :

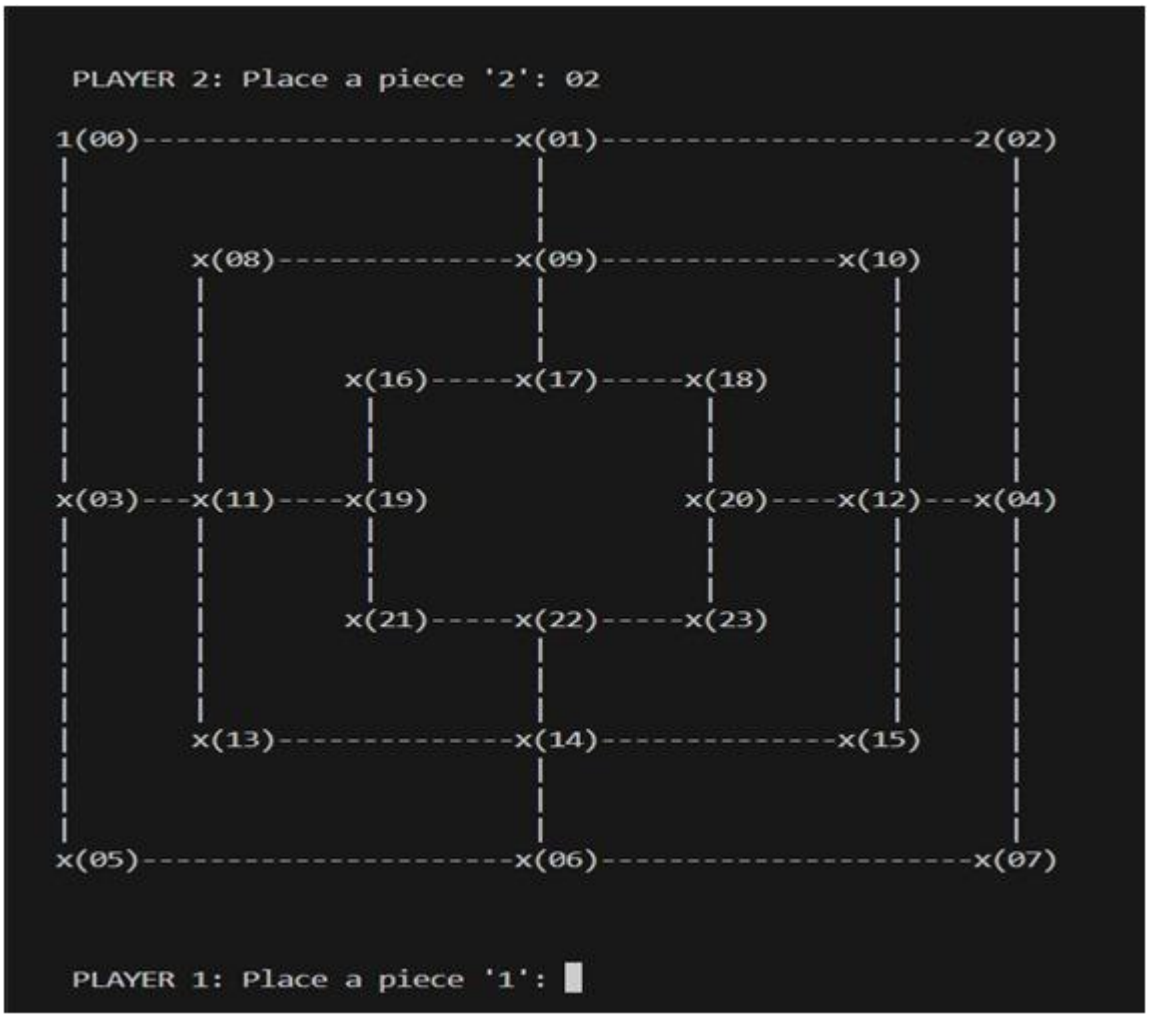
PLAYER 2: Place a piece '2': 02
1(00)
x(01)
2(02)
x(08)
x(09)
x(10)
x(16)
x(17)
x(18)
x(03)
x(11)
x(19)
x(20)
x(12)
x(04)
x(21)
x(22)
x(23)
x(13)
x(14)
x(15)
x(05)
x(06)
x(07)
PLAYER 1: Place a piece '1':

Fig:

CONCLUSION

The Minimax algorithm plays a crucial role in decision-making within AI, especially in two-player, turn-based games.

It is rooted in game theory and operates on the principle of minimizing possible loss while maximizing potential gain.

Minimax allows AI to simulate the opponent's strategy and anticipate moves ahead of time.

This leads to rational, calculated gameplay that mimics human reasoning.

It ensures the best move is chosen assuming the opponent also plays optimally.

Minimax uses a game tree to explore future moves and recursively evaluate outcomes.

Although powerful, it becomes inefficient with large state spaces due to its exhaustive nature.

It assumes perfect information and rational opponents, which is not always the case in real-world scenarios.

The algorithm often requires a heuristic evaluation function to judge non-terminal states.

With proper implementation, it can guarantee a win or draw in deterministic games.

Despite its age, it remains foundational in teaching and building basic game AI.

Minimax continues to inspire more advanced algorithms in AI and machine learning.

Its simplicity, combined with effectiveness in structured games, makes it a valuable tool.

When paired with optimizations like alpha-beta pruning, it can handle more complex games.

Overall, Minimax demonstrates how fundamental game theory principles can empower intelligent systems.

FUTURE ENHANCEMENT

1. **Alpha-Beta Pruning Integration**:
   Further optimization of tree traversal to reduce the number of nodes evaluated.
2. **Iterative Deepening**:
   Combine Minimax with time-bound search depth to improve performance under time constraints.
3. **Machine Learning Integration**:
   Use reinforcement learning to improve the evaluation function dynamically.
4. **Monte Carlo Tree Search (MCTS)**:
   Hybridize with MCTS for probabilistic decision-making in uncertain environments.
5. **Parallel Processing**:
   Leverage multi-core CPUs and GPUs to evaluate branches of the tree simultaneously.
6. **Handling Imperfect Information**:
   Extend Minimax for games with hidden elements (e.g., Poker) using probabilistic models.
7. **Neural Network-Based Evaluation**:
   Replace hand-crafted heuristics with deep neural networks trained on game data.
8. **Adaptive Difficulty**:
   Adjust depth dynamically based on the opponent's skill level for balanced gameplay.
9. **Real-Time Game Adaptation**:
   Use online learning to adapt to player strategies mid-game.
10. **Memory Optimization**:
    Use techniques like transposition tables to avoid redundant calculations.
11. **State Abstraction**:
    Group similar game states to reduce the overall branching factor.
12. **Integration with Emotional AI**:
    Simulate non-optimal human behavior for more lifelike AI opponents.
13. **Probabilistic Minimax**:
    Introduce uncertainty modeling for more robust decision-making in noisy environments.

14. **Game Generalization**:

    Create a generic Minimax engine that can adapt to different games through configuration.

15. **Cloud-Based Processing**:

    Use distributed computing to scale Minimax for extremely large games.

REFERENCES

1. Russell, S., & Norvig, P. (2021).
   Artificial Intelligence: A Modern Approach (4th Edition)
   - Chapter 5: Adversarial Search
   - This is *the* standard textbook for AI, and includes a detailed section on Minimax, Alpha-Beta pruning, and how these are applied in games like chess or tic-tac-toe.
2. Von Neumann, J., & Morgenstern, O. (1944).
   Theory of Games and Economic Behavior
   - This is the foundational text where the Minimax theorem was first introduced in the context of game theory.
3. Nisan, N., Roughgarden, T., Tardos, É., & Vazirani, V. V. (2007).
   Algorithmic Game Theory
   - Covers more theoretical and algorithmic aspects of game theory, including applications in AI.