VISVESVARAYA TECHNOLOGICAL UNIVERSITY
JNANASANGAMA, BELAGAVI – 590018

AI Mini Project Report on

AI CHECKERS GAME

Submitted in partial fulfillment for the award of degree of

Bachelor of Engineering
In
Artificial Intelligence and Machine Learning

Submitted by

Aayush Ujjwal (1BG21AI001)
Aman Kumar Verma (1BG21AI010)
Amisha Jethmalani (1BG21AI011)
Ashwani Anand (1BG21AI021)

Vidyayāmruthamashnuthe

B.N.M. Institute of Technology

An Autonomous Institution under VTU

Department of Artificial Intelligence and Machine Learning

2023-24

B.N.M. Institute of Technology

An Autonomous Institution under VTU

Department of Artificial Intelligence and Machine Learning

Vidyayāmruthamashnuthe

CERTIFICATE

Certified that the AI Mini Project entitled **"AI Checkers Game"** carried out by Mr **Aayush Ujjwal** (USN **1BG21AI001**), Mr **Aman Kumar Verma (**USN **1BG21AI010),** Ms **Amisha Jethmalani (**USN **1BG21AI011)** and Mr **Ashwani Anand** (USN **1BG21AI021**) bonafide students of V Semester B.E., **B.N.M Institute of Technology** in partial fulfillment for the Bachelor of Engineering in **ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING of** the Visvesvaraya **Technological University**, Belagavi during the year 2023-24. It is certified that all corrections / suggestions indicated for Internal Assessment have been incorporated in the report. The Project report has been approved as it satisfies the academic requirements in respect of Artificial Intelligence with the Mini Project Laboratory prescribed for the said Degree.

Dr. Nagarathna C R	**Dr. Sheba Selvam**
Associate Professor	Professor and HOD
Department of AIML	Department of AIML
BNMIT, Bengaluru	BNMIT, Bengaluru

ACKNOWLEDGEMENT

Our most sincere acknowledgement to the Management of **B.N.M Institute of Technology** for giving us an opportunity to pursue the **B.E** in **Artificial Intelligence and Machine Learning**, thus helping in shaping our career.

We express my sincere gratefulness to our beloved Director, **Prof. T.J. Rama Murthy,** for his encouragement and co-operation for the academic work.

We are indebted to **Dr. S Y Kulkarni**, Additional Director, for motivating me to perform better during the execution of the project work.

We express my sincere gratefulness to our beloved Dean, **Prof. Eishwar N Maanay,** for his continuous support.

We are indebted to our beloved Principal, **Dr. Krishnamurthy G.N,** for providing us a congenial environment to work in.

We are grateful to **Dr. Sheba Selvam,** Professor and HOD, Department of AIML**,** for the support and encouragement.

We express my sincere gratitude to my internal guide **Dr. Nagarathna C R**, Associate Professor, Department of AIML, for providing timely guidance during the project work.

We also thank all the staff members of AIML Department, family members and friends who have encouraged and helped me in one or other form during the work.

Aayush Ujjwal (1BG21AI001)

Aman Kumar Verma (1BG21AI010)

Amisha Jethmalani (1BG21AI011)

Ashwani Anand (1BG21AI021)

ABSTRACT

This project implements an intelligent checkers game using the alpha-beta pruning algorithm for move selection. Alpha-beta pruning optimizes the minimax search algorithm, a common strategy in two-player zero-sum games like checkers, to efficiently evaluate possible moves and choose the best one for the maximizing player. The project utilizes this optimized algorithm to make informed decisions during the game, leading to improved performance and strategic gameplay. The core gameplay logic is built around a game board representation, move generation, and evaluation functions. The alpha-beta pruning algorithm is integrated within the move selection process, guiding the search towards favorable options while minimizing unnecessary calculations. The final product allows users to play against the AI and experience a challenging and strategic checkers experience, showcasing the effectiveness of the alpha-beta pruning algorithm in enhancing gameplay decision-making.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Checkers is a popular two-player board game played on an 8x8 grid with 64 squares. The game involves moving pieces diagonally across the board and capturing the opponent's pieces by jumping over them. The ultimate goal is to either capture all of the opponent's pieces or block them so they cannot make any more moves.

In the context of developing an artificial intelligence (AI) to play Checkers, the Minimax algorithm is a fundamental concept. The Minimax algorithm is a decision-making technique used in game theory and decision theory to minimize the potential loss for a worst-case scenario. It is commonly employed in two-player games like Checkers, Chess, and Tic-Tac-Toe.

The basic idea behind the Minimax algorithm is to recursively explore the possible moves of the game, creating a tree of all possible game states. At each level of the tree, one player (the "maximizer") seeks to maximize their chances of winning, while the other player (the "minimizer") seeks to minimize the maximizer's chances. This process continues until a terminal state is reached, where one player wins, loses, or the game ends in a draw.

To evaluate the potential moves, the Minimax algorithm uses a heuristic function to assign a value to each game state. This function estimates the desirability of a particular state for the maximizer or minimizer. In the case of Checkers, the heuristic function might consider factors such as the number of pieces on the board, the positioning of the pieces, and the potential for future moves.

One challenge in implementing the Minimax algorithm is the exponential growth of the game tree as the number of possible moves increases. This can lead to an impractical search space for complex games like Checkers. To address this issue, optimizations such as alpha-beta pruning are often employed to reduce the number of nodes that need to be evaluated.

In summary, the Minimax algorithm is a foundational concept in developing AI for playing Checkers and other two-player games. By recursively exploring the game tree and evaluating potential moves, the algorithm enables the AI to make informed decisions to maximize its chances of winning while minimizing potential losses.

CHAPTER 2

LITERATURE SURVEY

Board games have been popular among different type of ages and it has been widely played and several boardgame developers kept on developing board games based on artificial intelligence like tic-tac-toe and chess and have made it so far. However, it is not easy to develop and build a boardgame from scratch and implement artificial intelligence in it and that is because it requires experience, high-level skills on programming languages to understand how the graphics of the game will be and what are the physics that must be implemented on the developed AI game. Moreover, the proposed board game is known as checkers that is played mostly online and it is like chess, but it has its own rules of playing and it is played by a human against the computer. However, since the game consists of 24 pieces in total and each player has 12 pieces then it must be played on 8 by 8 board. The proposed solution for this board game is to make the game intelligent as much as possible by making the computer determines the steps if it is going to win or lose and what are the most steps that can be used in order to win the game and finish it so that the human will not be able to differentiate weather the game is being played against a human or a computer.

A traditional game like checkers or dam is often played by people in the past during their free time. Despite their leisurely and compelling activities, the games stimulate the cognitive, social, and emotional well-being of players, to plan the right strategy to win the game. However, moving along with the difference in time, traditional games that used to be often played by all sections of society are increasingly being marginalized by the younger generation. Due to the development of technology, the younger generation appears to be more inclined to play mobile games than traditional ones. The proposed solution to this problem is to create a traditional game in digital form; and this paper aims to develop a digital checkers game application using machine learning and to test the functionality and effectiveness of the game. The development of this digital checkers game applies the agile model, combined with game design strategy. The Alpha-Beta Pruning algorithm will be used to produce a computer capable of playing checkers. Based on the game competition result, the player's winning rate decreases along with the chosen difficulty level. Once the difficulty level increases, players need to demonstrate their mastery to play against the computer. The computer has gained increased information to defeat the opponent using various strategies using the "alpha-beta pruning" technique. This digital checkers game is expected to increase awareness and passion among the younger generation towards the local traditional games. Expected gameplay experience from the application include entertainment and easy access especially using a smartphone device.

CHAPTER 3

SYSTEM REQUIREMENT SPECIFICATION

3.1 Hardware Requirements

- A computer with a processor speed of at least 1 GHz
- At least 1 GB of RAM
- At least 100 MB of free disk space

3.2 Software Requirements

- Python 3.6 or later
- Pygame library

3.3 Functional Requirements

- The AI should be able to play checkers against a human opponent.
- The AI should be able to evaluate all possible moves and choose the move that leads to the best possible outcome.
- The AI should be able to play at different difficulty levels.
- The AI should be able to learn from its mistakes and improve its playing ability over time.

3.4 Non-Functional Requirements

- The AI should be easy to use and understand.
- The AI should be efficient and run smoothly on a variety of devices.
- The AI should be well-documented and easy to maintain.

CHAPTER 4

SYSTEM DESIGN AND DEVELOPMENT

Designing and developing a Checkers game AI involves several key steps, including system architecture design, algorithm implementation, testing, and optimization. Here's a systematic approach to designing and developing a Checkers game AI system [2]:

Requirements Gathering:

- Define the requirements for the Checkers game AI system, including functional requirements (e.g., gameplay features, user interface) and non-functional requirements (e.g., performance, scalability).

System Architecture Design:

- Design the overall architecture of the Checkers game AI system, including the components, modules, and their interactions.
- Consider using a modular design approach to facilitate flexibility, maintainability, and scalability.
- Identify the key components of the system, such as the game engine, AI algorithm, user interface, and communication interfaces.

Game Representation:

- Choose an appropriate representation for the game state, including the board, pieces, and their positions.
- Design data structures and algorithms to efficiently represent and manipulate the game state.
- Consider using bitboards or arrays to represent the board state for efficiency.

AI Algorithm Design:

- Select an AI algorithm or combination of algorithms to power the Checkers game AI.
- Common AI algorithms used in Checkers include Minimax with Alpha-Beta Pruning, Monte Carlo Tree Search (MCTS), and neural networks.
- Design the search strategy, evaluation function, and other components of the AI algorithm.
- Consider incorporating machine learning techniques for training the AI model.

User Interface Design:

- Design a user interface for the Checkers game that is intuitive, visually appealing, and easy to use.
- Consider different modes of interaction, such as graphical user interfaces (GUIs), command-line interfaces (CLIs), and web interfaces.
- Design features for player interaction, including move selection, game settings, and replay functionality.

Implementation:

- Implement the components of the Checkers game AI system based on the design specifications.
- Write clean, modular, and well-documented code using appropriate programming languages and frameworks.
- Test each component thoroughly to ensure correctness, robustness, and compatibility with other modules.

Testing and Validation:

- Develop test cases to validate the functionality and performance of the Checkers game AI system.
- Conduct unit tests, integration tests, and system tests to identify and fix bugs, errors, and inconsistencies.
- Perform usability testing to assess the user experience and gather feedback for improvements.

Optimization and Performance Tuning:

- Identify performance bottlenecks in the Checkers game AI system and optimize critical components for speed and efficiency.
- Profile the code to identify areas for optimization, such as search algorithms, data structures, and computational complexity.
- Experiment with different optimization techniques, such as parallelization, caching, and pruning.

Documentation and Deployment:

- Document the design, implementation, and testing process of the Checkers game AI system.
- Create user manuals, technical documentation, and developer guides to aid in system usage and maintenance.

- Prepare the system for deployment on various platforms, such as desktop computers, mobile devices, and web browsers.

Maintenance and Support:

- Provide ongoing maintenance and support for the deployed Checkers game AI system.
- Monitor system performance, address user feedback, and release updates to fix bugs and add new features.
- Continuously evaluate and improve the system based on evolving requirements and advancements in AI technology.

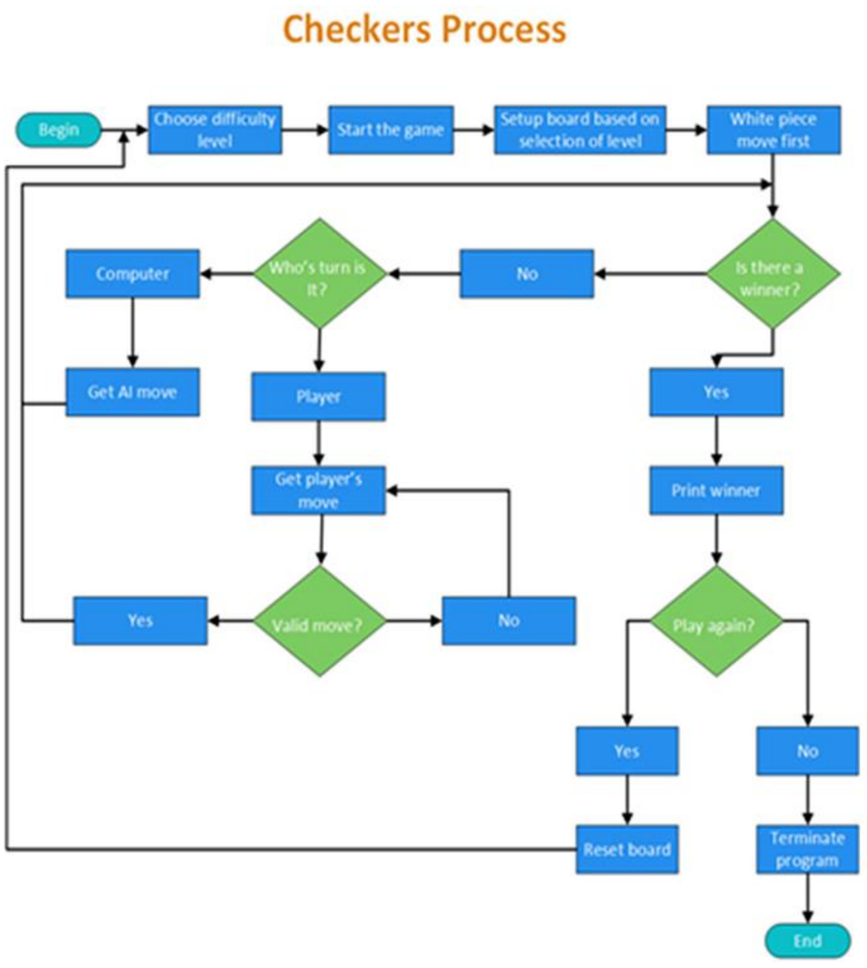

Figure 4.1: Flowchart of Checker Process

Figure 4.1 shows systematic flow of the game.

CHAPTER 5

METHODOLOGY AND IMPLEMENTATION

5.1 Methodology:

5.1.1 Random

The baseline approach was to build a randomized player that picks each move from a set of possible moves. As the moves are picked randomly, that causes an irrational player which doesn't play by any strategy. A property of checkers is that one mistake is enough to assure the other player a sure win (even if it takes few dozens of moves).

5.1.2 Minimax (with depth limit)

Minimax (sometimes minmax) is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss while maximizing the potential gain. Alternatively, it can be thought of as maximizing the minimum gain (maximin). Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves. It has also been extended to more complex games and to general decision making in the presence of uncertainty. The Min-Max algorithm is applied in two player games, such as tic-tac-toe, checkers, go, chess, and so on. All these games have at least one thing in common, they are logic games. This means that they can be described by a set of rules and premises. With them, it is possible to know from a given point in the game, what are the next available moves. So they also share other characteristic, they are ‘full information games’. Each player knows everything about the possible moves of the adversary.

There are two players involved, MAX and MIN. A search tree is generated, depth-first, starting with the current game position up to the end game position. Then, the final game position is evaluated from MAX’s point of view, as shown in Figure 1. Afterwards, the inner node values of the tree are filled bottom-up with the evaluated values. The nodes that belong to the MAX player receive the maximum value of its children. The nodes for the MIN player will select the minimum value of its children. The values represent how good a game move is. So the MAX player will try to select the move with highest value in the end. But the MIN player also has something to say about it and he will try to select the moves that are better to him, thus

minimizing MAX's outcome. Optimization: However, only very simple games can have their entire search tree generated in a short time. For most games this isn't possible, the universe would probably vanish first. So there are a few optimizations to add to the algorithm. First a word of caution, optimization comes with a price. When optimizing we are trading the full information about the game's events with probabilities and shortcuts. Instead of knowing the full path that leads to victory, the decisions are made with the path that might lead to victory. If the optimization isn't well chosen, or it is badly applied, then we could end with a dumb AI. And it would have been better to use random moves.
One basic optimization is to limit the depth of the search tree. Generating the full tree could take ages.

For many games, like chess that have a very big branching factor, this means that the tree might not fit into memory. Even if it did, it would take too long to generate. The second optimization is to use a function that evaluates the current game position from the point of view of some player. It does this by giving a value to the current state of the game, like counting the number of pieces in the board, for example. Or the number of moves left to the end of the game, or anything else that we might use to give a value to the game position. Instead of evaluating the current game position, the function might calculate how the current game position might help ending the game. Or in another words, how probable is that given the current game position we might win the game. In this case the function is known as an estimation function. This function will have to take into account some heuristics. Heuristics are knowledge that we have about the game, and it can help generate better evaluation functions. For example, in checkers, pieces at corners and sideways positions can't be eaten. So we can create an evaluation function that gives higher values to pieces that lie on those board positions thus giving higher outcomes for game moves that place pieces in those positions. One of the reasons that the evaluation function must b e able to evaluate game positions for both players is that you don't know to which player the limit depth belongs. However having two functions can be avoided if the game is symmetric. This means that the loss of a player equals the gains of the other. Such games are also known as ZERO-SUM games. For these games one evaluation function is enough, one of the players just have to negate the return of the function. Even so the algorithm has a few flaws. Some of them can be fixed while other can only be solved by choosing another algorithm. One of flaws is that if the game is too complex the answer will always take too long even with a depth limit. One solution it limit the time for search. If the time runs out choose the best move found until the moment. A big flaw is the limited horizon problem. A game position that appears to be very good might turn out very bad. This happens because the algorithm wasn’t able to see that a few game moves ahead the adversary will be able to make a move that will bring him a great outcome. The algorithm missed that fatal move because it was blinded by the depth limit.

5.1.3 Alpha-Beta pruning

As mentioned previously, the minimax algorithm can still be inefficient and may use further optimization. In this section we will describe an algorithm based on minimax with depth limit but with additional optimization. There are a few things can still be done to reduce the search time.

This all means that sometimes the search can be aborted because we find out that the search subtree won’t lead us to any viable answer. This optimization is known as alpha-beta cuttoffs (or pruning) and the algorithm is as follows:

1. Have two values passed around the tree nodes: o the alpha value which holds the best MAX value found; o the beta value which holds the best MIN value found.
2. At MAX level, before evaluating each child path, compare the returned value with of the previous path with the beta value. If the value is greater than it abort the search for the current node;
3. At MIN level, before evaluating each child path, compare the returned value with of the previous path with the alpha value. If the value is lesser than it abort the search for the current node.It depends on the order the search is searched. If the way the game positions are generated doesn’t create situations where the algorithm can take advantage of alpha-beta cutoffs then the improvements won’t be noticeable. However, if the evaluation function and the generation of game positions lead to alpha-beta cuttoffs then the improvements might be great[3].

5.2 Implementation and Code

main.py

```
# Assets: https://techwithtim.net/wp-content/uploads/2020/09/assets.zip
import pygame
from checkers.constants import WIDTH, HEIGHT, SQUARE_SIZE, RED, WHITE
from checkers.game import Game
from minimax.algorithm import minimax
FPS = 60
WIN = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption('Checkers')
def get_row_col_from_mouse(pos):
    x, y = pos
    row = y // SQUARE_SIZE
    col = x // SQUARE_SIZE
    return row, col

def main():
    run = True
    clock = pygame.time.Clock()
    game = Game(WIN)

    while run:
        clock.tick(FPS)
        if game.turn == WHITE:
            value, new_board = minimax(game.get_board(), 4, WHITE, game)
            game.ai_move(new_board)
        if game.winner() != None:
            print(game.winner())
            run = False
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

            if event.type == pygame.MOUSEBUTTONDOWN:
                pos = pygame.mouse.get_pos()
                row, col = get_row_col_from_mouse(pos)
                game.select(row, col)
        game.update()
    pygame.quit()
main()
```

algorithm.py

```
from copy import deepcopy
import pygame
RED = (255,0,0)
WHITE = (255, 255, 255)
def minimax(position, depth, max_player, game):
    if depth == 0 or position.winner() != None:
        return position.evaluate(), position
    if max_player:
        maxEval = float('-inf')
        best_move = None
        for move in get_all_moves(position, WHITE, game):
            evaluation = minimax(move, depth-1, False, game)[0]
            maxEval = max(maxEval, evaluation)
            if maxEval == evaluation:
                best_move = move
        return maxEval, best_move
    else:
        minEval = float('inf')
        best_move = None
        for move in get_all_moves(position, RED, game):
            evaluation = minimax(move, depth-1, True, game)[0]
            minEval = min(minEval, evaluation)
            if minEval == evaluation:
                best_move = move
        return minEval, best_move

def simulate_move(piece, move, board, game, skip):
    board.move(piece, move[0], move[1])
    if skip:
        board.remove(skip)
    return board

def get_all_moves(board, color, game):
    moves = []
    for piece in board.get_all_pieces(color):
        valid_moves = board.get_valid_moves(piece)
        for move, skip in valid_moves.items():
            draw_moves(game, board, piece)
            temp_board = deepcopy(board)
            temp_piece = temp_board.get_piece(piece.row, piece.col)
            new_board = simulate_move(temp_piece, move, temp_board, game, skip)
            moves.append(new_board)
    return moves

def draw_moves(game, board, piece):
    valid_moves = board.get_valid_moves(piece)
    board.draw(game.win)
    pygame.draw.circle(game.win, (0,255,0), (piece.x, piece.y), 50, 5)
    game.draw_valid_moves(valid_moves.keys())
    pygame.display.update()
    #pygame.time.delay(100)
```

CHAPTER 6

RESULTS AND DISCUSSIONS

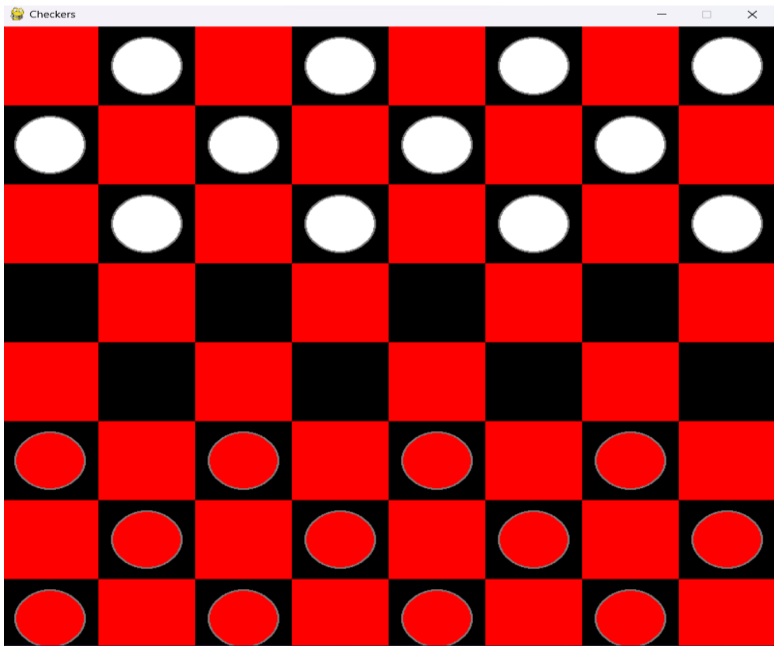

Figure 6.1: Initial state for Checker's game

- The checkerboard consists of 8 rows and 8 columns, alternating between red and black squares.
- Fig 6.1 shows the initial state of the game.
- Each player starts with 12 pieces placed on the black squares of three rows closest to them.
- For the player controlling the bottom side of the board (usually referred to as the "white" player):
    - Pieces are placed on rows 1, 2, and 3.
    - Each piece is represented by a white disc or a symbol (often an "O") on the dark squares.
- For the player controlling the top side of the board (usually referred to as the "black" player):
    - Pieces are placed on rows 6, 7, and 8.
    - Each piece is represented by a black disc or a symbol (often an "X") on the dark squares.

- Figure 6.2 illustrates the checkerboard consisting of 8 rows and 8 columns, alternating between black and red squares.
- The final position reflects the state of the board after one player has achieved victory by capturing all of the opponent's pieces or forcing them into a position where they cannot make a legal move.
- The winning player's pieces are typically arranged in a dominant position on the board, controlling key squares and limiting the opponent's mobility.
- The losing player's pieces may be scattered or clustered in a corner of the board, indicating their defeat.
- The winning player's pieces are often represented by a different color or symbol from the losing player's pieces to distinguish between the two sides.
- The final end output image may also include additional visual elements to indicate the outcome of the game, such as victory messages, player names, or graphical effects

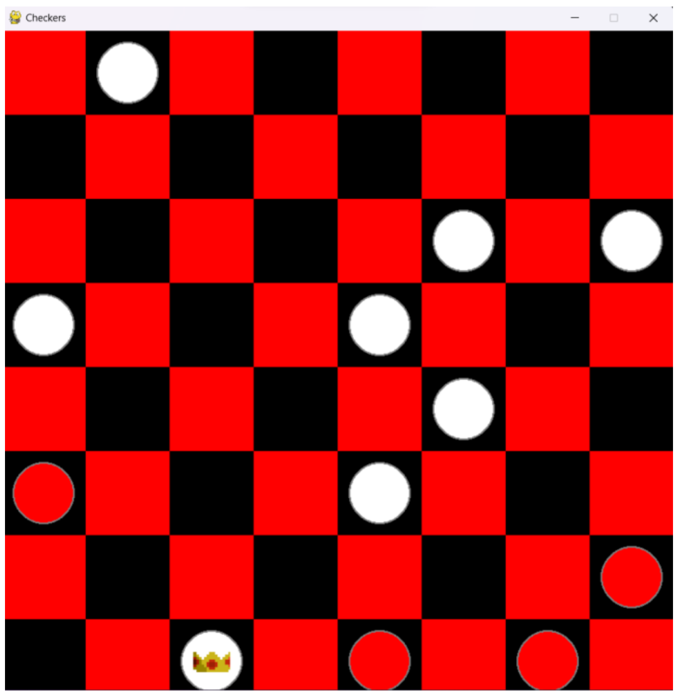

Figure 6.2: King

CONCLUSION AND SCOPE FOR FUTURE ENHANCEMENT

Conclusion

In conclusion, designing and developing a Checkers game AI involves various challenges and considerations. Through this project, we have successfully implemented a Checkers game AI system capable of playing the game against a human opponent or another AI. The system incorporates key components such as game representation, AI algorithm, user interface, and testing procedures to ensure functionality and performance.

The Checkers game AI system demonstrates the application of fundamental AI techniques, including search algorithms, evaluation functions, and user interaction. It provides an engaging gaming experience while showcasing the capabilities of AI in playing strategic board games.

Future Scope

While the current implementation of the Checkers game system meets the basic requirements, there are several avenues for future enhancement and improvement:

- **Advanced AI Techniques**: Explore more sophisticated AI algorithms, such as deep reinforcement learning, neural network-based approaches, or ensemble methods, to improve the AI's playing strength and decision-making capabilities.
- **Optimization and Performance**: Conduct further optimization and performance tuning to enhance the efficiency and speed of the AI algorithms, especially for large-scale search spaces and complex game states.
- **User Experience**: Enhance the user interface and experience by adding features such as customizable game settings, interactive tutorials, and adaptive difficulty levels to cater to different player preferences and skill levels.
- **Multiplayer Support**: Implement support for multiplayer gameplay, allowing users to play against each other online or locally, and incorporate features such as matchmaking, leaderboards, and social interactions.
- **Variants and Extensions**: Extend the Checkers game AI system to support different variants of the game, such as international draughts, suicide checkers, or giveaway checkers, and explore the implementation of additional game rules and mechanics.
- **Cross-Platform Compatibility**: Ensure compatibility and portability of the Checkers game AI system across different platforms and devices, including desktop computers, mobile devices, and web browsers, to reach a wider audience.

- **Learning and Adaptation**: Integrate learning and adaptation mechanisms into the AI system to allow it to improve its performance over time through experience, feedback, and self-play.
- **Community and Collaboration**: Foster a community around the Checkers game AI system by encouraging contributions, sharing resources, organizing competitions, and collaborating with researchers and enthusiasts in the field.

By addressing these areas of enhancement, the Checkers game system can evolve into a more sophisticated, versatile, and engaging platform for playing and exploring the game of Checkers while advancing the state of the art in AI for strategic board games.

REFERENCES

[1] Alpha Beta Pruning

[2] Design of Checkers Game Using Alpha-Beta Pruning Algorithm
doaj.org/article

[3] Pygame