Checkers AI Project Proposal

Ahmed Nour
Lubna Al Rifaie
Anahita Sumoreeah

Wilfrid Laurier University

alri1590@myluarier.ca
nour2050@mylaurier.ca
sumo5204@mylaurier.ca

***Abstract* - This paper details the creation of Checkers AI, an artificial intelligence system designed to excel in the game of checkers. By integrating machine learning algorithms, game theory principles, and strategic analysis, the project aims to develop an AI capable of adapting to and exploiting various game dynamics. The AI's foundation was built on a dataset of human and AI checker games. Utilizing a minimax algorithm with future improvement through alpha-beta pruning, the AI optimizes its play strategy through iterative learning processes.**

The Checkers AI demonstrates remarkable competencies across different levels of play, showcasing its ability to challenge expert human players. This advancement not only elevates the AI's strategic gameplay but also contributes to the broader field of AI research, highlighting the potential of artificial intelligence in complex decision-making scenarios.

***Keywords* - Artificial Intelligence (AI), Checkers, Machine Learning, Game Theory, Strategic Analysis, Minimax Algorithm, Decision-Making tree, Competitive Play**

I. INTRODUCTION

Creating a Checkers game for an AI project offers a fascinating blend of challenges and opportunities that make it an excellent choice for both educational and research purposes. At its core, Checkers is a game with simple rules but deep strategic complexity, making it an ideal sandbox for AI development. This balance allows us to focus on the fundamental aspects of AI, such as decision-making, strategic planning, and opponent prediction, without getting bogged down by overly complex game mechanics. Also, the fun fact the first ever computer program to win a world championship against a human was a checker's AI called Chinook created by our very own Canadian computer science students from the University of Alberta.

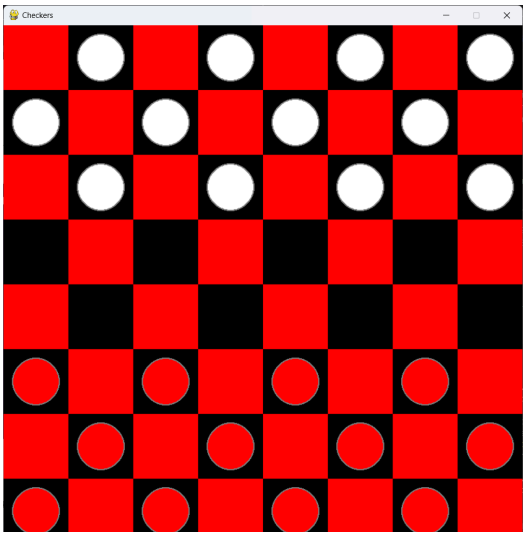
II. Game Rules

Checkers is played by two opponents on opposite sides of the game board. One player has dark pieces, the other has light pieces. A player moves first, then players alternate turns. A move consists of moving a piece forward to an adjacent unoccupied

square. If the adjacent square contains an opponent's piece, and the square immediately beyond it is vacant, the piece may be captured (and removed from the game) by jumping over it. There are several different types of rules for the game of Checkers depending on where in the world the game is taking place. Our checker's AI follows the standard Hasbro rules where play consists of advancing a piece diagonally forward to an adjoining vacant square. Black moves first. If an opponent's piece is in such an adjoining vacant square, with a vacant space beyond, it must be captured and removed by jumping over it to the empty square. If this square presents the same situation, successive jumps forward in a straight or zigzag direction must be completed in the same play. When there is more than one way to jump, the player has a choice. When a piece first enters the king row, the opponent's back row, it must be crowned by the opponent, who places another piece of the same colour on it. The piece, now called a king, has the added privilege of moving and jumping backward; if it moved to the last row with a capture, it must continue capturing backward if possible. No other piece other than the king can jump backwards to capture. A win is scored when an opponent's pieces are all captured or blocked so that they cannot move. When neither side can force a victory and the trend of play becomes repetitious, a draw game is declared.

III. Game Design

Before we were able to implement an AI in our game, the first task at hand was to create the board, the pieces, and the basic algorithm providing the logic for the game of checkers, adhering to the rules we've decided to have for our version of Checkers AI. Firstly we will begin with our constants file. Within our constants file we have a series of constants for the dimensions of our game which are listed and defined below.



CONSTANTS:

- **constants.py:** Contains the following constants used in the game:
- **WIDTH, HEIGHT:** Dimensions of the game window.
- **ROW, COLS**: Number of rows and columns on the checkers board.
- **SQAURE_SIZE:** Size of each square on the checker's board, calculated based on WIDTH and COLS.

```
1    import pygame
2
3    WIDTH, HEIGHT = 800, 800
4    ROWS, COLS = 8, 8
5    SQUARE_SIZE = WIDTH//COLS
```

Usage of Constants

The constants defined in constants.py are used throughout the game to maintain consistency in dimensions, colors and other parameters. These constants are utilized in various components such as the game board, pieces and user interface to ensure uniformity and ease of maintenance.

Next, we create the piece.py file which contains the class Piece which represents a singular piece on the checker's board. This class includes methods such as

PIECE:

- **_ _ init _ _:** Initializes the piece with its position and color.
- **calc_pos:** Calculates the position of the piece on the screen.
- **make_king:** Makes the piece a king.
- **draw:** Draw the piece on the screen.
- **move:** Move the piece to a new position on the board.

```python
from .constants import RED, WHITE, SQUARE_SIZE, GREY, CROWN
import pygame

class Piece:
    PADDING = 15
    OUTLINE = 2

    def __init__(self, row, col, color):
        self.row = row
        self.col = col
        self.color = color
        self.king = False
        self.x = 0
        self.y = 0
        self.calc_pos()

    def calc_pos(self):
        self.x = SQUARE_SIZE * self.col + SQUARE_SIZE // 2
        self.y = SQUARE_SIZE * self.row + SQUARE_SIZE // 2

    def make_king(self):
        self.king = True

    def draw(self, win):
        radius = SQUARE_SIZE//2 - self.PADDING
        pygame.draw.circle(win, GREY, (self.x, self.y), radius + self.OUTLINE)
        pygame.draw.circle(win, self.color, (self.x, self.y), radius)
        if self.king:
            win.blit(CROWN, (self.x - CROWN.get_width()//2, self.y - CROWN.get_height()//2))
```

A key portion of this file to pay attention to is the calculate position method. What this does is calculate our x and y position based on the row and columns that we're in. In the first line of this method, you can see that self.x = square size * self.col because in pygame when you move horizontally you move on the x-axis which is columns and when you move vertically you move on the y-axis. Another key portion is def draw. The idea for this portion is to begin drawing the circle and then outlining the circle so that we can see it well enough. So for this, we need to determine what we want our radius to be and that will be based on how much padding we want between the edge of the square and the circle so we need to define that amount of pixels through the class variable padding and we also define the outline. Since the piece we are drawing is a circle the radius would be from the middle of the square so square size divided by 2, minus the padding we defined. For the outline, we had to make a different colour (grey), so that the piece could be seen, therefore, we took the radius plus the outline.

Usage of Piece Class:

The Piece class in piece.py represents a singular piece on the checker's board. It stores attributes such as the position, color and whether the piece is a king. The class provides methods for drawing a piece on the screen, making it a king and moving it to a new position.

Lastly, we move on to the board.py file which contains the class Board which handles the state of the actual checkers board. It will hold all of the pieces, allowing us to check what the valid moves are. This class includes methods such as

BOARD:

- **draw_squares:** Draws the squares of the checker's board.
- **evaluate:** Evaluate the current board configuration.
- **get_all_pieces:** Gets the piece at a given position.
- **create_board:** Initializes the checker's board with pieces.
- **draw**: Draw the checkerboard and pieces.
- **remove**: Removes pieces from the board.
- **winner:** Determines the winner of the game.
- **get_valid_moves:** Gets valid moves for a given piece.

```
71      def get_valid_moves(self, piece):
72          moves = {}
73          left = piece.col - 1
74          right = piece.col + 1
75          row = piece.row
76
77          if piece.color == RED or piece.king:
78              moves.update(self._traverse_left(row -1, max(row-3, -1), -1, piece.color, left))
79              moves.update(self._traverse_right(row -1, max(row-3, -1), -1, piece.color, right))
80          if piece.color == WHITE or piece.king:
81              moves.update(self._traverse_left(row +1, min(row+3, ROWS), 1, piece.color, left))
82              moves.update(self._traverse_right(row +1, min(row+3, ROWS), 1, piece.color, right))
83
84          return moves
```

A key portion of this file is the get_valid_moves function which is responsible for the logistical algorithm for the game of checkers. In this snapshot here, we can see the basic algorithm for determining the traversal search depending on piece colour. We set a depth of search no greater than 2 rows ahead to ensure efficiency.

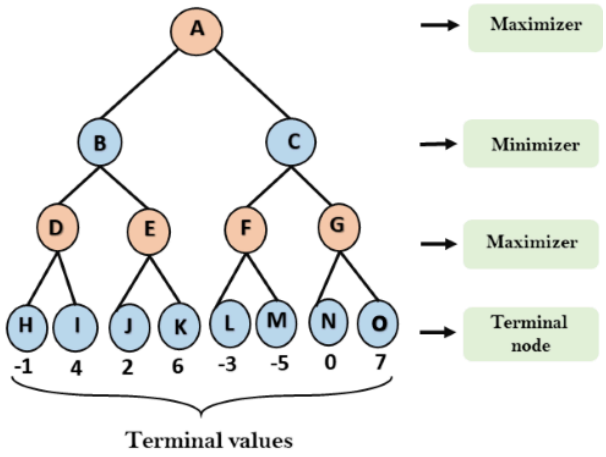
IV. Minimax Algorithm: An In-depth Analysis

Introduction to Minimax

Minimax is a pivotal decision-making algorithm predominantly utilized in the realm of two-player turn-based games. Its fundamental purpose is to ascertain the most advantageous move for a player under the presumption that their adversary is also making optimally calculated decisions. The algorithm accomplishes this by methodically examining potential future game scenarios, ultimately aiming to optimize the player's probability of success while concurrently minimizing the opponent's chances.

Exploring the Decision Tree: A Visual Representation

A core component of understanding the Minimax algorithm is visualizing its decision-making process through a Decision Tree. The root of this tree symbolizes the present state of the game, typically depicted as the initial positioning on the game board. As one navigates through each node of the tree, one encounters various possible moves that could be executed by the current player. This branching structure allows the algorithm to explore a multitude of game dynamics and their respective outcomes.



Algorithmic Flow Through the Tree

Initiating from the root, the algorithm traverses the Decision Tree via a depth-first search methodology. It alternates between maximizing and minimizing players at each tree level. The objective for maximizing players is to increase the evaluation function's value, whereas minimizing players aims to decrease it. At the terminal nodes, the algorithm assesses the game's state using an evaluation function, which quantitatively reflects the state's desirability from the player's perspective. Through a process of backtracking and aggregating these evaluation scores, the algorithm can determine the most beneficial move for the maximizing player based on the current game state.

Key Components of the Minimax Algorithm

The efficiency and effectiveness of the Minimax algorithm are contingent upon several key components, as highlighted below:

- **Evaluation Function (`position.evaluate()`):** This function plays a crucial role in determining the desirability of a given board state. It assesses the state based on various metrics, such as the number of pieces each player possesses, thereby providing a numerical value indicative of the state's benefit to the player.
- **Legal Moves Enumeration (`get_all_moves()`):** Identifying all permissible moves from a current board state is vital. This function generates a list of all potential board states resulting from legal player actions.
- **Depth of Search (`depth` in `minimax()`):** The `depth` parameter is instrumental in defining the algorithm's foresight, dictating how far ahead in the game it analyzes. By recursively calling itself with a decrementing depth, the algorithm delves deeper into potential future game states, thus broadening its evaluation horizon.
- **Terminal State Check:** The minimax function incorporates a mechanism to identify terminal game states, either when a winner is ascertained or when the search depth is exhausted. This halts further recursion and triggers the return of the current board state's evaluation, aligning with the Minimax algorithm's principle of identifying conclusive game outcomes.

In summation, the Minimax algorithm's intricate design and strategic components underscore its significance in game theory and artificial intelligence, providing a robust framework for optimizing decision-making in a competitive gaming scenario.

V. The implementation

Components:

main.py: This is the main program that drives the game and user interface (UI). It allows players to interact with the game, and select and move pieces. The main program gets the user's input and calls the function minimax to evaluate the best possible move for the AI. It also checks who is the winner and if someone has won the game, it will print the winner and it will quit the game.

game.py: Contains the class Game which represents the state of the game. This class includes methods such as:

- **_ _ init _ _:** initializes the game.
- **update:** Updates the game display.
- **_ init:** Initializes game variables.
- **winner:** Determines the winner of the game.
- **reset:** Resets the game.
- **select:** Select a piece on the board.
- **_move:** Moves a selected piece on the board.
- **draw_valid_moves:** Draws valid moves for a selected piece.
- **change_turn:** Changes the turn of players.
- **get_board**: Returns the game board.
- **ai_move**: Performs the AI move.

algorithm.py: Contains the function minimax which implements the minimax algorithm for AI decision-making. This file also includes helper functions such as simulate_move, get_all_moves and draw_moves:

minimax: Implements the minimax algorithm to evaluate the best move for the AI player.

```python
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
```

Simulator_move: Simulates a move on the board.

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

    return board
```

get_all_moves: Returns all possible moves for a given player.

```python
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
```

draw_moves: Draws valid moves for a selected piece on the board.

```
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)
```

Game Loop:

The game loop in main.py continually updates the game state, handles player input and displays the game window. It runs until a winner is determined or the user quits the game by closing the window.

Usage of Game Class:

The Game class in game.py represents the state of the game and provides methods for managing the game flow. It handles player moves, updates the game display, determines the winner, and allows for resetting the game. Additionally, it includes functionality for selecting pieces, moving pieces and drawing valid moves for a selected piece. The ai_move method allows the AI to make a move on the board.

Usage of Minimax Algorithm:

The minimax function in algorithm.py evaluates the optimal board arrangement as we play. It considers the current board configuration, the depth of the search tree and whether we are maximizing or minimizing the score. The function recursively evaluates possible future game states to determine the best move for the current player. Helper functions such as simulate_move and get_all_moves assist in the evaluation process by simulating moves and generating all possible moves for a player. It also handles terminal states and returns the best move found.

Max Evaluation:

In the minimax algorithm, during the evaluation of potential moves for the maximizing player (AI player), we aim to find the move that maximizes the score for that player.

Since we want to find the maximum possible score, we initialize the max evaluation to a very low value that acts as a placeholder for the maximum score found during the evaluation process.

Setting the max evaluation to negative infinity ensures that any score encountered during the evaluation will be greater than this initial value, allowing us to update it accordingly as we find better moves.

Min Evaluation:

Similarly, during the evaluation of potential moves for the minimizing player (opponent), we aim to find the move that minimizes the score for that player.

Since we want to find the minimum possible score, we initialize the minimum evaluation to a very high value that acts as a placeholder for the minimum score found during the evaluation process.

Setting the min evaluation to positive infinity ensures that any score encountered during the evaluation will be less than this initial value, allowing us to update it accordingly as we find better moves.

Depth Selection:

In the line value, new_board = minimax(game. getboard( ), 3, WHITE, game) the parameter 3 represents the depth of the minimax algorithm. The depth determines how many moves ahead the algorithm will consider when evaluating potential game states. The reasons for choosing a depth of 3 for this checkers game are illustrated as follows:

Computational Complexity:

Increasing the depth of the minimax algorithm results in exponentially increasing computational

complexity. Each additional level of depth leads to a branching factor, where the algorithm explores all possible moves for each player at each level.

A higher depth would lead to significantly more computations and would take more time to compute the optimal move. This can make the game less friendly, especially for real-time or interactive game scenarios.

Suitability for Checkers:

Checkers is a complex board game with numerous possible moves and strategies. However, considering too many moves ahead may not necessarily lead to better gameplay.

A depth of 3 was deemed suitable for the checkers game as it strikes a balance between computational complexity and the quality of AI gameplay.

With a depth of 3, the AI can anticipate and plan several moves, making it challenging for human players without causing excessive computational burden.

Effect on AI Performance

Using a depth of less than 3 may result in deficient AI performance. With a shallower search depth, the AI may not be able to explore enough potential game states to make informed decisions.

Conversely, using a depth greater than 3 may not significantly improve AI performance while substantially increasing computational requirements.

A depth of 3 ensures that the AI makes reasonable intelligent moves while maintaining acceptable performance for real-time gameplay. This provides players with a challenging AI opponent that strikes a balance between intelligence and computational efficiency.

VI. Future Improvements

The Future improvement we plan to implement is alpha-beta pruning. Alpha-beta pruning is an optimization technique for the minimax algorithm, often used in two-player games like checkers. The basic idea is to skip evaluating parts of the game tree that won't affect the final decision. When it's clear that one player has a better move available, further exploration of alternatives can be "pruned" away, saving computational time while still finding the optimal move. A quick step-by-step process of how we can implement that is by using our already made framework for the game tree and minimax algorithm that explores the current game and then during the traversal of the tree, two values are maintained: alpha (the best already explored option along the path for the maximizer) and beta (the best already explored option for the minimizer). When the minimizer is evaluating its move, if it finds an option worse than alpha (meaning the maximizer already has a better choice), it can stop looking further down that branch. This pruning avoids evaluating many branches of the game tree that do not need to be explored because they cannot influence the final decision, making it much more efficient than our initial version.

VII. Conclusion

In conclusion, the Checkers AI project represents a significant milestone in the realm of artificial intelligence, melding machine learning, game theory, and strategic analysis to forge a formidable opponent in the game of checkers. Through meticulous design and implementation, including the use of advanced algorithms such as minimax with potential future use of alpha-beta pruning, this AI has not only mastered the intricacies of checkers but has also set a precedent for the development of intelligent systems in other complex domains. The

success of the Checkers AI underscores the remarkable potential of AI to mimic, understand, and even surpass human strategic thinking. As this project illuminates the pathway for future research, it reaffirms the transformative power of artificial intelligence in enhancing decision-making processes and solving intricate challenges across various fields. This advancement not only elevates the strategic gameplay of artificial intelligence but also significantly contributes to our understanding of AI's potential in navigating and excelling within complex environments.

REFERENCES

[1] Techwithtim. (n.d.). Techwithtim/python-checkers-ai: A checkers AI using the minimax algorithm. GitHub. techwithtim/Python-Checkers-AI