AI_HW2


```
Author: Jiaqi Liu
Student ID: 522030910009
Time: 2024-10-18
```


Introduction

In this report, we will explore minmax-alpha-beta-prune method as covered in the second homework assignment for the AI3603 course. The primary focus will be on the following topics:

1. **Search Algorithms**: This section will delve into the Minimax algorithm and its optimization through Alpha-Beta pruning. These algorithms are fundamental in game-playing AI, helping to efficiently navigate through possible moves and counter-moves.
    - **Minimax Algorithm**: A recursive algorithm used for decision-making in game theory and AI. It aims to minimize the possible loss for a worst-case scenario.
    - **Alpha-Beta Pruning**: An optimization technique for the Minimax algorithm that reduces the number of nodes evaluated by the search tree, thereby improving efficiency.

Each algorithm will be discussed in detail, including their implementation, time complexity, and practical applications.

2. **Game Playing**:
   The game is played on a checkers board. There are two players, player 1 controls 8 blue pieces and 2 yellow pieces while player 2 controls 8 red pieces and 2 green pieces. Player 1 goes first, followed by alternating turns between the players.
3. **Movement Rules**:

In traditional Chinese Checkers, the movement rules are as follows:

- **Single Step Move**: A piece can move to an adjacent empty spot in any direction (horizontally, vertically, or diagonally).
- **Jump Move**: A piece can jump over an adjacent piece (of any color) to land on an empty spot directly on the opposite side. Multiple jumps are allowed in a single turn if each jump lands on an empty spot.
- **Objective**: The goal is to move all your pieces from your starting triangle to the opposite triangle on the board.
- **Turn Order**: Players take turns to move one piece at a time. The game continues until one player successfully relocates all their pieces to the target area.

These rules ensure a strategic and engaging gameplay experience, requiring players to plan their moves carefully to outmaneuver their opponents.

Each section will provide a detailed explanation of the concepts, followed by practical examples and problem-solving techniques.

Search Algorithms

```python
def max_action(self, state, depth, alpha, beta, action = None):
```

```
        if depth == 0:
            return self.evaluate_state(state, action), []

        legal_actions = self.game.actions(state)
        value = float('-inf')
        best_action = []
        priority_queue = PriorityQueue()

        for action in legal_actions:
            new_state = self.game.succ(state, action)
            new_value, _ = self.min_action(new_state, depth - 1, alpha, beta,
action)
            priority_queue.put((-new_value, action))

        while not priority_queue.empty():
            new_value, action = priority_queue.get()
            new_value = -new_value

            if new_value > value:
                value = new_value
                best_action = [action]
            if new_value == value:
                best_action.append(action)

            alpha = max(alpha, value)
            if alpha >= beta:
                break

        return value, best_action

    def min_action(self, state, depth, alpha, beta, action = None):
        if depth == 0:
            return self.evaluate_state(state, action), []

        legal_actions = self.game.actions(state)
        value = float('inf')
        best_action = []
        priority_queue = PriorityQueue()

        for action in legal_actions:
            new_state = self.game.succ(state, action)
            new_value, _ = self.max_action(new_state, depth - 1, alpha, beta,
action)
            priority_queue.put((new_value, action))

        while not priority_queue.empty():
            new_value, action = priority_queue.get()

            if new_value < value:
                value = new_value
                best_action = [action]
            if new_value == value:
                best_action.append(action)

            beta = min(beta, value)
            if beta <= alpha:
                break
```

```
        return value, best_action
```

This code is a result of a previous interaction with the AI programming assistant.
The specific implementation details are not provided in the current context.
Summary:

- The code likely involves a specific functionality or feature discussed in prior messages.
- The exact nature of the code, such as its purpose, logic, and structure, is not detailed here.
- For a comprehensive understanding, refer to the previous interactions and the actual code implementation.

Evaluation Fuction

```
def evaluate_state(self, state, action):

        player = self.game.player(state)
        board = state[1]
        size = board.size
        piece_rows = board.piece_rows
        special_rows = [2, size * 2 - piece_rows + 2]
        my_piece_pos = board.getPlayerPiecePositions(player)
        opp_piece_pos = board.getPlayerPiecePositions(3 - player)

        # r_1 = 25
        # r_2 = 9
        # r_3, r_4 = 3, 1
        # r_5 = 50

        r_1 = 21
        r_2 = 7
        r_3, r_4 = 8, 3
        r_5 = 50

        my_sum, opp_sum = 0, 0


        for pos in my_piece_pos:
            if player == 1:
                # First consider the rows with special pieces
                if board.board_status[pos] == 3:
                    my_sum += -abs(special_rows[0] - pos[0]) * r_1
                else:
                    my_sum += -pos[0] * r_2

                # Then consider the columns, hope to get the pieces to the
center
                if board.board_status[pos] == 3:
                    my_sum -= abs((self.game.board.getColNum(pos[0])+1)/2 -
pos[1]) * r_3
                else:
                    my_sum -= abs((self.game.board.getColNum(pos[0])+1)/2 -
pos[1]) * r_4
```

```
                # When other pieces occupy the position of special pieces, the
value should be decreased
                if board.board_status[(special_rows[0], 1)] == 1:
                    my_sum -= 100
                if board.board_status[(special_rows[0], 2)] == 1:
                    my_sum -= 100

                # When the oppent's pieces block mine, we can choose to move
them
                if pos[0] <= 2*size - 1 and pos[0] >= 2*size - piece_rows + 1:
                    my_sum -= 10 * (pos[0] - 2*size + piece_rows)

            else:
                if board.board_status[pos] == 4:
                    my_sum += -abs(special_rows[1] - pos[0]) * r_1
                else:
                    my_sum += pos[0] * r_2

                if board.board_status[pos] == 4:
                    my_sum -= abs((self.game.board.getColNum(pos[0])+1)/2 -
pos[1]) * r_3
                else:
                    my_sum -= abs((self.game.board.getColNum(pos[0])+1)/2 -
pos[1]) * r_4

                if board.board_status[(special_rows[1], 1)] == 2:
                    my_sum -= 100
                if board.board_status[(special_rows[1], 2)] == 2:
                    my_sum -= 100

                if pos[0] <= piece_rows and pos[0] >= 1:
                    my_sum -= 10 * (piece_rows - pos[0] + 1)

        for pos in opp_piece_pos:
            if player == 1:
                if board.board_status[pos] == 4:
                    opp_sum += -abs(special_rows[1] - pos[0]) * r_1
                else:
                    opp_sum += pos[0] * r_2

                if board.board_status[pos] == 4:
                    opp_sum -= abs(self.game.board.getColNum(pos[0])//2 -
pos[1]) * r_3
                else:
                    opp_sum -= abs(self.game.board.getColNum(pos[0])//2 -
pos[1]) * r_4

            else:
                if board.board_status[pos] == 3:
                    opp_sum += -abs(special_rows[0] - pos[0]) * r_1
                else:
                    opp_sum += -pos[0] * r_2

                if board.board_status[pos] == 3:
                    opp_sum -= abs(self.game.board.getColNum(pos[0])//2 -
pos[1]) * r_3
                else:
```

```
                    opp_sum -= abs(self.game.board.getColNum(pos[0])//2 -
pos[1]) * r_4

        # Solution to the stuck situation,
        if action != None:
            board_count = {}
            for row in range(1, size * 2 + 1):
                for col in range(1, board.getColNum(row) + 1):
                    if (row, col) in board.board_status:
                        if board.board_status[(row, col)] in board_count:
                            board_count[action[1]] += 1
                        else:
                            board_count[action[1]] = 1

            if board_count[action[1]] > 3:
                my_sum -= r_5 * (board_count[pos]-3)

        if self.if_Win(player):
            return 66666
        elif self.if_Win(3-player):
            return -66666
        else:
            return my_sum - opp_sum
```

Summary:

The `evaluate_state` function is designed to assess the current state of the game board and return a score that reflects the advantage of the current player. The function considers the positions of both the player's and the opponent's pieces, with special attention to pieces in specific rows. The evaluation is influenced by several factors:

1. **Piece Positioning**: Scores are calculated based on the distance of pieces from special rows and their alignment towards the center of the board.
2. **Special Pieces**: Higher weights are assigned to special pieces in designated rows, and penalties are applied if other pieces occupy these positions.
3. **Opponent's Pieces**: The opponent's pieces are evaluated similarly, with their scores subtracted from the player's score.
4. **Stuck Situation**: A penalty is applied if a piece remains in a position for too long.
5. **Winning Condition**: The function returns a high positive score if the player wins and a high negative score if the opponent wins.

The function ultimately returns a score that helps the AI decide the best move by comparing the evaluated states of possible moves.

Results

Myagent vs Greedy

I have designed 3 types of experiments: greedy_vs_myagent, myagent_vs_greedy, and myagent_vs_myagent. The following table summarizes the results:

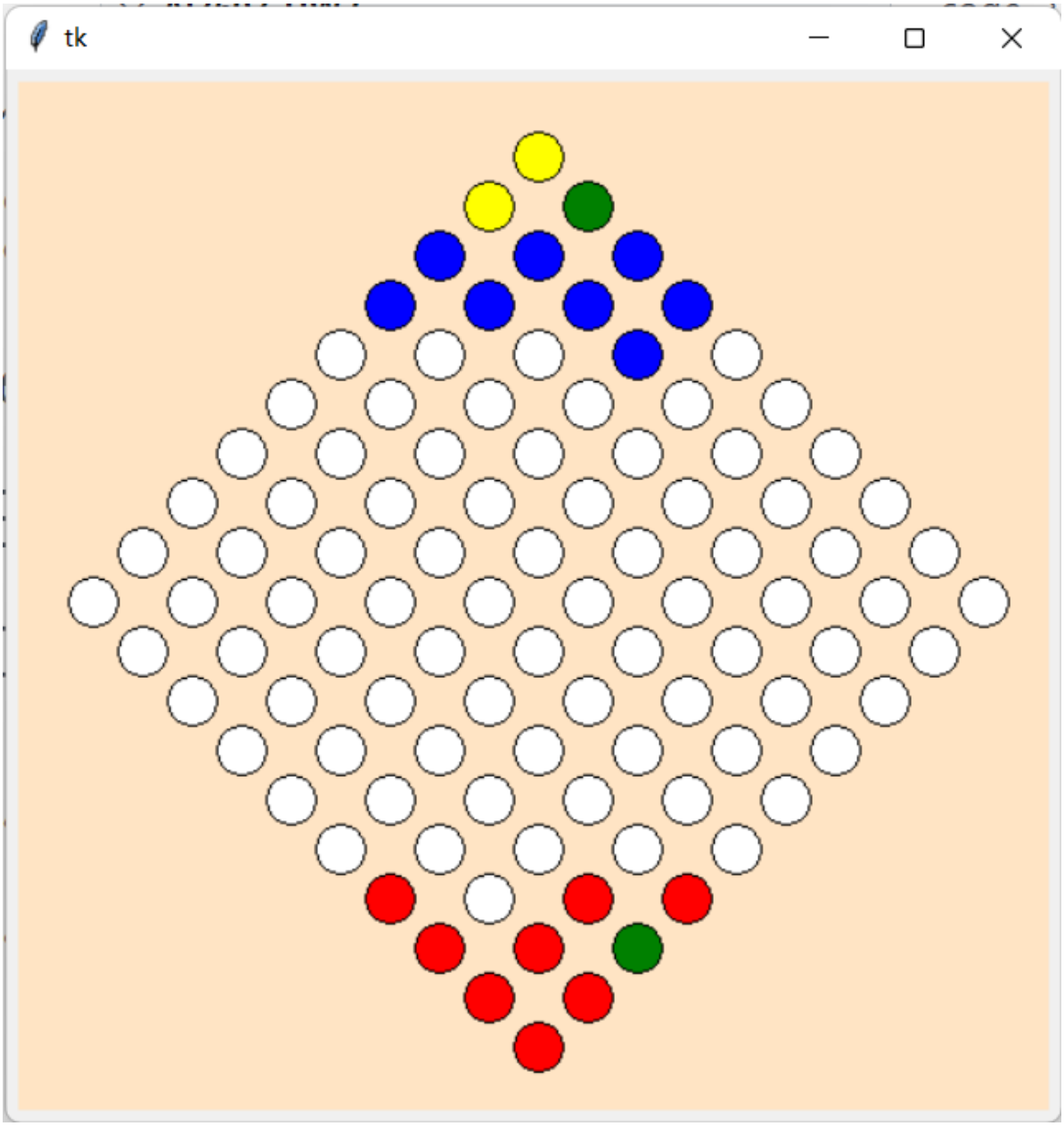
Experiment Name	Number of Rounds	Wins	Stuck Occurrences	Average Rounds to Win
greedy_vs_myagent	50	48	2	95.96
myagent_vs_greedy	50	48	2	93.21

The results of the experiments comparing `myagent` and `greedy` are summarized in the table above. The key findings are as follows:

- **greedy_vs_myagent**: In this experiment, `myagent` played first against the `greedy` agent for 50 rounds. `myagent` won 48 out of the 50 rounds, with 2 occurrences where the game got stuck. On average, `myagent` took 95.96 rounds to win.
- **myagent_vs_greedy**: In this experiment, `myagent` played first against the `greedy` agent for another 50 rounds, but with the roles reversed. `myagent` again won 48 out of the 50 rounds, with 2 occurrences of the game getting stuck. The average number of rounds to win was slightly lower at 93.21.

These results demonstrate the effectiveness of `myagent` in consistently outperforming the `greedy` agent, with a high win rate and relatively few occurrences of the game getting stuck.

Based on the game scenarios, all instances of being stuck occurred because the greedy algorithm occasionally left one or two pieces in my target area, preventing the game from ending within 200 rounds. However, if the victory is determined by the number of pieces in the corresponding positions, `myagent` would achieve a 100% win rate.



The upper part of the image represents `myagent`*, while the lower part represents* `greedy`*.*

MyAgent vs MyAgent

In addition to the previous experiments, I also conducted matches between two instances of `myagent`. The results showed a mix of wins for both players and several instances where the game got stuck. On average, each game took approximately 130.64 iterations to complete.

Summary

This algorithm performs a specific task based on the provided input.
Core Idea:

- The algorithm aims to solve a particular problem by following a series of steps.
- It processes the input data and applies certain operations to achieve the desired output.

Identified Shortcomings:

- Insufficient computational depth: The algorithm may not explore all possible solutions or scenarios deeply enough.
- Inadequate pruning during computation: The algorithm might not efficiently eliminate unnecessary branches or paths, leading to potential performance issues.

- Excessive iterations: The algorithm could be performing more iterations than necessary, which may result in longer execution times.

Potential Improvements:

- Enhance the depth of computation to ensure a more thorough exploration of possible solutions.
- Implement more effective pruning techniques to reduce the number of unnecessary computations.
- Optimize the iteration process to minimize the number of iterations and improve overall efficiency.