-99442326-

July 28, 2024

1 AI Project(Minimax Algorithm)

Implements the Pentago game with an AI opponent using the Minimax algorithm enhanced with Alpha-Beta pruning. Pentago is a two-player strategy game where the objective is to get five of your pieces in a row on a 6x6 board divided into four 3x3 quadrants.

2 Pentago Game with AI using Minimax Algorithm

2.1 Introduction

Pentago is a two-player abstract strategy game where the goal is to get five of your pieces in a row, either horizontally, vertically, or diagonally, on a 6x6 board. The board is divided into four 3x3 quadrants. In this modified version of Pentago, each player can place two pieces on the board and then rotate one of the quadrants 90 degrees to the left or right.

In our project, we will implement the game in Python, allowing a human to play against an AI opponent. The AI will use the Minimax algorithm to predict and make optimal moves.

2.2 Game Board Layout

The board is divided into four quadrants, numbered as follows: - Upper left quadrant: 1 - Upper right quadrant: 2 - Lower left quadrant: 3 - Lower right quadrant: 4

Each quadrant is a 3x3 grid, with cells numbered as follows:

1 2 3

4 5 6

7 8 9

The game screen is displayed as:

+——-+——-+

. . . | . . . |

. . . | . . . |

. . . | . . . |

+——-+——-+

. . . | . . . |

. . . | . . . |

. . . | . . . |

+——-+——-+

Here, "." represents an empty cell.

2.3 Input Format

During each turn, the input is given in the following format: A/B C/D X

- `A/B` and `C/D` indicate the positions to place the two pieces. `A` and `C` are the quadrant numbers (1 to 4), and `B` and `D` are the cell numbers (1 to 9) within the respective quadrants.
- `X` indicates the rotation operation, where the first digit (1 to 4) represents the quadrant number to be rotated, and the second letter (L or R) represents the direction of rotation (Left or Right).

2.3.1 Example

1/1 1/2 1L

This means: - Place the first piece in the 1st cell of the 1st quadrant. - Place the second piece in the 2nd cell of the 1st quadrant. - Rotate the 1st quadrant 90 degrees to the left.

2.4 Minimax Algorithm

The Minimax algorithm is a decision-making algorithm used in two-player games to minimize the possible loss for a worst-case scenario. When the opponent is also playing optimally, it minimizes the maximum loss. This algorithm evaluates the possible moves using a recursive approach to choose the best possible move.

2.4.1 Steps Involved

1. **Generate possible moves**: Create a list of all possible moves.
2. **Evaluate moves**: For each possible move, simulate the move and evaluate the board state.
3. **Minimax recursion**: For each possible move, call the Minimax function recursively to evaluate the resulting board state.
4. **Choose optimal move**: The AI will choose the move that maximizes its minimum gain.

2.5 How to Get Input

The game will prompt the user to enter their move in the specified format. The input will be parsed and validated to ensure it adheres to the rules. The AI will then make its move using the Minimax algorithm and update the board state accordingly.

2.5.1 Example User Interaction

1. **Initial Prompt**: Enter your move (format: A/B C/D X):
2. **User Input**: 1/1 2/5 3R
3. **Board Update and AI Move**: The game will update the board with the user's move, rotate the specified quadrant, and then the AI will calculate and make its move.

By following this structure, players can input their moves in an intuitive manner, and the AI will respond with optimized moves to create an engaging gameplay experience.

2.5.2 Explanation of the Code and the Minimax Algorithm

Class Definitions

1. **Move Class**:
    - Used to store a move in the game.
    - Attributes:
        - `place_board`: The board/quadrant where a piece is placed.
        - `position`: The position within the quadrant where the piece is placed.
        - `twist_board`: The board/quadrant to be twisted.
        - `direction`: The direction of the twist ('L' for left, 'R' for right).
2. **Node Class**:
    - Represents a node in the game tree.
    - Attributes:
        - `children`: List of child nodes.
        - `boards`: The current state of the game board.
        - `depth`: Depth of the node in the game tree.
        - `token`: The current player's token.
        - `move`: The move that led to this node.
        - `is_max`: Boolean indicating if the node is a maximizer or minimizer.
        - `best_node`: The best child node (used for backtracking the best move).
        - `utility_value`: The utility value of the node (used in Minimax evaluation).
        - `alpha`, `beta`: Alpha and Beta values for Alpha-Beta pruning.
3. **simulate_twist Function**:
    - Simulates a twist on a given quadrant of the board.
    - Parameters:
        - `the_current_boards`: The current state of the boards.
        - `the_board`: The quadrant to be twisted.
        - `the_direction`: The direction of the twist ('L' for left, 'R' for right).
4. **AI Class**:
    - Implements the AI logic using the Minimax algorithm with Alpha-Beta pruning.
    - Attributes:
        - `name`: Name of the AI.
        - `token`: The token used by the AI ('b' or 'w').
        - `victory`: Boolean indicating if the AI has won.
        - `is_max`: Boolean indicating if the AI is a maximizer.
        - `root`: The root node of the game tree.
        - `max_depth`: The maximum depth for the game tree.
        - `nodes_total`: The total number of nodes generated.

Methods in the AI Class

1. **generate_tree**:
    - Generates the game tree recursively until the maximum depth is reached.
    - Parameters:
        - `current_node`: The current node in the game tree.

  - The function generates possible moves, simulates them, and creates child nodes.
  - It uses Alpha-Beta pruning to cut off branches that don't need to be explored, optimizing the search.
2. **play**:
  - The main method to generate and return the AI's move.
  - Parameters:
    - `the_current_board`: The current state of the game board.
  - It initializes the root node, generates the game tree, and returns the best move.
3. **utility**:
  - Evaluates the utility value of a board state.
  - Parameters:
    - `the_boards`: The current state of the boards.
  - It calculates the utility value based on the number of AI's and human's pieces in rows, columns, and diagonals.

Minimax Algorithm with Alpha-Beta Pruning

- **Minimax Algorithm**: A recursive algorithm used in decision-making and game theory to find the optimal move for a player, assuming the opponent is also playing optimally.
- **Alpha-Beta Pruning**: An optimization technique for the Minimax algorithm that eliminates branches in the game tree that don't need to be explored, improving efficiency.

Steps in the Minimax Algorithm:

1. **Generate Possible Moves**:
  - Generate all possible moves for the current player by placing pieces on the board and simulating twists.
2. **Evaluate Moves**:
  - For each possible move, simulate the move and create a child node.
  - Recursively call the Minimax function to evaluate the child nodes.
3. **Maximizing Player**:
  - If the current node is a maximizer, choose the move with the highest utility value.
  - Update the alpha value (maximum lower bound).
4. **Minimizing Player**:
  - If the current node is a minimizer, choose the move with the lowest utility value.
  - Update the beta value (minimum upper bound).
5. **Alpha-Beta Pruning**:
  - If alpha is greater than or equal to beta, prune the branch (skip evaluating further nodes in that branch).
6. **Utility Calculation**:
  - At the maximum depth or terminal nodes, calculate the utility value of the board state using the `utility` function.

The AI class uses these steps to build the game tree, evaluate the possible moves, and choose the optimal move for the AI player. The `play` method orchestrates this process, generating the AI's move based on the current board state.

[21]:
```python
import copy
import math
```

```
# This class is used to store a move.
class Move:
    def __init__(self, the_place_board: int, the_position: int, the_twist_board:
 ↪ int, the_direction: str):
        self.place_board = the_place_board
        self.position = the_position
        self.twist_board = the_twist_board
        self.direction = the_direction


# This class is used to build the AI's game tree.
class Node:
    def __init__(self, the_boards: list, depth: int, token, move: Move, is_max,␣
 ↪alpha, beta):
        self.children = []
        self.boards = copy.deepcopy(the_boards)
        self.depth = depth
        self.token = token
        self.move = move
        self.is_max = is_max
        self.best_node = None
        self.utility_value = math.inf
        if self.is_max:
            self.utility_value = -math.inf
        self.alpha = alpha
        self.beta = beta


# This class simulates a twist. It is used to generate the possible moves.
def simulate_twist(the_current_boards, the_board: int, the_direction: str):
    board = the_current_boards[the_board]
    new_board = ['.'] * 9
    new_board[4] = board[4]
    if the_direction.lower() == 'l':
        new_board[0] = board[2]
        new_board[1] = board[5]
        new_board[2] = board[8]
        new_board[3] = board[1]
        new_board[5] = board[7]
        new_board[6] = board[0]
        new_board[7] = board[3]
        new_board[8] = board[6]
    else:
        new_board[0] = board[6]
        new_board[1] = board[3]
        new_board[2] = board[0]
        new_board[3] = board[7]
        new_board[5] = board[1]
        new_board[6] = board[8]
        new_board[7] = board[5]
        new_board[8] = board[2]
    the_current_boards[the_board] = new_board


# This class can generate the game tree and assign utility values to the game␣
 ↪states using the Minimax algorithm.
class AI:
    def __init__(self, the_token: str):
        self.name = 'AI'
        self.token = the_token
        self.victory = False
        self.is_max = False
        self.root = None
        self.max_depth = 3
        self.nodes_total = 0

    # Generates the game tree. It is call recursively until max depth is␣
 ↪reached.
    def generate_tree(self, current_node: Node):
        if current_node.depth < self.max_depth:
            new_token = 'b'
            if current_node.token == new_token:
                new_token = 'w'
            seen_moves = []
            possible_moves = []
            for i in range(4):
                for j in range(9):
                    if current_node.boards[i][j] == '.':
                        possible_moves.append([i, j])
            exit_signal = False
            for move in possible_moves:
                boards = copy.deepcopy(current_node.boards)
                boards[move[0]][move[1]] = current_node.token
                for board in range(4):
                    for direction in ['l', 'r']:
                        simulate_twist(boards, board, direction)
                        if boards not in seen_moves:
                            seen_moves.append(copy.deepcopy(boards))
                            node = Node(boards, current_node.depth + 1,␣
 ↪new_token,
                                        Move(move[0] + 1, move[1] + 1, board +␣
 ↪1, direction),
                                        not current_node.is_max, current_node.
↪alpha, current_node.beta)
                            current_node.children.append(node)
                            self.generate_tree(node)
                            if current_node.is_max:
                                if current_node.alpha < node.utility_value:
                                    current_node.alpha = node.utility_value
                                if current_node.utility_value < node.
↪utility_value:
                                    current_node.utility_value = node.
↪utility_value
                                    current_node.best_node = node

                            else:
                                if current_node.beta > node.utility_value:
                                    current_node.beta = node.utility_value
                                if current_node.utility_value > node.
↪utility_value:
                                    current_node.utility_value = node.
↪utility_value
                                    current_node.best_node = node
                            if current_node.alpha > current_node.beta or␣
↪current_node.alpha == current_node.beta:
                                exit_signal = True
                                break
                    if exit_signal:
                        break

                if exit_signal:
                    break

        else:
            current_node.utility_value = self.utility(current_node.boards)

    def play(self, the_current_board: list):
        self.root = Node(the_current_board, 0, self.token, None, self.is_max,␣
↪-math.inf, math.inf)
        self.generate_tree(self.root)
        best_move_node = self.root.best_node
        move1 = best_move_node.move
        new_boards = copy.deepcopy(the_current_board)
        new_boards[move1.place_board - 1][move1.position - 1] = self.token

        # Generate the second move from the new board state
        self.root = Node(new_boards, 0, self.token, None, self.is_max, -math.
↪inf, math.inf)
```

```
        self.generate_tree(self.root)
        best_move_node = self.root.best_node
        move2 = best_move_node.move

        # The twist move
        twist_move = move2  # Use the second move node's twist info

        return move1, move2, twist_move

    def utility(self, the_boards: list):
        value = 0
        modifier = -1
        if self.is_max:
            modifier = 1
        # Calculate columns
        for i in range(2):
            for j in range(3):
                ai_tally = 0
                human_tally = 0
                if the_boards[i][j] == self.token:
                    ai_tally += 1
                elif the_boards[i][j] != '.':
                    human_tally += 1

                if the_boards[i][j + 3] == self.token:
                    ai_tally += 1
                elif the_boards[i][j + 3] != '.':
                    human_tally += 1

                if the_boards[i][j + 6] == self.token:
                    ai_tally += 1
                elif the_boards[i][j + 6] != '.':
                    human_tally += 1

                if the_boards[i + 2][j] == self.token:
                    ai_tally += 1
                elif the_boards[i + 2][j] != '.':
                    human_tally += 1

                if the_boards[i + 2][j + 3] == self.token:
                    ai_tally += 1
                elif the_boards[i + 2][j + 3] != '.':
                    human_tally += 1

                if the_boards[i + 2][j + 6] == self.token:
                    ai_tally += 1
                elif the_boards[i + 2][j + 6] != '.':
```

```
                human_tally += 1

            if ai_tally > 1:
                value += modifier * ai_tally
            if human_tally > 1:
                value -= modifier * human_tally

    # Calculate rows
    for i in range(0, 3, 2):
        for j in range(0, 9, 3):
            ai_tally = 0
            human_tally = 0
            if the_boards[i][j] == self.token:
                ai_tally += 1
            elif the_boards[i][j] != '.':
                human_tally += 1

            if the_boards[i][j + 1] == self.token:
                ai_tally += 1
            elif the_boards[i][j + 1] != '.':
                human_tally += 1

            if the_boards[i][j + 2] == self.token:
                ai_tally += 1
            elif the_boards[i][j + 2] != '.':
                human_tally += 1

            if the_boards[i + 1][j] == self.token:
                ai_tally += 1
            elif the_boards[i + 1][j] != '.':
                human_tally += 1

            if the_boards[i + 1][j + 1] == self.token:
                ai_tally += 1
            elif the_boards[i + 1][j + 1] != '.':
                human_tally += 1

            if the_boards[i + 1][j + 2] == self.token:
                ai_tally += 1
            elif the_boards[i + 1][j + 2] != '.':
                human_tally += 1

            if ai_tally > 1:
                value += modifier * ai_tally
            if human_tally > 1:
                value -= modifier * human_tally
    return value
```

2.6 Class: Player

2.6.1 Purpose:

The `Player` class is used to represent a player in the game, either human or AI.

2.6.2 Attributes:

- `name`: A string representing the name of the player.
- `token`: A string representing the token of the player ('b' for black or 'w' for white).
- `victory`: A boolean value indicating whether the player has won the game.

2.6.3 Methods:

- `__init__(self, the_name: str, the_token: str)`: The constructor method initializes a new player with a given name and token. The `victory` attribute is set to `False` by default.

2.7 Class: GameBoard

2.7.1 Purpose:

The `GameBoard` class manages the state of the game, including the game board, player moves, and determining the winner.

2.7.2 Attributes:

- `boards`: A list of lists representing the four 3x3 game boards.
- `human`: An instance of the `Player` class representing the human player.
- `ai`: An instance of the `Player` class representing the AI player.

2.7.3 Methods:

- `__init__(self)`: The constructor method initializes the game board, sets up the players, and assigns tokens. It prompts the human player to choose a token ('b' for black or 'w' for white). The AI player is assigned the opposite token. The game board is initialized with empty spaces ('.').

- `start(self)`: Determines which player makes the first move. It randomly decides and calls either `self.human_turn()` or `self.ai_turn()`.

- `human_turn(self)`: Handles the human player's turn. It prompts the player to enter their move in the format "1/1 1/2 1L" (place pieces and twist a board). It validates the input, places the pieces, and twists the board accordingly. After the human's turn, it calls `self.ai_turn()`.

- `ai_turn(self)`: Handles the AI player's turn. It generates a move using the AI logic, places the pieces, and twists the board accordingly. After the AI's turn, it calls `self.human_turn()`.

- `display(self)`: Displays the current state of the game board in a formatted manner.

- `place(self, the_piece: str, the_board: int, the_position: int)`: Places a piece on the specified board and position. It validates the move and updates the board. If the move is invalid, it prompts the human player to enter a new move.

- `twist(self, the_board: int, the_direction: str)`: Twists the specified board in the specified direction ('L' for left, 'R' for right). It validates the twist and updates the board accordingly.
- `victory_check(self)`: Checks if there is a winner. It verifies rows, columns, and diagonals across the combined boards. If a winning condition is met, it sets the `victory` attribute of the respective player to `True`. If both players meet a winning condition, the game is declared a tie. The game ends when a winner is determined.

2.7.4 Example of Usage:

The game begins by creating an instance of the `GameBoard` class and calling the `start()` method.

```
game = GameBoard()
game.start()
```

2.7.5 Notes:

- The `place` and `twist` methods ensure that moves are validated before updating the board state.
- The `victory_check` method comprehensively checks all possible winning conditions, ensuring that a winner is declared as soon as a winning condition is met.
- The `display` method provides a clear visual representation of the game board, helping players to see the current state of the game easily.

[22]:
```
class Player:
    def __init__(self, the_name: str, the_token: str):
        self.name = the_name
        self.token = the_token
        self.victory = False
```

[23]:
```
import sys
import random


# Game board that can place, twist, and check if a winner is found.
class GameBoard:
    def __init__(self):
        self.boards = []
        token = input('Please pick your token (b/w): ').lower()
        while token != 'b' and token != 'w':
            print('Invalid token.')
            token = input('Please pick your token (b/w): ').lower()
        self.human = Player('human', token)
        if self.human.token == 'b':
            self.ai = AI('w')
        else:
            self.ai = AI('b')
        print("AI's token: " + self.ai.token)
        for i in range(4):
            block = []
            for j in range(9):
                block.append('.')
            self.boards.append(block)

    def start(self):
        if random.choice([1, 2]) == 1:
            print('You make the first move.')
            self.human_turn()
        else:
            print('AI makes the first move.')
            self.ai.is_max = True
            self.ai_turn()

    def human_turn(self):
      move = input("Enter your move: ")
      move_parts = move.split()  # Split the input into parts
      if len(move_parts) != 3:
          print("Invalid input format. Enter two placements and one twist (e.g.
↪, '1/3 2/3 1L').")
          self.human_turn()
          return

      try:
          place1_board, place1_pos = map(int, move_parts[0].split('/'))
          place2_board, place2_pos = map(int, move_parts[1].split('/'))
          twist_board, twist_dir = int(move_parts[2][0]), move_parts[2][1].
↪lower()
      except (ValueError, IndexError):
          print("Invalid input format. Enter two placements and one twist (e.g.
↪, '1/3 2/3 1L').")
          self.human_turn()
          return

      self.place(self.human.token, place1_board, place1_pos)
      self.place(self.human.token, place2_board, place2_pos)
      self.twist(twist_board, twist_dir)
      self.ai_turn()

    def ai_turn(self):
      print("AI's turn...")
      move = self.ai.play(self.boards)
      print("AI's move: {}/{} {}/{} {}{}".format(move[0].place_board, move[0].
↪position, move[1].place_board, move[1].position, move[2].twist_board,␣
↪move[2].direction.upper()))
```

```
      self.place(self.ai.token, move[0].place_board, move[0].position)
      self.place(self.ai.token, move[1].place_board, move[1].position)
      self.twist(move[2].twist_board, move[2].direction)
      self.human_turn()

    def display(self):
        print("+-------+-------+")
        for i in range(3):
            print('| {} {} {} | {} {} {} |'.format(self.boards[0][0 + 3 * i],
                                                    self.boards[0][1 + 3 *␣
↪i],
                                                    self.boards[0][2 + 3 *␣
↪i],
                                                    self.boards[1][0 + 3 *␣
↪i],
                                                    self.boards[1][1 + 3 *␣
↪i],
                                                    self.boards[1][2 + 3 *␣
↪i]))
        print("+-------+-------+")
        for i in range(3):
            print('| {} {} {} | {} {} {} |'.format(self.boards[2][0 + 3 * i],
                                                    self.boards[2][1 + 3 *␣
↪i],
                                                    self.boards[2][2 + 3 *␣
↪i],
                                                    self.boards[3][0 + 3 *␣
↪i],
                                                    self.boards[3][1 + 3 *␣
↪i],
                                                    self.boards[3][2 + 3 *␣
↪i]))
        print("+-------+-------+")

    # Board: 1-4
    # Position: 1-9
    def place(self, the_piece: str, the_board: int, the_position: int):
      if the_board < 1 or the_board > 4 or the_position < 1 or the_position > 9␣
↪or \
              self.boards[the_board - 1][the_position - 1] != '.':
          print('Invalid move')
          self.human_turn()
      else:
          self.boards[the_board - 1][the_position - 1] = the_piece
          print('Placing: ' + str(the_board) + '/' + str(the_position))
          self.display()
          self.victory_check()

    def twist(self, the_board: int, the_direction: str):
      if the_board < 1 or the_board > 4 or the_direction.lower() != 'l' and␣
  ↪the_direction.lower() != 'r':
          print('Invalid twist')
          self.human_turn()
      else:
          board = self.boards[the_board - 1]
          new_board = ['.'] * 9
          new_board[4] = board[4]
          if the_direction.lower() == 'l':
              new_board[0] = board[2]
              new_board[1] = board[5]
              new_board[2] = board[8]
              new_board[3] = board[1]
              new_board[5] = board[7]
              new_board[6] = board[0]
              new_board[7] = board[3]
              new_board[8] = board[6]
          else:
              new_board[0] = board[6]
              new_board[1] = board[3]
              new_board[2] = board[0]
              new_board[3] = board[7]
              new_board[5] = board[1]
              new_board[6] = board[8]
              new_board[7] = board[5]
              new_board[8] = board[2]
          self.boards[the_board - 1] = new_board
          print('Twisting: ' + str(the_board) + the_direction.upper())
          self.display()
          self.victory_check()

    def victory_check(self):
        # Check rows
        for i in range(0, 3, 2):
            for j in range(3):
                if self.boards[i][1 + 3 * j] != '.':
                    if self.boards[i][0 + 3 * j] == self.boards[i][1 + 3 * j]␣
  ↪== self.boards[i][2 + 3 * j] == \
                            self.boards[i + 1][0 + 3 * j] == self.boards[i +␣
  ↪1][1 + 3 * j] \
                            or self.boards[i][1 + 3 * j] == self.boards[i][2 +␣
  ↪3 * j] == self.boards[i + 1][
                            0 + 3 * j] == self.boards[i + 1][1 + 3 * j] == self.
  ↪boards[i + 1][2 + 3 * j]:
```

```
                        if self.boards[i][1 + 3 * j] == self.human.token:
                            self.human.victory = True
                        else:
                            self.ai.victory = True

        # Check columns
        for i in range(2):
            for j in range(3):
                if self.boards[i][j + 3] != '.':
                    if self.boards[i][j] == self.boards[i][j + 3] == self.
↪boards[i][j + 6] == \
                            self.boards[i + 2][j] == self.boards[i + 2][j + 3] \
                            or self.boards[i][j + 3] == self.boards[i][j + 6]␣
↪== self.boards[i + 2][j] == \
                            self.boards[i + 2][j + 3] == self.boards[i + 2][j +␣
↪6]:
                        if self.boards[i][j + 3] == self.human.token:
                            self.human.victory = True
                        else:
                            self.ai.victory = True

        # Check diagonals
        if self.boards[0][8] != '.':
            if self.boards[0][0] == self.boards[0][4] == self.boards[0][8] ==␣
↪self.boards[3][0] == self.boards[3][4] or \
                    self.boards[0][4] == self.boards[0][8] == self.boards[3][0]␣
↪== self.boards[3][4] == self.boards[3][
                8] or self.boards[1][1] == self.boards[1][3] == self.
↪boards[0][8] == self.boards[2][1] == \
                    self.boards[2][3]:
                if self.boards[0][8] == self.human.token:
                    self.human.victory = True
                else:
                    self.ai.victory = True

        if self.boards[1][6] != '.':
            if self.boards[1][2] == self.boards[1][4] == self.boards[1][6] ==␣
↪self.boards[2][2] == self.boards[2][4] or \
                    self.boards[1][4] == self.boards[1][6] == self.boards[2][2]␣
↪== self.boards[2][4] == self.boards[2][
                6] or self.boards[0][1] == self.boards[0][5] == self.
↪boards[1][6] == self.boards[3][1] == \
                    self.boards[3][5]:
                if self.boards[1][6] == self.human.token:
                    self.human.victory = True
                else:
                    self.ai.victory = True

        if self.boards[3][0] != '.':
            if self.boards[1][5] == self.boards[1][7] == self.boards[3][0] ==␣
↪self.boards[2][5] == self.boards[2][7]:
                if self.boards[3][0] == self.human.token:
                    self.human.victory = True
                else:
                    self.ai.victory = True

        if self.boards[2][2] != '.':
            if self.boards[0][3] == self.boards[0][7] == self.boards[2][2] ==␣
↪self.boards[3][4] == self.boards[3][7]:
                if self.boards[2][2] == self.human.token:
                    self.human.victory = True
                else:
                    self.ai.victory = True

        if self.human.victory and self.ai.victory:
            print('We have a tie!')
            sys.exit()
        elif self.human.victory:
            print('You won!')
            sys.exit()
        elif self.ai.victory:
            print('You lost.')
            sys.exit()
```

[24]:
```
game = GameBoard()
game.start()
```

```
Please pick your token (b/w): w
AI's token: b
You make the first move.
Enter your move: 1/1 1/2 1L
Placing: 1/1
+-------+-------+
| w . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
Placing: 1/2
+-------+-------+
| w w . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
Twisting: 1L
+-------+-------+
| . . . | . . . |
| w . . | . . . |
| w . . | . . . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
AI's turn…
AI's move: 1/1 1/2 1L
Placing: 1/1
+-------+-------+
| b . . | . . . |
| w . . | . . . |
| w . . | . . . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
Placing: 1/2
+-------+-------+
| b b . | . . . |
| w . . | . . . |
| w . . | . . . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
Twisting: 1L
+-------+-------+
| . . . | . . . |
| b . . | . . . |
| b w w | . . . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
Enter your move: 2/7 2/8 3L
Placing: 2/7
+-------+-------+
| . . . | . . . |
| b . . | . . . |
| b w w | w . . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
Placing: 2/8
+-------+-------+
| . . . | . . . |
| b . . | . . . |
| b w w | w w . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
Twisting: 3L
+-------+-------+
| . . . | . . . |
| b . . | . . . |
| b w w | w w . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
AI's turn…
AI's move: 2/6 1/1 1R
Placing: 2/6
+-------+-------+
| . . . | . . . |
| b . . | . . b |
| b w w | w w . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
Placing: 1/1
+-------+-------+
| b . . | . . . |
| b . . | . . b |
| b w w | w w . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
Twisting: 1R
+-------+-------+
| b b b | . . . |
| w . . | . . b |
| w . . | w w . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . . | . . . |
+-------+-------+
Enter your move: 3/9 4/1 1L
Placing: 3/9
+-------+-------+
| b b b | . . . |
| w . . | . . b |
| w . . | w w . |
+-------+-------+
| . . . | . . . |
| . . . | . . . |
| . . w | . . . |
+-------+-------+
Placing: 4/1
+-------+-------+
| b b b | . . . |
| w . . | . . b |
| w . . | w w . |
+-------+-------+
| . . . | w . . |
| . . . | . . . |
| . . w | . . . |
+-------+-------+
Twisting: 1L
+-------+-------+
| b . . | . . . |
| b . . | . . b |
| b w w | w w . |
+-------+-------+
| . . . | w . . |
| . . . | . . . |
| . . w | . . . |
+-------+-------+
AI's turn…
AI's move: 2/2 2/1 2L
Placing: 2/2
+-------+-------+
| b . . | . b . |
| b . . | . . b |
| b w w | w w . |
+-------+-------+
| . . . | w . . |
| . . . | . . . |
| . . w | . . . |
+-------+-------+
Placing: 2/1
+-------+-------+
| b . . | b b . |
| b . . | . . b |
| b w w | w w . |
+-------+-------+
| . . . | w . . |
| . . . | . . . |
| . . w | . . . |
+-------+-------+
Twisting: 2L
+-------+-------+
| b . . | . b . |
| b . . | b . w |
| b w w | b . w |
+-------+-------+
| . . . | w . . |
| . . . | . . . |
| . . w | . . . |
+-------+-------+
Enter your move: 2/3 3/1 2R
Placing: 2/3
+-------+-------+
| b . . | . b w |
| b . . | b . w |
| b w w | b . w |
+-------+-------+
| . . . | w . . |
| . . . | . . . |
| . . w | . . . |
+-------+-------+
Placing: 3/1
+-------+-------+
| b . . | . b w |
| b . . | b . w |
| b w w | b . w |
+-------+-------+
| w . . | w . . |
| . . . | . . . |
| . . w | . . . |
+-------+-------+
Twisting: 2R
+-------+-------+
| b . . | b b . |
| b . . | . . b |
| b w w | w w w |
+-------+-------+
| w . . | w . . |
| . . . | . . . |
| . . w | . . . |
+-------+-------+
You won!
```

```
An exception has occurred, use %tb to see the full traceback.

SystemExit
```

```
/usr/local/lib/python3.10/dist-packages/IPython/core/interactiveshell.py:3561:
UserWarning: To exit: use 'exit', 'quit', or Ctrl-D.
  warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)
```