Morabaraba : the MAIC sixth edition game.

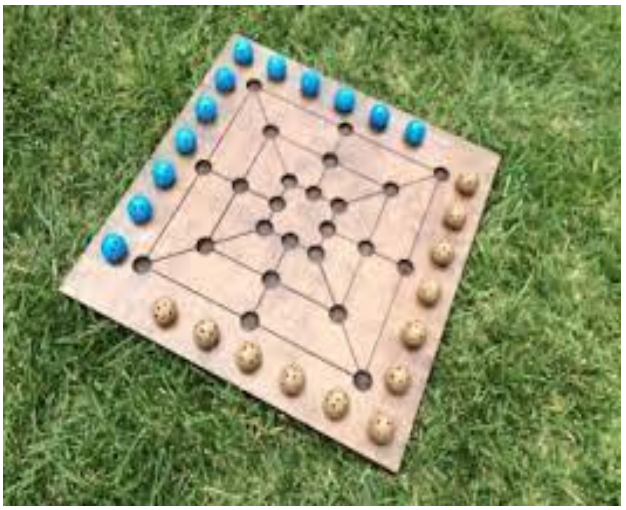

Figure: A Morabaraba board

Morabaraba is a traditional two-player strategy board game mostly played in South Africa, Botswana and Lesotho. It is an accessible and easy to learn game with deep strategic and tactical aspects whose objective is to create a "mill" (a line of three pieces) to knock down the opposing pieces.

Morabaraba Rules

- **The Board**

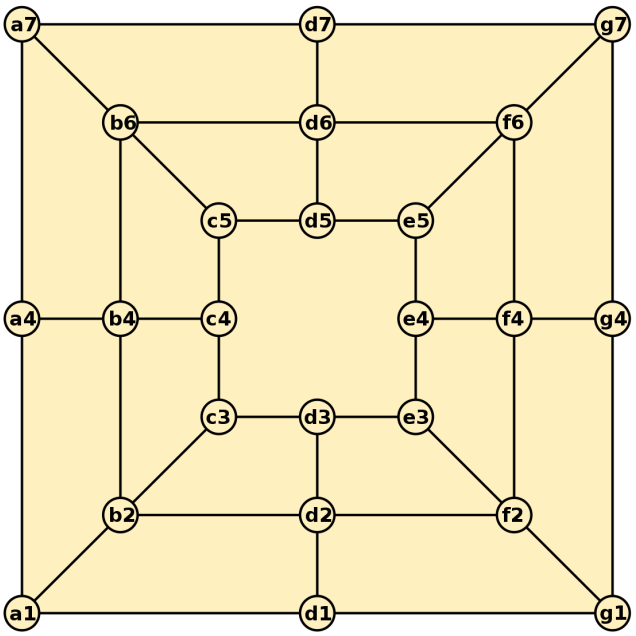

Figure: Morabaraba board

Morabaraba is played on a board of 3 nested squares with cells at the corners and in the middle of each side of the squares. Lines indicate normal moves.

- **Gameplay**

There are three main phases to the game:

1. Placing the cows
2. Moving the cows
3. Flying the cows

Placing the cows

- The board is empty when the game begins. Each player has 12 pieces, known as "cows".
-
- The player with the dark cows moves first
- Each turn consists of placing a cow on an empty intersection on the board.

- The aim is to create a "mill": a row of three cows on any line drawn on the board.
- If a player forms a mill, he or she may remove or "shoot" one of the opponent's cows. The shot cow is removed from the board and not placed again. A cow in a mill may not be shot unless all of the opponent's cows are in mills, in which case any cow may be shot.
- Even if a move creates more than one mill, only one cow can be shot in a single move.

Moving the cows

- After all the cows have been placed, each turn consists of moving a cow to an empty adjacent intersection.
- As before, completing a mill allows a player to shoot one of the opponent's cows. Again, this must be a cow which is not in a mill, unless all of the opponent's cows are in mills.
- Players are allowed to "break" their own mills.
- A mill may be broken and remade repeatedly by shuffling cows back and forth. Each time the mill is remade, one of the opponent's cows is shot.
- A mill which is broken to form a new mill can not be formed again on the next move.

Flying the cows

- When a player has only three cows remaining, desperate measures are called for. This player's cows are allowed to "fly" to any empty intersection, not just adjacent ones.
- If one player has three cows and the other player has more than three cows, only the player with three cows is allowed to fly.

End of the game

- A win occurs if one opponent has just two cows or if there are no moves.
- If either player has only three cows and neither player shoots a cow within ten moves, the game is drawn.

- After 50 moves without a capture, the player with the highest score wins, otherwise it's a draw.

The code structure

The game is defined by five main classes that combine all the necessary elements for its progress. These are MorabarabaBoard, MorabarabaPlayer, MorabarabaAction, MorabarabaRules and MorabarabaState classes.

The ***MorabarabaBoard*** class represents the game board and gathers all the functions that act on the board and the pieces. It keeps all the information relating to the empty intersections, to the locations of the opponent's pawns and own pawns.

The ***MorabarabaPlayer*** class defines a player with his features and especially its 'play' method which allows him to perform actions in the game.

The ***MorabarabaAction*** class defines the structure of all types of actions that the player can perform during the game. Here, four types of actions are possible:

- ‘ADD’ action
- ‘MOVE’ action
- ‘STEAL’ action and
- ‘FLY’ action

The ***MorabarabaRules*** class represents through its methods the rules of the game.

The ***MorabarabaState*** class keeps the situation of the players and the board at each move.

What you will be required to do

You have to provide your player as a AI. Your AI will be an extension of the MorabarabaPlayer class.

```python
class MorabarabaPlayer(object):

    def __init__(self, color):
        self.color = color
        self._reset_player_info()

    def _reset_player_info(self):
        self.pieces_in_hand = 12

    def play(self, state):
        raise NotImplementedError

    def set_score(self, new_score):
        self.score = new_score

    def update_player_infos(self, infos):
        self.in_hand = infos['in_hand']
        self.score = infos['score']

    def reset_player_informations(self):
        self.in_hand = 12
        self.score = 0
```

Figure: MorabarabaPlayer class

Your main work is to **write a "play" function for your AI**. This method returns an action among the four possible types of action during the game for a given state (players and board).

The functions you will need

- ***get_effective_cell_moves*** (from MorabarabaRules class)
  - ➢ gives the only possibles movements of a piece on the board
  - ➢ takes: the game state and the coordinates of a piece as a tuple
  - ➢ returns: a list of all the coordinates where the piece can move.
  - ➢ example
    - ```
      print(rules.get_effective_cell_moves(state, (0,0)))
      ```
    - ```
      [(3, 0), (1, 1), (0, 3)]
      ```

- ***get_player_actions*** (from MorabarabaRules class)
  - ➢ determines the possibles actions of a player in a state of the game
  - ➢ takes: the game state and the player characteristic number
  - ➢ returns: a list of all the possible actions of the player at the given.
  - ➢ example
    - ```
      print(rules.get_player_actions(state, -1))
      ```
      ```

```
[{'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (0, 0)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (0, 3)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (0, 6)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (1, 1)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (1, 3)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (1, 5)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (2, 2)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (2, 3)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (2, 4)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (3, 0)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (3, 1)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (3, 2)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (3, 4)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (3, 5)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (3, 6)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (4, 2)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (4, 3)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (4, 4)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (5, 1)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (5, 3)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (5, 5)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (6, 0)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (6, 3)}},
 {'action_type': <MorabarabaActionType.ADD: 1>, 'action': {'to': (6, 6)}}]
```

❖ ***is_making_mill*** (from MorabarabaRules class)

- ➢ determines if a move/addiction creates a mill
- ➢ takes: the board, the player characteristic number and the coordinates of the moved/added piece as a tuple
- ➢ returns: a list whose first element is a boolean (true if there is a mill and false otherwise) and the second a list of the mill(s) formed
- ➢ example

```
print(rules.is_making_mill(board, 1, (0,0)))
```

```
[True, [[(0, 0), (3, 0), (6, 0)]]]
```

❖ ***stealables*** (from MorabarabaRules class)

- ➢ gives the opponent pieces that can be steal
- ➢ takes: the player characteristic number and the board
- ➢ returns: a list of all the coordinates of stealables pieces.
- ➢ example

```
print(rules.stealables(-1, board))
```

```
[(0, 0), (1, 1), (6, 6)]
```

OBJ

❖ ***get_board*** (from MorabarabaState class)

- ➢ returns: the board object
- ➢ example
  - ```
    print(state.get_board())
    ```
  - ```
    <morabaraba.morabaraba_board.MorabarabaBoard object at
    0x7f0fc0a1bd30>
    ```

❖ ***get_latest_move*** (from MorabarabaState class)

- ➢ returns: the last action in the game
- ➢ example
  - ```
    print(state.get_latest_move())
    ```
  - ```
    {'action_type': 'ADD', 'action': {'to': (6, 3)}}
    ```

OBJ OBJ

❖ ***get_all_empty_cells*** (from MorabarabaBoard class)

- ➢ gives back the coordinates of all empties intersections on the board
- ➢ takes: no argument
- ➢ returns: a list of the coordinates of all empties intersections.
- ➢ example
  - ```
    print(board.get_all_empty_cells())
    ```
  - ```
    [(0, 3), (0, 6), (1, 3), (1, 5), (2, 2), (2, 3), (3, 0),
     (3, 1), (3, 2), (3, 4), (3, 5), (3, 6), (4, 3), (4, 4),
     (5, 1), (5, 3), (5, 5), (6, 0), (6, 6)]
    ```

❖ ***get_player_pieces_on_board*** (from MorabarabaBoard class)

- ➢ gives back the coordinates of all pieces of a player
- ➢ takes: the color of the player's pieces
- ➢ returns: a list of the coordinates of all player's pieces.
- ➢ example
  - ```
    print(board.get_player_pieces_on_board(Color(1)))
    ```
  - ```
    [(0, 0), (1, 1), (4, 2), (6, 3)]
    ```

❖ ***mills*** (from MorabarabaBoard class)

- ➢ gives all mill formation possibilities
- ➢ takes: no argument

- ➢ returns: a list of all mill combinations on the board.
- ➢ example
  - ```
    print(board.mills())
    ```
  - ```
    [[(0, 0), (3, 0), (6, 0)], [(1, 1), (3, 1), (5, 1)],
     [(2, 2), (3, 2), (4, 2)], [(0, 3), (1, 3), (2, 3)],
     [(4, 3), (5, 3), (6, 3)], [(2, 4), (3, 4), (4, 4)],
     [(1, 5), (3, 5), (5, 5)], [(0, 6), (3, 6), (6, 6)],
     [(0, 0), (0, 3), (0, 6)], [(1, 1), (1, 3), (1, 5)],
     [(2, 2), (2, 3), (2, 4)], [(3, 0), (3, 1), (3, 2)],
     [(3, 4), (3, 5), (3, 6)], [(4, 2), (4, 3), (4, 4)],
     [(5, 1), (5, 3), (5, 5)], [(6, 0), (6, 3), (6, 6)],
     [(0, 0), (1, 1), (2, 2)], [(4, 2), (5, 1), (6, 0)],
     [(0, 6), (1, 5), (2, 4)], [(4, 4), (5, 5), (6, 6)]]
    ```

❖ ***player_mills*** (from MorabarabaBoard class)

- ➢ determines a player's mills on the board
- ➢ takes: the player characteristic number
- ➢ returns: a list of all player's mills.
- ➢ example
  - ```
    print(board.player_mills(1))
    ```
  - ```
    [[(0, 0), (3, 0), (6, 0)], [(2, 2), (3, 2), (4, 2)]]
    ```

❖ ***names*** (from MorabarabaBoard class)

- ➢ gives the cell's alias from its coordinates
- ➢ takes: the cell's coordinates
- ➢ returns: the cell's alias
- ➢ example
  - ```
    print(board.names((1,1)))
    ```
  - ```
    b2
    ```

❖ ***coordinates*** (from MorabarabaBoard class)

- ➢ gives the cell's coordinates from its alias
- ➢ takes: the cell's alias
- ➢ returns: the cell's coordinates
- ➢ example
  - ```
    print(board.coordinates("a1"))
    ```
  - ```
    (0, 0)
    ```

Write an AI class

```python
from morabaraba.morabaraba_player import MorabarabaPlayer
from morabaraba.morabaraba_action import MorabarabaAction
from morabaraba.morabaraba_action import MorabarabaActionType
from morabaraba.morabaraba_rules import MorabarabaRules


class AI(MorabarabaPlayer):
    name = "Lord of War"

    def __init__(self, color):
        super(AI, self).__init__(color)
        self.position = color.value

    def play(self, state, remain_time):
        """ Return Morabara action.
            Example : an add to (0, 1) is equivalent to
                MorabarabaAction(action_type=MorabarabaActionType.ADD, to=(0, 1))
            Example : a move from (0, 1) to (0, 2) is equivalent to
                MorabarabaAction(action_type=MorabarabaActionType.MOVE, at=(0, 1), to=(0, 2))
            Example : a fly from (0, 1) to (0, 2) is equivalent to
                MorabarabaAction(action_type=MorabarabaActionType.FLY, at=(0, 1), to=(0, 2))
            Example : a steal at (0, 1) is equivalent to
                MorabarabaAction(action_type=MorabarabaActionType.STEAL, at=(0, 1))

        Utils:
            you can retrieve the coordinates of a cell through the alias or vice versa
            Example : Retrieve cell coordinates of b2 is equivalent to :
                state.get_board().coordinates('b2'), which return (1, 1)
            Example : Retrieve alias of cell with coordinates (1, 1) is equivalent to :
                state.get_board().names((1,1)), which return 'b2'

        Args:
            action_type (MorabarabaActionType): The type of the performed action.
        """
        board = state.get_board()
        return MorabarabaRules.random_play(state, self.position)
```

Figure: An example of a AI

To write the AI class,

- You need to import the MorabarabaPlayer class.
- You will import also the MorabarabaRules class to access all it's methods.
- You will need MorabarabaActionType and MorabarabaAction if you want to write the actions yourself.
- You will define a name for your AI.
- Finally, you will write the function play : all the functions of the MorabarabaState class are accessible from the ***'state'*** argument of the 'play' function. You will also need to use the MorabarabaBoard class methods. You can get the board with the method

‘***get_board()***’ of the state. The ***‘remain_time’*** argument is time your AI have to return an action.