Minmax algorithm

from libs.games import TicTacToe

from IPython.display import HTML, display

import random

# Define the AdversarialAgents class with rational and random agents

class AdversarialAgents:

    def \_\_init\_\_(*self*):

        self.ttt = TicTacToe()

    def rational\_agent(*self*, *state*):

        # Rational agent always maximizes its utility

        return max(self.ttt.actions(state), *key*=lambda *move*: self.utility(self.ttt.result(state, move)))

    def random\_agent(*self*, *state*):

        # Random agent selects a move randomly among the two best moves (lowest utility)

        actions = self.ttt.actions(state)

        best\_moves = [move for move in actions if self.utility(self.ttt.result(state, move)) == min(self.utility(self.ttt.result(state, a)) for a in actions)]

        return random.choice(best\_moves) if best\_moves else random.choice(actions)

    def utility(*self*, *state*):

        if self.ttt.utility(state, self.ttt.to\_move(self.ttt.initial)) == 0:

            return 0  # Draw

        elif self.ttt.utility(state, self.ttt.to\_move(self.ttt.initial)) > 0:

            return 1  # Rational agent wins

        else:

            return -1  # Random agent wins

# Define the Canvas class (required for the TicTacToe game visualization)

\_canvas = """

<script type="text/javascript" src="./js/canvas.js"></script>

<div>

<canvas id="{0}" width="{1}" height="{2}" style="background:rgba(158, 167, 184, 0.2);" onclick='click\_callback(this, event, "{3}")'></canvas>

</div>

<script> var {0}\_canvas\_object = new Canvas("{0}");</script>

"""  # noqa

class Canvas:

    """Inherit from this class to manage the HTML canvas element in Jupyter notebooks.

    To create an object of this class, use any\_name\_xyz = Canvas("any\_name\_xyz")

    The first argument given must be the name of the object being created.

    IPython must be able to reference the variable name that is being passed."""

    def \_\_init\_\_(*self*, *varname*, *width*=800, *height*=600, *cid*=None):

        self.name = varname

        self.cid = cid or varname

        self.width = width

        self.height = height

        self.html = \_canvas.format(self.cid, self.width, self.height, self.name)

        self.exec\_list = []

        display\_html(self.html)

    def mouse\_click(*self*, *x*, *y*):

        """Override this method to handle mouse click at position (x, y)"""

        raise NotImplementedError

    def mouse\_move(*self*, *x*, *y*):

        raise NotImplementedError

    def execute(*self*, *exec\_str*):

        """Stores the command to be executed to a list which is used later during update()"""

        if not isinstance(exec\_str, str):

            print("Invalid execution argument:", exec\_str)

            self.alert("Received invalid execution command format")

        prefix = "{0}\_canvas\_object.".format(self.cid)

        self.exec\_list.append(prefix + exec\_str + ';')

    def fill(*self*, *r*, *g*, *b*):

        """Changes the fill color to a color in rgb format"""

        self.execute("fill({0}, {1}, {2})".format(r, g, b))

    def stroke(*self*, *r*, *g*, *b*):

        """Changes the colors of line/strokes to rgb"""

        self.execute("stroke({0}, {1}, {2})".format(r, g, b))

    def strokeWidth(*self*, *w*):

        """Changes the width of lines/strokes to 'w' pixels"""

        self.execute("strokeWidth({0})".format(w))

    def rect(*self*, *x*, *y*, *w*, *h*):

        """Draw a rectangle with 'w' width, 'h' height and (x, y) as the top-left corner"""

        self.execute("rect({0}, {1}, {2}, {3})".format(x, y, w, h))

    def rect\_n(*self*, *xn*, *yn*, *wn*, *hn*):

        """Similar to rect(), but the dimensions are normalized to fall between 0 and 1"""

        x = round(xn \* self.width)

        y = round(yn \* self.height)

        w = round(wn \* self.width)

        h = round(hn \* self.height)

        self.rect(x, y, w, h)

    def line(*self*, *x1*, *y1*, *x2*, *y2*):

        """Draw a line from (x1, y1) to (x2, y2)"""

        self.execute("line({0}, {1}, {2}, {3})".format(x1, y1, x2, y2))

    def line\_n(*self*, *x1n*, *y1n*, *x2n*, *y2n*):

        """Similar to line(), but the dimensions are normalized to fall between 0 and 1"""

        x1 = round(x1n \* self.width)

        y1 = round(y1n \* self.height)

        x2 = round(x2n \* self.width)

        y2 = round(y2n \* self.height)

        self.line(x1, y1, x2, y2)

    def arc(*self*, *x*, *y*, *r*, *start*, *stop*):

        """Draw an arc with (x, y) as centre, 'r' as radius from angles 'start' to 'stop'"""

        self.execute("arc({0}, {1}, {2}, {3}, {4})".format(x, y, r, start, stop))

    def arc\_n(*self*, *xn*, *yn*, *rn*, *start*, *stop*):

        """Similar to arc(), but the dimensions are normalized to fall between 0 and 1

        The normalizing factor for radius is selected between width and height by

        seeing which is smaller."""

        x = round(xn \* self.width)

        y = round(yn \* self.height)

        r = round(rn \* min(self.width, self.height))

        self.arc(x, y, r, start, stop)

    def clear(*self*):

        """Clear the HTML canvas"""

        self.execute("clear()")

    def font(*self*, *font*):

        """Changes the font of text"""

        self.execute('font("{0}")'.format(font))

    def text(*self*, *txt*, *x*, *y*, *fill*=True):

        """Display a text at (x, y)"""

        if fill:

            self.execute('fill\_text("{0}", {1}, {2})'.format(txt, x, y))

        else:

            self.execute('stroke\_text("{0}", {1}, {2})'.format(txt, x, y))

    def text\_n(*self*, *txt*, *xn*, *yn*, *fill*=True):

        """Similar to text(), but with normalized coordinates"""

        x = round(xn \* self.width)

        y = round(yn \* self.height)

        self.text(txt, x, y, fill)

    def alert(*self*, *message*):

        """Immediately display an alert"""

        display\_html('<script>alert("{0}")</script>'.format(message))

    def update(*self*):

        """Execute the JS code to execute commands queued by execute()"""

        exec\_code = "<script>\n" + '\n'.join(self.exec\_list) + "\n</script>"

        self.exec\_list = []

        display\_html(exec\_code)

# Define the Canvas\_TicTacToe class with integrated adversarial agents

class Canvas\_TicTacToe(Canvas):

    def \_\_init\_\_(*self*, *varname*, *player\_1*='Player1', *player\_2*='Player2',

*width*=300, *height*=350, *cid*=None):

        super().\_\_init\_\_(varname, width, height, cid)

        self.ttt = TicTacToe()

        self.adversarial\_agents = AdversarialAgents()

        self.state = self.ttt.initial

        self.turn = 0

        self.strokeWidth(5)

        self.players = (player\_1, player\_2)

        self.font("20px Arial")

        self.draw\_board()

    def take\_turn(*self*, *x*, *y*):

        player = self.players[self.turn]

        if self.ttt.terminal\_test(self.state):

            # Game over, show final result

            utility = self.ttt.utility(self.state, self.ttt.to\_move(self.ttt.initial))

            if utility == 0:

                self.text\_n('Game Draw!', offset, 6 / 7 + offset)

            else:

                self.text\_n('Player {} wins!'.format("XO"[utility < 0]), offset, 6 / 7 + offset)

            return

        # Make move

        x, y = x, y

        if (x, y) not in self.ttt.actions(self.state):

            # Invalid move

            return

        move = (x, y)

        self.state = self.ttt.result(self.state, move)

        self.turn ^= 1  # Switch turns

        self.draw\_board()  # Update board after each move

    def computer\_move(*self*):

        if self.turn == 1:  # Player 2 (computer) turn

            move = self.adversarial\_agents.random\_agent(self.state)

            self.state = self.ttt.result(self.state, move)

            self.turn ^= 1

            self.draw\_board()

    def draw\_board(*self*):

        self.clear()

        self.stroke(0, 0, 0)

        offset = 1 / 20

        self.line\_n(0 + offset, (1 / 3) \* 6 / 7, 1 - offset, (1 / 3) \* 6 / 7)

        self.line\_n(0 + offset, (2 / 3) \* 6 / 7, 1 - offset, (2 / 3) \* 6 / 7)

        self.line\_n(1 / 3, (0 + offset) \* 6 / 7, 1 / 3, (1 - offset) \* 6 / 7)

        self.line\_n(2 / 3, (0 + offset) \* 6 / 7, 2 / 3, (1 - offset) \* 6 / 7)

        board = self.state.board

        for mark in board:

            if board[mark] == 'X':

                self.draw\_x(mark)

            elif board[mark] == 'O':

                self.draw\_o(mark)

        if self.ttt.terminal\_test(self.state):

            # End game message

            utility = self.ttt.utility(self.state, self.ttt.to\_move(self.ttt.initial))

            if utility == 0:

                self.text\_n('Game Draw!', offset, 6 / 7 + offset)

            else:

                self.text\_n('Player {} wins!'.format("XO"[utility < 0]), offset, 6 / 7 + offset)

                # Find the 3 and draw a line

                self.stroke([255, 0][self.turn], [0, 255][self.turn], 0)

                for i in range(3):

                    if all([(i + 1, j + 1) in self.state.board for j in range(3)]) and \

                            len({self.state.board[(i + 1, j + 1)] for j in range(3)}) == 1:

                        self.line\_n(i / 3 + 1 / 6, offset \* 6 / 7, i / 3 + 1 / 6, (1 - offset) \* 6 / 7)

                    if all([(j + 1, i + 1) in self.state.board for j in range(3)]) and \

                            len({self.state.board[(j + 1, i + 1)] for j in range(3)}) == 1:

                        self.line\_n(offset, (i / 3 + 1 / 6) \* 6 / 7, 1 - offset, (i / 3 + 1 / 6) \* 6 / 7)

                if all([(i + 1, i + 1) in self.state.board for i in range(3)]) and \

                        len({self.state.board[(i + 1, i + 1)] for i in range(3)}) == 1:

                    self.line\_n(offset, offset \* 6 / 7, 1 - offset, (1 - offset) \* 6 / 7)

                if all([(i + 1, 3 - i) in self.state.board for i in range(3)]) and \

                        len({self.state.board[(i + 1, 3 - i)] for i in range(3)}) == 1:

                    self.line\_n(offset, (1 - offset) \* 6 / 7, 1 - offset, offset \* 6 / 7)

            # restart button

            self.fill(0, 0, 255)

            self.rect\_n(0.5 + offset, 6 / 7, 0.4, 1 / 8)

            self.fill(0, 0, 0)

            self.text\_n('Restart', 0.5 + 2 \* offset, 13 / 14)

        else:  # Print which player's turn it is

            self.text\_n("Player {}'s move({})".format("XO"[self.turn], self.players[self.turn]),

                        offset, 6 / 7 + offset)

        self.update()

    def draw\_x(*self*, *position*):

        self.stroke(0, 255, 0)

        x, y = [i - 1 for i in position]

        offset = 1 / 15

        self.line\_n(x / 3 + offset, (y / 3 + offset) \* 6 / 7, x / 3 + 1 / 3 - offset, (y / 3 + 1 / 3 - offset) \* 6 / 7)

        self.line\_n(x / 3 + offset, (y / 3 + offset) \* 6 / 7, x / 3 + 1 / 3 - offset, (y / 3 + 1 / 3 - offset) \* 6 / 7)

    def draw\_o(*self*, *position*):

        self.stroke(255, 0, 0)

        x, y = [i - 1 for i in position]

        self.arc\_n(x / 3 + 1 / 6, (y / 3 + 1 / 6) \* 6 / 7, 1 / 9, 0, 360)

# Initialize the TicTacToe environment and adversarial agents

adversarial\_agents = AdversarialAgents()

ttt = adversarial\_agents.ttt

# Define a function to play the game between the agents

def play\_game():

    state = ttt.initial

    turn = 0  # Rational agent takes the first turn

    print("Initial State:")

    ttt.display(state)  # Display the initial state of the game

    while not ttt.terminal\_test(state):

        if turn == 0:  # Rational agent's turn

            move = adversarial\_agents.rational\_agent(state)

            player = "Rational Agent"

        else:  # Random agent's turn

            move = adversarial\_agents.random\_agent(state)

            player = "Random Agent"

        print(f"\nPlayer: {player}. Making move {move}...")

        state = ttt.result(state, move)  # Update the state after the move

        ttt.display(state)  # Display the updated state of the game

        turn ^= 1  # Switch turns

    # Determine the winner or if it's a draw

    utility = ttt.utility(state, ttt.to\_move(ttt.initial))

    if utility == 0:

        print("\nGame Draw!")

    elif utility > 0:

        print(f"\nRational Agent ({ttt.to\_move(ttt.initial)}) wins!")

    else:

        print(f"\nRandom Agent ({ttt.to\_move(ttt.initial)}) wins!")

# Play the game

play\_game()

task 2

from libs.games import TicTacToe

from IPython.display import HTML, display

import random

# Define the AdversarialAgents class with rational and random agents

class AdversarialAgents:

    def \_\_init\_\_(*self*):

        self.ttt = TicTacToe()

    def rational\_agent(*self*, *state*):

        # Alpha-beta pruning algorithm for rational agent

        return self.alpha\_beta\_search(state)

    def random\_agent(*self*, *state*):

        # Random agent selects a move randomly among available actions

        actions = self.ttt.actions(state)

        return random.choice(actions) if actions else None

    def alpha\_beta\_search(*self*, *state*):

        infinity = float('inf')

        best\_score = -infinity

        beta = infinity

        best\_action = None

        for action in self.ttt.actions(state):

            result\_state = self.ttt.result(state, action)

            score = self.min\_value(result\_state, -infinity, beta)

            if score > best\_score:

                best\_score = score

                best\_action = action

            beta = min(beta, best\_score)

        return best\_action

    def max\_value(*self*, *state*, *alpha*, *beta*):

        if self.ttt.terminal\_test(state):

            return self.utility(state)

        value = -float('inf')

        for action in self.ttt.actions(state):

            value = max(value, self.min\_value(self.ttt.result(state, action), alpha, beta))

            if value >= beta:

                return value

            alpha = max(alpha, value)

        return value

    def min\_value(*self*, *state*, *alpha*, *beta*):

        if self.ttt.terminal\_test(state):

            return self.utility(state)

        value = float('inf')

        for action in self.ttt.actions(state):

            value = min(value, self.max\_value(self.ttt.result(state, action), alpha, beta))

            if value <= alpha:

                return value

            beta = min(beta, value)

        return value

    def utility(*self*, *state*):

        # Utility function to evaluate the state

        if self.ttt.utility(state, self.ttt.to\_move(self.ttt.initial)) == 0:

            return 0  # Draw

        elif self.ttt.utility(state, self.ttt.to\_move(self.ttt.initial)) > 0:

            return 1  # Rational agent wins

        else:

            return -1  # Random agent wins

# Define the Canvas class (required for the TicTacToe game visualization)

\_canvas = """

<script type="text/javascript" src="./js/canvas.js"></script>

<div>

<canvas id="{0}" width="{1}" height="{2}" style="background:rgba(158, 167, 184, 0.2);" onclick='click\_callback(this, event, "{3}")'></canvas>

</div>

<script> var {0}\_canvas\_object = new Canvas("{0}");</script>

"""  # noqa

class Canvas:

    """Inherit from this class to manage the HTML canvas element in Jupyter notebooks.

    To create an object of this class, use any\_name\_xyz = Canvas("any\_name\_xyz")

    The first argument given must be the name of the object being created.

    IPython must be able to reference the variable name that is being passed."""

    def \_\_init\_\_(*self*, *varname*, *width*=800, *height*=600, *cid*=None):

        self.name = varname

        self.cid = cid or varname

        self.width = width

        self.height = height

        self.html = \_canvas.format(self.cid, self.width, self.height, self.name)

        self.exec\_list = []

        display\_html(self.html)

    def mouse\_click(*self*, *x*, *y*):

        """Override this method to handle mouse click at position (x, y)"""

        raise NotImplementedError

    def mouse\_move(*self*, *x*, *y*):

        raise NotImplementedError

    def execute(*self*, *exec\_str*):

        """Stores the command to be executed to a list which is used later during update()"""

        if not isinstance(exec\_str, str):

            print("Invalid execution argument:", exec\_str)

            self.alert("Received invalid execution command format")

        prefix = "{0}\_canvas\_object.".format(self.cid)

        self.exec\_list.append(prefix + exec\_str + ';')

    def fill(*self*, *r*, *g*, *b*):

        """Changes the fill color to a color in rgb format"""

        self.execute("fill({0}, {1}, {2})".format(r, g, b))

    def stroke(*self*, *r*, *g*, *b*):

        """Changes the colors of line/strokes to rgb"""

        self.execute("stroke({0}, {1}, {2})".format(r, g, b))

    def strokeWidth(*self*, *w*):

        """Changes the width of lines/strokes to 'w' pixels"""

        self.execute("strokeWidth({0})".format(w))

    def rect(*self*, *x*, *y*, *w*, *h*):

        """Draw a rectangle with 'w' width, 'h' height and (x, y) as the top-left corner"""

        self.execute("rect({0}, {1}, {2}, {3})".format(x, y, w, h))

    def rect\_n(*self*, *xn*, *yn*, *wn*, *hn*):

        """Similar to rect(), but the dimensions are normalized to fall between 0 and 1"""

        x = round(xn \* self.width)

        y = round(yn \* self.height)

        w = round(wn \* self.width)

        h = round(hn \* self.height)

        self.rect(x, y, w, h)

    def line(*self*, *x1*, *y1*, *x2*, *y2*):

        """Draw a line from (x1, y1) to (x2, y2)"""

        self.execute("line({0}, {1}, {2}, {3})".format(x1, y1, x2, y2))

    def line\_n(*self*, *x1n*, *y1n*, *x2n*, *y2n*):

        """Similar to line(), but the dimensions are normalized to fall between 0 and 1"""

        x1 = round(x1n \* self.width)

        y1 = round(y1n \* self.height)

        x2 = round(x2n \* self.width)

        y2 = round(y2n \* self.height)

        self.line(x1, y1, x2, y2)

    def arc(*self*, *x*, *y*, *r*, *start*, *stop*):

        """Draw an arc with (x, y) as centre, 'r' as radius from angles 'start' to 'stop'"""

        self.execute("arc({0}, {1}, {2}, {3}, {4})".format(x, y, r, start, stop))

    def arc\_n(*self*, *xn*, *yn*, *rn*, *start*, *stop*):

        """Similar to arc(), but the dimensions are normalized to fall between 0 and 1

        The normalizing factor for radius is selected between width and height by

        seeing which is smaller."""

        x = round(xn \* self.width)

        y = round(yn \* self.height)

        r = round(rn \* min(self.width, self.height))

        self.arc(x, y, r, start, stop)

    def clear(*self*):

        """Clear the HTML canvas"""

        self.execute("clear()")

    def font(*self*, *font*):

        """Changes the font of text"""

        self.execute('font("{0}")'.format(font))

    def text(*self*, *txt*, *x*, *y*, *fill*=True):

        """Display a text at (x, y)"""

        if fill:

            self.execute('fill\_text("{0}", {1}, {2})'.format(txt, x, y))

        else:

            self.execute('stroke\_text("{0}", {1}, {2})'.format(txt, x, y))

    def text\_n(*self*, *txt*, *xn*, *yn*, *fill*=True):

        """Similar to text(), but with normalized coordinates"""

        x = round(xn \* self.width)

        y = round(yn \* self.height)

        self.text(txt, x, y, fill)

    def alert(*self*, *message*):

        """Immediately display an alert"""

        display\_html('<script>alert("{0}")</script>'.format(message))

    def update(*self*):

        """Execute the JS code to execute commands queued by execute()"""

        exec\_code = "<script>\n" + '\n'.join(self.exec\_list) + "\n</script>"

        self.exec\_list = []

        display\_html(exec\_code)

# Define the Canvas\_TicTacToe class with integrated adversarial agents

class Canvas\_TicTacToe(Canvas):

    def \_\_init\_\_(*self*, *varname*, *player\_1*='Player1', *player\_2*='Player2',

*width*=300, *height*=350, *cid*=None):

        super().\_\_init\_\_(varname, width, height, cid)

        self.ttt = TicTacToe()

        self.adversarial\_agents = AdversarialAgents()

        self.state = self.ttt.initial

        self.turn = 0

        self.strokeWidth(5)

        self.players = (player\_1, player\_2)

        self.font("20px Arial")

        self.draw\_board()

    def take\_turn(*self*, *x*, *y*):

        player = self.players[self.turn]

        if self.ttt.terminal\_test(self.state):

            # Game over, show final result

            utility = self.ttt.utility(self.state, self.ttt.to\_move(self.ttt.initial))

            if utility == 0:

                self.text\_n('Game Draw!', offset, 6 / 7 + offset)

            else:

                self.text\_n('Player {} wins!'.format("XO"[utility < 0]), offset, 6 / 7 + offset)

            return

        # Make move

        x, y = x, y

        if (x, y) not in self.ttt.actions(self.state):

            # Invalid move

            return

        move = (x, y)

        self.state = self.ttt.result(self.state, move)

        self.turn ^= 1  # Switch turns

        self.draw\_board()  # Update board after each move

    def computer\_move(*self*):

        if self.turn == 1:  # Player 2 (computer) turn

            move = self.adversarial\_agents.random\_agent(self.state)

            self.state = self.ttt.result(self.state, move)

            self.turn ^= 1

            self.draw\_board()

    def draw\_board(*self*):

        self.clear()

        self.stroke(0, 0, 0)

        offset = 1 / 20

        self.line\_n(0 + offset, (1 / 3) \* 6 / 7, 1 - offset, (1 / 3) \* 6 / 7)

        self.line\_n(0 + offset, (2 / 3) \* 6 / 7, 1 - offset, (2 / 3) \* 6 / 7)

        self.line\_n(1 / 3, (0 + offset) \* 6 / 7, 1 / 3, (1 - offset) \* 6 / 7)

        self.line\_n(2 / 3, (0 + offset) \* 6 / 7, 2 / 3, (1 - offset) \* 6 / 7)

        board = self.state.board

        for mark in board:

            if board[mark] == 'X':

                self.draw\_x(mark)

            elif board[mark] == 'O':

                self.draw\_o(mark)

        if self.ttt.terminal\_test(self.state):

            # End game message

            utility = self.ttt.utility(self.state, self.ttt.to\_move(self.ttt.initial))

            if utility == 0:

                self.text\_n('Game Draw!', offset, 6 / 7 + offset)

            else:

                self.text\_n('Player {} wins!'.format("XO"[utility < 0]), offset, 6 / 7 + offset)

                # Find the 3 and draw a line

                self.stroke([255, 0][self.turn], [0, 255][self.turn], 0)

                for i in range(3):

                    if all([(i + 1, j + 1) in self.state.board for j in range(3)]) and \

                            len({self.state.board[(i + 1, j + 1)] for j in range(3)}) == 1:

                        self.line\_n(i / 3 + 1 / 6, offset \* 6 / 7, i / 3 + 1 / 6, (1 - offset) \* 6 / 7)

                    if all([(j + 1, i + 1) in self.state.board for j in range(3)]) and \

                            len({self.state.board[(j + 1, i + 1)] for j in range(3)}) == 1:

                        self.line\_n(offset, (i / 3 + 1 / 6) \* 6 / 7, 1 - offset, (i / 3 + 1 / 6) \* 6 / 7)

                if all([(i + 1, i + 1) in self.state.board for i in range(3)]) and \

                        len({self.state.board[(i + 1, i + 1)] for i in range(3)}) == 1:

                    self.line\_n(offset, offset \* 6 / 7, 1 - offset, (1 - offset) \* 6 / 7)

                if all([(i + 1, 3 - i) in self.state.board for i in range(3)]) and \

                        len({self.state.board[(i + 1, 3 - i)] for i in range(3)}) == 1:

                    self.line\_n(offset, (1 - offset) \* 6 / 7, 1 - offset, offset \* 6 / 7)

            # restart button

            self.fill(0, 0, 255)

            self.rect\_n(0.5 + offset, 6 / 7, 0.4, 1 / 8)

            self.fill(0, 0, 0)

            self.text\_n('Restart', 0.5 + 2 \* offset, 13 / 14)

        else:  # Print which player's turn it is

            self.text\_n("Player {}'s move({})".format("XO"[self.turn], self.players[self.turn]),

                        offset, 6 / 7 + offset)

        self.update()

    def draw\_x(*self*, *position*):

        self.stroke(0, 255, 0)

        x, y = [i - 1 for i in position]

        offset = 1 / 15

        self.line\_n(x / 3 + offset, (y / 3 + offset) \* 6 / 7, x / 3 + 1 / 3 - offset, (y / 3 + 1 / 3 - offset) \* 6 / 7)

        self.line\_n(x / 3 + offset, (y / 3 + offset) \* 6 / 7, x / 3 + 1 / 3 - offset, (y / 3 + 1 / 3 - offset) \* 6 / 7)

    def draw\_o(*self*, *position*):

        self.stroke(255, 0, 0)

        x, y = [i - 1 for i in position]

        self.arc\_n(x / 3 + 1 / 6, (y / 3 + 1 / 6) \* 6 / 7, 1 / 9, 0, 360)

# Initialize the TicTacToe environment and adversarial agents

adversarial\_agents = AdversarialAgents()

ttt = adversarial\_agents.ttt

# Define a function to play the game between the agents

def play\_game():

    state = ttt.initial

    turn = 0  # Rational agent takes the first turn

    print("Initial State:")

    ttt.display(state)  # Display the initial state of the game

    while not ttt.terminal\_test(state):

        if turn == 0:  # Rational agent's turn

            move = adversarial\_agents.rational\_agent(state)

            player = "Rational Agent"

        else:  # Random agent's turn

            move = adversarial\_agents.random\_agent(state)

            player = "Random Agent"

        print(f"\nPlayer: {player}. Making move {move}...")

        state = ttt.result(state, move)  # Update the state after the move

        ttt.display(state)  # Display the updated state of the game

        turn ^= 1  # Switch turns

    # Determine the winner or if it's a draw

    utility = ttt.utility(state, ttt.to\_move(ttt.initial))

    if utility == 0:

        print("\nGame Draw!")

    elif utility > 0:

        print(f"\nRational Agent ({ttt.to\_move(ttt.initial)}) wins!")

    else:

        print(f"\nRandom Agent ({ttt.to\_move(ttt.initial)}) wins!")

# Play the game

play\_game()

|  |  |  |  |
| --- | --- | --- | --- |
| Minmax algo | B^d=O(4^9) | O(d)=O(9) | Agent X |
| Alpha beta pruning | O((b\*)d\*)=O(32) | O(1) | Agent X |