M.SRIRAM

2303A52165

B-34

Part-01: List of Tasks to Perform

1. Install the Python Libraries for Game Strategy
2. Implement a Game Class Constructor with - actions, is_terminal, result, utility
3. Implement a Player Game using game class function

```
from collections import namedtuple, Counter, defaultdict
import random
import math
import functools
cache = functools.lru_cache(10**6)


class Game:


    def actions(self, state):

        raise NotImplementedError

    def result(self, state, move):

        raise NotImplementedError

    def is_terminal(self, state):

        return not self.actions(state)

    def utility(self, state, player):

        raise NotImplementedError

def play_game(game, strategies: dict, verbose=False):

    state = game.initial
    while not game.is_terminal(state):
        player = state.to_move
        move = strategies[player](game, state)
        state = game.result(state, move)
        if verbose:
            print('Player', player, 'move:', move)
            print(state)
    return state
```

Part-02: Implementation of Game Strategy Algorithm

1. MiniMax Tree
2. Alpha-Beta Search Algorithm

```
def minimax_search(game, state):


    player = state.to_move

    def max_value(state):
        if game.is_terminal(state):
            return game.utility(state, player), None
        v, move = -infinity, None
        for a in game.actions(state):
            v2, _ = min_value(game.result(state, a))
            if v2 > v:
                v, move = v2, a
        return v, move
```

```
    def min_value(state):
        if game.is_terminal(state):
            return game.utility(state, player), None
        v, move = +infinity, None
        for a in game.actions(state):
            v2, _ = max_value(game.result(state, a))
            if v2 < v:
                v, move = v2, a
        return v, move

    return max_value(state)

infinity = math.inf


def alphabeta_search(game, state):


    player = state.to_move

    def max_value(state, alpha, beta):
        if game.is_terminal(state):
            return game.utility(state, player), None
        v, move = -infinity, None
        for a in game.actions(state):
            v2, _ = min_value(game.result(state, a), alpha, beta)
            if v2 > v:
                v, move = v2, a
                alpha = max(alpha, v)
            if v >= beta:
                return v, move
        return v, move

    def min_value(state, alpha, beta):
        if game.is_terminal(state):
            return game.utility(state, player), None
        v, move = +infinity, None
        for a in game.actions(state):
            v2, _ = max_value(game.result(state, a), alpha, beta)
            if v2 < v:
                v, move = v2, a
                beta = min(beta, v)
            if v <= alpha:
                return v, move
        return v, move

    return max_value(state, -infinity, +infinity)
```

Part-03: Implement the Game Strategy using TicTacToe

```
class TicTacToe(Game):

    def __init__(self, height=3, width=3, k=3):
        self.k = k # k in a row
        self.squares = {(x, y) for x in range(width) for y in range(height)}
        self.initial = Board(height=height, width=width, to_move='X', utility=0)

    def actions(self, board):

        return self.squares - set(board)

    def result(self, board, square):

        player = board.to_move
        board = board.new({square: player}, to_move=('O' if player == 'X' else 'X'))
        win = k_in_row(board, player, square, self.k)
        board.utility = (0 if not win else +1 if player == 'X' else -1)
        return board

    def utility(self, board, player):

        return board.utility if player == 'X' else -board.utility

    def is_terminal(self, board):
```

```
        return board.utility != 0 or len(self.squares) == len(board)

    def display(self, board): print(board)


def k_in_row(board, player, square, k):

    def in_row(x, y, dx, dy): return 0 if board[x, y] != player else 1 + in_row(x + dx, y + dy, dx, dy)
    return any(in_row(*square, dx, dy) + in_row(*square, -dx, -dy)-1>=k
               for (dx, dy) in ((0, 1), (1, 0), (1, 1), (1, -1)))


class Board(defaultdict):

    empty = '.'
    off = '#'

    def __init__(self, width=8, height=8, to_move=None, **kwds):
        self.__dict__.update(width=width, height=height, to_move=to_move, **kwds)

    def new(self, changes: dict, **kwds) -> 'Board':

        board = Board(width=self.width, height=self.height, **kwds)
        board.update(self)
        board.update(changes)
        return board

    def __missing__(self, loc):
        x, y = loc
        if 0 <= x < self.width and 0 <= y < self.height:
            return self.empty
        else:
            return self.off

    def __hash__(self):
        return hash(tuple(sorted(self.items()))) + hash(self.to_move)

    def __repr__(self):
        def row(y): return ' '.join(self[x, y] for x in range(self.width))
        return '\n'.join(map(row, range(self.height))) +  '\n'


def random_player(game, state): return random.choice(list(game.actions(state)))

def player(search_algorithm):

    return lambda game, state: search_algorithm(game, state)[1]
```

Evaluate the Game Strategy using TicTokToe

```
play_game(TicTacToe(), dict(X=random_player, O=player(alphabeta_search)), verbose=True).utility
```

```
Player X move: (0, 1)
. . .
X . .
. . .

Player O move: (2, 1)
. . .
X . O
. . .

Player X move: (1, 2)
. . .
X . O
. X .

Player O move: (0, 0)
O . .
X . O
. X .

Player X move: (1, 0)
O X .
X . O
. X .

Player O move: (1, 1)
```

```
O X .
X O O
. X .

Player X move: (0, 2)
O X .
X O O
X X .

Player O move: (2, 2)
O X .
X O O
X X O

-1
```

play_game(TicTacToe(), dict(X=player(alphabeta_search), O=player(minimax_search)), verbose=True).utility

Start coding or generate with AI.