

School of Computer Science Engineering and Technology
Assignment-03

Course- B.Tech	Type- Core
Code- 23CS106	Course Name- Artificial Intelligence & Machine Learning
Year- 2024-2025	Semester- Even, Instructor: Prof. E.L.N. Kiran
Date- 23-01-2024	Batch- AIML-A,B

1 Implement the AI Game Strategy

Part 1 –(a). Install the Python Libraries required for Game Strategy

1. Install the python libraries - collections, random, math, functools,
`cache = functools.lru_cache(10**6)`
2. Implement a Game Class Constructor using action, is_terminal, result, utility functions
3. A game is similar to a problem, but it has a terminal test instead of a goal test, and a utility for each terminal state.
4. Create a game subclass and implement *actions*, *result*, *is_terminal*, and *utility*.
5. You will also need to set the initial attribute to the initial state; this can be done in the constructor.

```
class Game:
    def actions(self, state):
        """Return a collection of the allowable moves from this state."""
        raise NotImplementedError

    def result(self, state, move):
        """Return the state that results from making a move from a state."""
        raise NotImplementedError

    def is_terminal(self, state):
        """Return True if this is a final state for the game."""
        return not self.actions(state)

    def utility(self, state, player):
        """Return the value of this final state to player."""
        raise NotImplementedError
```

6. Implement a Player Game using the Game Class Constructor.

```

def play_game(game, strategies: dict, verbose=False):
    """Play a turn-taking game. 'strategies' is a {player_name: function} dict,
    where function(state, game) is used to get the player's move."""
    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 2 – Implement the Game Strategy Algorithms

1. Implement the MiniMax Search Algorithm

```

def minimax_search(game, state):
    """Search game tree to determine best move; return (value, move) pair."""
    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

```

2. Implement the Alpha-Beta Search Algorithm

```
def alphabeta_search(game, state):
    """Search game to determine best action; use alpha-beta pruning.
    """Search all the way to the leaves."""
    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 3 – Implement the Game Strategy using TicTacToe

1. Implement TicToCToe game using *init*_, *actions*, *result*, *is_terminal*, *utility*, *display* constructors

```
class TicTacToe(Game):

    """Play TicTacToe on an 'height' by 'width' board, needing 'k' in a row to win.
    'X' plays first against 'O'."""
```

```

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):
    """Legal moves are any square not yet taken."""
    return self.squares - set(board)

def result(self, board, square):
    """Place a marker for current player on 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 the value to player; 1 for win, -1 for loss, 0 otherwise."""
    return board.utility if player == 'X' else -board.utility

def is_terminal(self, board):
    """A board is a terminal state if it is won or there are no empty squares."""
    return board.utility != 0 or len(self.squares) == len(board)

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

def k_in_row(board, player, square, k):
    """True if player has k pieces in a line through square."""
    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)))

```

2. Implement a Game Board using *defaultdict* using *_init_*, *new*, *_missing_*, *_hash_*, *_repr_*

```

class Board(defaultdict):
    """A board has the player to move, a cached utility value,
    and a dict of {(x, y): player} entries, where player is 'X' or 'O'."""
    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':
    "Given a dict of {(x, y): contents} changes, return a new Board with the changes."
    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'

```

3. Implement *random_player(game, state)* and *player(search_algorithm)*

```

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

def player(search_algorithm):
    """A game player who uses the specified search algorithm"""
    return lambda game, state: search_algorithm(game, state)[1]

```

Part 4 – Evaluate the AI Game Strategy using TicTacToe

1. Implement Game Strategy using *play_game(TicTacToe(), dict(X=random_player, O=player(alphabeta_search)), verbose=True).utility*
2. Implement Game strategy using *play_game(TicTacToe(), dict(X=player(alphabeta_search), O=player(minimax_search)), verbose=True).utility*