4 IMPLEMENTATION OF DIFFERENT GAME PLAYING TECHNIQUES.

Contents

Introduction:

Implementing game-playing techniques involves coding algorithms for decision-making in games. Here, I'll provide examples for two classic game-playing techniques: Minimax algorithm for turn-based games and Q-learning for reinforcement learning in games.

1. Minimax Algorithm (Tic-Tac-Toe Example):

```python
import math


def evaluate(board):
    # Evaluate the current state of the board
    # Returns 1 if the player wins, -1 if the opponent wins, 0 if it's a tie


def minimax(board, depth, maximizing_player):
    if depth == 0 or evaluate(board) != 0:
        return evaluate(board)

    if maximizing_player:
        max_eval = -math.inf
        for move in get_possible_moves(board):
            eval = minimax(make_move(board, move, 'X'), depth - 1, False)
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = math.inf
        for move in get_possible_moves(board):
            eval = minimax(make_move(board, move, 'O'), depth - 1, True)
            min_eval = min(min_eval, eval)
        return min_eval


def get_best_move(board):
```

```
    best_move = None
    best_eval = -math.inf
    for move in get_possible_moves(board):
        eval = minimax(make_move(board, move, 'X'), 2, False)
        if eval > best_eval:
            best_eval = eval
            best_move = move
    return best_move

# Example functions (to be implemented):
def make_move(board, move, player):
    # Make a move on the board

def get_possible_moves(board):
    # Get a list of possible moves

# Example usage:
tic_tac_toe_board = [' ', ' ', ' ',
                     ' ', ' ', ' ',
                     ' ', ' ', ' ']

best_move = get_best_move(tic_tac_toe_board)
print("Best Move:", best_move)
```

This example demonstrates a simplified version of the Minimax algorithm for Tic-Tac-Toe. You need to implement the `evaluate`, `make_move`, and `get_possible_moves` functions based on the rules of the game.

2. Q-learning (Q-Learning in a Grid World):

```python
import numpy as np

# Define the environment (a 3x3 grid)
env = np.zeros((3, 3))

# Define Q-table
Q = np.zeros_like(env)

# Set parameters
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
epsilon = 0.1  # Exploration-exploitation trade-off

# Training Q-learning
for episode in range(1000):
    state = (0, 0)  # Starting state
    while state != (2, 2):  # Goal state
        if np.random.rand() < epsilon:
            action = np.random.choice(['up', 'down', 'left', 'right'])
        else:
            action = np.argmax(Q[state])

        next_state = take_action(state, action)
        reward = get_reward(next_state)
        Q[state][action] = Q[state][action] + alpha * (reward + gamma * np.max(Q[next_state]) - Q[state][action])

        state = next_state
```

```
# Example functions (to be implemented):
def take_action(state, action):
    # Perform the given action from the current state and return the next state

def get_reward(state):
    # Return the reward for the current state

# Example usage:
start_state = (0, 0)
while start_state != (2, 2):  # Goal state
    action = np.argmax(Q[start_state])
    start_state = take_action(start_state, action)

print("Q-Learning Path:", start_state)
```