REINFORCEMENT LEARNING

2211cs020685
Ravuri Mourya
Aiml-Zeta

[1]:
```
import math
import random

class UCBGame:
    def __init__(self, num_arms):
        """
        Initialize the UCBGame instance.
        :param num_arms: Number of arms in the multi-armed bandit.
        """
        self.num_arms = num_arms
        self.counts = [0] * num_arms  # Number of times each arm has been played
        self.values = [0.0] * num_arms  # Average reward for each arm
        self.total_plays = 0

    def select_arm(self):
        """
        Selects the arm to play based on the Upper Confidence Bound (UCB)␣
↪algorithm.
        """
        if self.total_plays < self.num_arms:
            # Play each arm at least once
            return self.total_plays

        # Calculate UCB values for all arms
        ucb_values = [
            self.values[i] + math.sqrt((2 * math.log(self.total_plays)) / self.
↪counts[i])
            for i in range(self.num_arms)
        ]
        # Return the arm with the maximum UCB value
        return ucb_values.index(max(ucb_values))

    def update(self, chosen_arm, reward):
        """
        Updates the counts and values for the chosen arm based on the received␣
↪reward.
        :param chosen_arm: The arm that was chosen.
```

```
        :param reward: The reward received from the chosen arm.
        """
        self.counts[chosen_arm] += 1
        self.total_plays += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]

        # Update the value using incremental formula
        self.values[chosen_arm] = ((n - 1) / n) * value + (1 / n) * reward


def simulate_game(num_arms, num_rounds, reward_probabilities):
    """
    Simulates a game where a player uses the UCB algorithm to maximize rewards.
    :param num_arms: Number of actions (arms) available to the player.
    :param num_rounds: Total number of rounds to play.
    :param reward_probabilities: List of probabilities for each arm to give a reward.
    :return: Total rewards accumulated, counts of arm plays, and estimated values for each arm.
    """
    game = UCBGame(num_arms)
    total_rewards = 0

    for _ in range(num_rounds):
        chosen_arm = game.select_arm()
        # Simulate reward: 1 with given probability, 0 otherwise
        reward = 1 if random.random() < reward_probabilities[chosen_arm] else 0
        game.update(chosen_arm, reward)
        total_rewards += reward

    return total_rewards, game.counts, game.values


if __name__ == "__main__":
    # Define game parameters
    num_arms = 5
    num_rounds = 1000
    reward_probabilities = [0.1, 0.2, 0.3, 0.5, 0.8]  # True probabilities of rewards

    # Run simulation
    total_rewards, counts, values = simulate_game(num_arms, num_rounds, reward_probabilities)

    # Output results
    print(f"Total rewards: {total_rewards}")
```

```
    print(f"Number of times each arm was played: {counts}")
    print(f"Estimated values for each arm: {values}")
```

```
Total rewards: 703
Number of times each arm was played: [19, 27, 55, 76, 823]
Estimated values for each arm: [0.05263157894736842, 0.18518518518518517,
0.3999999999999998, 0.47368421052631576, 0.7764277035236941]
```

[2]:
```
import math
import random

class UCBOptimizer:
    def __init__(self, num_modes):
        """
        Initializes the UCBOptimizer.
        :param num_modes: Number of possible settings (modes) for the devices.
        """
        self.num_modes = num_modes  # Number of possible settings for devices
        self.counts = [0] * num_modes  # Number of times each mode has been␣
 ↪selected
        self.values = [0.0] * num_modes  # Average efficiency for each mode
        self.total_selections = 0  # Total number of selections made

    def select_mode(self):
        """
        Selects the mode to use based on the Upper Confidence Bound (UCB)␣
 ↪algorithm.
        """
        if self.total_selections < self.num_modes:
            # Ensure each mode is selected at least once
            return self.total_selections

        # Calculate UCB values for each mode
        ucb_values = [
            self.values[i] + math.sqrt((2 * math.log(self.total_selections)) /␣
 ↪self.counts[i])
            for i in range(self.num_modes)
        ]

        # Return the mode with the highest UCB value
        return ucb_values.index(max(ucb_values))

    def update(self, chosen_mode, efficiency):
        """
        Updates the counts and values for the chosen mode based on the observed␣
 ↪efficiency.
        :param chosen_mode: The mode that was chosen.
```

```
        :param efficiency: The observed efficiency value.
        """
        self.counts[chosen_mode] += 1
        self.total_selections += 1
        n = self.counts[chosen_mode]
        value = self.values[chosen_mode]

        # Update the average efficiency using the incremental formula
        self.values[chosen_mode] = ((n - 1) / n) * value + (1 / n) * efficiency


def simulate_smart_home(num_modes, num_rounds, efficiency_factors):
    """
    Simulates an IoT smart home system optimizing energy usage across multiple␣
 ↪modes.
    :param num_modes: Number of settings (modes) available for devices.
    :param num_rounds: Total number of iterations for optimization.
    :param efficiency_factors: List of real efficiency probabilities for each␣
 ↪mode.
    :return: Total efficiency score and mode selection statistics.
    """
    optimizer = UCBOptimizer(num_modes)
    total_efficiency = 0

    for _ in range(num_rounds):
        chosen_mode = optimizer.select_mode()

        # Simulate efficiency: value based on real efficiency factor plus␣
 ↪random noise
        efficiency = efficiency_factors[chosen_mode] + random.uniform(-0.05, 0.
 ↪05)
        efficiency = max(0, min(1, efficiency))  # Clamp efficiency between 0␣
 ↪and 1

        optimizer.update(chosen_mode, efficiency)
        total_efficiency += efficiency

    return total_efficiency, optimizer.counts, optimizer.values


if __name__ == "__main__":
    # Define simulation parameters
    num_modes = 4  # Example: Low, Medium, High, Auto settings for devices
    num_rounds = 500  # Number of rounds to simulate
    efficiency_factors = [0.7, 0.8, 0.9, 0.85]  # Base efficiency for each mode

    # Run the simulation
```

```
    total_efficiency, counts, values = simulate_smart_home(num_modes,␣
↪num_rounds, efficiency_factors)

    # Display results
    print(f"Total efficiency score: {total_efficiency:.2f}")
    print(f"Number of times each mode was selected: {counts}")
    print(f"Estimated efficiencies for each mode: {values}")
```

```
Total efficiency score: 421.84
Number of times each mode was selected: [62, 101, 202, 135]
Estimated efficiencies for each mode: [0.7020655744272584, 0.8017331823941076,
0.9048560301754909, 0.8485900024313898]
```

[3]:
```
import random
import math

class PACChessGame:
    def __init__(self, board_size, max_depth, epsilon, delta):
        """
        Initialize the PAC Chess game.
        :param board_size: Size of the chessboard.
        :param max_depth: Number of moves ahead to evaluate.
        :param epsilon: Error tolerance for PAC decision-making.
        :param delta: Confidence threshold for PAC decision-making.
        """
        self.board_size = board_size
        self.max_depth = max_depth
        self.epsilon = epsilon
        self.delta = delta
        self.board = self.initialize_board()

    def initialize_board(self):
        """Initializes a simplified chessboard."""
        board = [[None for _ in range(self.board_size)] for _ in range(self.
↪board_size)]

        # Place pawns for both players
        for i in range(self.board_size):
            board[1][i] = 'P1'  # Player 1 pawns
            board[-2][i] = 'P2'  # Player 2 pawns

        # Place rooks
        board[0][0], board[0][-1] = 'R1', 'R1'  # Player 1 rooks
        board[-1][0], board[-1][-1] = 'R2', 'R2'  # Player 2 rooks

        # Place kings
        board[0][self.board_size // 2] = 'K1'  # Player 1 king
```

```
        board[-1][self.board_size // 2] = 'K2'  # Player 2 king

        return board

    def possible_moves(self, player):
        """Generates all possible moves for the given player."""
        moves = []
        for row in range(self.board_size):
            for col in range(self.board_size):
                if self.board[row][col] and self.board[row][col].
↪endswith(player):
                    # Example: Simplified pawn movement
                    if row + 1 < self.board_size:
                        moves.append(((row, col), (row + 1, col)))  # Move␣
↪forward
        return moves

    def evaluate_board(self):
        """Evaluates the board state for scoring."""
        score = 0
        for row in self.board:
            for piece in row:
                if piece == 'P1':
                    score += 1
                elif piece == 'P2':
                    score -= 1
                elif piece == 'K1':
                    score += 10
                elif piece == 'K2':
                    score -= 10
        return score

    def pac_decision(self, moves):
        """Selects the best move using PAC principles."""
        best_move = None
        best_score = float('-inf')

        # Calculate the number of samples needed for PAC guarantee
        samples = int((1 / (self.epsilon ** 2)) * math.log(1 / self.delta))

        for move in moves:
            total_score = 0
            for _ in range(samples):
                total_score += self.simulate_move(move)
            avg_score = total_score / samples

            if avg_score > best_score:
```

```
                best_score = avg_score
                best_move = move

        return best_move

    def simulate_move(self, move):
        """Simulates the effect of a move and evaluates the board."""
        start, end = move
        piece = self.board[start[0]][start[1]]

        # Make the move
        self.board[start[0]][start[1]] = None
        self.board[end[0]][end[1]] = piece

        # Evaluate the board
        score = self.evaluate_board()

        # Undo the move
        self.board[end[0]][end[1]] = None
        self.board[start[0]][start[1]] = piece

        return score

    def play_game(self):
        """Plays a game between two PAC-based players."""
        current_player = '1'

        for turn in range(50):  # Maximum number of turns
            moves = self.possible_moves(current_player)
            if not moves:
                print(f"Player {current_player} has no moves left. Game over.")
                break

            chosen_move = self.pac_decision(moves)
            print(f"Player {current_player} chooses move {chosen_move}")

            start, end = chosen_move
            piece = self.board[start[0]][start[1]]
            self.board[start[0]][start[1]] = None
            self.board[end[0]][end[1]] = piece

            current_player = '2' if current_player == '1' else '1'

        print("Game ended.")


if __name__ == "__main__":
```

```python
# Define game parameters
game = PACChessGame(board_size=6, max_depth=3, epsilon=0.1, delta=0.05)
game.play_game()
```

```
Player 1 chooses move ((0, 0), (1, 0))
Player 2 chooses move ((4, 3), (5, 3))
Player 1 chooses move ((0, 3), (1, 3))
Player 2 chooses move ((4, 0), (5, 0))
Player 1 chooses move ((0, 5), (1, 5))
Player 2 chooses move ((4, 1), (5, 1))
Player 1 chooses move ((1, 0), (2, 0))
Player 2 chooses move ((4, 2), (5, 2))
Player 1 chooses move ((1, 1), (2, 1))
Player 2 chooses move ((4, 4), (5, 4))
Player 1 chooses move ((1, 2), (2, 2))
Player 2 chooses move ((4, 5), (5, 5))
Player 1 chooses move ((1, 3), (2, 3))
Player 2 has no moves left. Game over.
Game ended.
```

[ ]:

```python

```