MONTE CARLO CONTROL ALGORITHM

AIM

To develop a Python program to find the optimal policy for the given RL environment using the Monte Carlo algorithm.

PROBLEM STATEMENT

Develop a Python program that implements a Monte Carlo control algorithm to find the optimal policy for navigating the FrozenLake environment. The program should initialize the environment, define parameters (discount factor, learning rate, exploration rate), and implement decay schedules for efficient learning. It must generate trajectories based on an epsilon-greedy action selection strategy and update the action-value function using sampled episodes. Evaluate the learned policy by calculating the probability of reaching the goal state and the average undiscounted return. Finally, print the action-value function, state-value function, and optimal policy.

MONTE CARLO CONTROL ALGORITHM

1. Initialize Q(s, a) arbitrarily for all state-action pairs
2. Initialize returns(s, a) to empty for all state-action pairs
3. Initialize policy π(s) to be arbitrary (e.g., ε-greedy)
4. For each episode:
   a. Generate an episode using policy π
   b. For each state-action pair (s, a) in the episode:

```
i. Calculate G (return) for that (s, a) pair

ii. Append G to returns(s, a)

iii. Calculate the average of returns(s, a)

iv. Update Q(s, a) using the average return
```

   c. Update policy π(s) based on Q(s, a)

MONTE CARLO CONTROL FUNCTION

```
def mc_control(env, gamma=1.0,
               init_alpha=0.5, min_alpha=0.01, alpha_decay_ratio=0.5,
               init_epsilon=1.0, min_epsilon=0.1, epsilon_decay_ratio=0.9,
               n_episodes=3000, max_steps=200, first_visit=True):
    nS, nA = env.observation_space.n, env.action_space.n

    # Calculate discount factors
    discounts = np.logspace(
        0, max_steps,
        num=max_steps, base=gamma,
        endpoint=False
    )

    # Calculate learning rates (alphas)
    alphas = decay_schedule(
        init_alpha, min_alpha,
        alpha_decay_ratio, n_episodes
    )
```

```
    # Calculate epsilon values for exploration
    epsilons = decay_schedule(
        init_epsilon, min_epsilon,
        epsilon_decay_ratio, n_episodes
    )

    # Initialize variables for tracking policy and Q-values
    pi_track = []
    Q = np.zeros((nS, nA), dtype=np.float64)  # Q-table initialization
    Q_track = np.zeros((n_episodes, nS, nA), dtype=np.float64)

    # Define epsilon-greedy action selection strategy
    select_action = lambda state, Q, epsilon: (
        np.argmax(Q[state]) if np.random.random() > epsilon
        else np.random.randint(len(Q[state]))
    )

    # Loop over episodes
    for e in tqdm(range(n_episodes), leave=False):
        trajectory = generate_trajectory(select_action, Q, epsilons[e], env, max_steps)
        visited = np.zeros((nS, nA), dtype=bool)

        # Loop over trajectory steps (t for time step)
        for t, (state, action, reward, _, _) in enumerate(trajectory):
            # Skip if the state-action pair has been visited and we're using first-visit
            if visited[state][action] and first_visit:
                continue
            visited[state][action] = True

            # Calculate return G (discounted sum of rewards) for this trajectory
            n_steps = len(trajectory[t:])
            G = np.sum(discounts[:n_steps] * trajectory[t:, 2])  # Trajectory[:, 2] is t

            # Update Q-value using the calculated return G and learning rate (alpha)
            Q[state][action] = Q[state][action] + alphas[e] * (G - Q[state][action])

        # Track Q-values and policy for analysis
        Q_track[e] = Q
        pi_track.append(np.argmax(Q, axis=1))

    # Compute the value function (V) by taking the max Q-value for each state
    V = np.max(Q, axis=1)

    # Define greedy policy (pi)
    pi = lambda s: {s: a for s, a in enumerate(np.argmax(Q, axis=1))}[s]

    # Return the Q-values, state-value function, policy, and tracking information
    return Q, V, pi
```

OUTPUT:

Name: SANJAY S
Register Number: 212222230132

Action value function

```
Name: SANJAY   Register Number:212222230132
Action-value function:
| 00 [0.05 0.05 0.07 0.05] | 01 [0.03 0.02 0.01 0.05] | 02 [0.04 0.06 0.06 0.04] | 03 [0.03 0.02 0.   0.  ] |
| 04 [0.1  0.01 0.02 0.03] |          | 06 [0.1  0.03 0.02 0.  ] |          |
| 08 [0.04 0.04 0.08 0.04] | 09 [0.01 0.18 0.08 0.07] | 10 [0.14 0.27 0.17 0.01] |          |
|          | 13 [0.07 0.06 0.15 0.23] | 14 [0.28 0.45 0.6  0.47] |          |
```

optimal value function

```
State-value function:
| 00   0.07 | 01   0.05 | 02   0.06 | 03   0.03 |
| 04    0.1 |           | 06    0.1 |           |
| 08   0.08 | 09   0.18 | 10   0.27 |           |
|           | 13   0.23 | 14    0.6 |           |
```

optimal policy

```
Policy:
| 00      > | 01      ^ | 02      v | 03      < |
| 04      < |           | 06      < |           |
| 08      > | 09      v | 10      v |           |
|           | 13      ^ | 14      > |           |
```

success rate for the optimal policy.

```
Name:SANJAY   Register Number:212222230132
Reaches goal 66.00%. Obtains an average undiscounted return of 0.6600.
```

RESULT:

The Monte Carlo Control Algorithm effectively learns the optimal policy for a given reinforcement learning environment by averaging returns for state-action pairs over multiple episodes. Through exploration and exploitation, it converges to an optimal action-value function, enabling informed decision-making. This approach demonstrates the power of simulation in discovering optimal strategies in complex environments.