MONTE CARLO CONTROL ALGORITHM

AIM

The aim is to use Monte Carlo Control in a specific environment to learn an optimal policy, estimate state-action values, iteratively improve the policy, and optimize decision-making through a functional reinforcement learning algorithm.

PROBLEM STATEMENT

Monte Carlo Control is a reinforcement learning method, to figure out the best actions for different situations in an environment. The provided code is meant to do this, but it's currently having issues with variables and functions.

MONTE CARLO CONTROL ALGORITHM

Step 1:

Initialize Q-values, state-value function, and the policy.

Step 2:

Interact with the environment to collect episodes using the current policy.

Step 3:

For each time step within episodes, calculate returns (cumulative rewards) and update Q-values.

Step 4:

Update the policy based on the improved Q-values.

Step 5:

Repeat steps 2-4 for a specified number of episodes or until convergence.

Step 6:

Return the optimal Q-values, state-value function, and policy.

MONTE CARLO CONTROL FUNCTION

Name: HARINI V

Reg no: 212222230044

```
from tqdm import tqdm
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

  discounts = np.logspace(0,max_steps, num=max_steps,
                          base=gamma, endpoint = False)

  alphas = decay_schedule(init_alpha, min_alpha, alpha_decay_ratio, n_episodes)

  epsilons = decay_schedule(init_epsilon, min_epsilon, epsilon_decay_ratio, n_episodes)

  pi_track = []

  Q = np.zeros((nS, nA), dtype = np.float64)
  Q_track = np.zeros((n_episodes, nS, nA), dtype = np.float64)

  select_action = lambda state, Q, epsilon: np.argmax(Q[state]) if np.random.random() > epsilon_decay_ratio else np

  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)
    for t, (state, action, reward, _, _) in enumerate(trajectory):
      if visited[state][action] and first_visit:
        continue
      visited[state][action] = True
      n_steps = len(trajectory[t:])
      G = np.sum(discounts[:n_steps] * trajectory[t:, 2])
      Q[state][action] += alphas[e] * (G - Q[state][action])
    Q_track[e] = Q
    pi_track.append(np.argmax(Q, axis = 1))
  V = np.max(Q, axis = 1)
  pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis = 1))}[s]
  return Q, V, pi
Q, V, pi = mc_control (env,n_episodes=150000)
print_state_value_function(Q, P, n_cols=4, prec=2, title='Action-value function:')
print_state_value_function(V, P, n_cols=4, prec=2, title='State-value function:')
print_policy(pi, P)
```

OUTPUT:

```
HARINI V
212222230044
Action-value function:
| 00 [0.02 0.02 0.02 0.01] | 01 [0.   0.01 0.01 0.03] | 02 [0.04 0.03 0.06 0.02] | 03 [0.   0.01 0.01 0.02] |
| 04 [0.02 0.02 0.02 0.01] |           | 06 [0.11 0.04 0.06 0.03] |           |
| 08 [0.02 0.04 0.04 0.06] | 09 [0.07 0.14 0.15 0.06] | 10 [0.23 0.2  0.19 0.04] |           |
|           | 13 [0.13 0.25 0.27 0.19] | 14 [0.27 0.65 0.51 0.44] |           |
State-value function:
| 00   0.02 | 01   0.03 | 02   0.06 | 03   0.02 |
| 04   0.02 |           | 06   0.11 |           |
| 08   0.06 | 09   0.15 | 10   0.23 |           |
|           | 13   0.27 | 14   0.65 |           |
Policy:
| 00      < | 01      ^ | 02      > | 03      ^ |
| 04      < |           | 06      < |           |
| 08      ^ | 09      > | 10      < |           |
|           | 13      > | 14      v |           |
```

RESULT:

Thus the program to use Monte Carlo Control in a specific environment to learn an optimal policy, estimate state-action values, iteratively improve the policy, and optimize decision-making through a functional reinforcement learning algorithm is successfully completed.