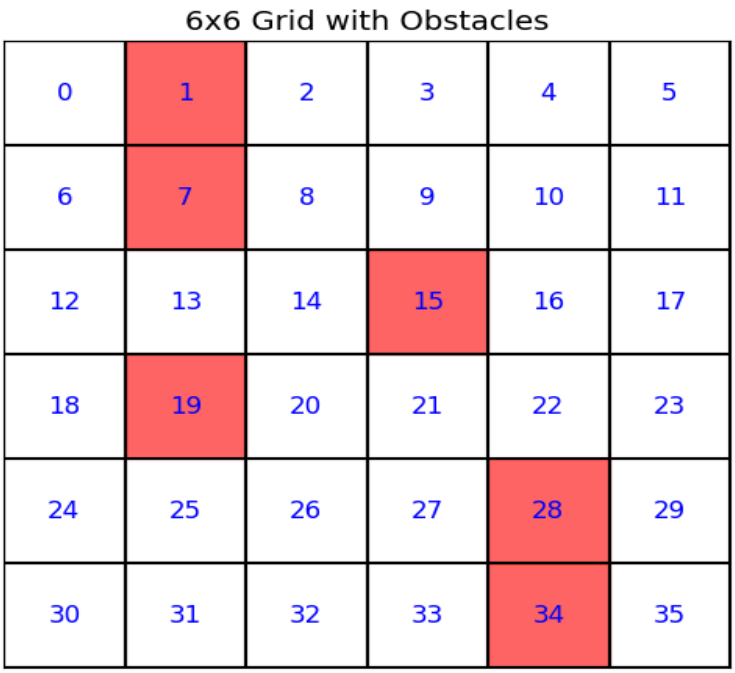
CS F317 REINFORCEMENT LEARNING
2024-25

Treasure Hunt Grid Problem

Under the Supervision of
Dr. Paresh Saxena

Bhaarath K (2021A7PS1816H)
Harshith Kumar Borundia (2021B3A70727H)

Contents



Problem Statement:

1. We have a 6*6 grid with start state at (0,0).
2. The treasure is located at (5,5) which is our terminal state.
3. All the cells colored with red are obstacles that an RL agent has to avoid.
4. The agent gets a reward of -15 upon stepping onto an obstacle.
5. The agent gets a reward of 30 upon reaching the treasure.
6. In all other cases the reward is -1.
7. If the agent tries to move out of the grid from a boundary cell, It will be teleported back to the same cell.

Monte Carlo: On Policy

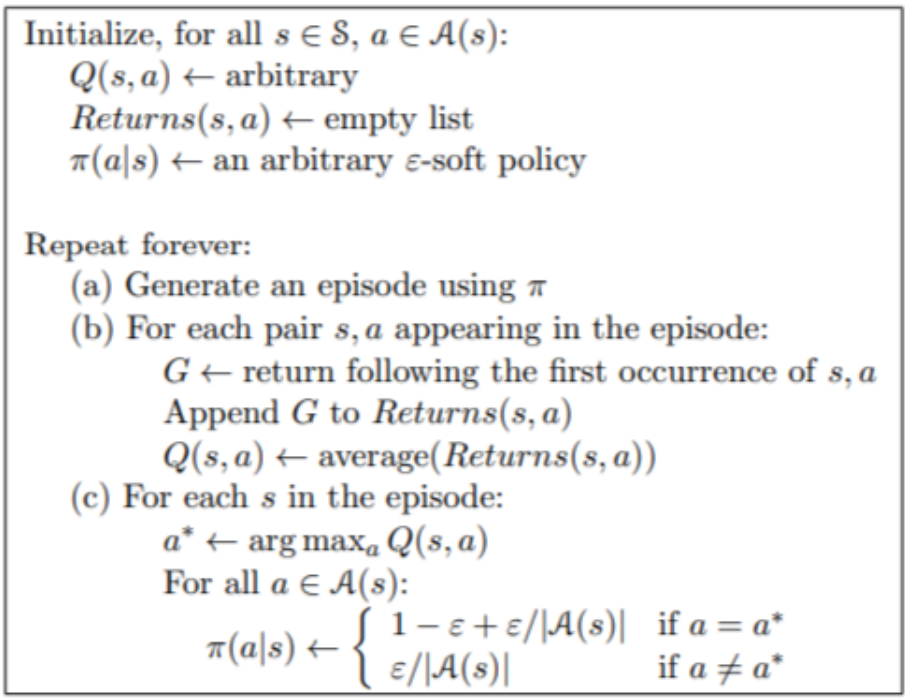


Figure 5.6: An on-policy first-visit MC control algorithm for ε-soft policies.

- The on policy monte carlo method used epsilon greedy policy.These greedy policies are closest to epsilon soft policies since probability of every action given state is greater than or equal to probability of choosing exploratory action over cardinality of A(s).
- The on policy approach is a comprise.It learns action values not for the optimal policy but for a near optimal policy that still explores.
- The overall idea of on-policy Monte Carlo is still GPI.
- Without the assumption of exploring starts, however, we cannot simply improve the policy by making it greedy with respect to the current value function, because that would prevent further exploration of non greedy actions.

Methodology:
We have taken the maximum value of episodes as 60000.Our discounting factor is 1. We are looping through each episode until we reach the treasure/terminal state.If the number of steps increases to more than 3000 we have stopped. After every 100th episode we are appending cumulative reward to an array rewards_list. Then we iterate backwards to update the action value pair.

Monte Carlo: Off Policy

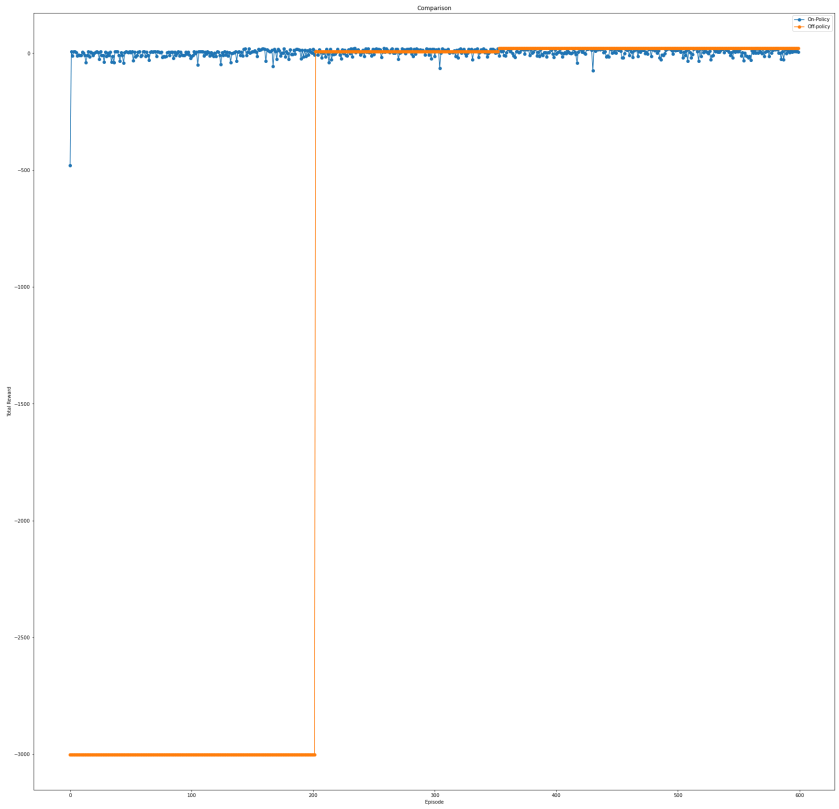
Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \in \mathbb{R}$ (arbitrarily)
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \operatorname{argmax}_a Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):
$b \leftarrow$ any soft policy
Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
$W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

- Here, we have two policies (Although it is not necessary to have two separate policies.), target and behavior policy. One policy learns from the other.We assume coverage which is behavior policy must be stochastic in states where it is not identical to the target policy.
- To find the state value function of the target policy we have two important things to consider: weighted and ordinary importance sampling.Weighted is biased because if there was a single return then it would be the state value of the behavioral policy but its variance is bounded. Ordinary importance sampling on other hand is unbiased but variance is unbounded.
- We require behavior policies to be soft, that is to select all actions in all states with non -zero probability.
- An advantage of this separation is that the target policy may be deterministic while the behavior policy can continue to sample all possible actions.

We ran some experiments and plotted the graph between total reward and episode number,
Note: Here the off-policy is getting a constant reward of -3000 initially since it was getting stuck in a loop and we manually capped the episode length to be at max 3000 steps.



Inference:

The blue line indicates on policy and the orange line indicates off policy. Clearly convergence is faster in on policy cases as compared to off policy cases. In case of off policy there is no noise/fluctuation since it is deterministic. On policy keeps exploring even after reaching convergence in the hope of finding a better optimal.

Code to run the Experiments and plot graph

```
import numpy as np
import matplotlib.pyplot as plt

class GridWorld:
    def __init__(self, size=6, treasure=(5,5), obstacles=[(1,1), (2,3), (3,1),
(5,4),(0,1),(4,4)]):
        self.size = size
        self.treasure = treasure
        self.obstacles = obstacles
        self.reset()

    def reset(self):
```

```
        self.agent_pos = (0, 0)  # Start position
        return self.agent_pos

    def step(self, action):
        x, y = self.agent_pos
        if action == 0: #up
            x = max(0, x - 1)
        elif action == 1:#down
            x = min(self.size - 1, x + 1)
        elif action == 2:#left
            y = max(0, y - 1)
        elif action == 3:#right
            y = min(self.size - 1, y + 1)
        self.agent_pos = (x, y)

        if self.agent_pos == self.treasure:
            return self.agent_pos, 30, True  # Found treasure
        elif self.agent_pos in self.obstacles:
            return self.agent_pos, -15, False  # Fell in obstacles
        else:
            return self.agent_pos, -1, False  # Continue

# Function to convert (x, y) coordinates to state number
def coords_to_state(c, grid_size=6):
    return c[0] * grid_size + c[1]

# Function to convert state number to (x, y) coordinates
def state_to_coords(state, grid_size=6):
    x = state // grid_size
    y = state % grid_size
    return (x, y)

def action_probs(A):
    epsilon = 0.25
    idx = np.argmax(A)
    probs = []
    A_ = len(A)
    for i, a in enumerate(A):
        if i == idx:
            probs.append(round(1-epsilon + (epsilon/A_),3))
        else:
            probs.append(round(epsilon/A_,3))
    err = sum(probs)-1
    substracter = err/len(A)
```

```
    return np.array(probs)-substracter




from tqdm import tqdm
import matplotlib.pyplot as plt



#########################################################################################
#########
# ON POLICY
#########################################################################################
#########
max_episodes = 60000
g = 1 # discounting factor
size = 6
grid  = GridWorld()
action_value_grid = np.zeros((size*size,4))

# creating Returns list, where each state has four possible actions to take
possible_states = size*size
possible_actions = 4
Returns = {}
for state in range(possible_states):
    for action in range(possible_actions):
        Returns[(state,action)] = []
rewards_list = []
for ep in tqdm(range(max_episodes)):
    G = 0
    l = 0
    r = 0
    state = grid.reset()
    trajectory = []
    while True:
        if(ep==0):
            action = np.random.randint(4)
        else:
            action_values = action_value_grid[coords_to_state(state)]
            probs = action_probs(action_values)
            action = np.random.choice(np.arange(4),p=probs)
```

```
        next_state, reward, done = grid.step(action)
        trajectory.append((coords_to_state(state), action, reward))
        state = next_state
        r+=reward
        l+=1
        if done or l > 3000:
            if(ep%100==0):
                rewards_list.append(r)
            break
    for idx, step in enumerate(trajectory[::-1]):
        G = g*G + step[2]
        if step[0] not in np.array(trajectory[::-1])[:,0][idx+1:]:
            Returns[(step[0],step[1])].append(G)
            action_value_grid[step[0]][step[1]] =
np.mean(Returns[(step[0],step[1])])




################################################################################
#########
# OFF POLICY
################################################################################
#########
max_episodes = 60000
g = 1 # discounting factor
size = 6
action_value_grid_off = np.zeros((size*size,4))
C = np.zeros((size*size,4))

b = lambda: np.random.randint(4)
t = lambda p: np.argmax(p)
rewards_list_off = []
for ep in tqdm(range(max_episodes)):
    if(ep%100==0):
        state = grid.reset()
        l=0
        r = 0
        while True:
            action_values = action_value_grid_off[coords_to_state(state)]
            action = t(action_values)
            next_state, reward, done = grid.step(action)
            state = next_state
            r+=reward
```

```
            l+=1
            if done or l > 3000:
                rewards_list_off.append(r)
                break

    G = 0
    W = 1
    state = grid.reset()
    trajectory = []
    while True:
        action = b()
        next_state, reward, done = grid.step(action)
        trajectory.append((coords_to_state(state), action, reward))
        state = next_state
        if done:
            break
    for idx, step in enumerate(trajectory[::-1]):
        G = g*G + step[2]

        C[step[0]][step[1]] += W
        action_value_grid_off[step[0]][step[1]] += (W/C[step[0]][step[1]]) *
(G-action_value_grid_off[step[0]][step[1]])
        action = t(action_value_grid_off[step[0]])
        if action != step[1]:
            break
        W = W*(1/0.25)



################################################################################
#########
# COMPARISON
################################################################################
#########
plt.figure(figsize=(30, 30))
plt.plot(rewards_list, label='On-Policy', marker='o')
plt.plot(rewards_list_off, label='Off-policy', marker='o')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Comparison')
plt.legend()
plt.show()
```

Comparison and benefits:

- Convergence to near optimal policy in case of on policy Monte Carlo is faster. This policy performs both exploration and exploitation on its own.
- On the other hand, off policy is slower to converge to an optimal policy but once it converges the cumulative rewards are mostly better than on policy.
- The above is mainly because we used a deterministic target policy in case of off-policy (since the exploration is taken care by behavioral policy), Once it converges to the optimal policy it deterministically performs the same optimal policy again whereas in on-policy even after it converges to a optimal policy it still explores.
- One could argue that we could have used a deterministic policy for on-policy MC as well but in that case it only learns the values for the states that appear in its episodes (No exploration, No guarantee for finding optimal policy) and we may need to handle it separately (Exploring starts).
- When the stakes of the problem are higher it is recommended to stick to off-policy.
- If off policy is too slow, then temporal differencing is the next method which can be applied.It updates value functions at each step which might reduce time to convergence.