E:\0NEU\CS5100\Mohammed\train_agent.py

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

# ---------------------------------------------------------------------
# 1. FIX SEEDS GLOBALLY
# ---------------------------------------------------------------------
random.seed(1234)
np.random.seed(1234)

from drone_env_limited import DroneCoverageEnvAdaptiveLimited

###############################################################################
# ALL CONFIGURATION IN ONE PLACE
###############################################################################
CONFIG = {
    "N": 10,
    "M": 10,
    "available_sizes": [3,5],
    "max_drones": 25,
    "obstacle_percent": 0.0,

    # Increase coverage more strongly
    "coverage_multiplier": 70.0,

    # Stronger overlap penalty
    "alpha_env": 25.0,
    "beta_env": 2.0,
    # Lower drone penalty so we can place more
    "gamma_penalty_env": 0.01,

    "stall_threshold_env": 500,   # more forgiving
    "max_steps_env": 1000,

    "num_episodes": 2000,
    "gamma_rl": 0.9,
    "alpha_rl": 0.05,
    "epsilon_rl": 1.0,
    "epsilon_decay": 0.999,
    "epsilon_min": 0.01,

    "test_mode": False
}


def state_to_str(obs):
    """
    Convert environment's observation => canonical string, sorting the drones
```

```
    by (size, cx, cy, active).
    """
    drones = obs["drones"]  # each is (cx, cy, size, active)
    canon = []
    for (cx, cy, sz, act) in drones:
        a_bit = 1 if act else 0
        canon.append((sz, cx, cy, a_bit))
    canon.sort()
    return str(canon)


def possible_actions(env, random_spawns=True):
    """
    - NOOP
    - SPAWN_RANDOM for each size in env.available_sizes (if len(env.drones)<env.max_drones)
       => if random_spawns=True, that means we won't require (cx,cy).
    - ACT for each drone => 'REMOVE' or 'STAY' only
      (we removed up/down/left/right toggles).
    """

    acts = [{"type": "NOOP"}]

    # 1) SPawns: Instead of enumerating x,y, do single "SPAWN_RANDOM" per size
    if len(env.drones) < env.max_drones:
        for s in env.available_sizes:
            acts.append({"type": "SPAWN_RANDOM", "size": s})

    # 2) For existing drones => "REMOVE" or "STAY"
    for i in range(len(env.drones)):
        acts.append({"type":"ACT","drone_index":i,"move":"REMOVE"})
        # "STAY" = do nothing but keep active
        acts.append({"type":"ACT","drone_index":i,"move":"STAY"})

    return acts


def safe_q(Q_table, s, a):
    if s not in Q_table:
        Q_table[s] = {}
    if a not in Q_table[s]:
        Q_table[s][a] = 0.0
    return Q_table[s][a]


################################################################################
# Q-LEARNING
################################################################################
def Q_learning_adaptive_limited(config):
    """
    Train a tabular Q-table with epsilon-greedy exploration.
```

```
    Keep track of coverage fraction as well as reward.
    """
    env = DroneCoverageEnvAdaptiveLimited(config)
    Q_table = {}

    best_Q_table = {}
    best_coverage_fraction = -1.0

    num_episodes = config["num_episodes"]
    gamma  = config["gamma_rl"]
    alpha  = config["alpha_rl"]
    epsilon= config["epsilon_rl"]
    eps_decay = config["epsilon_decay"]
    eps_min   = config["epsilon_min"]

    ep_rewards = []
    ep_coverages = []  # store coverage fraction each episode

    # (A) Count how often the agent visits the "empty" state => "[]"
    empty_visits = 0

    with open("training_output.txt", "w") as log_file:

        for ep in range(num_episodes):
            obs = env.reset()
            s_str = state_to_str(obs)
            done = False
            ep_reward = 0.0
            steps = 0

            if s_str == "[]":
                empty_visits += 1

            while not done:
                acts = possible_actions(env, random_spawns=True)

                # Epsilon-greedy for training
                if random.random() < epsilon:
                    act = random.choice(acts)
                else:
                    best_val = float("-inf")
                    chosen = None
                    for a in acts:
                        val = safe_q(Q_table, s_str, str(a))
                        if val > best_val:
                            best_val = val
                            chosen = a
                    act = chosen

                next_obs, reward, done, info = env.step(act)
```

```
            ep_reward += reward

            sp_str = state_to_str(next_obs)
            old_q  = safe_q(Q_table, s_str, str(act))

            if sp_str not in Q_table:
                Q_table[sp_str] = {}

            # Q-learning update
            if not done:
                nxt_acts = possible_actions(env, random_spawns=True)
                best_next = float("-inf")
                for na in nxt_acts:
                    v = safe_q(Q_table, sp_str, str(na))
                    if v > best_next:
                        best_next = v
                td_target = reward + gamma * best_next
            else:
                td_target = reward

            new_q = old_q + alpha*(td_target - old_q)
            Q_table[s_str][str(act)] = new_q

            s_str = sp_str
            steps += 1

            if s_str == "[]":
                empty_visits += 1

        # end of episode
        if epsilon > eps_min:
            epsilon *= eps_decay

        ep_rewards.append(ep_reward)

        coverage_fraction_episode = 0.0
        if env.num_free_cells > 0:
            coverage_fraction_episode = (
                env.previous_coverage / float(env.num_free_cells)
            )
        ep_coverages.append(coverage_fraction_episode)

        # If better coverage => copy entire Q-table
        if coverage_fraction_episode > best_coverage_fraction:
            best_coverage_fraction = coverage_fraction_episode
            best_Q_table = {}
            for st in Q_table:
                best_Q_table[st] = {}
                for ac in Q_table[st]:
                    best_Q_table[st][ac] = Q_table[st][ac]
```

```
                print(f"  --> Found new best coverage: {100.0*coverage_fraction_episode:.1f}%")

            line_str = (f"Episode {ep+1}/{num_episodes} => "
                        f"steps={steps}, reward={ep_reward:.3f}, "
                        f"coverage={env.previous_coverage}/{env.num_free_cells} "
                        f"({coverage_fraction_episode*100:.1f}%)")
            print(line_str)
            log_file.write(line_str + "\n")

    # Plot training curve
    plt.figure(figsize=(10,5))

    plt.subplot(1,2,1)
    plt.plot(ep_rewards, label="Episode Reward")
    plt.xlabel("Episode")
    plt.ylabel("Reward")
    plt.title("Training Rewards Over Episodes")
    plt.legend()

    plt.subplot(1,2,2)
    plt.plot(ep_coverages, label="Coverage Fraction")
    plt.xlabel("Episode")
    plt.ylabel("Coverage Fraction")
    plt.title("Coverage Fraction Over Episodes")
    plt.legend()

    plt.tight_layout()
    plt.savefig("training_progress.png", dpi=100)
    plt.close()

    print(f"\n[INFO] Visited the empty state '[]' {empty_visits} times in training!")

    # Return the best Q-table found
    return best_Q_table


def evaluate_policy(Q_table, config):
    """
    Evaluate purely greedily from Q_table until done or max_steps.
    We'll do a short forced spawn loop, and also do some mild epsilon exploration
    so that we actually try spawns if the Q-values are not well formed.
    """

    random.seed(1234)
    np.random.seed(1234)

    config = dict(config)
    config["test_mode"] = True
    config["max_steps_env"] = 500
```

```
    env = DroneCoverageEnvAdaptiveLimited(config)
    obs = env.reset()
    s_str = state_to_str(obs)

    done = False
    total_r = 0.0
    steps = 0
    max_steps = config["max_steps_env"]

    # We'll do a short forced spawn loop: 5 spawns
    # but we'll pick the best action among spawn actions
    # if it doesn't exist, we do a random spawn.
    # or we do an epsilon approach.

    # Let's do a small epsilon approach in final test:
    # (Even though "purely greedy" was the original, we want guaranteed coverage.)
    test_epsilon = 0.3

    for i in range(5):
        if done:
            break
        acts = possible_actions(env, random_spawns=True)  # random spawn available
        if random.random() < test_epsilon:
            act = random.choice(acts)
        else:
            best_val = float("-inf")
            chosen = acts[0]
            for a in acts:
                val = safe_q(Q_table, s_str, str(a))
                if val > best_val:
                    best_val = val
                    chosen = a
            act = chosen

        next_obs, r, done, _ = env.step(act)
        total_r += r
        s_str = state_to_str(next_obs)
        steps += 1

    # Now do the normal Q-based loop
    while not done and steps < max_steps:
        acts = possible_actions(env, random_spawns=True)

        if random.random() < test_epsilon:
            chosen = random.choice(acts)
        else:
            best_val = float("-inf")
            chosen = acts[0]
            for a in acts:
```

```
                val = safe_q(Q_table, s_str, str(a))
                if val > best_val:
                    best_val = val
                    chosen = a

        next_obs, r, done, _ = env.step(chosen)
        total_r += r
        s_str = state_to_str(next_obs)
        steps += 1

    return total_r, env._get_observation()
```