```python
import numpy as np
import random
from result import plot_policy_with_values, print_result
# ================================
# Gridworld Environment Definition
# ================================

class Gridworld:
    def __init__(self, rows, cols, terminal_states=set(), forbidden=set(), step_penalty=-0.1,
forbidden_penalty=-5):
        self.rows = rows
        self.cols = cols
        self.grid_size = (rows, cols)
        self.all_states = [(r, c) for r in range(rows) for c in range(cols)]
        self.terminal_states = terminal_states
        self.forbidden = forbidden
        self.actions = ["UP", "DOWN", "LEFT", "RIGHT"]
        self.step_penalty = step_penalty
        self.forbidden_penalty = forbidden_penalty

    def get_reward(self, state):
        if state in self.terminal_states:
            return 1
        elif state in self.forbidden:
            return self.forbidden_penalty
        else:
            return self.step_penalty

    def get_next_state(self, state, action):
        row, col = state
        if state in self.terminal_states:
            return state
        if action == "UP":
            return (max(row - 1, 0), col)
        elif action == "DOWN":
            return (min(row + 1, self.rows - 1), col)
        elif action == "LEFT":
            return (row, max(col - 1, 0))
        elif action == "RIGHT":
            return (row, min(col + 1, self.cols - 1))
        return state

# ================================
# Value Iteration
# ================================

def value_iteration_deterministic(env, theta=1e-4, gamma=0.95):
    """Deterministic Value Iteration with next-state reward"""
    state_values = {s: 0.0 for s in env.all_states}
    policy = {}

    while True:
        delta = 0
        for state in env.all_states:
            if state in env.terminal_states:
```

```
                policy[state] = None
                continue

            action_values = {}
            for a in env.actions:
                next_s = env.get_next_state(state, a)
                reward = env.get_reward(next_s)
                action_values[a] = reward + gamma * state_values[next_s]

            best_action = max(action_values, key=action_values.get)
            best_value = action_values[best_action]

            delta = max(delta, abs(state_values[state] - best_value))
            state_values[state] = best_value
            policy[state] = best_action

        if delta < theta:
            break

    return state_values, policy


def value_iteration_stochastic(env, theta=1e-4, gamma=0.95, epsilon=0.1):
    """Stochastic Value Iteration with next-state reward and ε-greedy policy"""
    state_values = {s: 0.0 for s in env.all_states}
    policy = {s: {a: 1/len(env.actions) for a in env.actions} for s in env.all_states}

    while True:
        delta = 0
        for state in env.all_states:
            if state in env.terminal_states:
                policy[state] = None
                continue

            action_values = {}
            for a in env.actions:
                next_s = env.get_next_state(state, a)
                reward = env.get_reward(next_s)
                action_values[a] = reward + gamma * state_values[next_s]

            best_action = max(action_values, key=action_values.get)
            best_value = action_values[best_action]

            delta = max(delta, abs(state_values[state] - best_value))
            state_values[state] = best_value

            # ε-greedy stochastic policy
            n_actions = len(env.actions)
            for a in env.actions:
                policy[state][a] = 1 - epsilon + epsilon/n_actions if a == best_action else
epsilon/n_actions

        if delta < theta:
            break

    return state_values, policy
```

```python
# ================================
# Policy Iteration
# ================================

def policy_iteration_deterministic(env, theta=1e-4, gamma=0.95):
    """Deterministic Policy Iteration with next-state reward"""
    state_values = {s: 0.0 for s in env.all_states}
    policy = {s: env.actions[0] for s in env.all_states if s not in env.terminal_states}

    while True:
        # Policy Evaluation
        while True:
            delta = 0
            for state in env.all_states:
                if state in env.terminal_states:
                    continue
                action = policy[state]
                next_s = env.get_next_state(state, action)
                reward = env.get_reward(next_s)
                new_value = reward + gamma * state_values[next_s]
                delta = max(delta, abs(state_values[state] - new_value))
                state_values[state] = new_value
            if delta < theta:
                break

        # Policy Improvement
        policy_stable = True
        for state in env.all_states:
            if state in env.terminal_states:
                policy[state] = None
                continue
            old_action = policy[state]
            action_values = {}
            for a in env.actions:
                next_s = env.get_next_state(state, a)
                reward = env.get_reward(next_s)
                action_values[a] = reward + gamma * state_values[next_s]
            best_action = max(action_values, key=action_values.get)
            policy[state] = best_action
            if old_action != best_action:
                policy_stable = False

        if policy_stable:
            break

    return state_values, policy


def policy_iteration_stochastic(env, theta=1e-4, gamma=0.95, epsilon=0.1):
    """Stochastic Policy Iteration with next-state reward and ε-greedy improvement"""
    state_values = {s: 0.0 for s in env.all_states}
    policy = {s: {a: 1/len(env.actions) for a in env.actions} for s in env.all_states}

    while True:
        # Policy Evaluation
```

```
        while True:
            delta = 0
            for state in env.all_states:
                if state in env.terminal_states:
                    continue
                expected_value = sum(prob * (env.get_reward(env.get_next_state(state, a)) +
                                             gamma * state_values[env.get_next_state(state,
a)])
                                     for a, prob in policy[state].items())
                delta = max(delta, abs(state_values[state] - expected_value))
                state_values[state] = expected_value
            if delta < theta:
                break

        # Policy Improvement
        policy_stable = True
        for state in env.all_states:
            if state in env.terminal_states:
                policy[state] = None
                continue

            old_action_probs = policy[state].copy()
            action_values = {a: env.get_reward(env.get_next_state(state, a)) +
                                 gamma * state_values[env.get_next_state(state, a)]
                                 for a in env.actions}
            best_action = max(action_values, key=action_values.get)

            n_actions = len(env.actions)
            for a in env.actions:
                policy[state][a] = 1 - epsilon + epsilon/n_actions if a == best_action else
epsilon/n_actions

            if policy[state] != old_action_probs:
                policy_stable = False

        if policy_stable:
            break

    return state_values, policy

# ================================
# Monte Carlo Policy Iteration
# ================================

def monte_carlo_policy_iteration_stochastic(env, gamma=0.95, episodes=10000, epsilon=0.1):
    # Initialize state values, returns, and policy
    state_values = {s: 0.0 for s in env.all_states}
    returns = {s: [] for s in env.all_states}
    policy = {s: {a: 1/len(env.actions) for a in env.actions}
              for s in env.all_states if s not in env.terminal_states}

    for ep in range(episodes):
        # --------------------------
        # Generate an episode
        # --------------------------
```

```
        episode = []
        state = random.choice([s for s in env.all_states if s not in env.terminal_states])
        done = False

        while not done:
            if state in env.terminal_states:
                # Terminal state reward
                episode.append((state, None, env.get_reward(state)))
                done = True
                continue

            # Choose action based on ε-greedy policy
            action_probs = policy[state]
            actions_list, probs_list = zip(*action_probs.items())
            action = np.random.choice(actions_list, p=probs_list)

            next_state = env.get_next_state(state, action)
            reward = env.get_reward(next_state)
            episode.append((state, action, reward))

            state = next_state
            if state in env.terminal_states:
                done = True

        # --------------------------
        # Policy Evaluation (Last-visit MC)
        # --------------------------
        G = 0
        visited = set()
        for t in reversed(range(len(episode))):
            s, a, r = episode[t]
            G = gamma * G + r
            if s not in visited:
                returns[s].append(G)
                state_values[s] = np.mean(returns[s])
                visited.add(s)

        # --------------------------
        # Policy Improvement (ε-greedy)
        # --------------------------
        for s in env.all_states:
            if s in env.terminal_states:
                policy[s] = None
                continue
            # Compute Q-values using next-state rewards
            action_values = {a: env.get_reward(env.get_next_state(s, a)) +
                                 gamma * state_values[env.get_next_state(s, a)]
                                 for a in env.actions}

            best_action = max(action_values, key=action_values.get)
            n_actions = len(env.actions)
            for a in env.actions:
                policy[s][a] = 1 - epsilon + epsilon / n_actions if a == best_action else
epsilon / n_actions
    return state_values, policy
```

```
# ========================================================
# Epsilon Greedy Policy used in SARSA and Q-Learning
# ========================================================

def derive_epsilon_greedy_policy(Q, actions, all_states, terminal_states, epsilon=0.1):
    policy = {}
    n_actions = len(actions)
    for s in all_states:
        if s in terminal_states:
            policy[s] = None
        else:
            best_action = max(Q[s], key=Q[s].get)
            policy[s] = {a: (1 - epsilon + epsilon / n_actions if a == best_action else
epsilon / n_actions)
                         for a in actions}
    return policy



# ==================================
# SARSA
# ==================================

def sarsa_stochastic(env, alpha=0.1, gamma=0.95, epsilon=0.1, episodes=10000, max_steps=100):
    Q = {s: {a: 0.0 for a in env.actions} for s in env.all_states}

    def epsilon_greedy_action(state):
        if random.random() < epsilon:
            return random.choice(env.actions)
        return max(Q[state], key=Q[state].get)

    for ep in range(episodes):
        state = random.choice([s for s in env.all_states if s not in env.terminal_states])
        action = epsilon_greedy_action(state)
        for _ in range(max_steps):
            next_state = env.get_next_state(state, action)
            reward = env.get_reward(next_state)
            if next_state in env.terminal_states:
                Q[state][action] += alpha * (reward - Q[state][action])
                break
            next_action = epsilon_greedy_action(next_state)
            Q[state][action] += alpha * (reward + gamma * Q[next_state][next_action] -
Q[state][action])
            state, action = next_state, next_action

    state_values = {s: max(Q[s].values()) for s in env.all_states}
    policy = derive_epsilon_greedy_policy(Q, env.actions, env.all_states, env.terminal_states,
epsilon)
    return state_values, policy
```

```python
# ================================
# Q-Learning
# ================================

def q_learning_stochastic(env, alpha=0.1, gamma=0.95, epsilon=0.1, episodes=10000,
max_steps=100):
    Q = {s: {a: 0.0 for a in env.actions} for s in env.all_states}

    def epsilon_greedy_action(state):
        if random.random() < epsilon:
            return random.choice(env.actions)
        return max(Q[state], key=Q[state].get)

    for ep in range(episodes):
        state = random.choice([s for s in env.all_states if s not in env.terminal_states])
        for _ in range(max_steps):
            action = epsilon_greedy_action(state)
            next_state = env.get_next_state(state, action)
            reward = env.get_reward(next_state)
            if next_state in env.terminal_states:
                Q[state][action] += alpha * (reward - Q[state][action])
                break
            Q[state][action] += alpha * (reward + gamma * max(Q[next_state].values()) -
Q[state][action])
            state = next_state

    state_values = {s: max(Q[s].values()) for s in env.all_states}
    policy = derive_epsilon_greedy_policy(Q, env.actions, env.all_states, env.terminal_states,
epsilon)
    return state_values, policy



# ================================
# Main Execution
# ================================

def main():
    # -------------------------
    # User Inputs for Gridworld
    # -------------------------
    rows = int(input("Enter number of rows: "))
    cols = int(input("Enter number of columns: "))

    forbidden = set()
    f_count = int(input("Enter number of forbidden cells: "))
    for i in range(f_count):
        r, c = map(int, input(f"Enter forbidden cell {i+1} (row col) indexed(0, 0): ").split())
        forbidden.add((r, c))

    terminal_states = set()
    t_count = int(input("Enter number of terminal cells: "))
    for i in range(t_count):
        r, c = map(int, input(f"Enter terminal cell {i+1} (row col) indexed(0, 0): ").split())
```

```python
            terminal_states.add((r, c))

    # Initialize environment
    env = Gridworld(rows, cols, terminal_states, forbidden)

    # --------------------------
    # Run all algorithms
    # --------------------------
    experiments = [
        ("Deterministic Value Iteration", value_iteration_deterministic, False),
        ("Stochastic Value Iteration", value_iteration_stochastic, True),
        ("Deterministic Policy Iteration", policy_iteration_deterministic, False),
        ("Stochastic Policy Iteration", policy_iteration_stochastic, True),
        ("Monte Carlo Policy Iteration", monte_carlo_policy_iteration_stochastic, True),
        ("SARSA Learned Policy", sarsa_stochastic, True),
        ("Q-Learning Policy", q_learning_stochastic, True)
    ]

    for title, algorithm, stochastic in experiments:
        print(f"{title}:")
        state_values, policy = algorithm(env)
        print_result(state_values, policy, env.grid_size, forbidden, stochastic=stochastic)
        plot_policy_with_values(state_values, policy, env.grid_size, terminal_states,
forbidden, title=title)
        print("------------------------------------------------------")

if __name__ == "__main__":
    main()
```