```
import numpy as np

class GridWorld:
    def __init__(self, grid_size, start_state, goal_state, hole_states, actions):
        self.grid_size = grid_size
        self.start_state = start_state
        self.goal_state = goal_state
        self.hole_states = hole_states
        self.actions = actions

    def step(self, state, action):
        next_state = np.array(state) + np.array(action)
        next_state = np.clip(next_state, [0, 0], [self.grid_size[0]-1, self.grid_size[1]-1])  # Ensure the agent stays within the grid
        if tuple(next_state) == self.goal_state:
            reward = 1
            done = True
        elif tuple(next_state) in self.hole_states:
            reward = -1
            done = True
        else:
            reward = 0
            done = False
        return tuple(next_state), reward, done

def monte_carlo_control(env, num_episodes=1000, epsilon=0.1, discount_factor=0.9):
    Q = {}
    N = {}  # Count of visits to state-action pairs
    returns_sum = {}  # Sum of returns for state-action pairs

    def policy(state):
      actions_array = np.array(env.actions)  # Convert actions to numpy array
      if np.random.rand() < epsilon:
          return tuple(actions_array[np.random.choice(len(env.actions))])  # Randomly select an action
      else:
          return tuple(actions_array[np.argmax([Q.get((state, a), 0) for a in env.actions])])  # Choose action with maximum Q-value

    for _ in range(num_episodes):
        episode = []
        state = env.start_state
        while True:
            action = policy(state)
            next_state, reward, done = env.step(state, action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state

        G = 0
        visited_state_actions = set()
        for t in range(len(episode)-1, -1, -1):
            state, action, reward = episode[t]
            G = discount_factor * G + reward
            sa_pair = (state, action)
            if sa_pair not in visited_state_actions:
                N[sa_pair] = N.get(sa_pair, 0) + 1
                returns_sum[sa_pair] = returns_sum.get(sa_pair, 0) + G
                Q[sa_pair] = returns_sum[sa_pair] / N[sa_pair]
                visited_state_actions.add(sa_pair)

    return Q

# Define the grid world parameters
grid_size = (4, 4)
start_state = (0, 0)
goal_state = (3, 3)
hole_states = [(1, 1), (2, 2)]
actions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# Create the grid world environment
env = GridWorld(grid_size, start_state, goal_state, hole_states, actions)

# Run Monte Carlo control to learn the optimal policy
optimal_policy = monte_carlo_control(env)

# Display the learned optimal policy
for i in range(grid_size[0]):
    for j in range(grid_size[1]):
        state = (i, j)
        if state == goal_state:
            print("G", end="\t")
        elif state in hole_states:
```

```
            print("H", end="\t")
        elif state == start_state:
            print("S", end="\t")
        else:
            action = optimal_policy.get(state, None)
            if action == (0, 1):
                print("→", end="\t")
            elif action == (0, -1):
                print("←", end="\t")
            elif action == (1, 0):
                print("↓", end="\t")
            elif action == (-1, 0):
                print("↑", end="\t")
    print()
```

```
S
H
H
G
```

```python
import numpy as np

class GridWorld:
    def __init__(self, grid_size, start_state, goal_state, hole_states, actions):
        self.grid_size = grid_size
        self.start_state = start_state
        self.goal_state = goal_state
        self.hole_states = hole_states
        self.actions = actions

    def step(self, state, action):
        next_state = np.array(state) + np.array(action)
        next_state = np.clip(next_state, [0, 0], [self.grid_size[0]-1, self.grid_size[1]-1])  # Ensure the agent stays within the gri
        if tuple(next_state) == self.goal_state:
            reward = 1
            done = True
        elif tuple(next_state) in self.hole_states:
            reward = -1
            done = True
        else:
            reward = 0
            done = False
        return tuple(next_state), reward, done

def monte_carlo_control(env, num_episodes=1000, epsilon=0.1, discount_factor=0.9):
    Q = {}
    N = {}  # Count of visits to state-action pairs
    returns_sum = {}  # Sum of returns for state-action pairs
    episode_rewards = []

    def policy(state):
        actions_array = np.array(env.actions)
        if np.random.rand() < epsilon:
            return tuple(actions_array[np.random.choice(len(env.actions))])  # Randomly select an action
        else:
            return tuple(actions_array[np.argmax([Q.get((state, a), 0) for a in env.actions])])  # Choose action with maximum Q-value

    for _ in range(num_episodes):
        episode = []
        state = env.start_state
        total_reward = 0
        while True:
            action = policy(state)
            next_state, reward, done = env.step(state, action)
            episode.append((state, action, reward))
            total_reward += reward
            if done:
                episode_rewards.append(total_reward)
                break
            state = next_state

        G = 0
        visited_state_actions = set()
        for t in range(len(episode)-1, -1, -1):
            state, action, reward = episode[t]
            G = discount_factor * G + reward
            sa_pair = (state, action)
            if sa_pair not in visited_state_actions:
                N[sa_pair] = N.get(sa_pair, 0) + 1
                returns_sum[sa_pair] = returns_sum.get(sa_pair, 0) + G
                Q[sa_pair] = returns_sum[sa_pair] / N[sa_pair]
                visited_state_actions.add(sa_pair)

    # Calculate the average reward
    avg_reward = np.mean(episode_rewards)

    return Q, avg_reward

def td_learning(env, num_episodes=1000, alpha=0.1, epsilon=0.1, discount_factor=0.9):
    Q = {}
    episode_rewards = []

    for _ in range(num_episodes):
        state = env.start_state
        total_reward = 0
        while True:
            if np.random.rand() < epsilon:
                actions = [action[0] for action in env.actions]  # Extract actions from tuples
                action = (np.random.choice(actions),)  # Randomly choose an action and convert it to tuple
            else:
                action_values = [Q.get((state, a), 0) for a, _ in env.actions]
                action = env.actions[np.argmax(action_values)]
```

```
            next_state, reward, done = env.step(state, action)
            total_reward += reward

            td_target = reward + discount_factor * max(Q.get((next_state, a), 0) for a, _ in env.actions)
            td_error = td_target - Q.get((state, action), 0)
            Q[(state, action)] = Q.get((state, action), 0) + alpha * td_error

            if done:
                episode_rewards.append(total_reward)
                break

            state = next_state

    # Calculate the average reward
    avg_reward = np.mean(episode_rewards)

    return Q, avg_reward

# Define the grid world parameters
grid_size = (4, 4)
start_state = (0, 0)
goal_state = (3, 3)
hole_states = [(1, 1), (2, 2)]
actions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# Create the grid world environment
env = GridWorld(grid_size, start_state, goal_state, hole_states, actions)

# Run Monte Carlo control to learn the optimal policy
optimal_policy_mc, avg_reward_mc = monte_carlo_control(env)
print("Monte Carlo Agent:")
print("Average Reward:", avg_reward_mc)
print("Learned State Values:")
for state in sorted(optimal_policy_mc.keys()):
    print(f"State: {state}, Value: {optimal_policy_mc[state]}")

print("\n")

# Run Temporal-Difference learning to learn the optimal policy
optimal_policy_td, avg_reward_td = td_learning(env)
print("Temporal-Difference Agent:")
print("Average Reward:", avg_reward_td)
print("Learned State Values:")
for state in sorted(optimal_policy_td.keys()):
```

```
Monte Carlo Agent:
Average Reward: 0.88
Learned State Values:
State: ((0, 0), (-1, 0)), Value: 0.3443947718251938
State: ((0, 0), (0, -1)), Value: 0.4476613827318654
State: ((0, 0), (0, 1)), Value: 0.49300303687387675
State: ((0, 0), (1, 0)), Value: 0.23552073761512093
State: ((0, 1), (-1, 0)), Value: 0.49093719995454566
State: ((0, 1), (0, -1)), Value: 0.36923552619639627
State: ((0, 1), (0, 1)), Value: 0.5941726054065737
State: ((0, 1), (1, 0)), Value: -1.0
State: ((0, 2), (-1, 0)), Value: 0.6341545862068968
State: ((0, 2), (0, -1)), Value: 0.3004875684
State: ((0, 2), (0, 1)), Value: 0.6757397684488391
```