**Batch: C-1** **Roll No: 16010121031**

**Experiment / assignment / tutorial No: 5**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

**TITLE:** To implement temporal difference approach

**AIM:** TD learning and prediction

---

**Expected OUTCOME of Experiment:**
CO-3: Apply different temporal difference learning policies.

---

**Books/ Journals/ Websites referred:**
Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning: An Introduction",* The MIT Press, Second Edition, 2018

---

**Pre-Lab/ Prior Concepts:**
TD methods learn directly from episodes of experience. TD is model-free: no knowledge of MDP transitions / rewards is required. TD learns from incomplete episodes, by bootstrapping. TD updates a guess towards a guess. TD can learn before knowing the final outcome. TD can learn online after every step.

**Chosen Problem Statement:** Driving Home

**Explain following concepts w.r.t. chosen problem statement:**

- **Policy:**
  The policy defines how the driver (agent) selects actions at each decision point, such as choosing routes or adjusting speed. In an epsilon-greedy policy:

  a) *Exploitation (1 - ε):* The driver chooses the route that minimizes expected travel time based on current knowledge.

  b) *Exploration (ε):* The driver explores alternative routes to potentially discover faster or more efficient paths.

- **Reward Function:**
  The reward function provides feedback based on travel time or cost:

  a) *Negative Reward:* The agent receives a negative reward proportional to the time taken for each route segment. Longer routes incur larger penalties (negative rewards).

  b) *Positive Reward:* Reaching home within a specific time frame may yield a positive reward, incentivizing the agent to optimize driving time.

- **Value Function:**
  The value function estimates the expected total reward, typically minimizing travel time, starting from a specific state:

  a) *State Value Function (V(s)):* The expected remaining travel time starting from state s.

  b) *Action Value Function (Q(s, a)):* The expected remaining travel time when taking action a in state s.

- **Model of the Environment:**
  The environment consists of external factors that influence driving decisions, such as:

  a) *Road Network:* The routes available between the starting point and home.

  b) *Traffic Conditions:* Varying traffic that can delay or speed up travel on different routes.

  c) *Weather:* External conditions like rain or snow that may affect driving speeds.

  d) *Signals and Rules:* Traffic lights, stop signs, and speed limits that impact the time taken at intersections.

<u>**Implementation**</u>

**Code:**

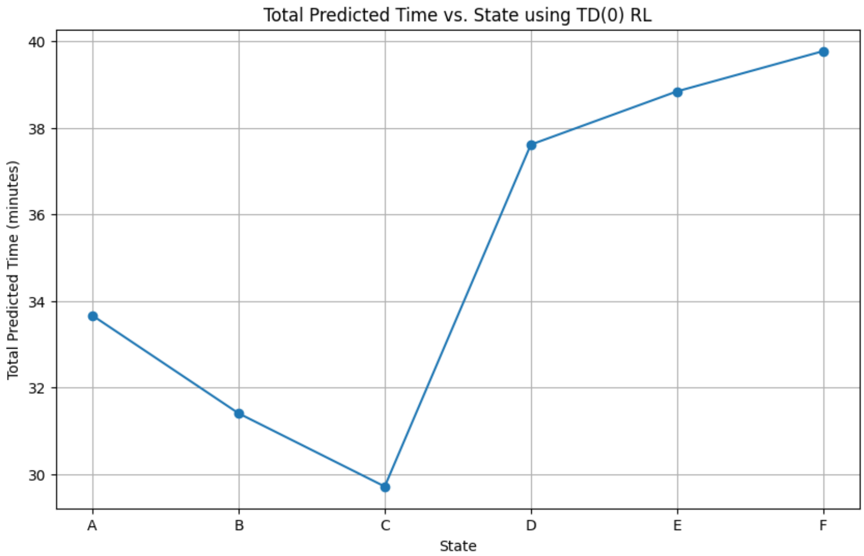
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Define states
states = ['A', 'B', 'C', 'D', 'E', 'F']
num_states = len(states)

# Define initial elapsed times with a 5 min gap
initial_elapsed_times = [0, 5, 10, 20, 30, 40]

# Add noise to the elapsed times
np.random.seed(42)
noise_elapsed = np.random.normal(0, 1, num_states)
final_elapsed_times = np.maximum(np.array(initial_elapsed_times) +
noise_elapsed, 0)

# Define fixed predicted times from each state to the final state
predicted_times_to_go = [30, 25, 20, 15, 10, 0]

# Add noise to the predicted times
noise_predicted = np.random.normal(0, 2, num_states)
final_predicted_times_to_go =
np.maximum(np.array(predicted_times_to_go) + noise_predicted, 0)
```

```
# Calculate total predicted times
total_predicted_times = final_elapsed_times +
final_predicted_times_to_go

# Round the times to two decimal places
final_elapsed_times = np.round(final_elapsed_times, 2)
final_predicted_times_to_go = np.round(final_predicted_times_to_go,
2)
total_predicted_times = np.round(total_predicted_times, 2)

# Initialize value function
V = {state: 0 for state in states}

# Parameters
alpha = 0.1  # Learning rate
gamma = 1.0  # Discount factor
num_episodes = 1000

# TD(0) Learning
for episode in range(num_episodes):
    state_idx = 0
    while state_idx < num_states - 1:
        state = states[state_idx]
        next_state = states[state_idx + 1]
        reward = -total_predicted_times[state_idx]  # Negative to
minimize time
        if next_state == 'F':
            V[state] += alpha * (reward - V[state])
        else:
            V[state] += alpha * (reward + gamma * V[next_state] -
V[state])
        state_idx += 1

# Create a DataFrame for tabular output
data = {
    'State': states,
    'Elapsed Time (minutes)': final_elapsed_times,
    'Predicted Time to Go (minutes)': final_predicted_times_to_go,
    'Predicted Total Time (minutes)': total_predicted_times
}
df = pd.DataFrame(data)

# Print the DataFrame
print(df)

# Plot the graph
plt.figure(figsize=(10, 6))
plt.plot(states, total_predicted_times, marker='o')
```

```
plt.xlabel('State')
plt.ylabel('Total Predicted Time (minutes)')
plt.title('Total Predicted Time vs. State using TD(0) RL')
plt.grid(True)
plt.show()
```

**Output:**

```
  State  Elapsed Time (minutes)  Predicted Time to Go (minutes)  \
0     A                    0.50                            33.16
1     B                    4.86                            26.53
2     C                   10.65                            19.06
3     D                   21.52                            16.09
4     E                   29.77                             9.07
5     F                   39.77                             0.00

   Predicted Total Time (minutes)
0                           33.66
1                           31.40
2                           29.71
3                           37.61
4                           38.84
5                           39.77
```




**<u>Conclusion</u>:**

Successfully implemented the Driving Home problem using Temporal difference learning.

**<u>Post Lab Descriptive Questions</u>:**

**Q - List advantages and disadvantages of MC and TD approach.**

| Aspect | Monte Carlo (MC) | Temporal Difference (TD) |
|---|---|---|
| **Learning Style** | Learn from complete episodes (episodic). | Learn from each step (online/bootstrapping). |
| **Exploration** | Requires exploration across episodes. | Learns with each step, can adapt faster to exploration changes. |
| **Convergence** | Converges slowly, only after complete episodes. | Typically converges faster since updates happen more frequently. |

| | | |
|---|---|---|
| **Computation** | Requires the entire episode to be stored and computed, which can be memory-intensive. | Updates incrementally, lower memory requirements. |
| **Variance** | High variance due to reliance on complete episodes. | Lower variance as it updates after each step. |
| **Bias** | Unbiased estimates of the return since it's based on full episodes. | Can introduce bias due to bootstrapping from estimates. |
| **Applicability** | Only works for episodic tasks. | Works for both episodic and continuous tasks. |
| **Handling of Non-Markovian Environments** | Performs poorly in non-Markovian environments as it waits until the end of an episode. | Handles non-Markovian environments better with step-by-step updates. |

**Date:** **Signature of faculty in-charge**