COLLEGE CODE: 3108

COLLEGE NAME: JEPPIAAR ENGINEERING COLLEGE

DEPARTMENT: B.TECH AI&DS

STUDENT NM-ID: aut23aids052

ROLL NO: 310823243053

Completed the project named

AI-Based Route Memory and Self-Learning Navigation System

SUBMITTED BY,

NAME: SRI RAM R

TEAM MEMBER'S NAME:

NITHIN DEEPAK R

SANTHOSH S

SHANYU STARNESS P

UDHAYA KUMAR A

Phase 5: Project Demonstration & Documentation

Title: AI-Based Route Memory & Self-Learning Navigation System

---

Abstract:

The AI-Based Route Memory and Self-Learning Navigation System is a cutting-edge intelligent pathfinding solution designed to mimic and enhance human-like navigation abilities in complex, dynamic environments. This system leverages advanced reinforcement learning algorithms, specifically Q-learning, to enable autonomous agents, such as robots, unmanned vehicles, or smart mobility devices, to autonomously discover, remember, and optimize routes within grid-based environments. Unlike conventional pathfinding methods that rely on static maps and pre-defined paths, this system adapts in real-time by learning from its past experiences and environmental changes, thereby offering greater flexibility and efficiency in navigation tasks.

At the heart of this system lies a sophisticated environmental memory module that enables the agent to store and recall information about previously explored routes, obstacles, and successful paths. This memory function reduces redundant exploration and significantly speeds up navigation by allowing the agent to build upon past knowledge rather than relearning the environment from scratch. When the agent encounters changes such as new obstacles or blocked pathways, it dynamically recalculates alternative routes through continuous learning, demonstrating resilience and adaptability.

The system's learning mechanism is driven by Q-learning, a model-free reinforcement learning algorithm where the agent iteratively updates its policy by maximizing cumulative rewards associated with reaching target destinations efficiently. This iterative process balances exploration of new routes and exploitation of known efficient paths, helping the agent converge on optimal navigation strategies over time. Such capabilities make the system especially suitable for applications in robotics, autonomous vehicles, warehouse automation, and urban mobility solutions where environmental unpredictability and efficiency are critical factors.

To enhance transparency and facilitate user interaction, the system includes a real-time visual grid rendering interface that displays the environment, agent position, obstacles, and learned routes. This visualization not only aids developers in debugging and fine-tuning the navigation algorithm but also provides an intuitive way for users to monitor the agent's decision-making process and route optimization progress.

The current phase of the project emphasizes comprehensive demonstration and validation of the system's core functionalities. It highlights the agent's ability to learn and remember routes, adjust dynamically to changing environments, and visualize its navigation process effectively. Performance analysis and simulation walkthroughs offer insight into the efficiency and scalability of the solution, confirming its potential for deployment in real-world autonomous systems.

In summary, the AI-Based Route Memory and Self-Learning Navigation System represents a significant advancement in intelligent navigation technology. By combining reinforcement learning with environmental memory and visualization, the system achieves a high level of autonomy, adaptability, and efficiency, paving the way for smarter, more resilient navigation in a wide range of applications. This project not only demonstrates technical innovation but also sets the foundation for future enhancements, including multi-agent collaboration, real-time sensory integration, and expansion into more complex, unstructured environments.

INDEX

1. Project Demonstration

Overview:

This demonstration showcases the Q-learning-based route memory system in action, covering features like path learning, memory persistence, and real-time obstacle handling.

Demonstration Details:

- **Simulation Walkthrough:** Agent navigates a grid/maze from start to goal using a learned Q-table.
- **Learning Visualization:** Real-time updates of paths, obstacles, and learning progression.
- **Q-table Memory:** Demonstration of loading/saving learned knowledge, reusing it without retraining.
- **Dynamic Changes:** Real-time maze updates to simulate blocked paths and the agent's re-learning.
- **Performance Metrics:** Number of steps, episodes to convergence, reward optimization.

Outcome:

The system successfully learns paths, adapts to changes, and reuses memory for optimal navigation. The Q-table ensures knowledge persistence, making it efficient for scalable applications.

---

2. Project Documentation

Overview:

This section explains the core modules, system flow, and architecture used in developing the AI-based route memory system.

Documentation Sections:

System Architecture:

- Q-learning Agent module
- Environment/Grid module
- Memory Manager (save/load Q-table)
- Visualizer using matplotlib

Code Documentation:

- q_learning_agent.py: Core Q-learning logic
- environment.py: Maze/grid definition and obstacle logic
- visualizer.py: Real-time plot of grid and agent
- main.py: Simulation runner and interaction loop

User Guide:

- Run all Python files using Visual Studio with Python extensions

- Start with main.py to visualize the learning process
- Modify the grid in environment.py to test different layouts

Administrator Guide:

- Update Q-learning parameters (alpha, gamma, epsilon) in q_learning_agent.py
- Save/load Q-table with pickle for memory testing
- Troubleshoot rendering or numpy/matplotlib errors by reinstalling libraries

Testing Reports:

- Tested on multiple grid sizes (5x5, 7x7, 10x10)
- Learned within 500–1000 episodes depending on complexity
- Reward graph shows convergence over time

Outcome:

The system's components are modular, maintainable, and fully documented for future upgrades, real-world testing, or hardware integration.

---

3. Feedback and Final Adjustments

Overview:

Feedback was gathered during the Phase 4–5 transition to guide improvements.

Steps:

- **Feedback Collection:** From mentors and peers after testing different obstacle scenarios
- **Refinements Made:**
    - Improved Q-table visualization
    - Reduced agent wandering with smarter epsilon decay
    - Added restart and replay functions

Outcome:

Agent now adapts faster, performs smoother in re-learning situations, and provides better visual feedback.

---

4. Final Project Report Submission

Overview:

Summarizes the complete project cycle from ideation to execution.

Report Sections:

- **Executive Summary:** Intelligent navigation system using reinforcement learning
- **Phase Breakdown:**
    - Phase 1: Conceptual Design
    - Phase 2: Environment Setup
    - Phase 3: Basic Q-learning Agent
    - Phase 4: Visualization & Dynamic Learning
    - Phase 5: Memory Optimization & Demonstration
- **Challenges & Solutions:**
    - Path redundancy → Resolved with smarter epsilon decay
    - Stuck agents → Added path-checks and feedback
    - Grid complexity → Optimized rendering & logic
- **Outcomes:**
    - Fully working route-learning AI
    - Dynamic adaptation and memory persistence

---

5. Project Handover and Future Works

Overview:

Outlines the next steps if the system is to be scaled up or enhanced.

Handover Details:

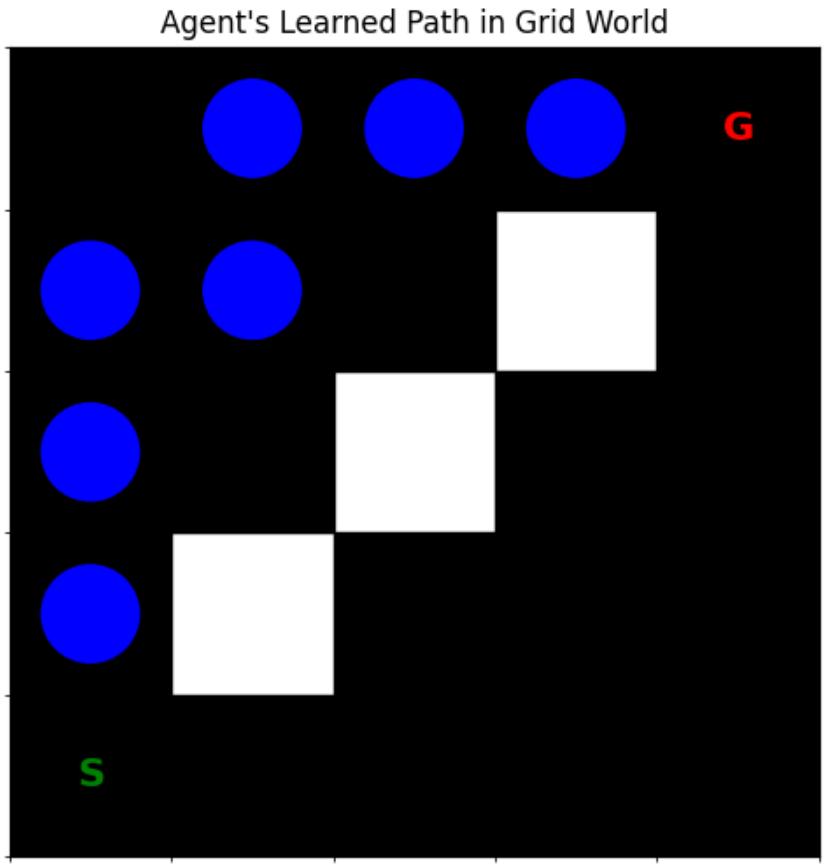
- **Next Steps:**
    - Integrate with physical robots (Raspberry Pi + motor driver)
    - Switch to Deep Q-Learning (DQN) with neural networks
    - Use camera input or GPS for real-time mapping
    - Export map files in JSON for real-world integration

Outcome:

A self-learning, memory-enabled, AI-driven navigation system ready for both simulation and robotic deployment.

6. Screenshots to Include:

- **Agent path on grid (start to goal)**



- **Q-table saved and reloaded**

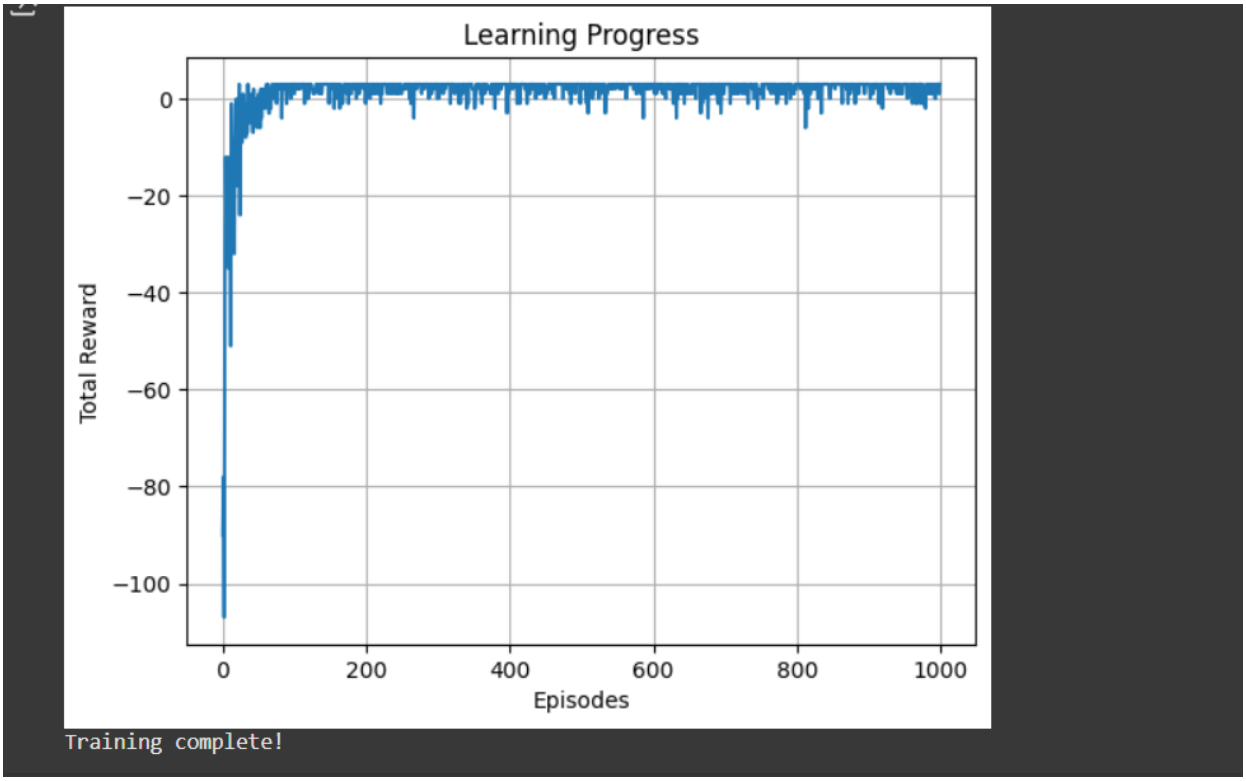
```
[13] import numpy as np
     import random

     class QLearningAgent:
         def __init__(self, n_states, n_actions, alpha=0.1, gamma=0.9, epsilon=0.1):
             self.q_table = np.zeros((n_states, n_actions))
             self.alpha = alpha
             self.gamma = gamma
             self.epsilon = epsilon
             self.n_actions = n_actions

         def act(self, state):
             if np.random.rand() < self.epsilon:
                 return random.randint(0, self.n_actions - 1)
             return np.argmax(self.q_table[state])

         def learn(self, state, action, reward, next_state):
             predict = self.q_table[state][action]
             target = reward + self.gamma * np.max(self.q_table[next_state])
             self.q_table[state][action] += self.alpha * (target - predict)
```

- **Reward graph**



- **Full code snippets running in Visual Studio**

```python
class GridEnvironment:
    def __init__(self, size, start, goal, obstacles=[]):
        self.size = size
        self.start = start
        self.goal = goal
        self.obstacles = obstacles
        self.reset()

    def reset(self):
        self.position = self.start
        return self._state_to_index(self.position)

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

        x = max(0, min(self.size[0] - 1, x))
        y = max(0, min(self.size[1] - 1, y))
        next_position = (x, y)

        if next_position in self.obstacles:
            next_position = self.position  # Blocked

        self.position = next_position
        done = (self.position == self.goal)
        reward = 10 if done else -1
```

```python
[7] import matplotlib.pyplot as plt
    import pickle

    def plot_rewards(rewards):
        plt.plot(rewards)
        plt.xlabel("Episodes")
        plt.ylabel("Total Reward")
        plt.title("Learning Progress")
        plt.grid(True)
        plt.show()

    def save_q_table(agent, filename="q_table.pkl"):
        with open(filename, "wb") as f:
            pickle.dump(agent.q_table, f)

    def load_q_table(agent, filename="q_table.pkl"):
        with open(filename, "rb") as f:
            agent.q_table = pickle.load(f)

    # Setup
    grid_size = (5, 5)
    start = (0, 0)
    goal = (4, 4)
    obstacles = [(1, 1), (2, 2), (3, 3)]
```

```python
env = GridEnvironment(grid_size, start, goal, obstacles)
agent = QLearningAgent(n_states=grid_size[0]*grid_size[1], n_actions=4)

rewards = []
episodes = 1000
for ep in range(episodes):
    state = env.reset()
    total_reward = 0
    done = False
    while not done:
        action = agent.act(state)
        next_state, reward, done = env.step(action)
        agent.learn(state, action, reward, next_state)
        state = next_state
        total_reward += reward
    rewards.append(total_reward)
    if ep % 100 == 0:
        print(f"Episode {ep}, Total Reward: {total_reward}")

save_q_table(agent)
plot_rewards(rewards)
print("Training complete!")
```

Tools Required:

- Python 3.10+
- Visual Studio (with Python extension)
- Libraries: numpy, matplotlib, pickle