1859

Politecnico di Torino

Resilient Task Sequence Planning for Industrial Mobile Robots

MECHATRONIC ENGINEERING
Software Technologies for Automation

Master's Degree Thesis
Academic Year 2023-2024

Supervisor:
Prof. DARIO ANTONELLI

Candidate:
HUSSEIN ZEIN ALDEEN

Table of Contents

List of Figures

List of Tables

Abstract

Robots play a crucial role in modern industrial environments, where they perform various tasks such as transporting components, assembling products, and conducting quality checks. Ensuring these robots operate efficiently in uncertain and dynamic conditions remains a significant challenge. This thesis explores the use of **Q-learning**, a type of reinforcement learning, to develop a robust decision-making framework for robots operating in such environments. The proposed system models the robot's tasks using a **Markov Decision Process (MDP)**, where the robot navigates through multiple states, such as charging_station, warehouse_kitting, assembly_station, and EOL (End of Line), and performs actions like moving between locations, assembling parts, and checking product quality.

A key feature of this project is the incorporation of **stochastic transitions**, which reflect real-world uncertainties, such as component mismatches or system failures. This allows the model to capture unpredictable outcomes, forcing the robot to learn optimal strategies despite the presence of probabilistic events. The learning process is governed by a **Q-table** that is updated iteratively as the robot interacts with its environment, receiving rewards for positive actions and penalties for undesirable ones. For example, the robot earns rewards for retrieving parts efficiently, while penalties are imposed for actions like waiting unnecessarily at idle stations or transporting faulty products.

This research employs an **epsilon-greedy policy** to balance exploration and exploitation, ensuring the robot explores various strategies before converging on the optimal policy. Two sets of hyperparameters are tested: the first set emphasizes long-term rewards by using a high discount factor (gamma = 0.9), while the second set prioritizes immediate outcomes with a lower discount factor (gamma = 0.3). The training process is evaluated over multiple episodes, with cumulative rewards tracked to monitor the system's learning progress.

The results show that the Q-learning algorithm successfully guides the robot toward an efficient policy, with the first hyperparameter set delivering superior performance by

favoring decisions that maximize rewards over time. The study demonstrates that reinforcement learning offers a powerful framework for addressing complex decision-making problems in robotics, especially in environments with uncertain outcomes.

This thesis concludes by suggesting avenues for future work, such as integrating **Deep Q-Networks (DQN)** to handle larger and more complex environments, deploying multi-agent systems to enable collaboration between multiple robots, and testing the model in real-world industrial settings. These improvements could further enhance the robot's ability to adapt and perform efficiently in dynamic environment.

Chapter 1: Introduction

1.1 Background

In today's rapidly evolving industrial landscape, robotics plays a crucial role in enhancing efficiency, reducing costs, and improving production quality. From warehouses to assembly lines, robots perform essential tasks such as transporting components, assembling products, and conducting quality inspections. However, as the complexity of these environments increases, so does the challenge of ensuring that robots operate smoothly and make intelligent decisions without constant human intervention. Traditional rule-based systems often struggle to adapt to unpredictable situations, such as sudden equipment failures or mismatches in assembly parts. This limitation motivates the search for more flexible, adaptive solutions, one of which is reinforcement learning (RL) [1].

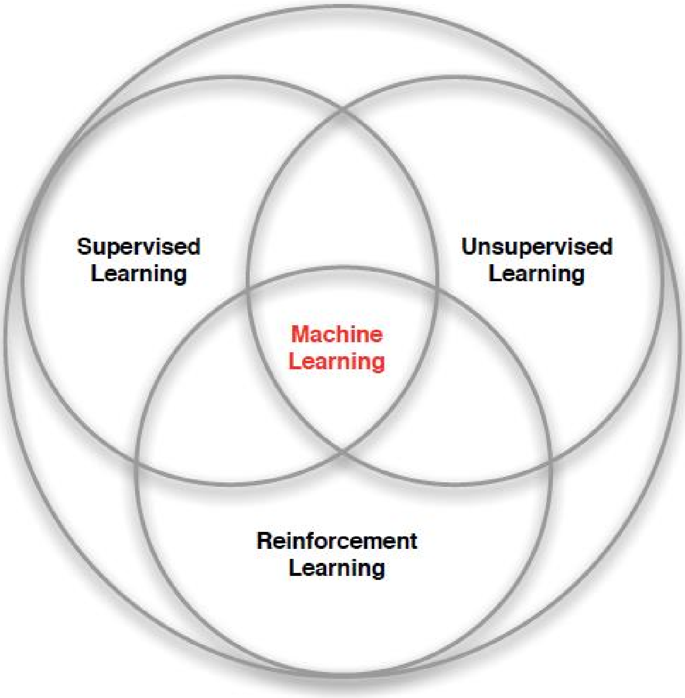


Figure 1 Branches of Machine Learning [2]

Reinforcement learning is a type of machine learning that allows an agent (in this case, a robot) to learn optimal behavior by interacting with its environment. Unlike supervised learning, which requires labeled data, reinforcement learning enables the agent to explore actions, receive feedback in the form of rewards or penalties, and gradually improve its performance. **Q-learning**, a widely used RL algorithm, is particularly effective for environments modeled as **Markov Decision Processes (MDPs)**, where decisions depend on both the current state and the future rewards associated with different actions [3].

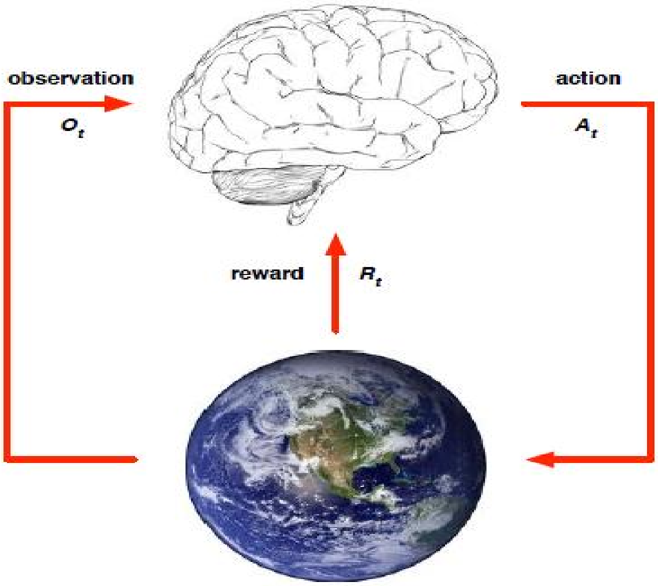


Figure 2 Agent and Environment [2]

The focus of this thesis is to explore how Q-learning can be applied to train a robot to navigate through a simulated industrial environment and perform essential tasks efficiently. The robot operates across multiple states such as charging_station, warehouse_kitting, and EOL (End of Line) and must select appropriate actions to achieve its objectives. The robot's actions include moving between locations, retrieving and assembling components, and conducting quality inspections. Importantly, this environment introduces **stochastic transitions**, meaning that some state changes are unpredictable, reflecting real-world uncertainties such as equipment malfunctions or incorrect part deliveries.

1.2 Problem Statement

Robotic systems often encounter situations where rigid, pre-programmed behavior limits their ability to respond effectively to dynamic environments. In industrial settings, conditions can change unpredictably due to sensor errors, component mismatches, or equipment failures. Current systems that rely on fixed decision rules struggle to adapt to such challenges. As a result, there is a need for robots that can **learn from experience** and develop policies that allow them to make **context-appropriate decisions** in uncertain environments.

The problem addressed by this thesis is: **How can Q-learning be applied to develop a robust decision-making policy for a robot operating in a dynamic, stochastic industrial environment?**

This problem is crucial because robots need to operate autonomously while handling unexpected events efficiently. For example, when a mismatch occurs during assembly, the robot should decide whether to replace the part immediately or recheck the components. Similarly, in case of an equipment failure, the robot must determine whether to continue working or switch to maintenance tasks. Without a flexible, data-driven policy, such decisions become difficult to manage effectively.

1.3 Thesis Objectives

The main objectives of this research focus on developing and implementing a reinforcement learning framework for navigating mobile robots in industrial settings. This involves enabling robots to identify free positions, avoid human-occupied spaces, and respond to potential worst-case scenarios. The specific objectives are:

- **Objective 1**: **Develop a Resilient RL-based Navigation Model** To create an RL model that allows mobile robots to identify and navigate to free positions while avoiding areas occupied by humans or other obstructions. This includes simulating environments where the robot learns optimal paths and behaviors through rewards for successful navigation and penalties for undesirable actions.
- **Objective 2**: **Integrate Worst-Case Scenario Training** Implement worst-case scenario simulations, including unexpected obstacles, operational delays, and system malfunctions, to train robots for enhanced adaptability. These scenarios aim to improve the robot's ability to respond effectively to challenging conditions, helping it to recover or adapt to maintain task efficiency and safety.
- **Objective 3**: **Optimize and Test RL Algorithms in MDP Environments** Explore various RL algorithms, including Q-learning, SARSA, and Dyna-Q, to identify the most suitable approach for achieving stable, efficient learning within an MDP framework. The focus will be on optimizing algorithm parameters to achieve high convergence rates, stability, and cumulative reward maximization.
- **Objective 4**: **Establish a Robust Framework for Continuous Learning and Adaptation**
  To build an RL framework that ensures continuous adaptation in dynamic industrial settings. This objective emphasizes designing a system that allows robots to continually update and refine their strategies as they encounter new challenges.

1.4 Research Question and Hypothesis

Based on the problem statement, the research question driving this thesis is:
How can Q-learning be used to train a robot to navigate and perform tasks optimally in a dynamic, stochastic industrial environment?

To address this question, the following **hypothesis** is proposed:

- **H1:** A Q-learning-based policy will allow the robot to improve its performance over time, selecting actions that maximize cumulative rewards in both predictable and unpredictable scenarios.
- **H2:** A higher discount factor (gamma) will result in better long-term performance, while a lower discount factor will prioritize short-term outcomes but may reduce overall efficiency.

1.5 Scope of the Study

This research focuses on the development, simulation, and evaluation of a reinforcement learning (RL)-based framework within a controlled yet dynamic environment that emulates an industrial setup. Specifically, it addresses the implementation of a Q-learning-based policy for a mobile robot performing task sequences across various operational states. These states include charging_station, warehouse_kitting, assembly_station, quality_control, and end-of-line (EOL), representing typical stages of an industrial workflow.

Each state in this environment requires a set of actions that the robot can perform, such as retrieving parts, moving between designated locations, or conducting quality checks on assembled products. The robot's actions are designed to simulate the decision-making processes required for adaptive task execution. The environment incorporates **stochastic transitions** to simulate real-world uncertainties, including unexpected part mismatches, component delays, equipment malfunctions, and other disruptions. These stochastic elements introduce variability, making it possible to assess the Q-learning policy's resilience and adaptability to real-world operational conditions, even though it is contained within a simulated environment.

1.5.1 Simulated Environment and Constraints

The industrial environment is entirely simulated, offering a virtual testing ground that allows for control over various parameters. This setup provides the flexibility to conduct numerous trials without the costs, logistical challenges, and potential risks associated with physical deployment. By controlling variables in a simulated setting, the study is able to **explore the Q-learning algorithm's behavior across a range of scenarios** that would be costly and complex to replicate in a real-world setup.

While this approach allows for extensive exploration, it has limitations. The absence of real-world sensory data and mechanical feedback means that certain physical factors (e.g., sensor noise, mechanical wear, spatial constraints) are not accounted for. Consequently, the trained policy’s performance may differ when deployed in a real-world environment. Future work may bridge this gap by testing the policy on actual robots, incorporating feedback from physical sensors, and accounting for physical constraints.

1.5.2 Focus on Q-Learning and Hyperparameter Tuning

This research is restricted to **Q-learning**, an effective yet relatively straightforward RL algorithm that is suitable for environments modeled as Markov Decision Processes (MDPs). Q-learning’s suitability lies in its ability to **learn an optimal policy through reward maximization** without requiring a model of the environment, making it a practical choice for the simulated setup. In this study, Q-learning is extensively tested and optimized through hyperparameter tuning, with particular focus on learning rate, discount factor, and exploration-exploitation strategies to achieve a policy that maximizes cumulative rewards in both regular and worst-case scenarios.

The research, however, does not extend to other RL algorithms, such as **advanced deep learning approaches like Deep Q-Networks (DQN)**, which could potentially enhance learning efficiency and adaptability in larger or more complex environments. While the current study provides a solid foundation with Q-learning, future research could involve benchmarking other RL methods, comparing their performance, and potentially integrating these methods for hybrid or multi-algorithmic strategies that enhance task adaptability. Although physical deployment and real-time testing are beyond the current scope, this research provides a framework that can serve as a stepping stone for **real-world application and further algorithmic development**. The insights gained from this study could be instrumental in adapting the trained policy for an actual industrial robot, where physical complexities and real-time data processing are necessary. Moreover, this study’s focus on Q-learning-based policy optimization could serve as a reference for future research aiming to integrate **advanced reinforcement learning techniques, such as Deep Q-Networks (DQN) or Policy Gradient Methods**, to enable more efficient learning and decision-making in highly variable or larger-scale industrial environments. The ultimate goal of this research is to create a scalable and adaptable RL-based policy for industrial task automation. By laying the groundwork with a simulated Q-learning framework, this study sets the stage for future work involving:

- **Real-world deployment** and validation of the trained policy.
- **Integration of physical sensors and actuators** for comprehensive feedback and decision-making.
- **Exploration of multi-agent systems** where multiple robots collaborate within the industrial environment.
- **Incorporation of more sophisticated algorithms** capable of handling higher-dimensional state spaces, such as deep reinforcement learning techniques.

In summary, this study confines its scope to the development and simulation of a Q-learning policy within a virtual industrial environment, emphasizing task execution and adaptability in the presence of stochastic uncertainties. Future work can build on this foundation to explore more advanced RL algorithms and implement real-world testing, ultimately advancing the potential for adaptive, resilient robotic systems in dynamic industrial applications.

1.6 Significance of the Study

The ability to develop **adaptive, self-improving robotic systems** has significant implications for industries that rely heavily on automation. A robot capable of learning from its environment and making optimal decisions can reduce downtime, improve productivity, and lower operational costs. This study demonstrates how **reinforcement learning** can address the limitations of rule-based robotic systems by enabling robots to **adapt to uncertainties**. The findings of this research also contribute to the broader field of **robotics and machine learning**, offering insights into how Q-learning can be applied to real-world problems.

Chapter 2: Related Work

Various studies have concerned the use of reinforcement learning for robots to move safely and efficiently. There exist previous works on applications of MDP environment setups for training robots to navigate through space environments. Some of the exciting research has illustrated that, especially in the industrial setup, reinforcement learning can effectively solve dynamic and unpredictable environmental challenges. However, a need for robust solutions still exists. Research on **reinforcement learning-based navigation for mobile robots** has shown RL's potential for teaching robots to navigate autonomously while handling uncertainties and obstacles. A significant study on **SARSA (λ) and Q-learning(λ)** algorithms, conducted by Altuntaş and İmal, explored how RL algorithms perform in realistic navigation tasks where robots autonomously avoid obstacles and reach designated goals. Their study revealed that **Q-learning(λ) achieved faster learning rates**, quickly navigating to target locations in simpler environments. However, **SARSA (λ) outperformed Q-learning(λ) in complex scenarios**, indicating a more robust performance due to its on-policy nature, which considers the next action to adjust the Q-values more optimally. This adaptive quality of SARSA (λ) was particularly valuable in managing exploration-exploitation challenges, a central issue in RL navigation, by using **eligibility traces** to improve learning efficiency and fine-tune actions in real-time [4].

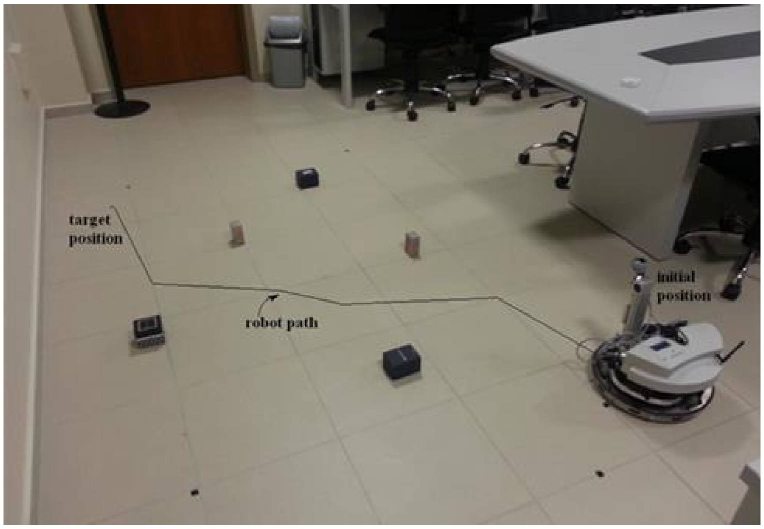


Figure 3 Robotino and Targets [4]

Another significant contribution to RL navigation is the **Robust Satisficing Markov Decision Process (RSMDP)** framework, which addresses uncertainties within traditional MDP models. Developed as an improvement over regular MDPs, **RSMDPs aim to balance performance and robustness** by introducing a user-defined return target, avoiding overly conservative or risky policies. This framework is particularly relevant for industrial setups, where robots frequently encounter unpredictable variables like part mismatches or sudden failures. The RSMDP framework includes a **first-order primal-dual algorithm** for efficient large-scale problem-solving, allowing it to outperform standard MDP approaches by maximizing returns more consistently under varied conditions [5].

Moreover, "Safe Reinforcement Learning for Human-Robot Collaboration" paper highlights the growing integration of robots into human-centric environments, combining robotic efficiency with human adaptability. Safety is a critical concern, particularly in dynamic scenarios like autonomous warehouses, where humans and robots share close physical spaces. Current ISO standards offer safety guidelines by restricting force and power but fall short in handling dynamic and unstructured environments. Reinforcement Learning (RL) has emerged as a powerful tool for optimizing robotic behavior in such settings. However, the simulation-to-reality gap and the potential for unsafe exploration during training remain significant challenges [6].

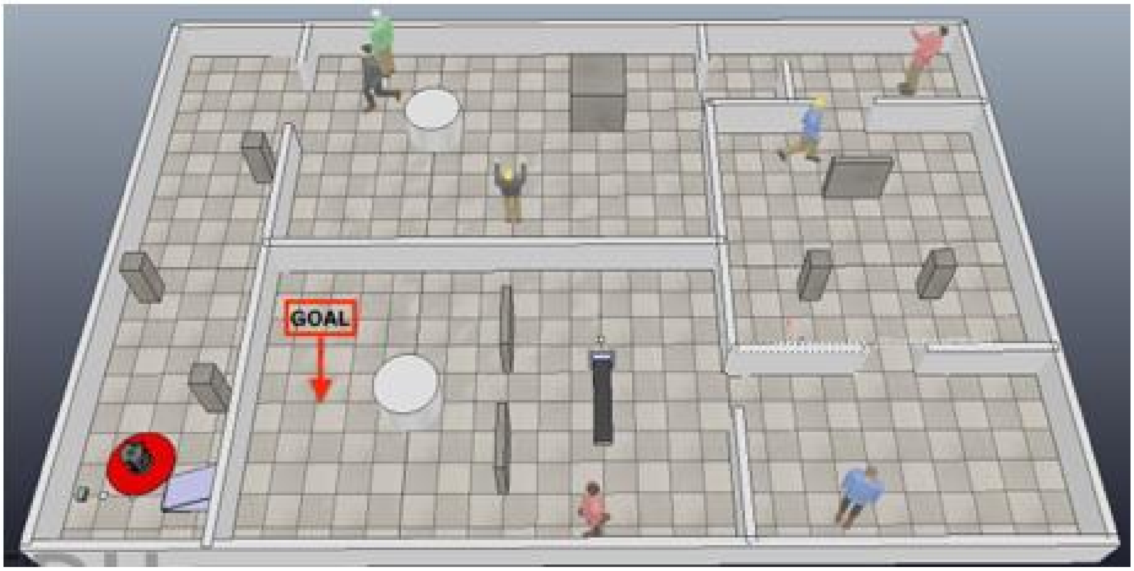


Figure 4 The robot has to navigate through the entire environment to reach the marked goal [6].

Safe Reinforcement Learning (Safe RL) addresses these issues by embedding safety measures into the learning process, either by modifying reward structures or incorporating external safety mechanisms like shielding. Shielding, in particular, has been effective in preventing unsafe actions during training and deployment, as seen in applications ranging from autonomous driving to robotic navigation in warehouses. While techniques like Linear Temporal Logic (LTL) and probabilistic models have demonstrated success in ensuring safety, they face challenges in scalability and efficiency trade-offs. This research builds on these advancements by integrating shielding and RL in a modular framework to improve safety and efficiency in dynamic, real-world HRC applications [6].

Furthermore, Lee and Jeong's 2021 paper on mobile robot path optimization using reinforcement learning (RL) in warehouse environments introduces a novel application of Q-Learning, an RL algorithm, for enhancing the pathfinding capabilities of autonomous robots. As path planning in dynamic and complex environments like warehouses poses significant challenges, traditional methods struggle with real-time adaptation to obstacles and changes. Reinforcement learning offers a solution by enabling robots to learn and optimize their behavior over time based on rewards. This paper builds on previous RL applications in robotics, particularly in logistics and warehouse settings, contributing to ongoing efforts to improve autonomous navigation in unpredictable environments [7].

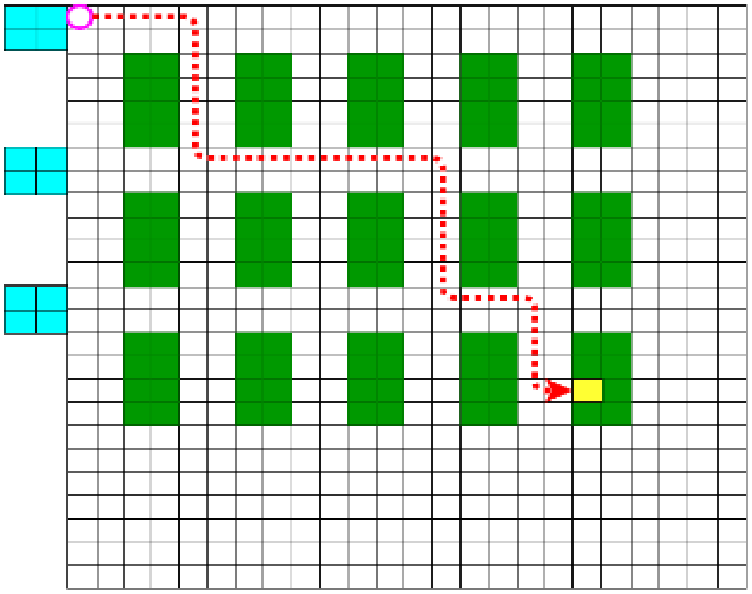
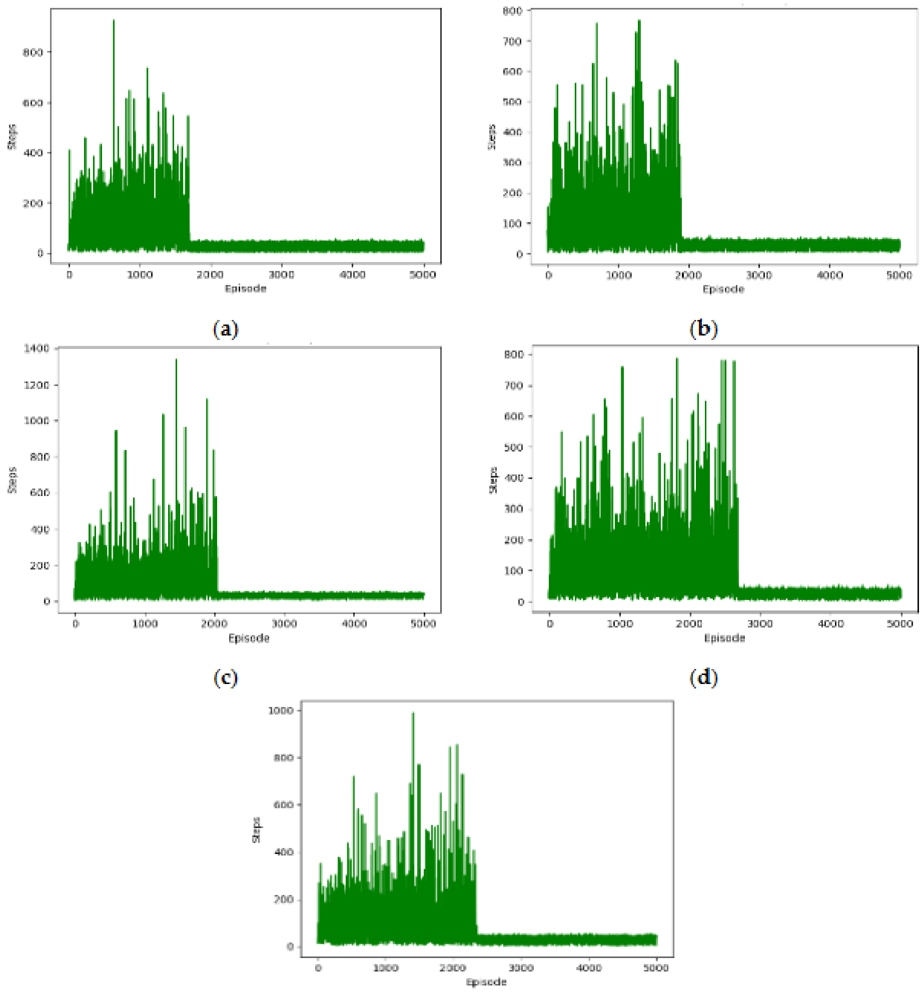


Figure 5 Target position of simulation experiment [7]



(a)

(b)

(c)

(d)

(e)

Figure 6 Path optimization learning process [7]

The research by Lee and Jeong is part of a broader trend exploring RL's potential in warehouse robotics, following works such as those by Yu et al. (2020) and Nguyen et al. (2021), which also focus on path optimization in dynamic environments. While RL shows great promise, challenges like large state-action spaces, slow learning times, and the need for real-time decision-making in complex, multi-agent settings remain. Future research may focus on deep reinforcement learning or multi-agent systems to further optimize robot performance in warehouses. Lee and Jeong's work represents an important step toward addressing these challenges, illustrating the value of RL in improving robot efficiency and adaptability in real-world applications [7].

Reinforcement learning (RL) has emerged as a transformative approach for addressing path planning challenges in mobile robots, especially in environments where prior knowledge of the terrain is minimal or nonexistent. Sichkar's work focuses on two popular RL algorithms—Q-Learning and its modification, Sarsa—comparing their effectiveness in global path planning within a virtual obstacle-laden environment. Q-Learning's reliance on maximizing the reward for each state-action pair enables faster convergence and efficient path optimization, albeit at the cost of increased risk in uncertain environments. In contrast, Sarsa prioritizes safer trajectories by modifying the reward update mechanism, resulting in more conservative path selections but requiring longer learning durations [8].

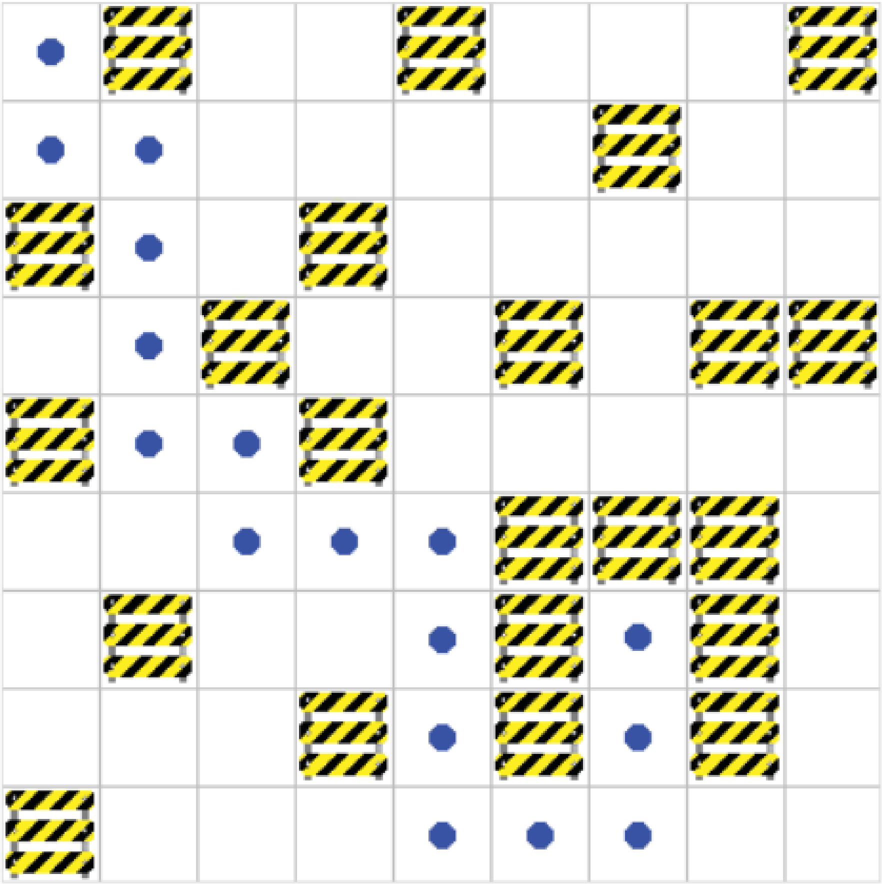


Figure 7 Virtual environment with obstacle and found path [8]

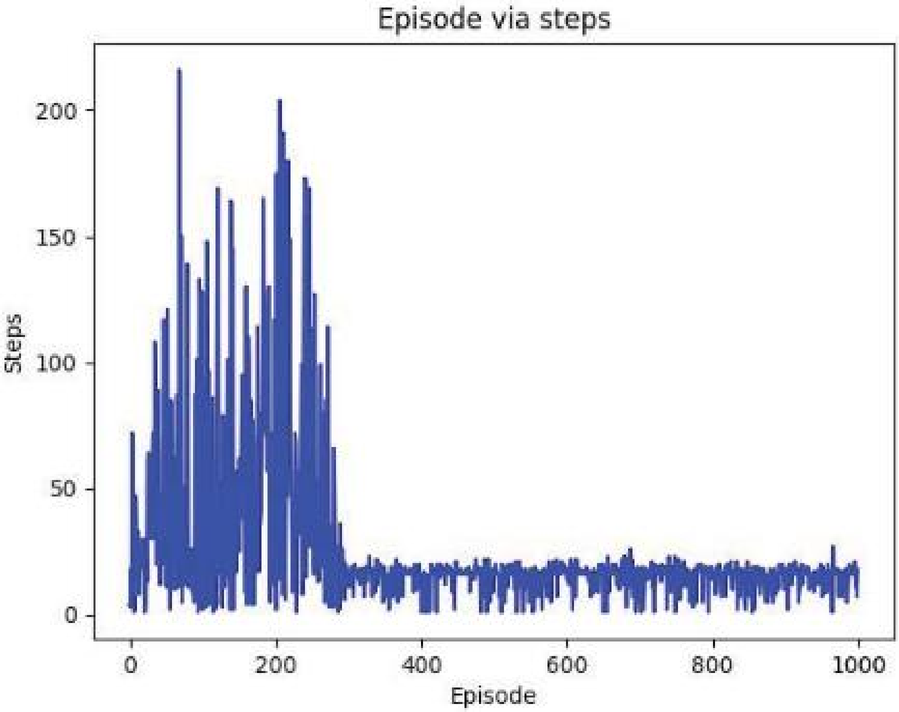


Figure 8 Episode via steps for Q-learning algorithm [8]

The study also highlights techniques to enhance RL performance, including Q-value approximation methods like discretization, radial basis functions (RBF), and clustering. These methods address scalability issues, enabling RL algorithms to handle larger and more complex state spaces. The experimental results demonstrate the trade-offs between the algorithms: while Q-Learning achieves faster convergence with shorter paths, Sarsa provides safer navigation, avoiding high-risk regions such as simulated cliffs. This work contributes valuable insights into optimizing RL-based path planning for various real-world applications, balancing speed and safety based on operational priorities [8].

Task-Motion Planning (TMP) is essential for integrating high-level task planning with low-level motion planning in robotic systems. TMP algorithms traditionally focus on creating efficient task plans in discrete spaces and refining them into executable motion plans in continuous spaces. Jiang et al. propose an innovative framework, TMP-RL, combining TMP with reinforcement learning (RL) to enhance adaptability in dynamic and uncertain environments. TMP-RL employs a dual-loop structure: the inner loop generates feasible task-motion plans using symbolic and motion planning, while the outer loop refines these plans through RL based on execution feedback. This integration enables robots to adapt to real-world changes, such as varying human interaction, environmental dynamics, and task complexities, ensuring robust performance over long-term operations [9].

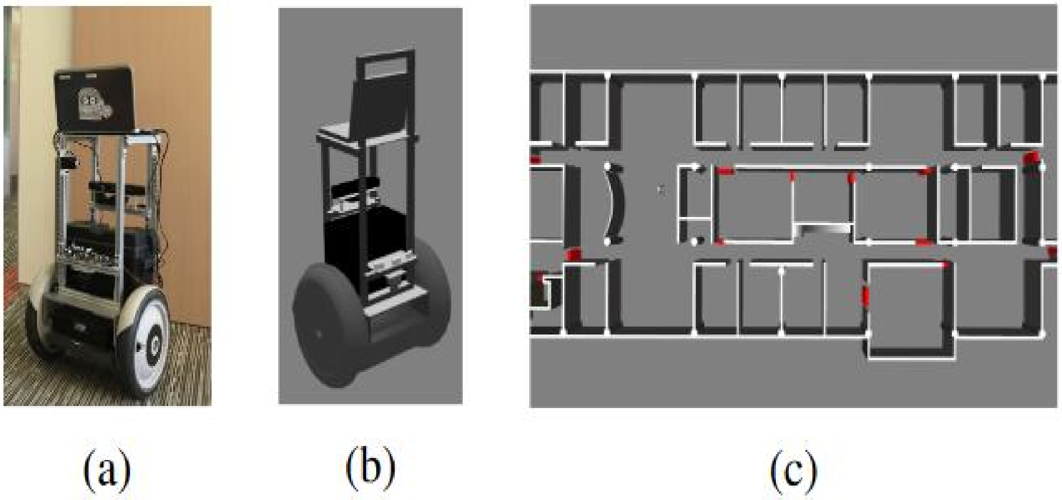


Figure 9 (a) a BWIBot (b) a simulated BWIBot (c) simulation environment [9].

Compared to state-of-the-art methods like PETLON (pure TMP) and PEORL (task planning with RL), TMP-RL offers superior adaptability and convergence speed. The use of RL in the outer loop enhances task plan quality iteratively by leveraging environmental rewards, reducing reliance on extensive motion planning computations. TMP-RL demonstrated significant improvements in task efficiency and adaptability in simulated and real-world tests with a service robot. The framework also supports knowledge transfer between tasks, allowing robots to generalize learned behaviors to new scenarios. This research advances the field of robotics by seamlessly integrating planning and learning, making it a promising solution for adaptable and intelligent robotic systems [9].

Moreover, the integration of Logistics 4.0 and Industry 5.0 technologies has transformed intralogistics processes, focusing on enhancing automation, efficiency, and collaboration. Pizoń et al. investigate the role of Autonomous Mobile Robots (AMRs) in optimizing intralogistics for automotive remanufacturing—a sector vital for sustainable manufacturing and circular economy practices. Remanufacturing processes involve restoring used automotive parts to like-new condition, and efficient auxiliary logistics, such as tool and material delivery, are critical to maintaining production flow. The study highlights a gap in best practices for applying advanced logistics technologies to ancillary processes, emphasizing the role of AMRs in addressing challenges like labor shortages, dynamic task requirements, and workplace efficiency [10].

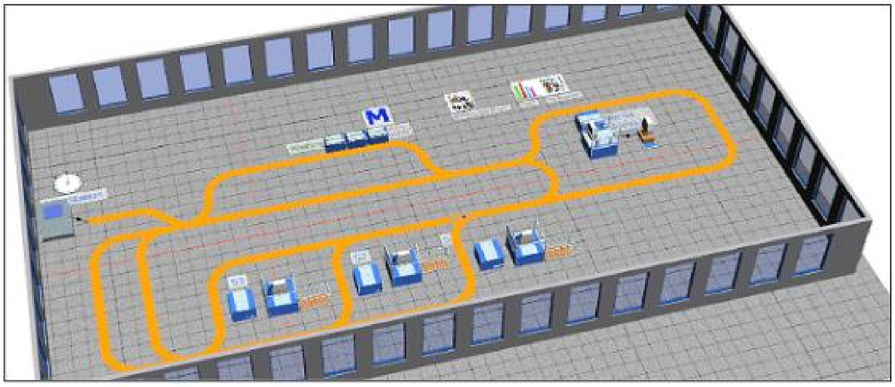


Figure 10 Intralogistics auxiliary processes – model 3D [10]

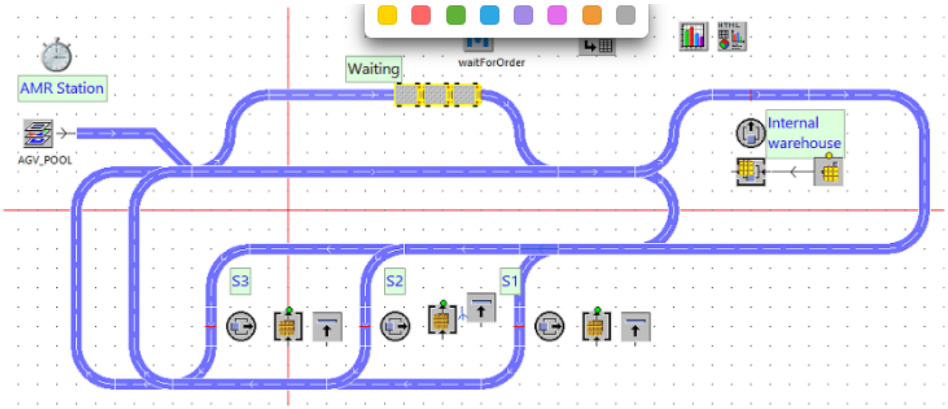


Figure 11 Intralogistics auxiliary processes – simulation model [10]

The authors adopt lean management principles, including Kaizen and 5S methodologies, to streamline intralogistics in collaboration with AMR deployment. Kaizen facilitates continuous improvement through iterative adjustments, while 5S ensures workplace organization and standardization, creating conditions conducive to seamless robot-human interaction. Simulation-based evaluations demonstrate the potential of AMRs in reducing disruptions in material handling. However, the study reveals diminishing returns when scaling AMR numbers, emphasizing the importance of strategic deployment tailored to specific operational needs. This work underscores the importance of balancing technological integration with lean principles to enhance productivity and maintain cost efficiency in industrial logistics [10].

Moreover, collaborative robots (cobots) have been increasingly integrated into industrial environments due to their adaptability and ability to safely coexist with humans in shared workspaces. Gomes et al. explore the application of reinforcement learning (RL) for enhancing cobot performance in pick-and-place tasks. The research addresses the limitations of traditional programming methods in dynamic environments, emphasizing the role of RL in enabling cobots to adapt to unforeseen changes. The study utilizes deep Q-learning integrated with convolutional neural networks (CNNs) for processing color and depth images, allowing cobots to grasp objects outside their training set effectively [11].

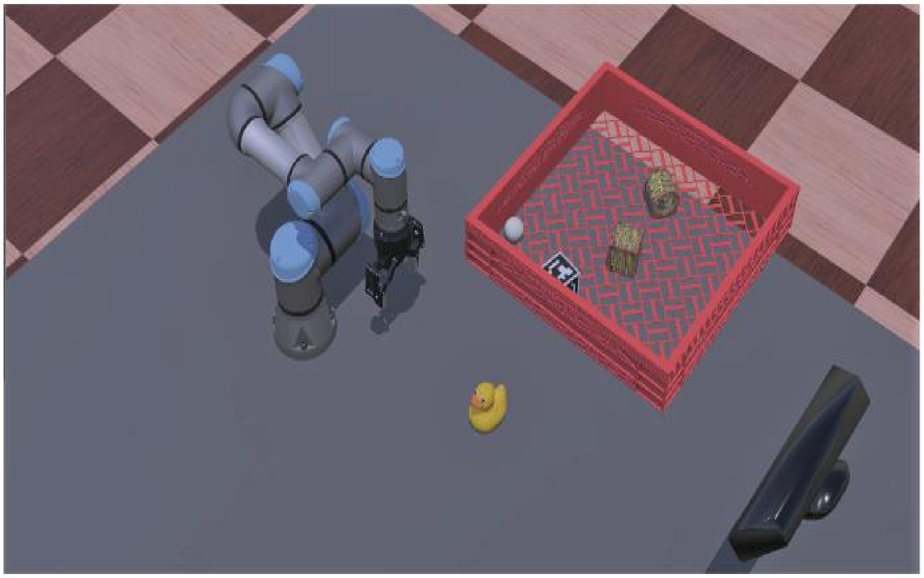


Figure 12 The virtual environment built on Webots: it consists of a table, a UR3e cobot (left) with a two-finger gripper, an RGBD camera (right) and the objects to be manipulated by the robot [11].

The work builds on existing literature that highlights the potential of deep learning in robotics, particularly for tasks involving vision-based object manipulation. It compares the efficacy of pre-trained CNN models, such as MobileNet, DenseNet, ResNext, and MNASNet, for feature extraction in RL frameworks. Simulation and real-world testing validate the system, achieving a success rate of 89.9% in grasping unfamiliar objects with MobileNet. This research aligns with advancements in combining RL and computer vision to enhance cobot flexibility, offering valuable insights for applications in environments requiring high adaptability and precision [11].

Chapter 3: Contribution

This thesis presents a complete framework for using reinforcement learning (RL) in mobile robot navigation within industrial settings. The framework focuses on both safety and efficiency, tackling the challenges of navigating areas where humans and machines share space. The main contributions of this research are detailed below:

- **Creation of a New Reinforcement Learning Model for Safe Navigation:** At the heart of this research is a new reinforcement learning model designed specifically for mobile robot navigation. This model allows robots to find their way to accessible areas while avoiding spaces occupied by humans. The innovation comes from using a Q-learning algorithm that has a special reward system. This system penalizes the robot for getting too close to people and rewards it for taking safer routes. By including safety in the reward system, the model improves the robot's decision-making process, allowing it to prioritize human safety while navigating effectively. This advancement is important for the robotics field, as it addresses the crucial issue of operating safely in environments with people.
- **Testing Through Worst-Case Scenario Simulations:** To ensure the reliability and adaptability of the proposed navigation framework, several tough simulations were performed. These tests were carefully designed to challenge the robot’s abilities in extreme situations, such as unexpected obstacles, sudden changes in the environment, narrow spaces, and equipment failures. By simulating these difficult conditions, we identified possible failure points in the robot’s navigation strategy. The results from these simulations helped refine the Q-learning algorithms, allowing us to add strong optimization methods. This back-and-forth process not only improved the robot’s adaptability but also enhanced its safety features, ensuring it can effectively handle real-world challenges.

- **Optimization of Advanced Reinforcement Learning Algorithms Using Python:**
  The reinforcement learning algorithms were developed using Python, a flexible programming language ideal for machine learning tasks. These algorithms were fine-tuned using popular libraries such as NumPy, SciPy, Matplotlib, and Pandas. These tools helped create complex Q-learning algorithms and other advanced RL methods. Tuning hyperparameters was crucial in this process, focusing on important settings like learning rate and how the robot balances exploration and exploitation. By carefully adjusting these factors, we ensured the algorithms worked efficiently and could handle the worst-case scenarios the robot might face. This robustness is vital for the reliable performance of mobile robots in changing and potentially dangerous environments.

In summary, this thesis significantly contributes to mobile robotics by providing a new framework that enhances both safety and efficiency in robot navigation in industrial areas. The combination of advanced reinforcement learning techniques, thorough testing through simulations, and practical implementation shows a complete approach to addressing real-world challenges in robot-human interactions.

Chapter 4: Hypothesis and Expected Impact

4.1 Hypothesis

This research posits that a **Q-learning-based policy for task execution in a simulated industrial environment will demonstrate measurable improvements in adaptability and efficiency**, particularly in the face of stochastic transitions and unexpected disruptions. The key hypotheses are as follows:

1. **Q-learning can train the robot to navigate an industrial task environment more efficiently than rule-based systems** by learning from rewards and penalties rather than relying on predefined instructions. The model's ability to adjust based on cumulative rewards will allow for a more flexible response to unpredictable situations, such as part mismatches, equipment failures, or sudden changes in task priority.
2. **Hyperparameter optimization (learning rate, discount factor, exploration-exploitation balance) will significantly impact the efficiency and stability of the learning process**, with optimized parameters leading to faster convergence and a more resilient task execution policy. This hypothesis suggests that fine-tuning these parameters will yield a policy that balances exploration with reliable task execution in industrial settings.
3. **Even in a simulated environment, a Q-learning model will demonstrate the potential for scalable learning**, showing promise for future deployment on physical robots. This hypothesis is based on the expectation that, by training in a diverse set of scenarios, the Q-learning policy can generalize well enough to serve as a foundation for real-world adaptation and further reinforcement learning methods.

4.2 Expected Impact

The anticipated impact of this research extends to several practical and theoretical areas within robotics, industrial automation, and reinforcement learning. The expected impacts are as follows:

1. **Enhanced Task Adaptability in Industrial Robotics:** The Q-learning-based policy is expected to enable robots to respond effectively to variations in an industrial environment, such as component availability or operational delays. This adaptability is crucial for industries where conditions are unpredictable, and robots need to adjust autonomously. By incorporating RL, industrial robots can move closer to achieving flexible, real-time decision-making, increasing overall operational efficiency.
2. **Foundation for Real-World Application and Advanced RL Integration:** Although the research is limited to a simulated environment, the developed Q-learning policy serves as a foundation for future real-world implementation. By validating Q-learning's effectiveness in simulation, this study paves the way for subsequent research that could incorporate advanced RL models, such as Deep Q-Networks (DQN), Policy Gradient Methods, or multi-agent RL. These advancements could further increase task efficiency, resilience, and collaborative capabilities in real-world industrial robotics.
3. **Insights into Hyperparameter Tuning for Task Resilience:** Through the optimization of Q-learning parameters, this study provides insights into how learning rate, discount factor, and exploration-exploitation strategies can affect a robot's ability to learn robust, adaptable policies in complex environments. This information is valuable for other researchers or engineers working on RL applications in similar settings, offering guidelines on hyperparameter tuning to maximize learning efficiency and task resilience.
4. **Potential for Improved Safety in Human-Robot Collaborative Spaces:** By training robots to recognize and avoid human-occupied spaces or prioritize safety in task execution, this research supports safer human-robot collaboration. The insights from this study can be used to develop safer task navigation protocols

that help robots operate in proximity to humans, with minimized risk of interference or accidents, making industrial environments more conducive to robot integration.

5. **Contribution to Sustainable and Cost-Efficient Industrial Operations:** With robots capable of adjusting to operational changes without constant reprogramming, industries can reduce downtime and maintenance costs. This adaptability can result in a more sustainable, cost-efficient industrial ecosystem, as robots trained via reinforcement learning can execute tasks with fewer interruptions, better resource management, and improved overall performance.
6. **Broadening the Scope of RL Applications in Industry:** This research highlights the applicability of Q-learning in industrial automation and expands the potential for using reinforcement learning to optimize task execution in various sectors, such as manufacturing, warehousing, and logistics. By focusing on RL's adaptability in task execution, this study could encourage more industrial sectors to adopt reinforcement learning to handle dynamic, task-intensive environments.

Chapter 5: Environmental Setup

5.1 Markov Decision Processes (MDPs)

The **MDP framework** is central to defining the robot's environment in this research, as it provides a structured, state-based approach to model the robot's navigation and task execution. An MDP is characterized by a set of states, actions, and transitions, which together determine the possible movements and decisions the robot can make while interacting with the environment. The goal of using an MDP is to provide the robot with a dynamic learning framework, enabling it to adapt to diverse scenarios, including routine tasks and unexpected disruptions [3], [12].

5.1.1 Stochastic Transitions in Markov Decision Processes (MDPs)

In the context of Markov Decision Processes (MDPs), stochastic transitions refer to the probabilistic nature of state changes that occur as a result of taking an action in a given state. Unlike deterministic transitions, where a specific action in a state leads to a fixed next state, stochastic transitions allow for multiple potential outcomes, each with an associated probability.

Formally, the transition dynamics in an MDP are defined by a transition probability function $P(s' \mid s, a)$, which represents the probability of transitioning to state s' after taking action a in state s. These probabilities must satisfy the following condition:

$$\sum_{s' \in S} P(s' \mid s, a) = 1 \ \forall s \in S, a \in A$$

where S is the set of states, and A is the set of actions.

Key characteristics of stochastic transitions include:

- **Probabilistic Uncertainty:** The next state is not fixed but sampled from a probability distribution defined by $P(s' \mid s, a)$.
- **Expressiveness:** They enable MDPs to capture complex dynamics and uncertainties, making them more applicable to real-world scenarios.
- **Computation and Planning:** Algorithms designed for solving MDPs with stochastic transitions, such as dynamic programming and reinforcement learning, must account for the expected value of outcomes across all possible transitions.

The use of stochastic transitions enhances the expressiveness of MDPs, allowing them to capture complex dynamics of uncertain systems. Algorithms designed to solve MDPs with stochastic transitions, such as dynamic programming and reinforcement learning, compute optimal policies by considering expected values of outcomes over the entire probability distribution of next states [3].

5.2 Spyder Platform:

Spyder, or the Scientific Python Development Environment, is an open-source Integrated Development Environment (IDE) specifically designed for scientific computing and data analysis in Python. It offers a user-friendly interface with features like code editing, interactive execution, debugging, and visualization tools. Spyder is widely used by scientists, engineers, and data analysts due to its seamless integration with popular scientific libraries like NumPy, SciPy, Matplotlib, and Pandas [13].

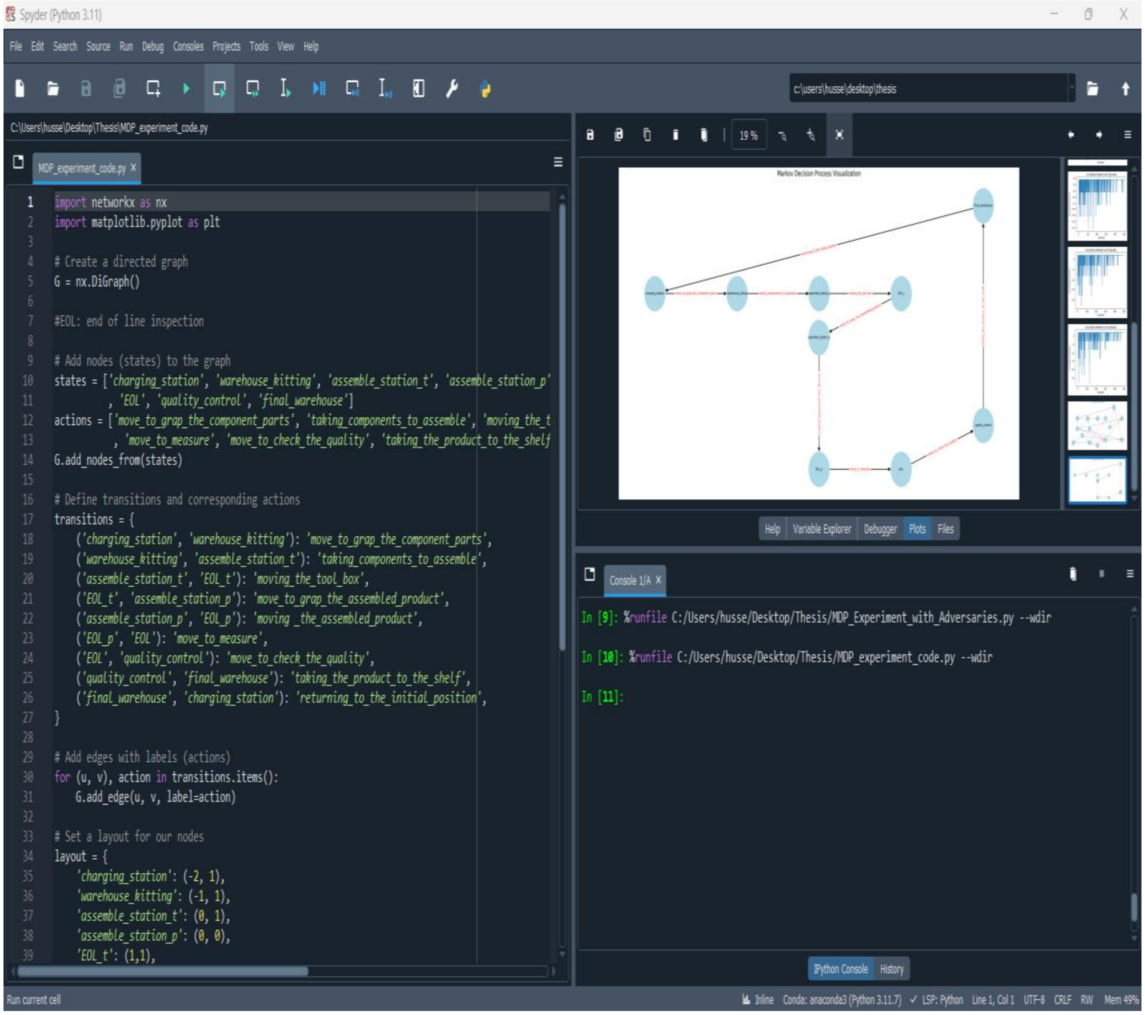
5.3 Setting up the environment and preliminary steps

This chapter is designed to offer the essential instructions for setting up the IDE and installing external libraries required to run various Python scripts. The reference guide for this process is based on a manual for RL, with the initial section focusing on the setup needed to begin developing implementations using Python. The first step is to install Anaconda, which can be done by following the specific installation instructions for your operating system found at https://docs.anaconda.com/anaconda/install/ [14], [15].

To proceed, run the 'Anaconda PowerShell Prompt' as an administrator (this program is included with Anaconda). Then, running the following commands:

- pip install networkx
- pip install matpolit.pyplot
- pip install pandas

The installation of the IDE and libraries is finished, now Spyder can be opened (which, similarly to Power shell, it is automatically installed together with Anaconda) from which scripts can be opened and ran.


Spyder (Python 3.11)
File Edit Search Source Run Debug Consoles Projects Tools View Help
c:\users\husse\desktop\thesis
C:\Users\husse\Desktop\Thesis\MDP_experiment_code.py
MDP_experiment_code.py
19 %
Markov Decision Process Visualization
Help Variable Explorer Debugger Plots Files
Console 1/A
IPython Console History
Run current cell
Inline Conda: anaconda3 (Python 3.11.7) LSP: Python Line 1, Col 1 UTF-8 CRLF RW Mem 49%

```
import networkx as nx
import matplotlib.pyplot as plt

# Create a directed graph
G = nx.DiGraph()

#EOL: end of line inspection

# Add nodes (states) to the graph
states = ['charging_station', 'warehouse_kitting', 'assemble_station_t', 'assemble_station_p'
          , 'EOL', 'quality_control', 'final_warehouse']
actions = ['move_to_grap_the_component_parts', 'taking_components_to_assemble', 'moving_the_t
           , 'move_to_measure', 'move_to_check_the_quality', 'taking_the_product_to_the_shelf
G.add_nodes_from(states)

# Define transitions and corresponding actions
transitions = {
    ('charging_station', 'warehouse_kitting'): 'move_to_grap_the_component_parts',
    ('warehouse_kitting', 'assemble_station_t'): 'taking_components_to_assemble',
    ('assemble_station_t', 'EOL_t'): 'moving_the_tool_box',
    ('EOL_t', 'assemble_station_p'): 'move_to_grap_the_assembled_product',
    ('assemble_station_p', 'EOL_p'): 'moving _the_assembled_product',
    ('EOL_p', 'EOL'): 'move_to_measure',
    ('EOL', 'quality_control'): 'move_to_check_the_quality',
    ('quality_control', 'final_warehouse'): 'taking_the_product_to_the_shelf',
    ('final_warehouse', 'charging_station'): 'returning_to_the_initial_position',
}

# Add edges with labels (actions)
for (u, v), action in transitions.items():
    G.add_edge(u, v, label=action)

# Set a layout for our nodes
layout = {
    'charging_station': (-2, 1),
    'warehouse_kitting': (-1, 1),
    'assemble_station_t': (0, 1),
    'assemble_station_p': (0, 0),
    'EOL_t': (1,1),
```

```
In [9]: %runfile C:/Users/husse/Desktop/Thesis/MDP_Experiment_with_Adversaries.py --wdir

In [10]: %runfile C:/Users/husse/Desktop/Thesis/MDP_experiment_code.py --wdir

In [11]:
```

Figure 13 Spyder Platform

5.4 Python Libraries Used

In this study, several Python libraries were utilized to perform data manipulation, analysis, visualization, and network-related tasks. Below is a brief overview of each library and its role in the code:

NumPy (import numpy as np): NumPy is a fundamental package for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. In this project, NumPy was primarily used for handling numerical data and performing vectorized operations, which significantly improves the speed and efficiency of computations [16].

Random (import random): The random module implements pseudo-random number generators for various distributions. It provides functions to generate random numbers, select random items from a list, and shuffle data. This library was used in the project to introduce randomness, such as selecting random state or action [17].

Matplotlib (import matplotlib.pyplot as plt): Matplotlib is a plotting library for creating static, animated, and interactive visualizations in Python. It is widely used for generating various types of plots, such as line charts, histograms, scatter plots, and more. In this project, Matplotlib was employed to visualize the data and results of the analysis, offering clear and informative graphical representations to interpret the findings effectively [18].

Pandas (import pandas as pd): Pandas is a powerful library for data manipulation and analysis. It provides data structures such as DataFrame and Series that are ideal for handling and analyzing structured data, such as time series or tabular data. With its rich functionality, including support for reading and writing data to various formats (CSV, Excel, etc.), and performing data cleaning, transformation, and aggregation tasks, Pandas was an essential tool in this project for managing datasets and performing exploratory data analysis [19].

NetworkX (import networkx as nx): NetworkX is a library designed for the creation, manipulation, and study of the structure and dynamics of complex networks. It provides tools to work with graphs, which are mathematical structures representing pairwise relationships between objects. In this study, NetworkX was used to model and analyze networks, structuring the MDP [20].

5.5 MiR100

The MiR100 is a mobile robot designed to automate the movement of goods and materials within a company, providing a fast and cost-effective solution. It enhances operational efficiency by allocating resources to employees, leading to increased productivity and reduced costs.

Figure 14 MiR100 Mobile Robot [21]

The key features of the MIR100 relevant to this project include:

- **Navigation in busy environments**: The robot is capable of safely operating in dynamic, crowded workspaces.
- **Path planning and local route adjustments**: The robot prioritizes finding the most efficient route to its destination, while also being able to modify its path when encountering obstacles.
- **Heavy load transportation**: It can carry loads weighing up to 100 kg.

- **Internal mapping**: The robot generates a map of its environment by manually navigating the workspace. During this process, it detects walls, doors, furniture, and other obstacles, creating a map based on these findings, which can later be edited through the MIR software.

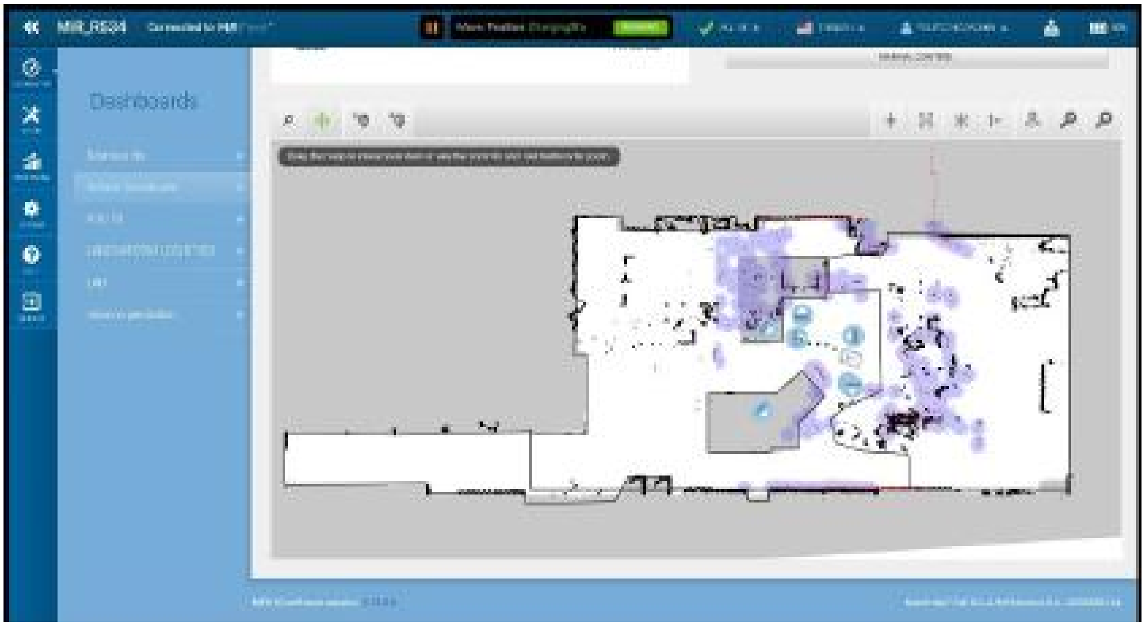


Figure 15 MiR100 Live Map

5.6 Robot Operating System (ROS)

The Robot Operating System (ROS) is an open-source software framework designed to support the development of robotic applications. Although it is called an operating system, it is actually a collection of tools and libraries aimed at aiding robot development. Its key features include [22]:

- **Middleware**: ROS acts as a mediator within a robotic system, enabling communication between different nodes, or code modules. The most commonly used programming languages for creating these nodes are Python and C++.
- **Development Tools**: ROS provides various tools that simplify tasks like data visualization, debugging, and programming. For example, *rqt* offers a graphical user interface (GUI) for managing and displaying data, while *rviz* is a visualization tool that allows users to view real-time sensor and node data.

- **Package Management**: ROS organizes and manages software through a package system. Each package may include nodes, libraries, configuration files, and other necessary tools for a specific function.
- **Hardware Abstraction**: ROS offers hardware abstractions, enabling developers to write code that interacts with different sensors and actuators without dealing with the specific details of the hardware.
- **Community and Ecosystem**: As an open-source project, ROS is supported by a large development and research community that provides additional packages, tools, and documentation. This ecosystem helps accelerate the adoption of new technologies and reduces development time.
- **Interoperability**: ROS enables communication between nodes using various message and service protocols, allowing different system components to work together seamlessly.

5.7 ROS Architecture

The architecture of ROS consists of several key elements [22], [23]:

- **Nodes**: Nodes are individual processes that execute specific functions within ROS. Each node performs a different task, and they can communicate with each other through the ROS network. For example, one node might handle a planning algorithm, another might control an actuator, and another might read sensor data. A node could, for instance, capture data from a camera, while another processes that data for object recognition. Nodes interact with each other using communication mechanisms such as Topics, Services, and Actions.
- **Topics**: Topics are communication channels that allow nodes to exchange messages by publishing and subscribing to them. The key characteristics of Topics include:
    - **Publish-Subscribe Model**: A node can either send (publish) or receive (subscribe) messages to or from a topic. Multiple nodes can publish or subscribe to messages within the same topic.
    - **Anonymous Communication**: Nodes communicating via a topic do not need to know the identity of the sender or receiver, which decouples them and increases the scalability of the network.

- ○ **Message Type**: Each topic is defined by a specific message type, so any message published to a topic must conform to that message type.
- ○ **Asynchronous Communication**: Messages published to or subscribed from a topic can be sent and received at different times and frequencies, allowing the nodes to operate independently of each other.

- **Services**: Services are used when a remote procedure call (RPC) interaction is required, meaning a request/reply communication pattern. A service consists of two message types: one for the request and one for the reply. A ROS node provides a service, where a client can make a request by sending a message, and the node will respond with the corresponding reply.

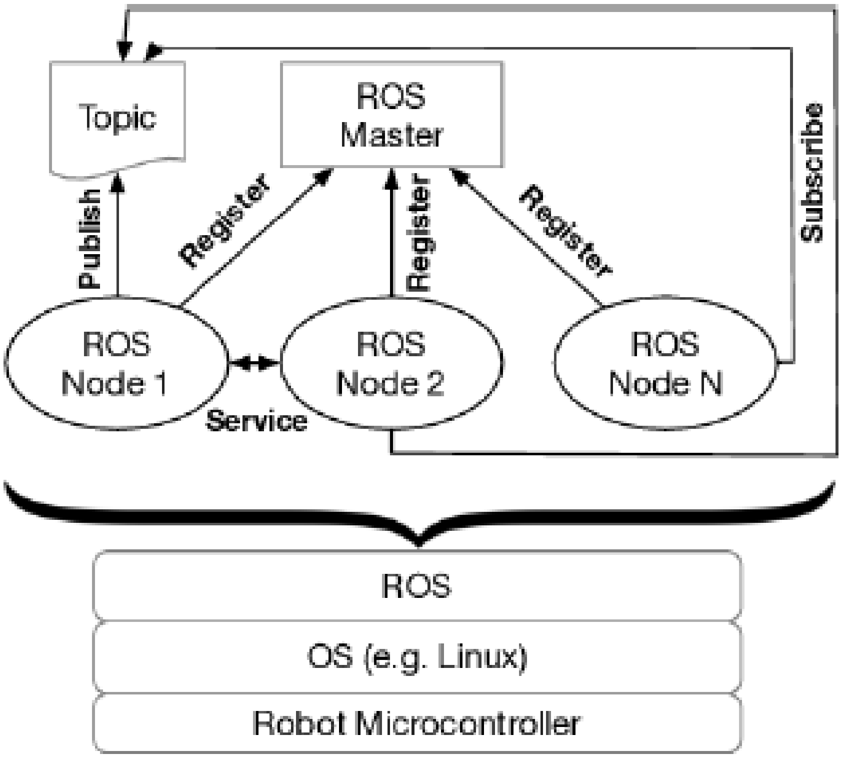


Figure 16 ROS Architecture [23]

Chapter 6: Methodology

6.1 Q-Learning in Reinforcement Learning

Q-learning is one of the most widely used model-free reinforcement learning (RL) algorithms. It enables an agent to learn optimal actions in an environment by maximizing cumulative rewards without needing a predefined model of the environment. This adaptability makes Q-learning highly applicable in dynamic, complex environments, such as industrial setups, where conditions and tasks can vary widely. Q-learning is particularly suitable for environments modeled as Markov Decision Processes (MDPs), where outcomes depend on both the agent's actions and inherent uncertainties in the environment [3].

6.1.1 Q-learning: Foundations and Core Concepts

Q-learning relies on an **action-value function**, or **Q-function**, which estimates the expected cumulative reward (known as the “Q-value”) for taking a particular action in a specific state. The agent uses this Q-function to learn which actions yield the highest rewards over time, converging to an optimal policy by updating Q-values as it explores the environment. The algorithm is designed to handle environments with stochastic transitions and rewards, making it robust in settings where outcomes are partly random [24].

The **Q-learning update rule** is expressed as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

where:

- Q(s,a) is the Q-value for state s and action a,
- Alpha α is the **learning rate**, determining how much new information overrides previous knowledge,
- r is the **reward** received after taking action a,
- γ is the **discount factor**, representing the importance of future rewards,

This update rule adjusts Q-values iteratively, allowing the agent to improve its understanding of the best actions to take in each state by balancing **immediate rewards** and **future rewards**.

6.1.2 Steps in Implementing Q-learning

The Q-learning algorithm proceeds through the following steps:

1. **Initialization**:
   - The Q-table is initialized with zeros or random values for each state-action pair. Each row represents a state, and each column represents an action, with values indicating the expected reward.
   - Parameters like the learning rate α, discount factor γ, and exploration parameter ϵ are set, often with fine-tuning through experimentation.
2. **Action Selection**:
   - Using an epsilon-greedy strategy, the agent either explores (with probability ϵ) by selecting a random action or exploits (with probability 1−ϵ) by choosing the action with the highest Q-value in its current state. This strategy helps balance exploration and exploitation in Q-learning.
3. **Execution and Reward Collection**:
   - The agent takes the chosen action and observes the resulting next state s′ and the immediate reward r for the action. These observed outcomes directly influence the Q-value updates.

4. **Q-value Update**:
    - The Q-value for the state-action pair (s,a) is updated based on the received reward and the maximum estimated Q-value for the next state s′, following the Q-learning update rule. This update is essential for learning the expected reward of each action over time.
5. **Repeat Until Convergence**:
    - These steps are repeated for each episode, where the agent continues to explore different state-action pairs, gradually improving its Q-table until Q-values converge. Convergence occurs when Q-values stabilize, indicating that the agent has learned an optimal policy.

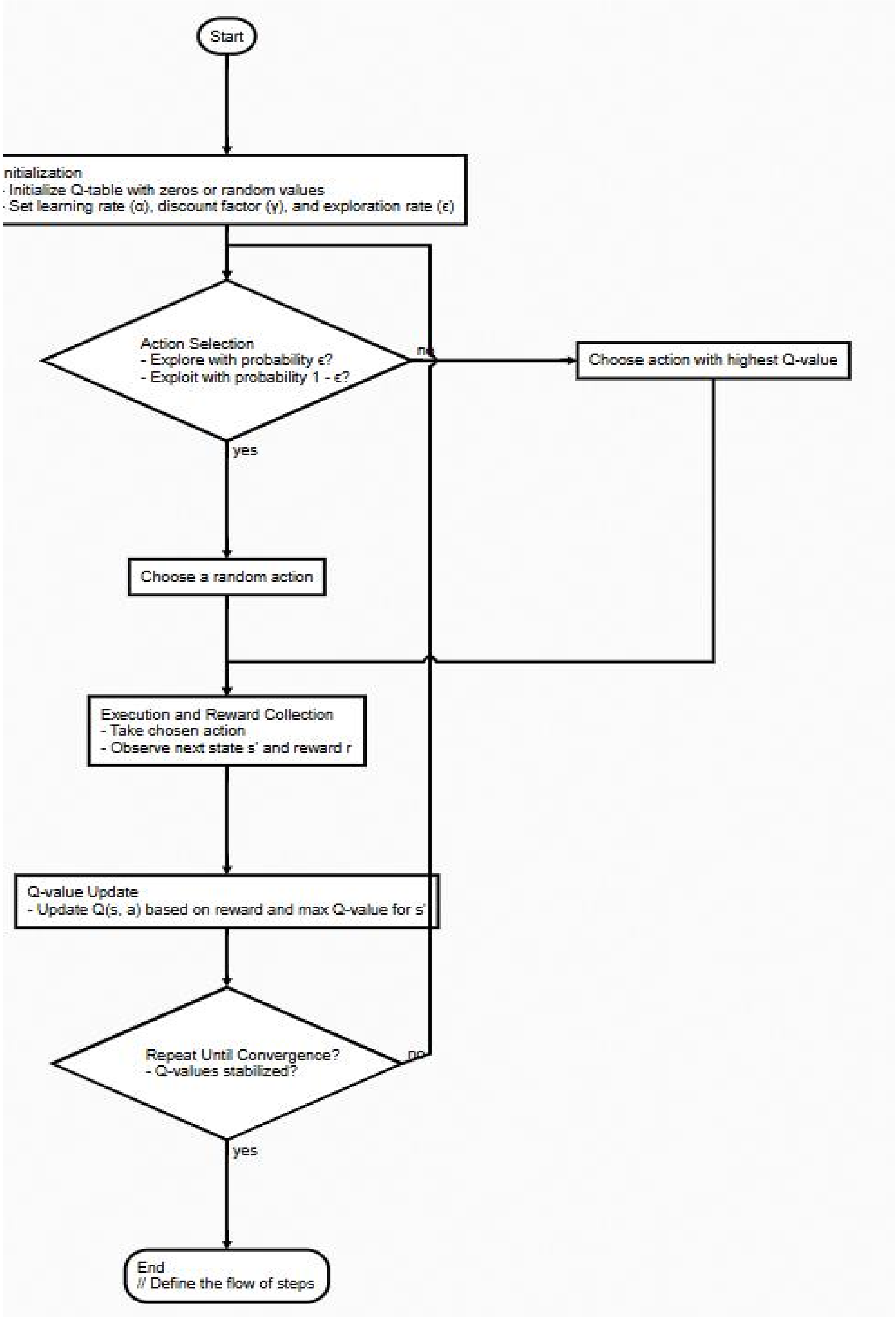

Start
nitialization
· Initialize Q-table with zeros or random values
· Set learning rate (α), discount factor (γ), and exploration rate (ε)
Action Selection
- Explore with probability ε?
- Exploit with probability 1 - ε?
no
Choose action with highest Q-value
yes
Choose a random action
Execution and Reward Collection
- Take chosen action
- Observe next state s' and reward r
Q-value Update
- Update Q(s, a) based on reward and max Q-value for s'
Repeat Until Convergence?
- Q-values stabilized?
no
yes
End
// Define the flow of steps

Figure 17 Q-learning Flowchart [25]

6.1.3 Advantages and Limitations of Q-learning

Q-learning offers notable benefits in its simplicity, adaptability, and ability to learn without a model of the environment. This makes it suitable for applications in dynamic industrial settings where unpredictability is common. Some key advantages of Q-learning include:

- **Model-Free Nature**: Q-learning does not require a predefined model of the environment, which makes it adaptable to unknown environments where transitions are not fully predictable.
- **Convergence Guarantees**: With appropriate learning parameters and enough episodes, Q-learning can converge to an optimal policy in discrete action spaces, maximizing cumulative rewards.
- **Scalability to Various Tasks**: Q-learning is flexible enough to handle a range of tasks by adjusting its hyperparameters, making it useful for robots performing different functions like navigation, assembly, or quality control.

However, Q-learning has limitations, particularly in environments with high-dimensional state-action spaces. As the number of states or actions increases, the **Q-table size** grows, leading to **scalability issues**. Additionally, Q-learning can struggle in environments where actions must be selected based on continuous variables, as the discrete Q-table is less effective without adjustments or alternative RL techniques.

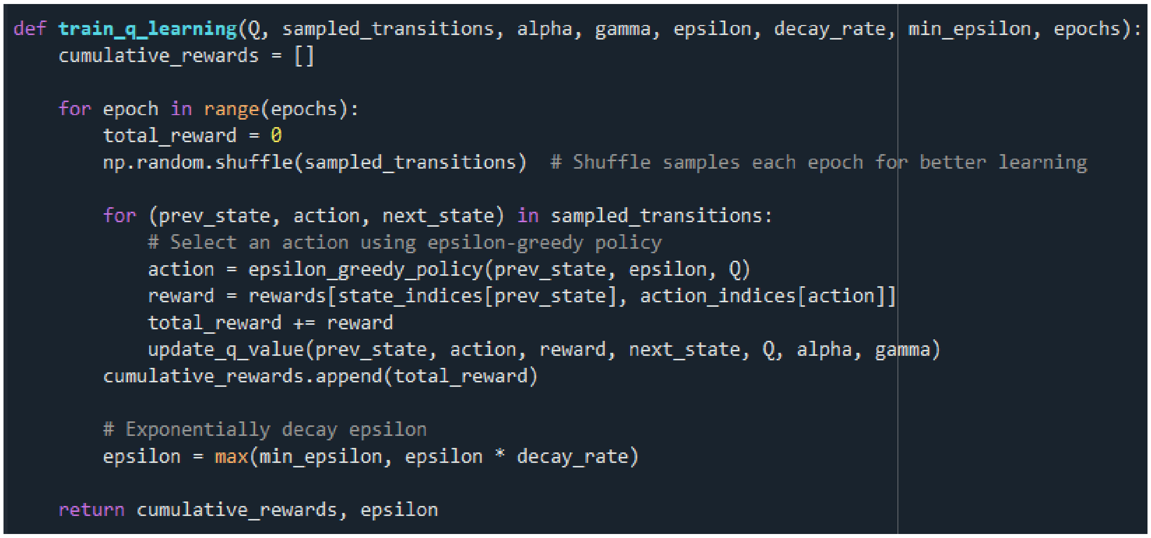
6.1.4 Practical Implementation in the Industrial MDP Environment

In this research, Q-learning is applied to train a robot in an industrial environment simulated as an MDP. The robot learns optimal actions for navigating and performing tasks across several states, including charging stations, warehouse kitting, assembly, quality control, and the final warehouse. Q-learning's flexibility in MDPs enables the robot to:

- Learn effective transitions between states such as moving from an assembly station to quality control, or between charging and work states, optimizing task sequences.
- Respond to **stochastic events** like part mismatches, battery threshold management, or sudden failures, by adjusting its actions based on the rewards or penalties received.

- Converge to an optimal policy for task completion, minimizing operational delays and enhancing task adaptability.

The Q-learning approach in this setting is particularly advantageous because it enables the robot to adjust to the dynamic nature of industrial tasks, even without having pre-programmed solutions for every possible scenario. By training through exploration and exploitation, the robot becomes adept at navigating complex workflows while handling unexpected challenges effectively.



```python
def train_q_learning(Q, sampled_transitions, alpha, gamma, epsilon, decay_rate, min_epsilon, epochs):
    cumulative_rewards = []

    for epoch in range(epochs):
        total_reward = 0
        np.random.shuffle(sampled_transitions)  # Shuffle samples each epoch for better learning

        for (prev_state, action, next_state) in sampled_transitions:
            # Select an action using epsilon-greedy policy
            action = epsilon_greedy_policy(prev_state, epsilon, Q)
            reward = rewards[state_indices[prev_state], action_indices[action]]
            total_reward += reward
            update_q_value(prev_state, action, reward, next_state, Q, alpha, gamma)
        cumulative_rewards.append(total_reward)

        # Exponentially decay epsilon
        epsilon = max(min_epsilon, epsilon * decay_rate)

    return cumulative_rewards, epsilon
```

Figure 18 Q-learning Training

6.2 SARSA in Reinforcement Learning

SARSA is an on-policy reinforcement learning (RL) algorithm that determines optimal actions in an environment modeled as a Markov Decision Process (MDP). Unlike Q-learning, which is an off-policy algorithm, SARSA learns from the actions taken by the agent based on its current policy. This "on-policy" approach allows SARSA to continuously refine actions using information derived from the agent's own behavior, making it particularly suitable for environments that require stable, gradual learning over time, such as dynamic or uncertain industrial settings [3].

6.2.1 Foundations of SARSA

The name **SARSA** stands for **State-Action-Reward-State-Action**, referring to the sequence of events the algorithm uses to update the action-value function (Q-values). In each episode, the agent observes its current state and takes an action, receiving a reward and transitioning to a new state, where it takes the next action according to its policy. SARSA then updates the Q-value based on this sequence, ensuring that the agent learns values tied to the policy it follows, rather than an idealized optimal policy [24].

The **SARSA update rule** is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right]$$

where:

- Q(s,a) is the Q-value for state s and action a.
- α is the **learning rate** controlling the weight of new information in updates.
- r is the **reward** obtained after taking action a.
- γ is the **discount factor**, prioritizing immediate vs. future rewards.
- Q(s′,a′) is the Q-value of the next state-action pair as per the agent's current policy.

The critical difference from Q-learning is that SARSA uses the next action a′ derived from the agent's policy, making it an **on-policy algorithm**.

6.2.2 Steps in Implementing SARSA

The SARSA algorithm progresses through the following steps:

1. **Initialization**:
    - ○ The Q-table is initialized with zeros or random values. Each Q-value represents the estimated cumulative reward for a state-action pair.
    - ○ Learning parameters such as α, γ, and ϵ (for exploration in epsilon-greedy policies) are set based on experimental tuning.

2. **Choosing an Action**:
   - The agent selects an action a from the current state sss using an epsilon-greedy strategy to balance exploration and exploitation.
   - Unlike Q-learning, SARSA's updates are directly influenced by the action the agent plans to take, tying learning closely to the agent's current policy.
3. **Executing the Action and Observing the Outcome**:
   - The agent performs the selected action a and transitions to the next state s′, where it receives a reward r from the environment. This reward informs the agent about the immediate impact of its action on achieving desired outcomes.
4. **Next Action Selection and Q-value Update**:
   - The agent selects the next action a′ in the new state s′ using the same policy (e.g., epsilon-greedy). The Q-value for (s,a) is updated based on the reward r and the Q-value of (s′,a′) as per the SARSA update rule.
   - By using Q(s′,a′) in the update, SARSA refines the action-value table according to the agent's current action policy rather than an idealized one.
5. **Repeating Until Convergence**:
   - The agent continues this process across episodes, progressively refining its policy by updating Q-values with each state-action transition. Convergence occurs when the Q-values stabilize, signaling that the agent has learned an optimal policy under the current parameters.

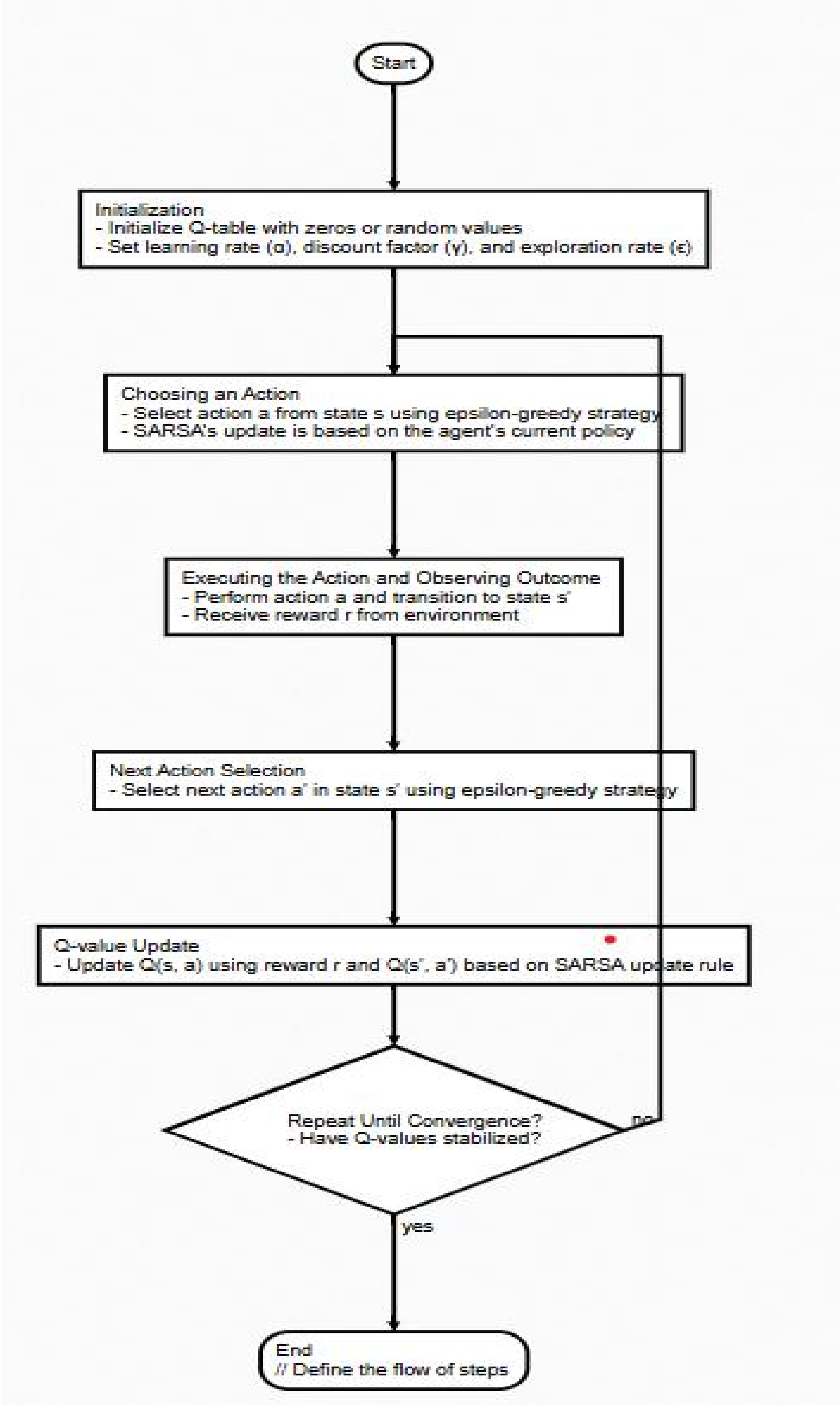

Start
Initialization
- Initialize Q-table with zeros or random values
- Set learning rate (α), discount factor (γ), and exploration rate (ε)
Choosing an Action
- Select action a from state s using epsilon-greedy strategy
- SARSA's update is based on the agent's current policy
Executing the Action and Observing Outcome
- Perform action a and transition to state s'
- Receive reward r from environment
Next Action Selection
- Select next action a' in state s' using epsilon-greedy strategy
Q-value Update
- Update Q(s, a) using reward r and Q(s', a') based on SARSA update rule
Repeat Until Convergence?
- Have Q-values stabilized?
no
yes
End
// Define the flow of steps

Figure 19 SARSA Flowchart [25]

6.2.3 Comparison of SARSA and Q-learning

While SARSA and Q-learning share similarities in updating action-value pairs, there are distinct differences in their approach and suitability for various environments:

- **On-policy vs. Off-policy**: SARSA's on-policy nature allows it to learn based on the agent's current behavior, which means it can adapt more conservatively, potentially resulting in smoother learning. This is advantageous in unstable environments where consistency is essential.
- **Stability in Dynamic Environments**: SARSA's on-policy updates mean that it adheres more closely to its current exploration-exploitation balance, which often makes it more stable in dynamic or high-risk environments. Q-learning, by contrast, learns an optimal policy assuming ideal actions, which may lead to erratic behavior if environmental conditions change unexpectedly.
- **Efficiency in Convergence**: Q-learning often converges faster due to its off-policy nature, which enables it to explore optimal actions aggressively. However, SARSA may provide more consistent performance in environments with fluctuating conditions or where exploratory actions must be more cautiously evaluated.

6.2.4 Application of SARSA in the Industrial MDP Environment

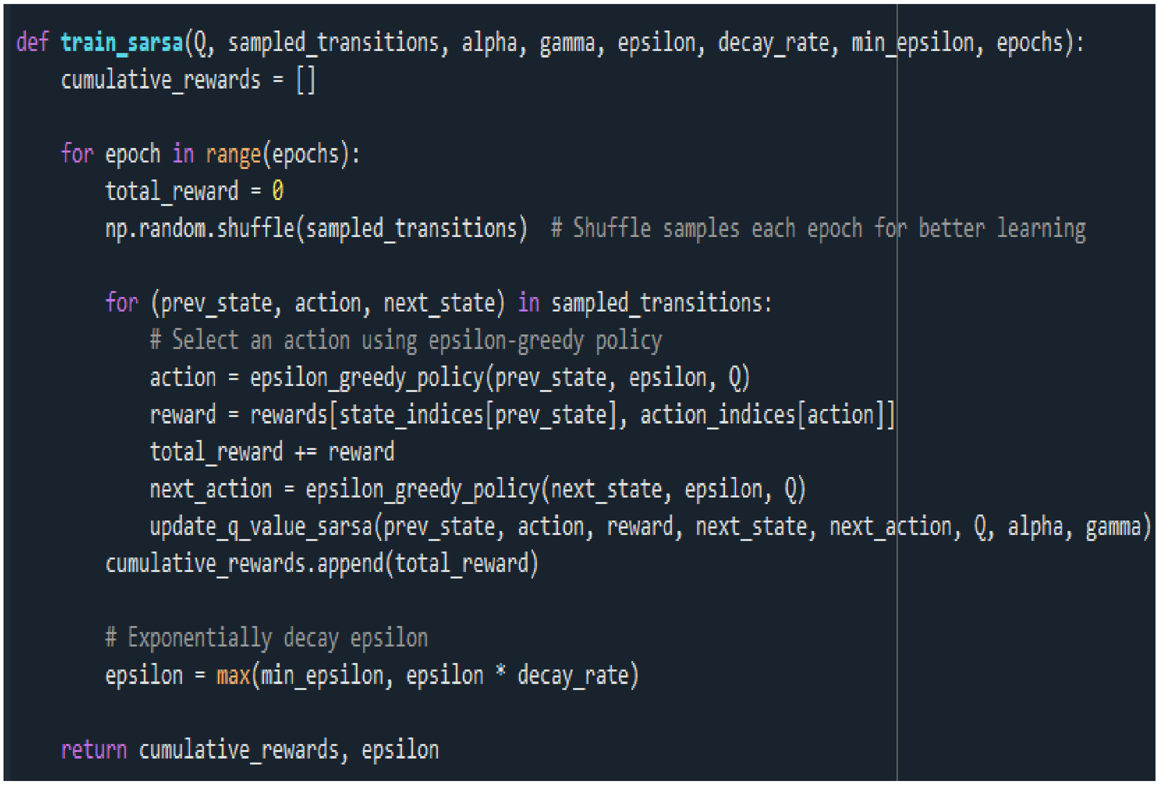
In this research, SARSA is applied within an industrial MDP environment, where the robot needs to manage various tasks, such as navigation between charging stations, quality control, assembly, and handling unexpected conditions. SARSA's on-policy learning approach offers specific advantages in this setup:

- **Adaptability in Task Sequences**: SARSA is particularly suitable for navigating through tasks that might change unpredictably. For instance, if a quality control issue arises mid-sequence, SARSA's policy-based updates help the robot adapt without compromising consistency.
- **Effective Resource Management**: SARSA can handle resource-sensitive states, such as battery management, more stably by updating Q-values in line with actions

taken. This allows the robot to factor in constraints like battery levels or maintenance needs without over-exploration.

- **Stability in Worst-Case Scenarios**: SARSA's conservative updates make it effective in worst-case scenarios, such as component mismatches or urgent tasks. The algorithm's stable convergence allows the robot to prioritize safer, policy-aligned actions, reducing the risk of failure in high-stakes environments.

The on-policy nature of SARSA helps the robot maintain a stable, adaptive policy that can handle the variability typical of industrial applications. By training within this structured MDP, the robot learns efficient actions for both routine and challenging tasks, improving operational resilience.



```python
def train_sarsa(Q, sampled_transitions, alpha, gamma, epsilon, decay_rate, min_epsilon, epochs):
    cumulative_rewards = []

    for epoch in range(epochs):
        total_reward = 0
        np.random.shuffle(sampled_transitions)  # Shuffle samples each epoch for better learning

        for (prev_state, action, next_state) in sampled_transitions:
            # Select an action using epsilon-greedy policy
            action = epsilon_greedy_policy(prev_state, epsilon, Q)
            reward = rewards[state_indices[prev_state], action_indices[action]]
            total_reward += reward
            next_action = epsilon_greedy_policy(next_state, epsilon, Q)
            update_q_value_sarsa(prev_state, action, reward, next_state, next_action, Q, alpha, gamma)
        cumulative_rewards.append(total_reward)

        # Exponentially decay epsilon
        epsilon = max(min_epsilon, epsilon * decay_rate)

    return cumulative_rewards, epsilon
```

Figure 20 SARSA Training

6.2.5 Advantages and Limitations of SARSA

SARSA offers several advantages in environments where stability and policy consistency are important, such as in real-time or industrial settings. Some of its key advantages include:

- **On-policy Learning**: SARSA's on-policy updates provide a smoother learning process, ideal for environments with variability, where adhering to the current policy reduces erratic actions.
- **Stability and Robustness**: SARSA adapts well to dynamic or uncertain environments, making it useful in industrial setups where tasks or conditions can change. This stability is particularly valuable in handling unexpected events.

However, SARSA also has limitations:

- **Slower Convergence**: SARSA may converge more slowly than Q-learning, as it requires adherence to the agent's own policy rather than an optimal policy based on maximum Q-values. In time-sensitive applications, this slower convergence could be a constraint.
- **Reduced Exploration**: Because SARSA relies on the agent's current policy for updates, it can limit exploration in favor of policy consistency, potentially overlooking optimal actions in favor of stability.

6.3 Dyna-Q in Reinforcement Learning

Dyna-Q is a hybrid reinforcement learning (RL) algorithm developed by Richard Sutton that combines elements of model-based and model-free learning. By integrating planning, Dyna-Q improves upon traditional Q-learning by enabling the agent to update its policy based on both real experiences and simulated experiences generated from an internal model of the environment. This approach allows Dyna-Q to accelerate learning, making it particularly useful in dynamic, complex environments where efficiency and adaptability are essential [3].

6.3.1 Foundations of Dyna-Q

Dyna-Q’s framework revolves around three main components: **direct RL**, **model learning**, and **planning**. The agent performs actions, updates its Q-values based on real experiences (as in traditional Q-learning), and simultaneously builds a model of the environment, storing state-action transitions and rewards. It then uses this model to simulate additional experiences and performs “planning updates” on the Q-table, reinforcing learning through both real and simulated interactions [26].

Dyna-Q’s **Q-value update rule** is similar to that of Q-learning:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

where:

- Q(s,a) represents the Q-value for state s and action a,
- α is the **learning rate** controlling the influence of new updates,
- r is the **reward** received after taking action a,
- gamma γ is the **discount factor**, balancing immediate and future rewards,
- $\max_{a'} Q(s',a')$ is the maximum estimated Q-value for the next state s′, assuming the agent acts optimally.

However, Dyna-Q differentiates itself by using **simulated experiences** generated from its learned model. This model allows the agent to explore state-action pairs more efficiently, leading to faster convergence and a more robust learning process.

6.3.2 Steps in Implementing Dyna-Q

The Dyna-Q algorithm integrates both real experience updates and simulated planning updates in the following steps:

1. **Initialize the Q-table and Model**:
   - The Q-table is initialized with zeros or random values for all state-action pairs.
   - A model of the environment, often implemented as a dictionary or table, is initialized to store transition information based on the agent's interactions.
2. **Real Experience Update**:
   - The agent selects an action using an epsilon-greedy strategy, balancing exploration and exploitation.
   - After performing the action and observing the resulting next state and reward, the agent updates the Q-value for the state-action pair using the Q-learning update rule.
   - The transition is then stored in the model, updating its understanding of the environment.
3. **Simulated Planning Updates**:
   - Using the stored model, the agent generates simulated experiences by randomly sampling previously encountered state-action pairs.
   - For each simulated experience, the agent retrieves the next state and reward from the model and performs additional Q-value updates on the sampled state-action pairs.
   - This process is repeated several times (determined by a n_planning_steps) for each real experience, enhancing the agent's learning without requiring additional real interactions.
4. **Repeat Until Convergence**:
   - The agent continues alternating between real experience updates and planning updates across episodes until Q-values stabilize. This hybrid approach accelerates learning, as the agent reinforces its policy using both real and simulated experiences.

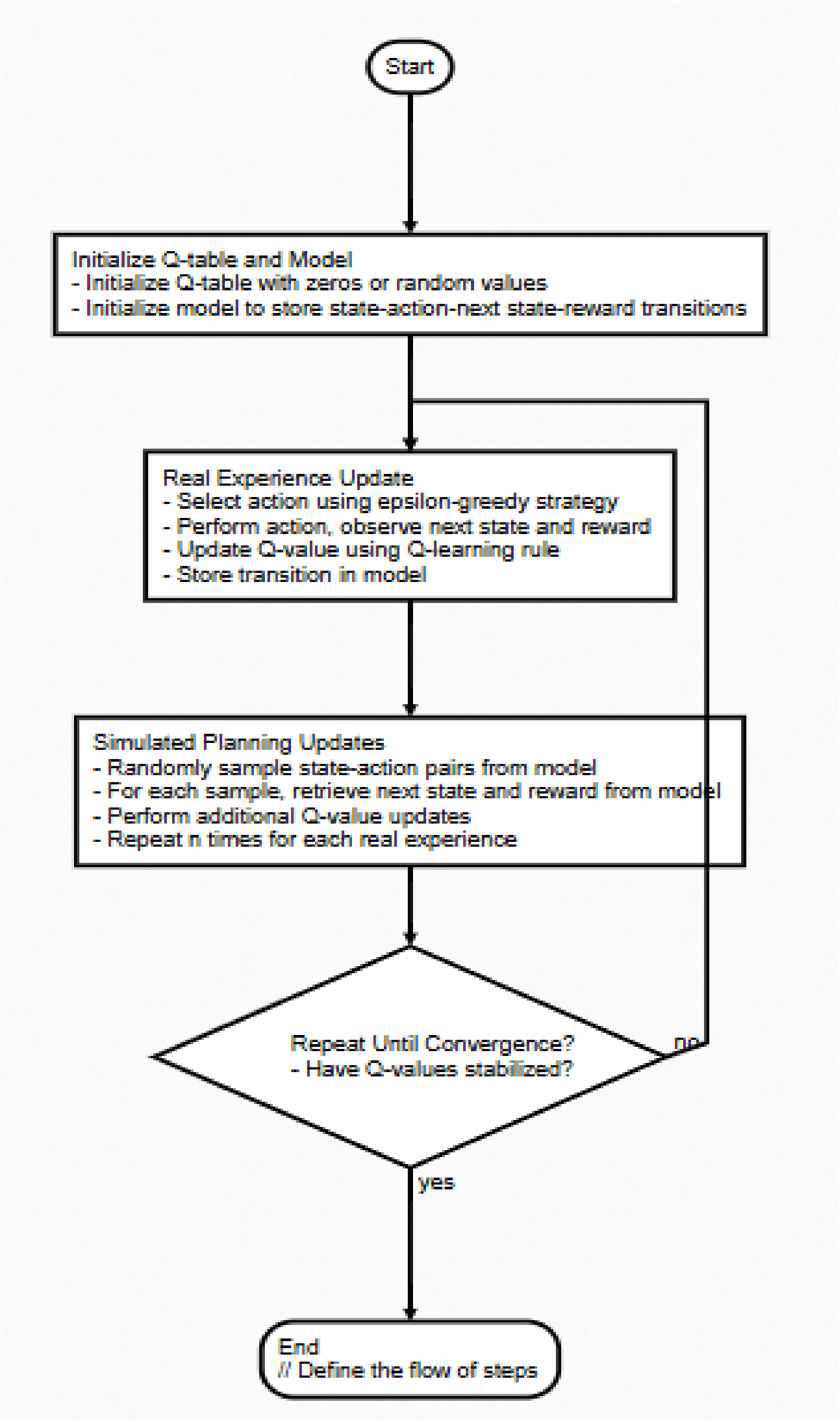

Start
Initialize Q-table and Model
- Initialize Q-table with zeros or random values
- Initialize model to store state-action-next state-reward transitions
Real Experience Update
- Select action using epsilon-greedy strategy
- Perform action, observe next state and reward
- Update Q-value using Q-learning rule
- Store transition in model
Simulated Planning Updates
- Randomly sample state-action pairs from model
- For each sample, retrieve next state and reward from model
- Perform additional Q-value updates
- Repeat n times for each real experience
Repeat Until Convergence?
- Have Q-values stabilized?
no
yes
End
// Define the flow of steps

Figure 21 Dyna-Q Flowchart [25]

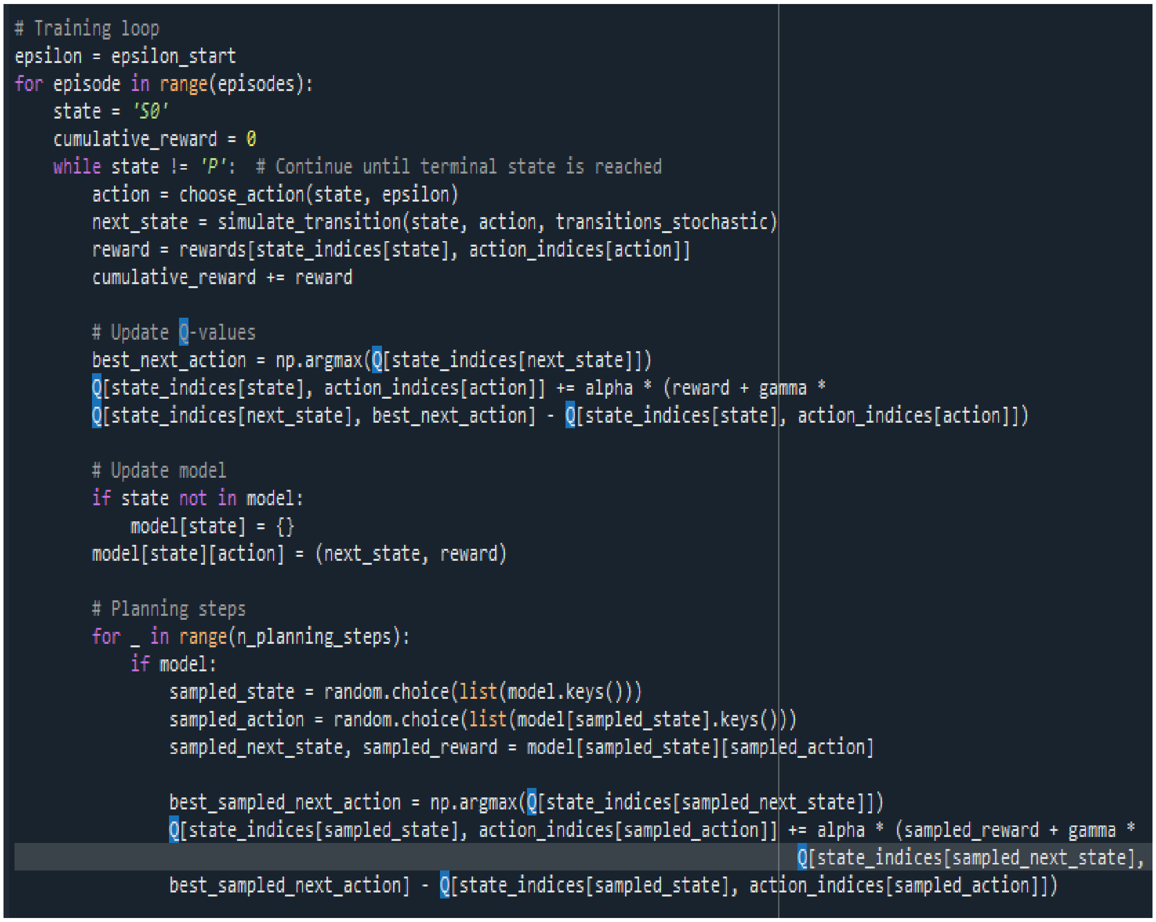
6.3.3 Limitations of Dyna-Q

However, Dyna-Q has limitations:

- **Dependence on an Accurate Model**: Dyna-Q relies on an accurate internal model for generating simulated experiences. Inaccuracies in this model can lead to suboptimal policy updates, as the agent may reinforce incorrect actions.
- **Complexity and Computational Overhead**: The need for planning updates introduces computational overhead, making Dyna-Q more resource-intensive than purely model-free approaches like Q-learning. In real-time applications, this added complexity can be challenging.

6.3.4 Application of Dyna-Q in the Navigation MDP Environment

One potential reason Dyna-Q may be an inappropriate choice for this navigation MDP environment is if the **environment is highly dynamic or has non-stationary elements** meaning that its rules, obstacles, or reward structures change over time. Dyna-Q relies heavily on building and updating an internal model of the environment, which is used for planning. This model assumes that the environment is **stationary** (i.e., its dynamics don't change). In a navigation environment with changing conditions, Dyna-Q's internal model can quickly become outdated or inaccurate, causing the agent to make suboptimal decisions based on an incorrect understanding of the environment.



```
# Training loop
epsilon = epsilon_start
for episode in range(episodes):
    state = 'S0'
    cumulative_reward = 0
    while state != 'P':  # Continue until terminal state is reached
        action = choose_action(state, epsilon)
        next_state = simulate_transition(state, action, transitions_stochastic)
        reward = rewards[state_indices[state], action_indices[action]]
        cumulative_reward += reward

        # Update Q-values
        best_next_action = np.argmax(Q[state_indices[next_state]])
        Q[state_indices[state], action_indices[action]] += alpha * (reward + gamma *
        Q[state_indices[next_state], best_next_action] - Q[state_indices[state], action_indices[action]])

        # Update model
        if state not in model:
            model[state] = {}
        model[state][action] = (next_state, reward)

        # Planning steps
        for _ in range(n_planning_steps):
            if model:
                sampled_state = random.choice(list(model.keys()))
                sampled_action = random.choice(list(model[sampled_state].keys()))
                sampled_next_state, sampled_reward = model[sampled_state][sampled_action]

                best_sampled_next_action = np.argmax(Q[state_indices[sampled_next_state]])
                Q[state_indices[sampled_state], action_indices[sampled_action]] += alpha * (sampled_reward + gamma *
                                                                                 Q[state_indices[sampled_next_state],
                best_sampled_next_action] - Q[state_indices[sampled_state], action_indices[sampled_action]])
```

Figure 22 Dyna-Q Training Loop

Imagine a navigation task where the layout of the location of the goal changes periodically. For instance, obstacles might move, or certain regions may switch from being high-penalty to low-penalty zones over time. In such an environment, an agent using Dyna-Q would struggle to keep its internal model up-to-date, as the Dyna-Q algorithm is not designed to continuously adapt to non-stationary dynamics. Instead, it would keep relying on simulated experiences generated from an outdated model, leading the agent to repeatedly make mistakes or avoid certain areas that it wrongly assumes are problematic.

This could explain the results seen in **figure 39**, where the cumulative reward stabilizes close to zero rather than reaching positive values. Each time the environment changes, the agent's policy becomes less effective, and Dyna-Q's model-based planning might actually reinforce suboptimal behaviors. The agent might end up revisiting safe but non-optimal routes, failing to learn an effective navigation strategy that adapts to the environment's changes.

In dynamic environments, model-free methods (such as Q-learning or SARSA) or adaptive model-based approaches are often more suitable. These methods either don't rely on a model or continuously update their model in response to changes. For a task where adaptability is essential, such approaches might allow the agent to respond better to changes in the environment, leading to more consistent positive rewards over time.

6.4 Sample-Based Planning in Reinforcement Learning

Sample-Based Planning is a technique in reinforcement learning where an agent leverages a learned model of the environment to simulate experiences and optimize its policy without interacting with the actual environment. Unlike traditional planning methods that compute outcomes deterministically, sample-based planning uses **random samples** from a model to approximate potential future states and rewards. This approach is especially beneficial in scenarios where interactions with the real environment are costly, time-consuming, or impractical, making it suitable for dynamic, complex environments such as those encountered in industrial automation [3].

6.4.1 Foundations of Sample-Based Planning

Sample-based planning sits between pure model-free and model-based RL methods. In model-free methods, the agent relies entirely on real experiences to update its policy, while in model-based methods, the agent has a complete model of the environment and uses this model for planning. Sample-based planning combines elements of both approaches by allowing the agent to generate simulated experiences based on a partial model of the environment, which helps accelerate learning and reduce dependence on real-time interactions [26].

The process involves three key elements:

- **Sampling**: Using the learned model to generate state-action pairs that represent potential outcomes based on the agent's current policy.
- **Planning Updates**: Updating action-value estimates (Q-values) based on the simulated outcomes, reinforcing the agent's understanding of optimal actions.
- **Efficient Exploration**: Simulating various scenarios allows the agent to explore a broader range of possible actions, refining its policy more efficiently.

This technique has proven particularly effective for environments with large state spaces, as it allows the agent to approximate an optimal policy without exhaustively exploring every possible action in the real world.

6.4.2 Steps in Implementing Sample-Based Planning

The steps for implementing sample-based planning in RL are as follows:

1. **Initialize Q-values and Model**:
   - The agent initializes Q-values for each state-action pair, often with zeros or random values.
   - A model of the environment, often a table or dictionary, is also initialized. This model stores information about state-action transitions and rewards based on the agent's interactions.
2. **Real Experience Collection**:
   - As the agent interacts with the environment, it records the observed transitions, such as the current state, chosen action, resulting next state, and received reward. These experiences are stored in the model and used to approximate the environment's dynamics.
3. **Sample-Based Planning Updates**:
   - The agent randomly selects previously encountered state-action pairs from the model. For each sampled pair, the agent retrieves the transition details (next state and reward) and performs Q-value updates on these pairs.

   - The **Q-learning update rule** or other similar value-based update rules can be used to adjust Q-values based on simulated experiences, refining the agent's understanding of optimal actions without additional real interactions.
4. **Reinforcement Through Repeated Planning**:
   - This process of generating samples and updating Q-values is repeated multiple times for each real experience. By simulating additional scenarios, the agent effectively "revisits" previous actions and reinforces learning without requiring new, costly interactions.
5. **Repeat Until Convergence**:
   - The process is repeated for multiple episodes, with the agent iterating between real experiences and sample-based planning. This hybrid approach accelerates convergence to an optimal policy, as the agent benefits from both real-world experience and simulated planning.

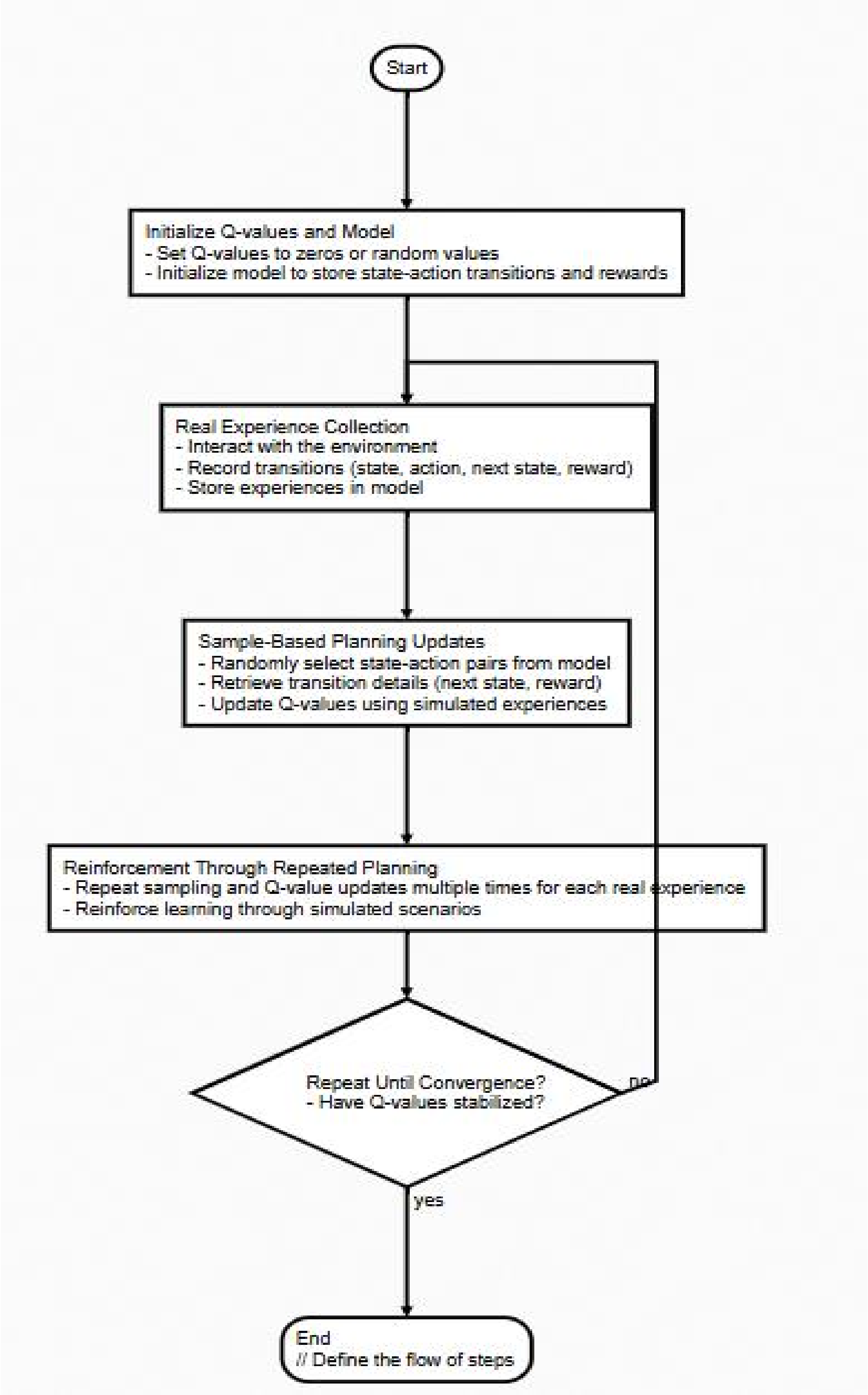

Start
Initialize Q-values and Model
- Set Q-values to zeros or random values
- Initialize model to store state-action transitions and rewards
Real Experience Collection
- Interact with the environment
- Record transitions (state, action, next state, reward)
- Store experiences in model
Sample-Based Planning Updates
- Randomly select state-action pairs from model
- Retrieve transition details (next state, reward)
- Update Q-values using simulated experiences
Reinforcement Through Repeated Planning
- Repeat sampling and Q-value updates multiple times for each real experience
- Reinforce learning through simulated scenarios
Repeat Until Convergence?
- Have Q-values stabilized?
no
yes
End
// Define the flow of steps

Figure 23 Sample-based Planning Flowchart [25]

6.4.3 Advantages and Limitations of Sample-Based Planning

Sample-based planning offers significant advantages, particularly in environments where the cost of real interactions is high or the state-action space is large:

- **Efficient Learning**: By generating simulated experiences, the agent can reinforce its policy without needing additional real interactions, making it ideal for situations where learning time or cost is a constraint.
- **Improved Exploration**: Sampling from a model enables the agent to explore various potential states and actions without direct risk, allowing for more thorough exploration of the environment.
- **Adaptability**: Sample-based planning allows the agent to respond to changes in the environment by simulating new scenarios, which is helpful for environments with unpredictable conditions.

However, sample-based planning also has limitations:

- **Dependence on Model Accuracy**: The effectiveness of sample-based planning depends heavily on the accuracy of the learned model. Errors in the model can lead to inaccurate Q-value updates and may result in suboptimal policy learning.
- **Increased Computational Complexity**: Generating multiple simulated experiences for each real interaction increases computational demands. In environments with large state spaces, this can be a limitation, as the planning updates require more processing power.

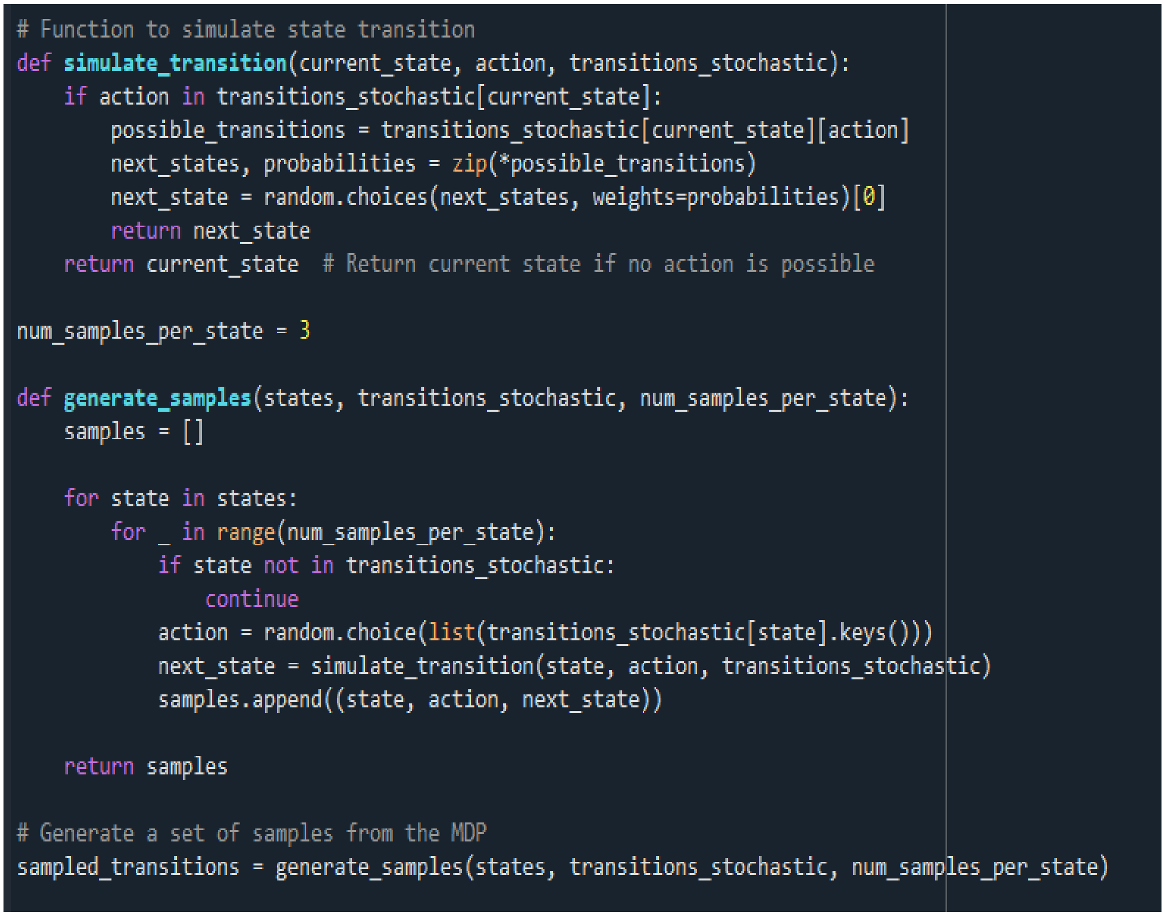
6.4.4 Application of Sample-Based Planning in the Industrial MDP Environment

In this research, sample-based planning is used within an industrial MDP environment to train a robot that performs complex, sequential tasks with occasional disruptions. Sample-based planning allows the robot to develop an efficient policy without requiring exhaustive real-world trials for each possible state-action pair.

Some specific benefits in this industrial MDP environment include:

- **Task Sequencing**: The robot can use sample-based planning to simulate various task sequences, such as moving from charging stations to assembly stations, quality control, and end-of-line inspection. By generating samples for these transitions, the robot efficiently learns optimal paths and sequences without repeating each task in real time.
- **Resilience to Disruptions**: With sample-based planning, the robot can simulate worst-case scenarios (e.g., battery depletion, urgent tasks) and update its policy based on these samples. This prepares the robot to handle unexpected events without requiring actual failures in the real environment.
- **Improved Resource Management**: Sample-based planning allows the robot to simulate different levels of resource availability, such as battery levels or component availability, and learn strategies to manage these constraints effectively. For example, the robot can learn when to prioritize charging or restocking actions based on simulated outcomes.

The use of sample-based planning in this industrial MDP environment enables the robot to learn more flexibly and responsively, as it can refine its policy based on simulated trials rather than relying solely on real interactions.



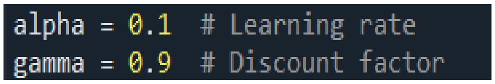
```python
# Function to simulate state transition
def simulate_transition(current_state, action, transitions_stochastic):
    if action in transitions_stochastic[current_state]:
        possible_transitions = transitions_stochastic[current_state][action]
        next_states, probabilities = zip(*possible_transitions)
        next_state = random.choices(next_states, weights=probabilities)[0]
        return next_state
    return current_state  # Return current state if no action is possible

num_samples_per_state = 3

def generate_samples(states, transitions_stochastic, num_samples_per_state):
    samples = []

    for state in states:
        for _ in range(num_samples_per_state):
            if state not in transitions_stochastic:
                continue
            action = random.choice(list(transitions_stochastic[state].keys()))
            next_state = simulate_transition(state, action, transitions_stochastic)
            samples.append((state, action, next_state))

    return samples

# Generate a set of samples from the MDP
sampled_transitions = generate_samples(states, transitions_stochastic, num_samples_per_state)
```

Figure 24 Generating Samples on python

6.5 Hyperparameters

The performance and efficiency of Q-learning depend heavily on three primary hyperparameters: **learning rate** α, **discount factor** γ, and **exploration rate** ϵ.

- **Learning Rate (α)**: The learning rate controls how much the agent's new experiences influence its Q-values. A high learning rate prioritizes recent experiences, while a low rate makes learning more conservative, averaging over multiple episodes. Optimal tuning of alpha (α) is critical; high values may lead to oscillations, while too low values slow down learning.



```python
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
```

Figure 25 Learning Rate and Discount Factor

- **Discount Factor (γ)**: The discount factor determines the importance of future rewards. With γ closer to 1, the agent values long-term rewards, ideal for complex tasks where cumulative rewards are more valuable. Lower γ values prioritize immediate rewards, which may be suitable for simpler tasks but can lead to short-sighted decision-making in complex environments.
- **Exploration Rate (ε)**: The exploration rate in epsilon-greedy strategies enables the agent to explore alternative actions. As learning progresses, decaying epsilon (ϵ) helps the agent shift towards exploiting its knowledge, gradually focusing on the actions it has identified as optimal.

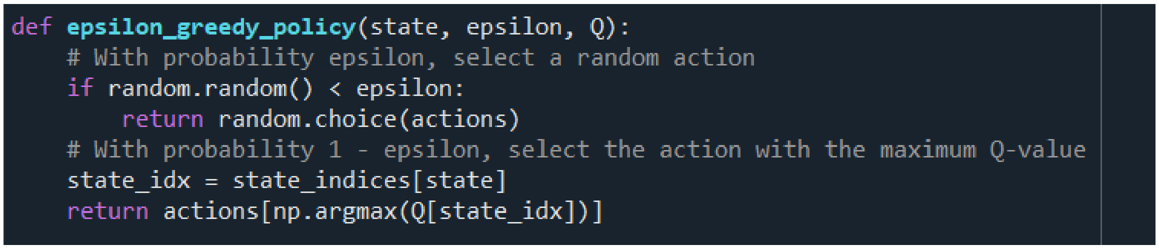
These parameters play a crucial role in **learning stability and convergence**, as improperly tuned values can cause either under-exploration or slow adaptation. Experimenting with and adjusting these parameters can lead to an optimal learning setup for different environments.

6.6 Exploration and Exploitation in Reinforcement Learning

In reinforcement learning, **exploration and exploitation** are crucial components that influence the agent’s decision-making process. Exploration involves the agent trying new actions to gather information about the environment, while exploitation focuses on selecting actions that maximize the agent's reward based on its current knowledge. Striking the right balance between these two aspects is essential for learning an optimal policy, where the agent not only learns effective actions but also adapts to dynamic environments and maximizes long-term rewards [27].

6.6.1 The Exploration-Exploitation Dilemma

The exploration-exploitation dilemma in reinforcement learning is the decision the agent must continually make between exploring new actions to improve its understanding of the environment and exploiting known actions that yield high rewards. This balance is critical, as excessive exploration can lead to inefficient, prolonged learning, whereas too much exploitation might prevent the agent from discovering better policies and adapting to new or dynamic conditions [27].



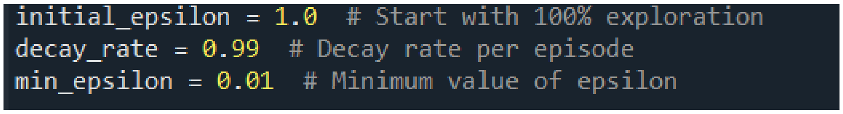
```python
def epsilon_greedy_policy(state, epsilon, Q):
    # With probability epsilon, select a random action
    if random.random() < epsilon:
        return random.choice(actions)
    # With probability 1 - epsilon, select the action with the maximum Q-value
    state_idx = state_indices[state]
    return actions[np.argmax(Q[state_idx])]
```

Figure 26 Epsilon-greedy Strategy

In this study, the **epsilon-greedy strategy** is employed to address this dilemma, allowing the agent to explore actions with a probability defined by the epsilon (ε) parameter while exploiting the best-known actions with a complementary probability (1 - ε) [27].

6.6.2 Epsilon-Greedy Strategy: Mechanism and Implementation

The epsilon-greedy strategy is a common approach used in Q-learning and other RL algorithms to control the balance between exploration and exploitation. This method allows the agent to explore a set percentage of the time while defaulting to exploitation for the remaining interactions. The probability ε gradually decays over time, meaning the agent explores less as it gains more experience and becomes more confident in its policy:



```python
initial_epsilon = 1.0  # Start with 100% exploration
decay_rate = 0.99  # Decay rate per episode
min_epsilon = 0.01  # Minimum value of epsilon
```

Figure 27 Epsilon Decay Rate

- **Initial Exploration (High Epsilon)**: At the start, ε is typically set to a high value (e.g., 1.0 or 0.9), encouraging exploration. In this phase, the agent chooses random actions to gather information about the rewards associated with various state-action pairs. This phase is essential for creating a comprehensive map of the environment, allowing the agent to develop a base understanding without being overly biased toward initial actions.
- **Gradual Reduction of Epsilon (Decay)**: As the agent gains more experience, ε decays, reducing the likelihood of random exploration and favoring exploitation. The decay rate is crucial and can follow either a **linear decay**, where ε decreases at

a constant rate, or an **exponential decay**, where ε reduces faster initially and slows down as it approaches a minimum threshold.

- **Final Stage (Low Epsilon)**: Toward the end of the learning process, ε reaches a small constant value (e.g., 0.01), which allows the agent to continue exploring minimally to handle any dynamic changes in the environment while primarily exploiting the learned policy. This final stage ensures that the agent maintains adaptability without frequent deviations from optimal actions.

The **decay rate of epsilon** affects the agent's ability to converge to an optimal policy. A slower decay may lead to more thorough exploration but can slow convergence, whereas a rapid decay can help achieve faster learning at the potential cost of suboptimal exploration.

6.6.3 Exploration Techniques Beyond Epsilon-Greedy

While the epsilon-greedy approach is effective, it has limitations in complex environments or where long-term exploration is beneficial. Some alternative techniques to balance exploration and exploitation are as follows [27]:

- **Boltzmann Exploration**: Also known as softmax action selection, this approach uses a temperature parameter to assign probabilities to each action based on its Q-value, allowing actions with higher Q-values to be chosen more frequently. This strategy is more refined than epsilon-greedy because it considers the Q-values when making exploratory decisions, favoring actions that are likely better but still occasionally exploring other options.
- **Upper Confidence Bound (UCB)**: UCB is a strategy commonly used in multi-armed bandit problems and RL, which factors in the confidence interval for each action's reward. Actions with higher uncertainty are chosen more often, especially early in the learning process, allowing the agent to focus on gathering more information where it is most beneficial.
- **Decay Schedules**: Beyond linear and exponential decay, adaptive schedules adjust ε based on the performance of the agent, where epsilon decays faster if the agent consistently finds high-reward actions or more slowly if uncertainty in rewards persists.

6.6.4 Implications of Exploration and Exploitation on Learning Efficiency

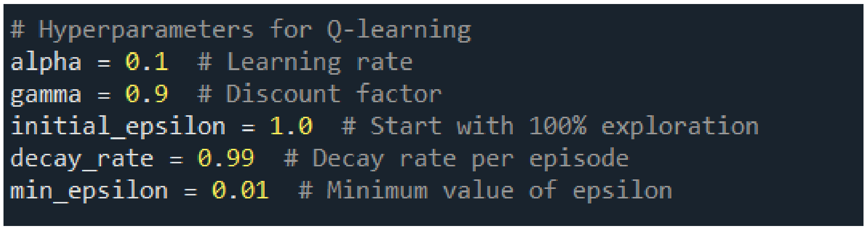
Balancing exploration and exploitation directly impact the agent's ability to learn efficiently and converge toward an optimal policy. In environments where dynamic elements, such as unpredictable obstacles, are present, the agent must learn to adjust its exploration patterns to account for these variations without losing efficiency.

In this research, an **epsilon decay strategy** is used to allow the agent to explore more in the initial learning stages and then shift gradually towards exploitation as it becomes more confident in its learned policy. By implementing this approach, the robot navigates through a complex environment, balancing short-term exploratory needs with the long-term objective of maximizing cumulative rewards.

- **Positive Impact of Exploration**: Adequate exploration helps prevent the agent from prematurely converging to suboptimal actions. It allows the robot to discover effective pathways, unexpected rewards, or better decision sequences that it might otherwise miss if focused solely on exploitation from the outset.
- **Risk of Over-Exploitation**: Excessive exploitation, especially in dynamic environments, can cause the robot to fail in adapting to new scenarios. This can lead to issues such as getting "stuck" in familiar routes or ignoring better paths due to lack of exploration, ultimately impacting the robot’s adaptability and efficiency.

6.6.5 Experimental Evaluation and Hyperparameter Tuning for Optimal Balance

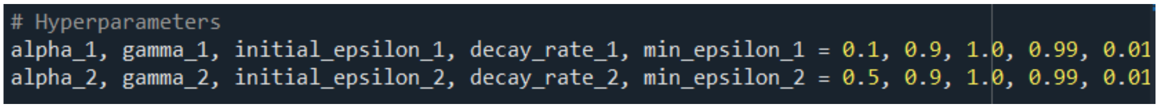
In this study, the performance of the epsilon-greedy exploration strategy is evaluated across various epsilon decay rates and initial epsilon values to identify an optimal configuration. Hyperparameters such as **initial epsilon value, decay rate, and minimum epsilon threshold** are tuned to ensure that the agent can learn effectively without excessive exploration:



```
# Hyperparameters for Q-learning
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
initial_epsilon = 1.0  # Start with 100% exploration
decay_rate = 0.99  # Decay rate per episode
min_epsilon = 0.01  # Minimum value of epsilon
```

Figure 28 Hyperparameters

- **Empirical Results**: The cumulative reward over episodes is analyzed to assess how the epsilon-greedy strategy affects the learning process. By experimenting with different decay schedules, the methodology identifies the configurations that yield the highest rewards while minimizing unnecessary exploratory actions in the later stages.
- **Hyperparameter Tuning**: Through iterative adjustments, the study finds that moderate initial epsilon values with exponential decay offer a balanced approach, allowing for quick adaptation in early learning stages while supporting stable exploitation in later phases.



```
# Hyperparameters
alpha_1, gamma_1, initial_epsilon_1, decay_rate_1, min_epsilon_1 = 0.1, 0.9, 1.0, 0.99, 0.01
alpha_2, gamma_2, initial_epsilon_2, decay_rate_2, min_epsilon_2 = 0.5, 0.9, 1.0, 0.99, 0.01
```

Figure 29 Hyperparameters Tuning

6.6.6 Practical Implications and Future Considerations

Effective exploration-exploitation strategies are particularly beneficial in environments where the agent faces varied tasks and uncertainties, such as in industrial setups. The insights gained from this study's epsilon-greedy implementation demonstrate that a well-tuned exploration policy can support resilient, autonomous learning in dynamic scenarios. For future research, exploring combinations of epsilon-greedy with advanced exploration techniques, such as UCB or Boltzmann, could further enhance the robot’s adaptability and robustness in more complex environments.

Chapter 7: Implementation

The study defines two key MDP environments for this purpose:

1. **Navigation to Free Position Environment**
2. **Task Sequence Environment**

Each environment is detailed with specific states and transitions that allow the robot to perform various actions, learning through rewards and penalties how to achieve optimal task completion. The next sections break down each environment, explaining the core components of the MDP and the challenges the robot encounters.

7.1 Navigation to Free Position Environment

This environment serves as the basis for training the robot to navigate between free and occupied spaces. The **MDP-Based Navigation Model for Mobile Robots** (see **figure 30** for visualization) illustrates the possible states and transitions, showing how the robot identifies and moves to available positions while avoiding areas occupied by humans.

7.1.1 States and Transitions

- **S0 (Initial Position)**: The robot begins here, starting each navigation process.
- **SA, SB, SC (Free Positions)**: These are the target positions where the robot can safely move. Each state represents an open space that is unoccupied and accessible.
- **HA, HB, HC (Occupied Positions)**: These states indicate areas currently occupied by humans. The robot must avoid these positions to prioritize safety and avoid interruptions.
- **P (Task State)**: This state represents the robot actively performing a task, which might be located near one of the free positions.

Transitions between these states are determined by the robot's actions, which include:

- **Action (F)**: The robot moves to a free position. For example, starting from S0, it can move to SA, SB, or SC.

- **Action (O)**: The robot attempts to move to an occupied position, but this action is typically discouraged by assigning penalties to reduce the likelihood of the robot attempting it again.

From any **free position** (SA, SB, SC), the robot can transition to **state P** to perform a task. Once completed, it may return to the initial position (S0) to begin a new navigation sequence. Additionally, transitions can occur between occupied and free positions, representing the robot's adaptive capabilities in handling different spatial configurations.

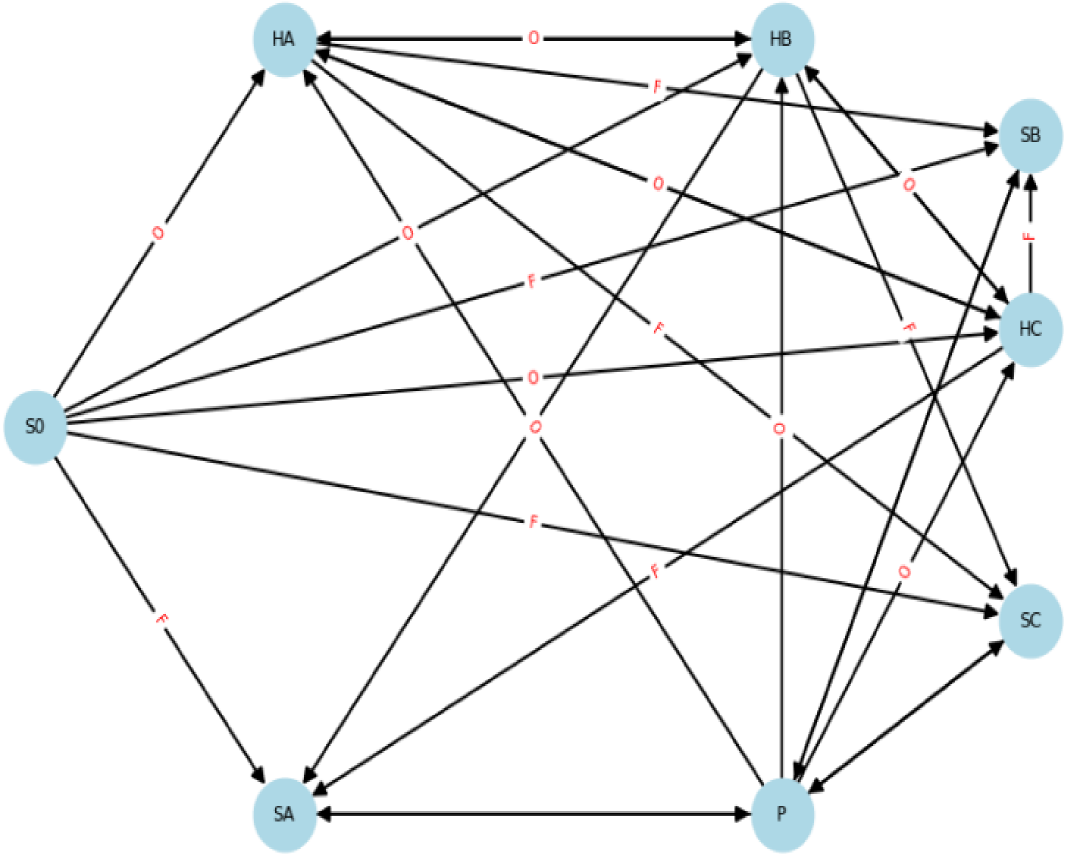
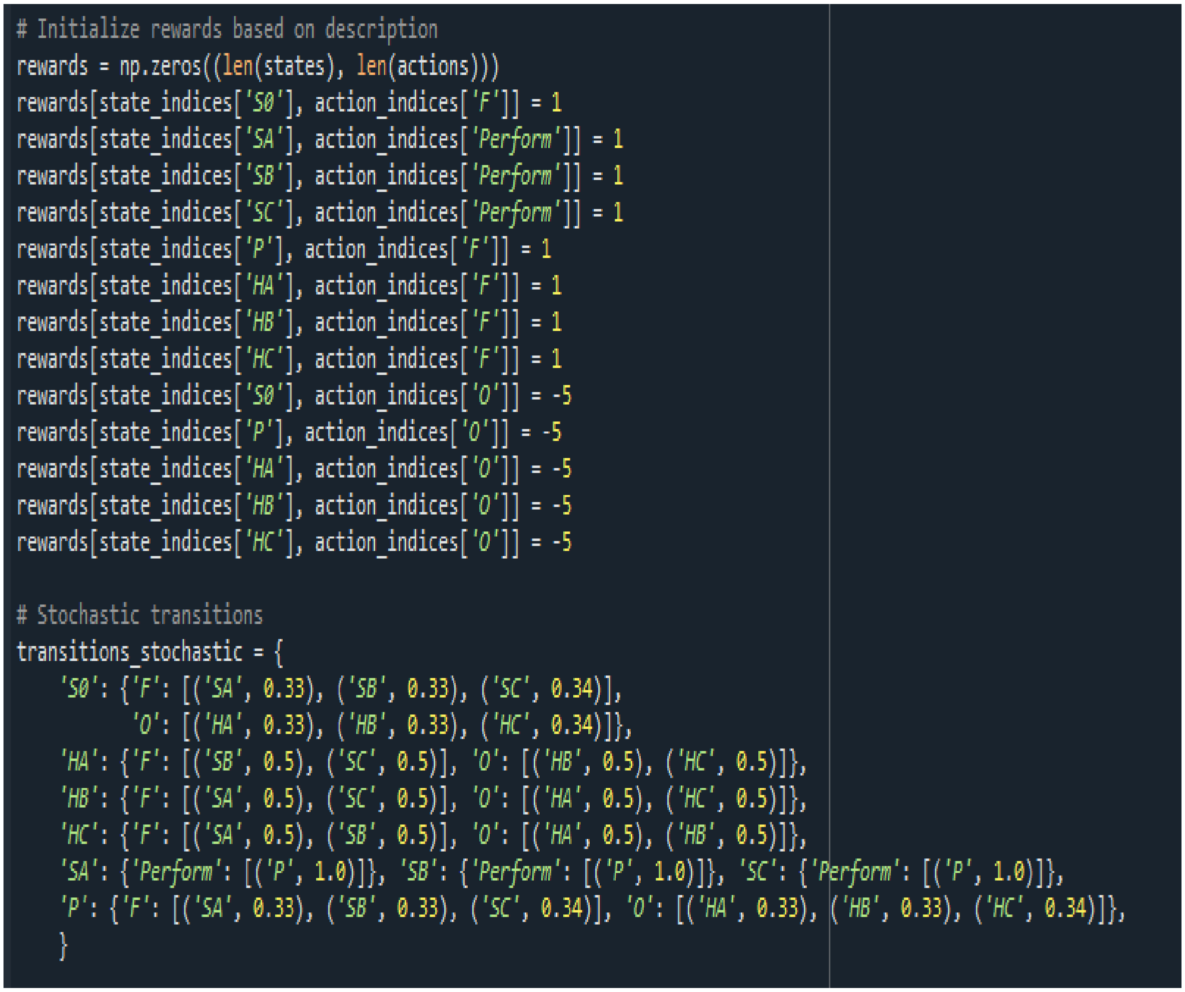


Figure 30 MDP Navigation Environment



```
# Initialize rewards based on description
rewards = np.zeros((len(states), len(actions)))
rewards[state_indices['S0'], action_indices['F']] = 1
rewards[state_indices['SA'], action_indices['Perform']] = 1
rewards[state_indices['SB'], action_indices['Perform']] = 1
rewards[state_indices['SC'], action_indices['Perform']] = 1
rewards[state_indices['P'], action_indices['F']] = 1
rewards[state_indices['HA'], action_indices['F']] = 1
rewards[state_indices['HB'], action_indices['F']] = 1
rewards[state_indices['HC'], action_indices['F']] = 1
rewards[state_indices['S0'], action_indices['O']] = -5
rewards[state_indices['P'], action_indices['O']] = -5
rewards[state_indices['HA'], action_indices['O']] = -5
rewards[state_indices['HB'], action_indices['O']] = -5
rewards[state_indices['HC'], action_indices['O']] = -5

# Stochastic transitions
transitions_stochastic = {
    'S0': {'F': [('SA', 0.33), ('SB', 0.33), ('SC', 0.34)],
           'O': [('HA', 0.33), ('HB', 0.33), ('HC', 0.34)]},
    'HA': {'F': [('SB', 0.5), ('SC', 0.5)], 'O': [('HB', 0.5), ('HC', 0.5)]},
    'HB': {'F': [('SA', 0.5), ('SC', 0.5)], 'O': [('HA', 0.5), ('HC', 0.5)]},
    'HC': {'F': [('SA', 0.5), ('SB', 0.5)], 'O': [('HA', 0.5), ('HB', 0.5)]},
    'SA': {'Perform': [('P', 1.0)]}, 'SB': {'Perform': [('P', 1.0)]}, 'SC': {'Perform': [('P', 1.0)]},
    'P': {'F': [('SA', 0.33), ('SB', 0.33), ('SC', 0.34)], 'O': [('HA', 0.33), ('HB', 0.33), ('HC', 0.34)]},
    }
```

Figure 31 Rewards Table and Stochastic Transitions for Navigation Environment

7.1.2 Flexibility in Navigation

The flexible transitions in this MDP allow the robot to make autonomous decisions about moving, interacting, and performing tasks across the environment. By using rewards for successful navigation and penalties for attempting to move to occupied spaces, the model helps the robot learn optimal navigation paths, improving efficiency and adaptability.

7.2 Task Sequence Environment

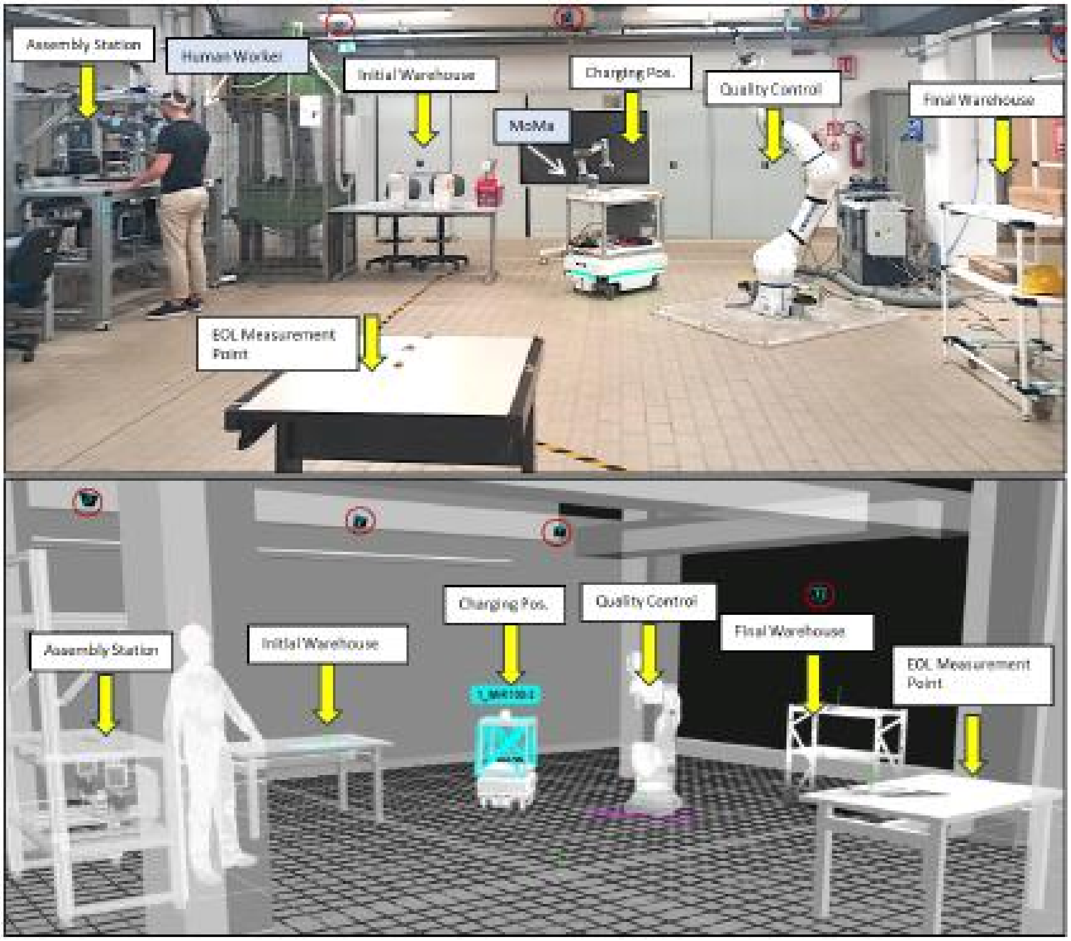


Figure 32 Task Environment in The Laboratory

The **Task Sequence Environment** simulates a complex task progression in an industrial setting, where the robot moves through various stages, including charging, assembly, quality control, and end-of-line inspection. The Environment layout consists of six stations, each with specific tasks for the mobile robot:

1. **Charging Station**: This is the starting and ending point of the production line, where the mobile robot waits for its tasks. If the robot is not at the Charging Station when instructed to begin, it will automatically navigate there before starting the first task. If the robot's battery drops below a set threshold during operation, it will halt its current task, move to the Charging Station, and wait until the battery reaches an appropriate level before continuing its work.

2. **Initial Warehouse**: At this station, the mobile robot retrieves skateboard parts and places them onto its shelf. The process is divided into two tasks: first, the robot picks the skateboard from a designated spot in the warehouse; then, it moves to the location where the skateboard wheels are stored and picks them up. Once all components are collected and placed on the shelf, the robot transports them to the next station.
3. **Assembly Station**: This is the first station where the robot interacts with a human worker. The robot delivers the parts to the worker, who will then begin assembling the skateboard. Before starting the assembly, the worker will give the robot tools to be transported to the assigned station. While the worker completes the assembly, the robot returns to wait for the finished skateboard, which it will then transport to the next station.
4. **End of Line Measurement Point (EOL) Station**: At this station, the mobile robot performs two tasks. The first task is to drop off the tools that the operator had previously provided. Then, the robot returns with the assembled skateboard. This marks the first collaboration between the human and the robot. Using the tools delivered earlier, the human inspects the skateboard for any obvious errors. During this process, the mobile robot assists by holding the skateboard for the operator. Once the inspection is completed, if the skateboard passes the check, both the operator and the robot continue with their respective tasks at their assigned stations. However, if the inspection identifies issues, the worker informs the robot, which will then return to the Assembly Station for adjustments.
5. **Quality Control Station**: This station marks the second and final collaboration between the human and the robot. The mobile robot repeats the same task as before, holding the assembled skateboard for the operator, who now performs a more thorough quality check. If the skateboard passes this inspection, the operator moves back to the Assembly Station, and the robot proceeds to the next station. If any issues are found during quality control, the operator halts the production line to address the problems or discard the defective product.

6. **Final Warehouse Station**: The final station in the production line is where the mobile robot deposits the completed skateboard. Afterward, the robot returns to the Charging Station to recharge and prepare for the next cycle. While the robot is returning to the charging point, the operator proceeds to the Final Warehouse to pack the product.

This environment builds on the **MDP structure** by incorporating possible **worst-case scenarios**, adding complexity and enhancing the robot's ability to respond to unexpected challenges.

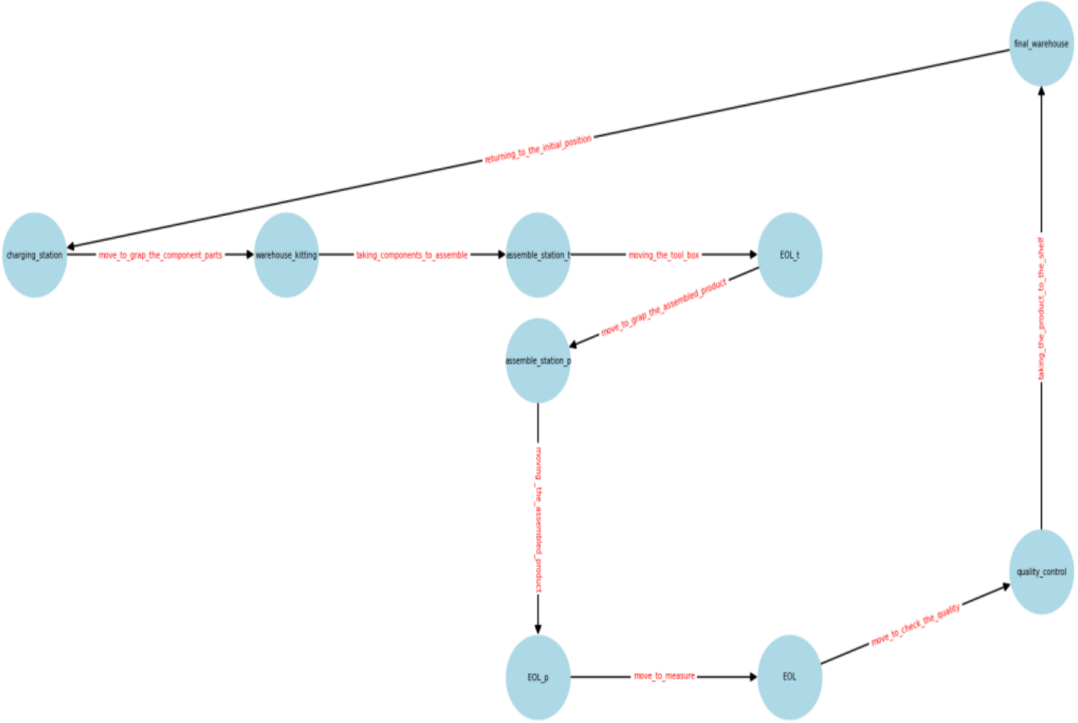

final_warehouse
returning_to_the_initial_position
charging_station
move_to_grap_the_component_parts
warehouse_kitting
taking_components_to_assemble
assemble_station_t
moving_the_tool_box
EOL_t
move_to_grap_the_assembled_product
assemble_station_p
taking_the_product_to_the_shelf
moving_the_assembled_product
quality_control
move_to_check_the_quality
EOL_p
move_to_measure
EOL

Figure 33 MDP for Original Task Sequence

By adding the possible worst-cases that could engage and affect the operation. The MDP transformed to be the below form.

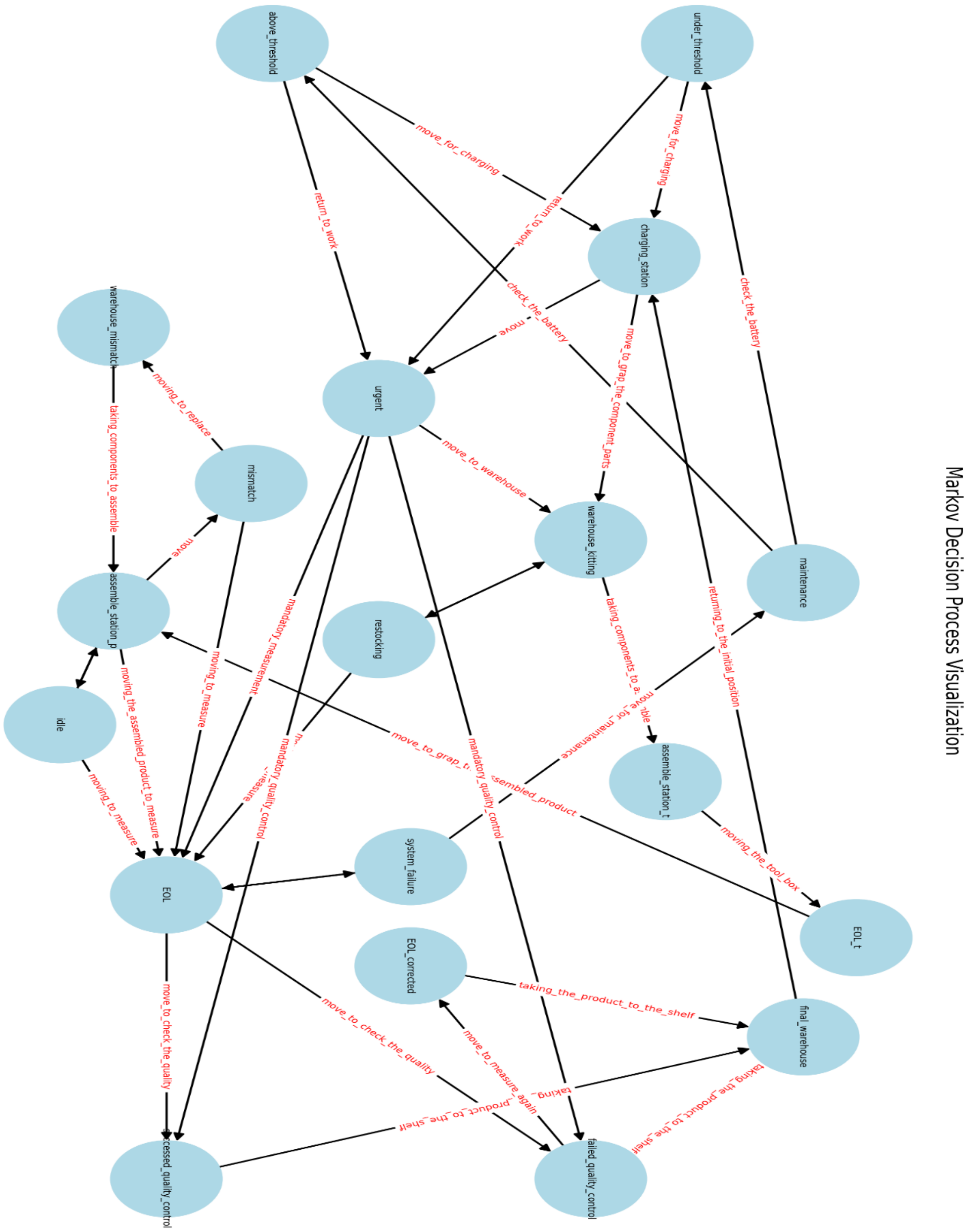
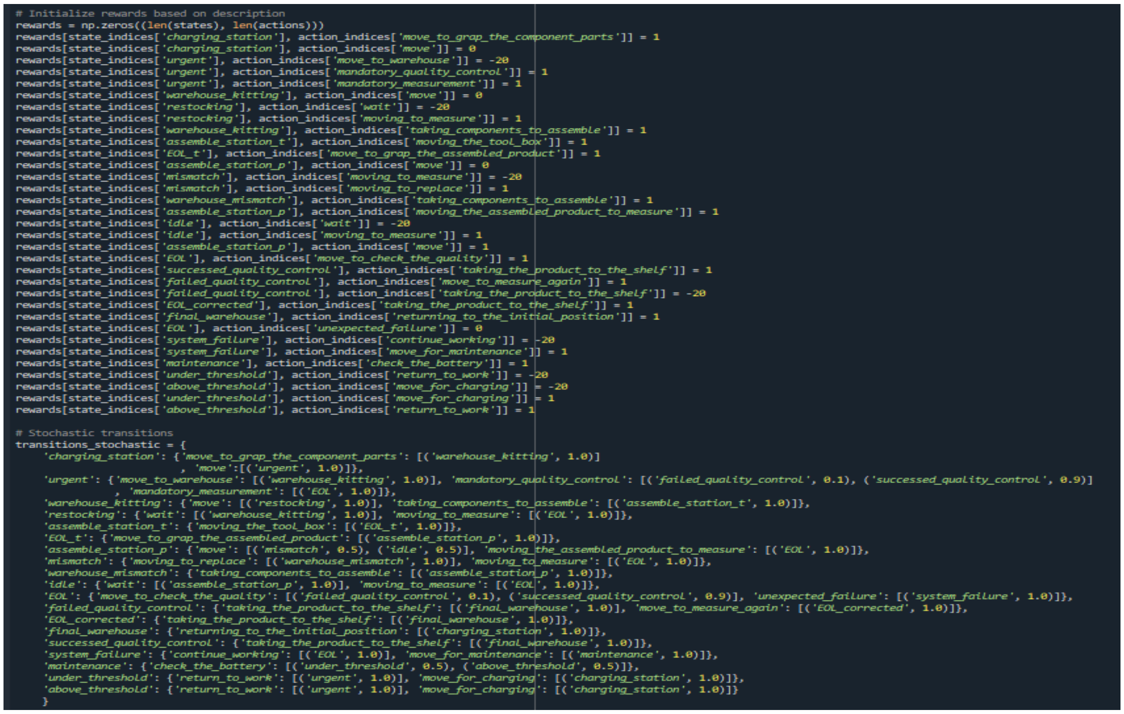


Figure 34 Markov Decision Process for Robot Task Management and Quality Control



```
# Initialize rewards based on description
rewards = np.zeros((len(states), len(actions)))
rewards[state_indices['charging_station'], action_indices['move_to_grap_the_component_parts']] = 1
rewards[state_indices['charging_station'], action_indices['move']] = 0
rewards[state_indices['urgent'], action_indices['move_to_warehouse']] = -20
rewards[state_indices['urgent'], action_indices['mandatory_quality_control']] = 1
rewards[state_indices['urgent'], action_indices['mandatory_measurement']] = 1
rewards[state_indices['warehouse_kitting'], action_indices['move']] = 0
rewards[state_indices['restocking'], action_indices['wait']] = -20
rewards[state_indices['restocking'], action_indices['moving_to_measure']] = 1
rewards[state_indices['warehouse_kitting'], action_indices['taking_components_to_assemble']] = 1
rewards[state_indices['assemble_station_t'], action_indices['moving_the_tool_box']] = 1
rewards[state_indices['EOL_t'], action_indices['move_to_grap_the_assembled_product']] = 1
rewards[state_indices['assemble_station_p'], action_indices['move']] = 0
rewards[state_indices['mismatch'], action_indices['moving_to_measure']] = -20
rewards[state_indices['mismatch'], action_indices['moving_to_replace']] = 1
rewards[state_indices['warehouse_mismatch'], action_indices['taking_components_to_assemble']] = 1
rewards[state_indices['assemble_station_p'], action_indices['moving_the_assembled_product_to_measure']] = 1
rewards[state_indices['idle'], action_indices['wait']] = -20
rewards[state_indices['idle'], action_indices['moving_to_measure']] = 1
rewards[state_indices['assemble_station_p'], action_indices['move']] = 1
rewards[state_indices['EOL'], action_indices['move_to_check_the_quality']] = 1
rewards[state_indices['successed_quality_control'], action_indices['taking_the_product_to_the_shelf']] = 1
rewards[state_indices['failed_quality_control'], action_indices['move_to_measure_again']] = 1
rewards[state_indices['failed_quality_control'], action_indices['taking_the_product_to_the_shelf']] = -20
rewards[state_indices['EOL_corrected'], action_indices['taking_the_product_to_the_shelf']] = 1
rewards[state_indices['final_warehouse'], action_indices['returning_to_the_initial_position']] = 1
rewards[state_indices['EOL'], action_indices['unexpected_failure']] = 0
rewards[state_indices['system_failure'], action_indices['continue_working']] = -20
rewards[state_indices['system_failure'], action_indices['move_for_maintenance']] = 1
rewards[state_indices['maintenance'], action_indices['check_the_battery']] = 1
rewards[state_indices['under_threshold'], action_indices['return_to_work']] = -20
rewards[state_indices['above_threshold'], action_indices['move_for_charging']] = -20
rewards[state_indices['under_threshold'], action_indices['move_for_charging']] = 1
rewards[state_indices['above_threshold'], action_indices['return_to_work']] = 1

# Stochastic transitions
transitions_stochastic = {
    'charging_station': {'move_to_grap_the_component_parts': [('warehouse_kitting', 1.0)]
                        , 'move':[('urgent', 1.0)]},
    'urgent': {'move_to_warehouse': [('warehouse_kitting', 1.0)], 'mandatory_quality_control': [('failed_quality_control', 0.1), ('successed_quality_control', 0.9)]
              , 'mandatory_measurement': [('EOL', 1.0)]},
    'warehouse_kitting': {'move': [('restocking', 1.0)], 'taking_components_to_assemble': [('assemble_station_t', 1.0)]},
    'restocking': {'wait': [('warehouse_kitting', 1.0)], 'moving_to_measure': [('EOL', 1.0)]},
    'assemble_station_t': {'moving_the_tool_box': [('EOL_t', 1.0)]},
    'EOL_t': {'move_to_grap_the_assembled_product': [('assemble_station_p', 1.0)]},
    'assemble_station_p': {'move': [('mismatch', 0.5), ('idle', 0.5)], 'moving_the_assembled_product_to_measure': [('EOL', 1.0)]},
    'mismatch': {'moving_to_replace': [('warehouse_mismatch', 1.0)], 'moving_to_measure': [('EOL', 1.0)]},
    'warehouse_mismatch': {'taking_components_to_assemble': [('assemble_station_p', 1.0)]},
    'idle': {'wait': [('assemble_station_p', 1.0)], 'moving_to_measure': [('EOL', 1.0)]},
    'EOL': {'move_to_check_the_quality': [('failed_quality_control', 0.1), ('successed_quality_control', 0.9)], 'unexpected_failure': [('system_failure', 1.0)]},
    'failed_quality_control': {'taking_the_product_to_the_shelf': [('final_warehouse', 1.0)], 'move_to_measure_again': [('EOL_corrected', 1.0)]},
    'EOL_corrected': {'taking_the_product_to_the_shelf': [('final_warehouse', 1.0)]},
    'final_warehouse': {'returning_to_the_initial_position': [('charging_station', 1.0)]},
    'successed_quality_control': {'taking_the_product_to_the_shelf': [('final_warehouse', 1.0)]},
    'system_failure': {'continue_working': [('EOL', 1.0)], 'move_for_maintenance': [('maintenance', 1.0)]},
    'maintenance': {'check_the_battery': [('under_threshold', 0.5), ('above_threshold', 0.5)]},
    'under_threshold': {'return_to_work': [('urgent', 1.0)], 'move_for_charging': [('charging_station', 1.0)]},
    'above_threshold': {'return_to_work': [('urgent', 1.0)], 'move_for_charging': [('charging_station', 1.0)]}
    }
```

Figure 35 Rewards and Stochastic Transitions for Task Sequence Environment

Figure 33 illustrates the task sequence, showing the standard workflow the robot follows. By adding new states and transitions for worst-case scenarios, the model evolves into a robust framework for training the robot in a realistic environment as shown in **figure 34**.

7.2.1 Worst-Case Scenario States and Transitions

The extended MDP model incorporates states that represent potential failures or interruptions in the workflow, training the robot to adapt by recognizing and handling each unique situation. Key worst-case scenario states include:

- **System Failure**: A "system_failure" state indicates a critical malfunction requiring the robot to stop operations and move to a maintenance state. This state simulates system downtimes and helps train the robot to handle unexpected breakdowns.
- **Urgent Tasks**: The "urgent" state prioritizes specific tasks requiring immediate attention. For example, if an urgent task arises while the robot is en route to another state, it must adapt and address the high-priority task first.
- **Quality Control Failure**: The states "failed_quality_control" and "move_to_measure_again" indicate that the robot has failed quality control checks.

The robot must then repeat or remeasure certain parameters before proceeding, introducing a realistic quality assurance process.

- **Battery Thresholds**: The states “under_threshold” and “above_threshold” represent battery levels. If battery levels are low, the robot must move to the charging station, simulating resource management and ensuring the robot does not run out of power mid-task.
- **Mismatch**: The “mismatch” state reflects component errors. If the robot encounters an incorrect or mismatched component, it must correct this by moving back to replace it, ensuring the accuracy of assembled products.
- **Idle State**: An “idle” state occurs if the robot is inactive, indicating inefficiencies in task management. Training the robot to minimize idle time enhances productivity.
- **Restocking**: The “restocking” state represents a scenario where resources or components need replenishing. Failures in restocking can delay tasks, so the robot must learn to handle this effectively.

Placement of Updated MDP Visualization: Consider including an updated MDP visualization here to show how the model handles these added complexities in worst-case scenarios.

7.2.2 Complete List of States and Actions in Task Sequence Environment

The expanded task environment incorporates all possible states and actions to simulate a realistic industrial setting. Below is a detailed breakdown of key states and actions:

States:

- **under_threshold**: Battery/resource levels are below a threshold.
- **ChargingStation**: The robot has reached a charging station.
- **maintenance**: Indicates the robot is undergoing maintenance.
- **warehouse_kitting**: The robot is gathering components in the warehouse.
- **urgent**: An urgent task requires immediate attention.
- **assemble_station_t**: Robot is at an assembly station, taking tools to EOL.

- **assemble_station_p**: Robot is transporting an assembled product to EOL.
- **EOL_t**: Robot places the toolbox at EOL.
- **EOL_corrected**: Robot has corrected a product.
- **EOL**: General end-of-line inspection state.
- **final_warehouse**: Robot has reached the final warehouse.
- **failed_quality_control**: The product has failed quality control.
- **system_failure**: A critical system failure has occurred.
- **above_threshold**: Battery/resource levels are above a threshold.
- **restocking**: Robot is restocking components in the warehouse.
- **IDLE**: Robot is not performing any task.
- **warehouse_mismatch**: A mismatch occurred in the warehouse.
- **mismatch**: General mismatch or error in component handling.
- **successed_quality_control**: Robot successfully passed quality control.

Actions:

- **move_for_charging**: Move to Charging Station.
- **return_to_work**: Resume work after charging/maintenance.
- **check_the_battery**: Check current battery level.
- **move_to_grab_the_component_parts**: Go to warehouse for parts.
- **Taking components to assemble**: Bring parts to assembly station.
- **mandatory_quality_control**: Perform required quality control.
- **measure_to_move**: Perform specific measurements.
- **Take the product to the shelf**: Place the finished product on the shelf.
- **moving_the_tool_box**: Move the toolbox.
- **returning_to_the_initial_position**: Return to starting point.
- **moving_to_replace**: Replace mismatched components.
- **Move_to_check_quality**: Perform quality inspection.
- **move_for_maintenance**: Go to maintenance for repair.

This complete list of states and actions ensures the robot can perform every task required, from gathering resources and assembly to handling maintenance and quality control.

Chapter 8: Simulation Results

8.1 Navigation to Free Position Environment

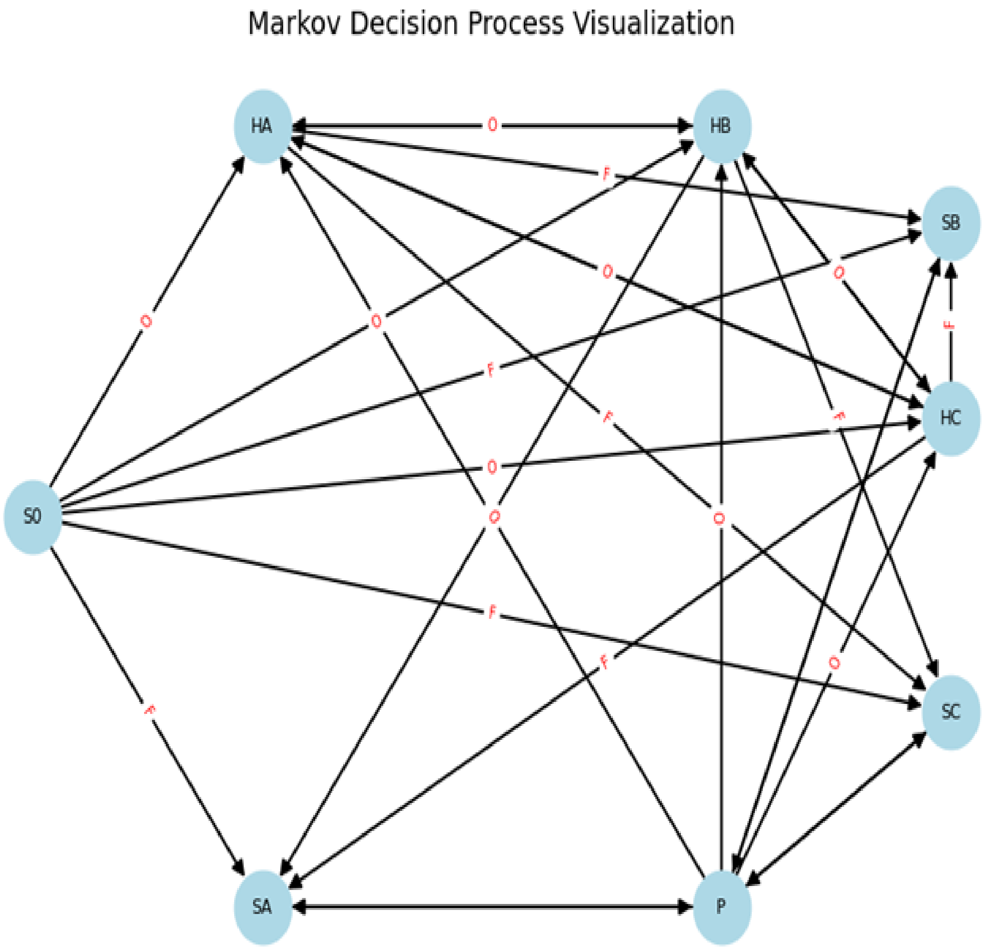
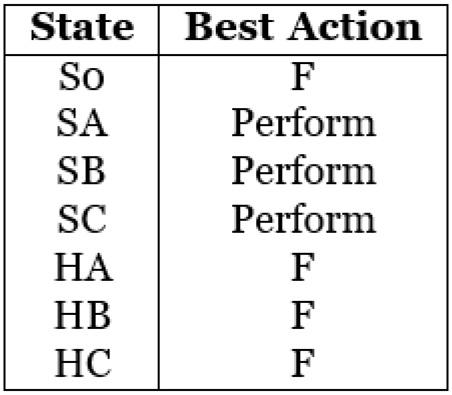


Figure 36 MDP Navigation Environment

The optimal actions were provided by Dyna-Q, and Sample-based planning with Q-learning. Through this process, the action-value function is updated, and through these simulated experiences, it plans to maximize the cumulative reward to achieve the best possible policy that is shown in **table 1.**



State	Best Action
So	F
SA	Perform
SB	Perform
SC	Perform
HA	F
HB	F
HC	F

Table 1 Optimal Policy for Free Navigation MDP

In reinforcement learning, cumulative reward over episodes provides one of the most valuable indicators for gauging learning efficiency and effectiveness in an algorithm. The performance of the Dyna-Q algorithm over three different of episodes (500, 1000 and 50000) is compared to understand how the learning process evolves with growing numbers of training episodes.

Figure 37 shows that there is fluctuation at the plateau. This is entirely expected since we are currently in the initial exploration phase within the environment, and the method has not collected enough experiences to settle down. Rewards might show some general tendency to go up, but the volatility means it is far from close to optimal.

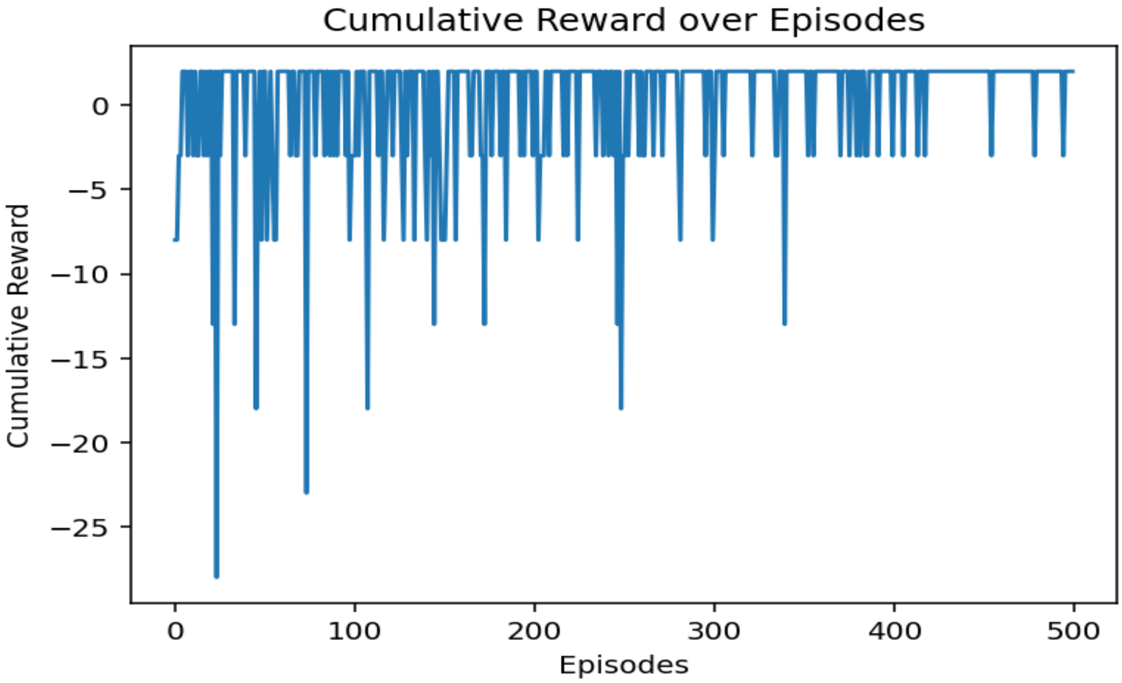


Figure 37 Cumulative Reward Over 500 Episodes for Dyna-Q Algorithm

In **figure 38**, the cumulative reward graph didn't show any changes with 1,000 episodes compared to the 500 episodes scenario. Fluctuations are there, but the trend is slightly consistent; that is, the Dyna-Q algorithm has started to exploit its knowledge of the environment better.

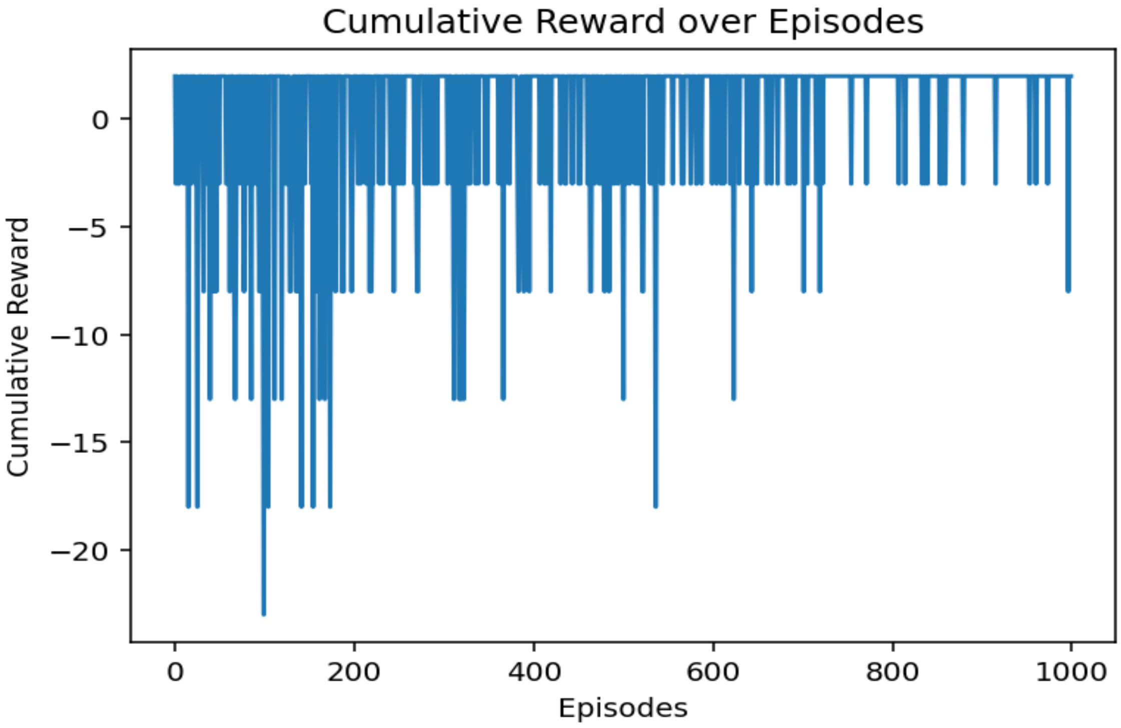


Figure 38 Cumulative Reward Over 1000 Episodes for Dyna-Q Algorithm

Finally, in **figure 39**, with 50,000 episodes, the cumulative graph would be the most stable and highest compared to all scenarios. Such an enormous number of episodes assures that the Dyna-Q algorithm gets enough opportunity to finetune its policy for maximum exploitation of the rewards in the environment. The graph has minimal fluctuations, with cumulative rewards approaching a plateau, which indicates that the policy is converging to be close to the optimal policy. Moreover, In the cumulative reward plot for 50000 episodes, after training, rewards in the last episode are still negative. This indicates that the Dyna-Q algorithm has failed to converge to positive rewards which means that the agent didn't finished exploring states with negative rewards. Thus, the agent keeps getting punished rather than obtaining positive accumulated rewards.

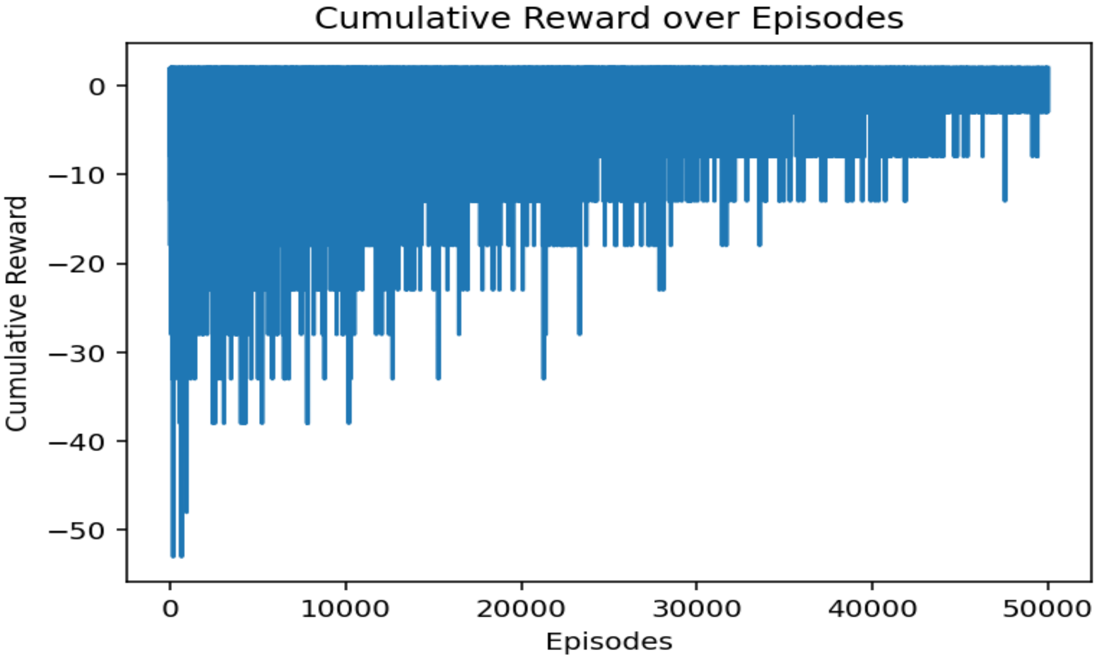


Figure 39 Cumulative Reward Over 50,000 Episodes for Dyna-Q Algorithm

The graph below depicts agent performance on a sample-based reinforcement learning algorithm over 500 episodes using Q-learning algorithm. Examination of this graph gives some sense of the learning dynamics, effectiveness, and stability of the algorithm.

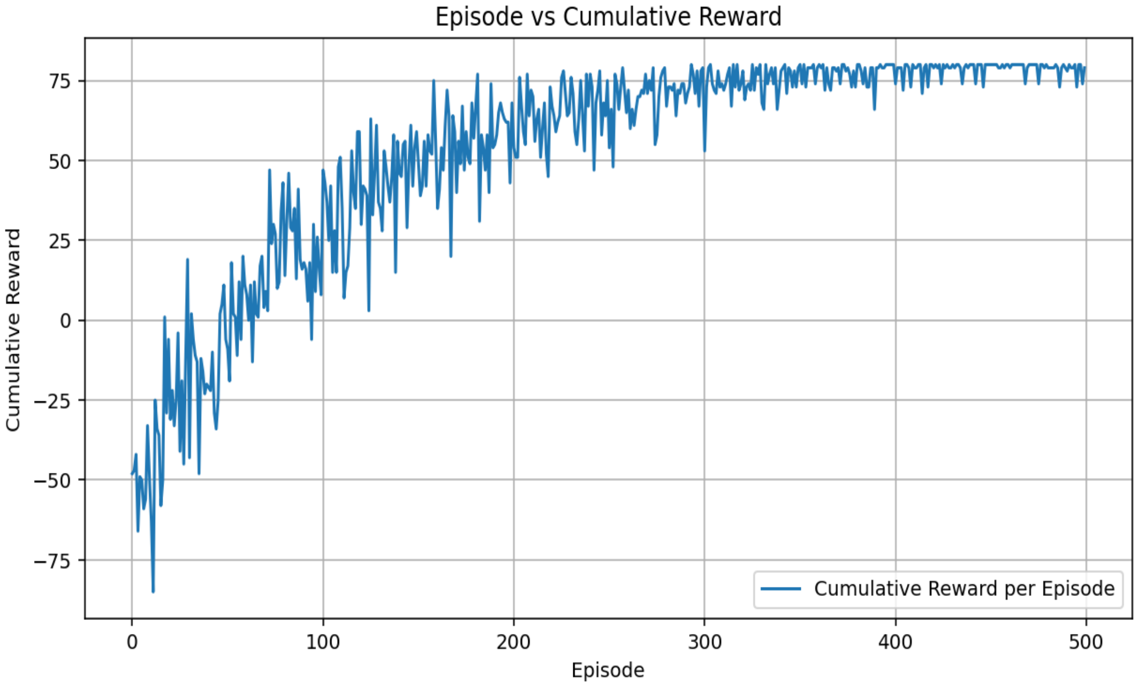


Figure 40 Cumulative Reward Over 500 Episodes for Sample-based planning

Initially, the cumulative reward has a very high variance, also with negative values. This is consistent with an agent of high exploratory character interacting with the environment. The high variance in rewards suggests the agent may be acting to sample a variety of actions to gain information about the dynamics and reward structure. As the number of episodes increases, a marked upward trend can be seen from the graph, indicating that the agent is now learning an effective policy. The fluctuations in cumulative reward are gradually decreasing, even though they are still present. It is a phase with a mixture of exploration and exploitation; the agent is now refining its policy based on all the experiences gathered. The fact that the graph generally slopes positively demonstrates agent performance improvement because cumulative rewards are monotonously increasing. The cumulative reward graph gets steadier, and fluctuations decrease in the later episodes. It reaches a return value of approximately 80, which hints that the agent has converged to some relatively optimal policy. The return values mainly exploit the learned policy, as judged by the reduced reward variance, indicating it is making consistent decisions to achieve high returns.

8.2 Task Sequence Environment

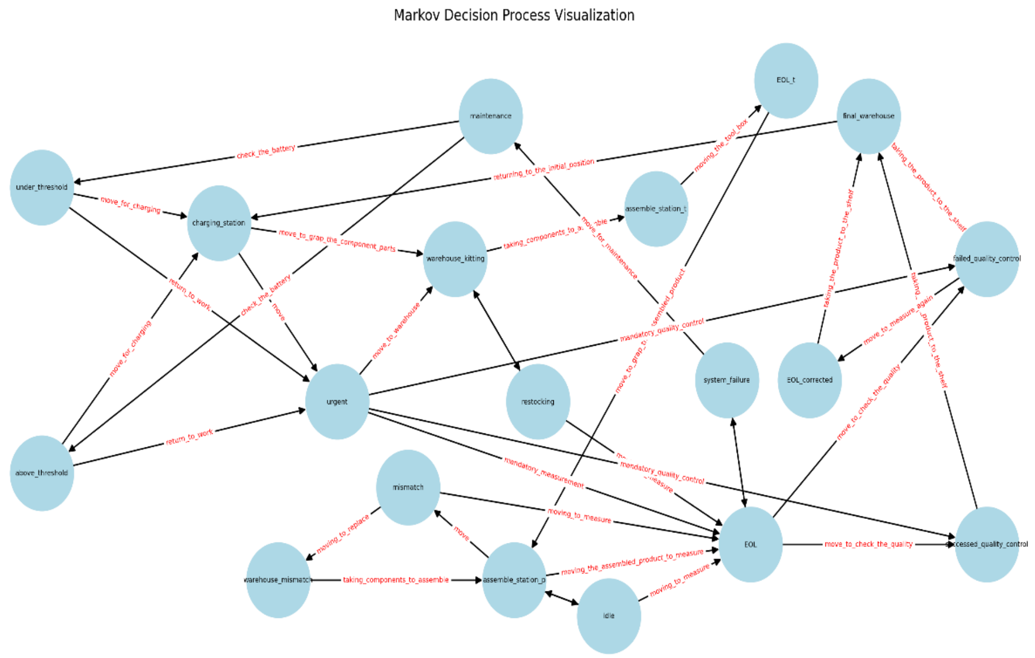


Figure 41 Markov Decision Process for Robot Task Management and Quality Control

Simulations of how well the policy works, using Sample-based planning with SARSA and Q-learning algorithms, are shown through the results in their respective graphs.

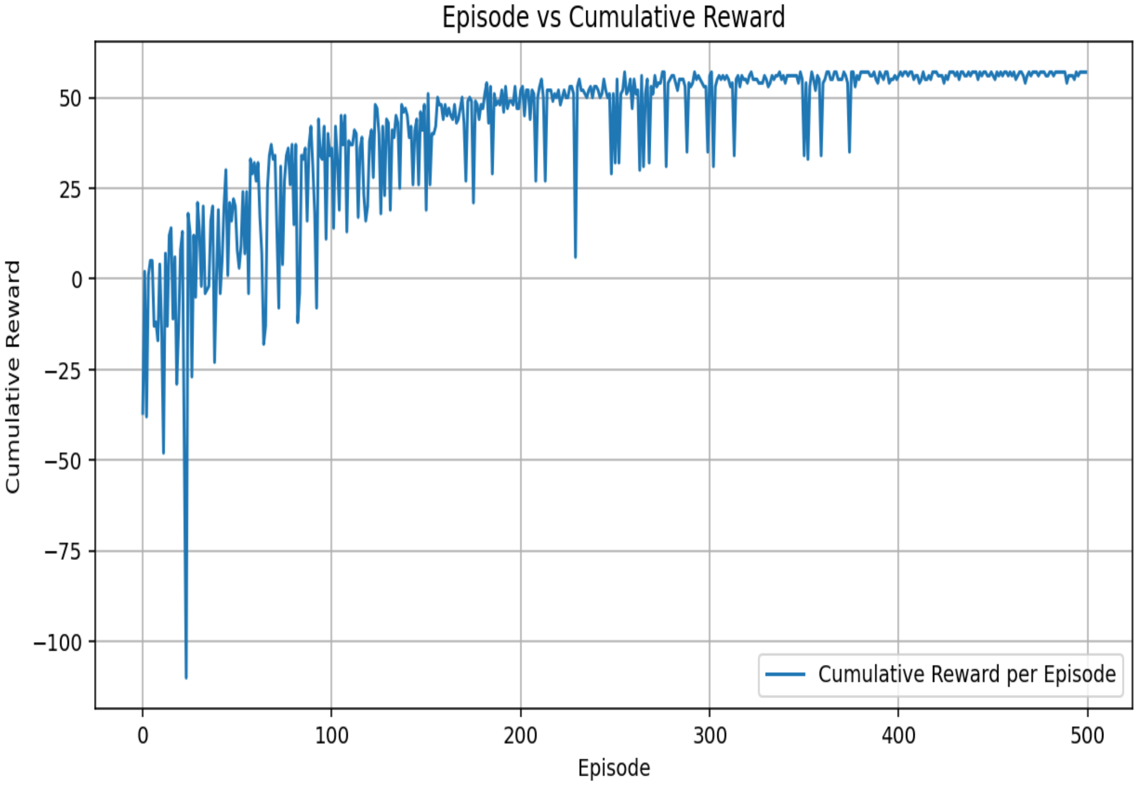


Figure 42 Task Sequence Episode vs Cumulative Reward for SARSA Algorithm

Figure 42 presents the results for the cumulative reward over 500 episodes while running the SARSA algorithm. In the early episodes, the cumulative reward changes considerably; thus, it is recognizable that there is an exploration of actions and their outcomes. Further on in the episodes, it keeps growing, which means that the policy is improving. From episode 200 onwards, the rewards stabilize, which means the agent picks up significant reward-returning actions. As can be seen, the cumulative reward saturates beyond 500 episodes, thus proving that the policy has converged and can efficiently navigate through an MDP.

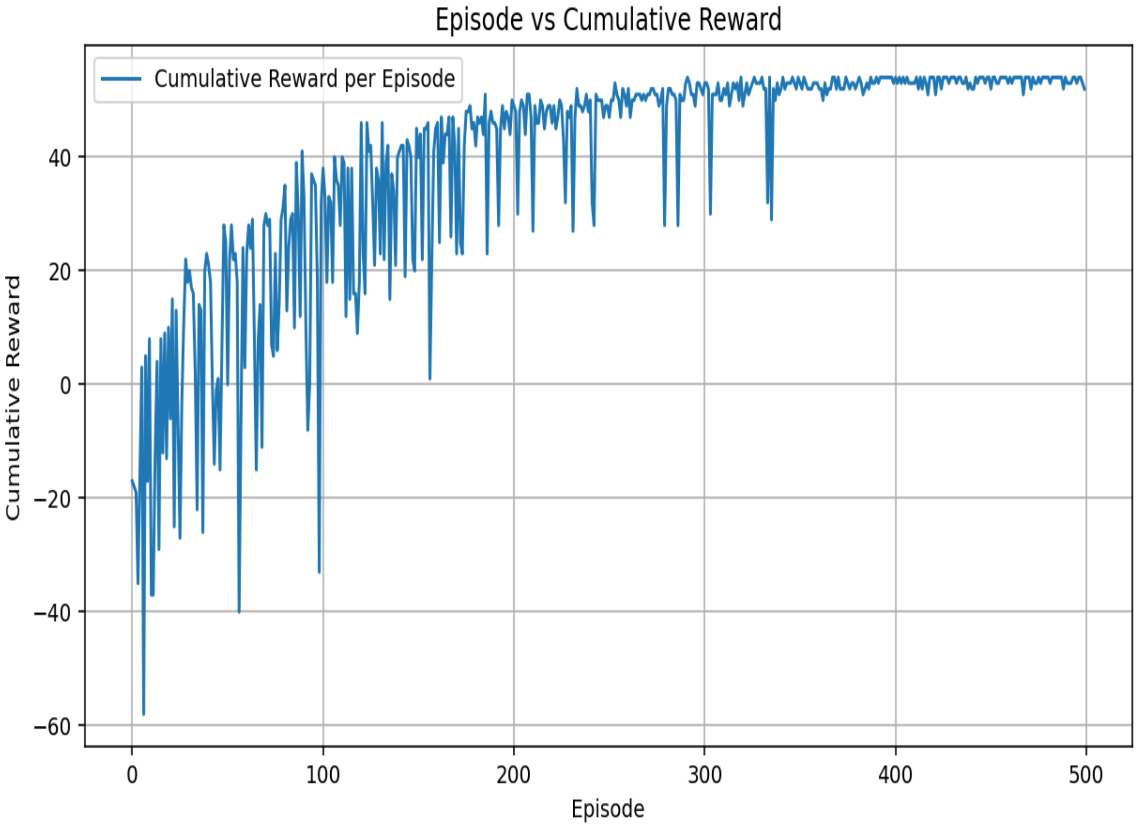
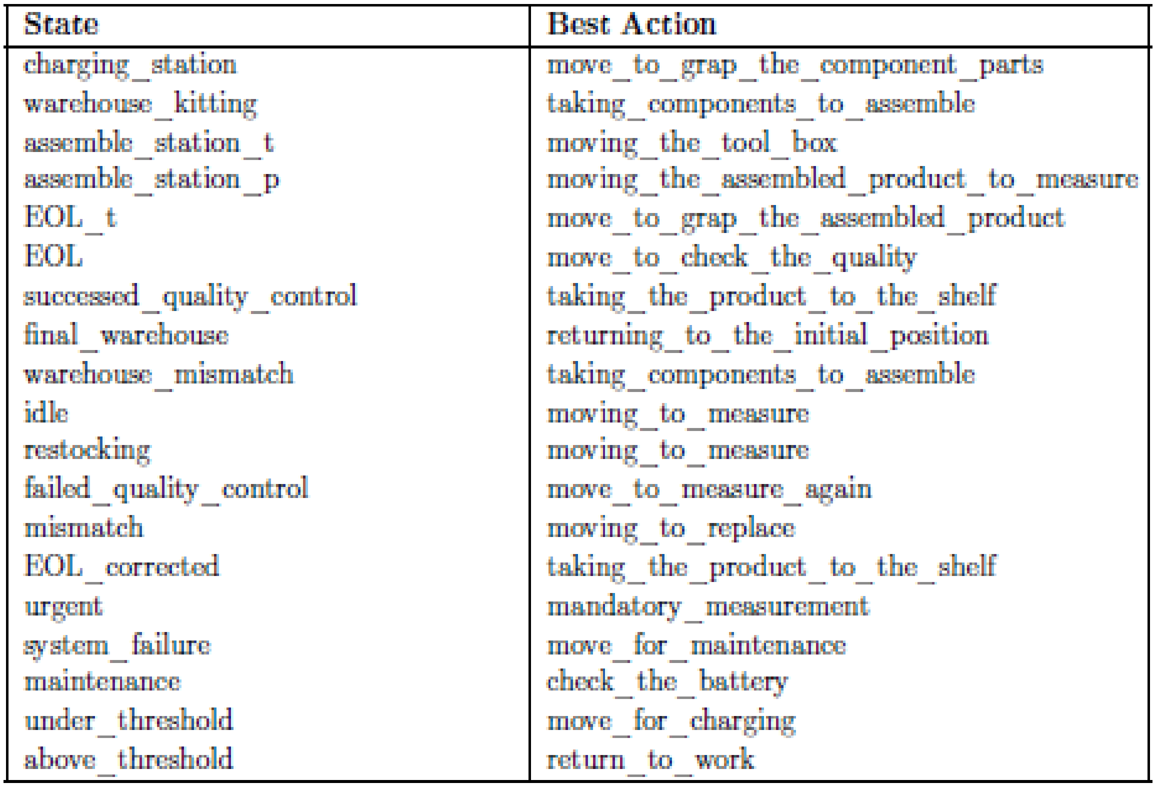


Figure 43 Task Sequence Episode vs Cumulative Reward for Q-learning Algorithm

Figure 43 illustrates the result of the run of the Q-learning algorithm for 500 episodes. As in the SARSA results, the early episodes have large fluctuations due to the exploration phase. However, Q-learning tends to have a better rate of reward improvement than SARSA. The graph rises steadily in cumulative rewards. This means more aggressive exploitation of the learned values. The cumulative rewards will start stabilizing around episode 150 and, at episode 500, will have plateaued, hence successful learning and policy optimization. Both algorithms achieved the same optimal policy which is shown in the table below.



State	Best Action
charging_station	move_to_grap_the_component_parts
warehouse_kitting	taking_components_to_assemble
assemble_station_t	moving_the_tool_box
assemble_station_p	moving_the_assembled_product_to_measure
EOL_t	move_to_grap_the_assembled_product
EOL	move_to_check_the_quality
successed_quality_control	taking_the_product_to_the_shelf
final_warehouse	returning_to_the_initial_position
warehouse_mismatch	taking_components_to_assemble
idle	moving_to_measure
restocking	moving_to_measure
failed_quality_control	move_to_measure_again
mismatch	moving_to_replace
EOL_corrected	taking_the_product_to_the_shelf
urgent	mandatory_measurement
system_failure	move_for_maintenance
maintenance	check_the_battery
under_threshold	move_for_charging
above_threshold	return_to_work

Table 2 Optimal Policy For Task Sequence MDP

In the mentioned MDP, the robot is trained across a variety of states and transitions to gain resilience against worst-case scenarios. In case of system failure, the robot is trained to move for maintenance and then check on the battery to quickly resume operations after attending to critical failures. The robot takes mandatory measures on urgent states to efficiently deal with highpriority tasks to avoid disruptions to the standard workflow. In case of failed quality control, the robot measures for reassessment and correction of issues to make sure that quality products move forward in the process. The robot manages its battery levels by moving to charge when under the threshold and returning to work when above the threshold, thus preventing resource shortages that might hinder the performability of tasks. It corrects the mismatches by replacing the incorrect components and taking the right ones to assembly, thus maintaining the assembly's integrity. While in idle states, the robot measures to stay productive, keeping itself ready for the following process, hence minimizing downtime and efficiency. At restocking, the robot avoids wait and waste of time by doing some obligated tasks that ensure continuous productivity. The policy ensures that the robot will be able to:

- Quickly address system failures and perform necessary maintenance.
- Effectively handle urgent tasks and quality control failures.
- Keep optimal levels of the batteries to avoid running short of resources
- Correct the mismatches and maintain the integrity of the warehouse.
- Minimize idle periods, maintaining a productive level.
- Ensure that end-of-line inspection products meet the standards before finalization.
- Prevent wasting time during restocking.

Moreover, an analysis is done to see how different settings of hyperparameters affect the learning process and the resulting policy performance. First, the change is done on learning rate. The first graph for SARSA and the second graph for Q-learning both enhanced with Sample-based planning.

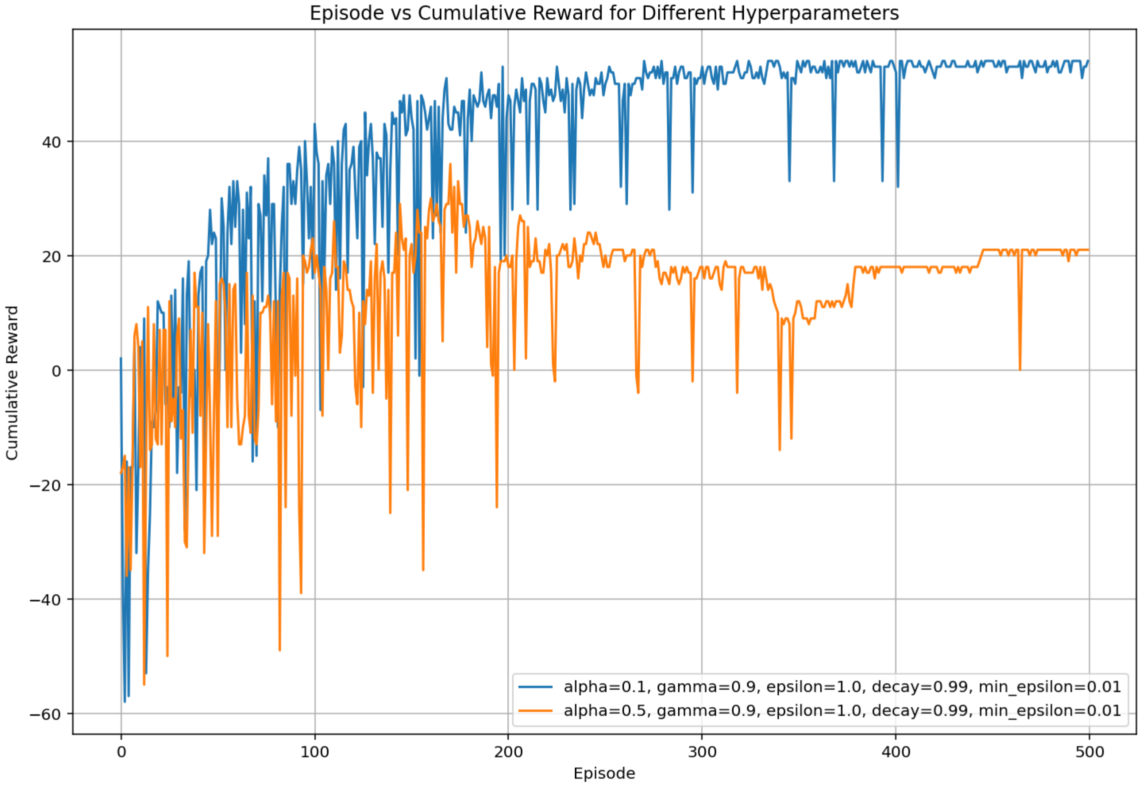


Figure 44 Episode vs Cumulative Reward for Different Hyperparameters in SARSA

The blue line represents the performance with a learning rate (α) of 0.1, discount factor (γ) of 0.9, an initial exploration rate (ϵ) of 1.0, a decay rate of 0.99, and a minimum ϵ of 0.01. The one in orange corresponds to a higher learning rate (α) of 0.5 with the same γ, ϵ, the decay rate, and a minimum ϵ. At the start of both lines, a large amount of fluctuation can be seen due to exploration; however, the blue line expands more rapidly away from 0. From episodes 50 through 200, the blue line remains steadily uphill, indicating good learning occurring because of a lower learning rate, while the orange line is much more volatile. Later in episodes, it levels off at a higher cumulative reward characteristic of a more optimal policy with less fluctuation. In contrast, the orange line remains more volatile and reaches lower cumulative rewards overall. These results suggest that a lower learning rate (α = 0.1) gives more stable learning and higher rewards, which suggests that hyperparameter tuning is very important in achieving optimal performance in reinforcement learning tasks.

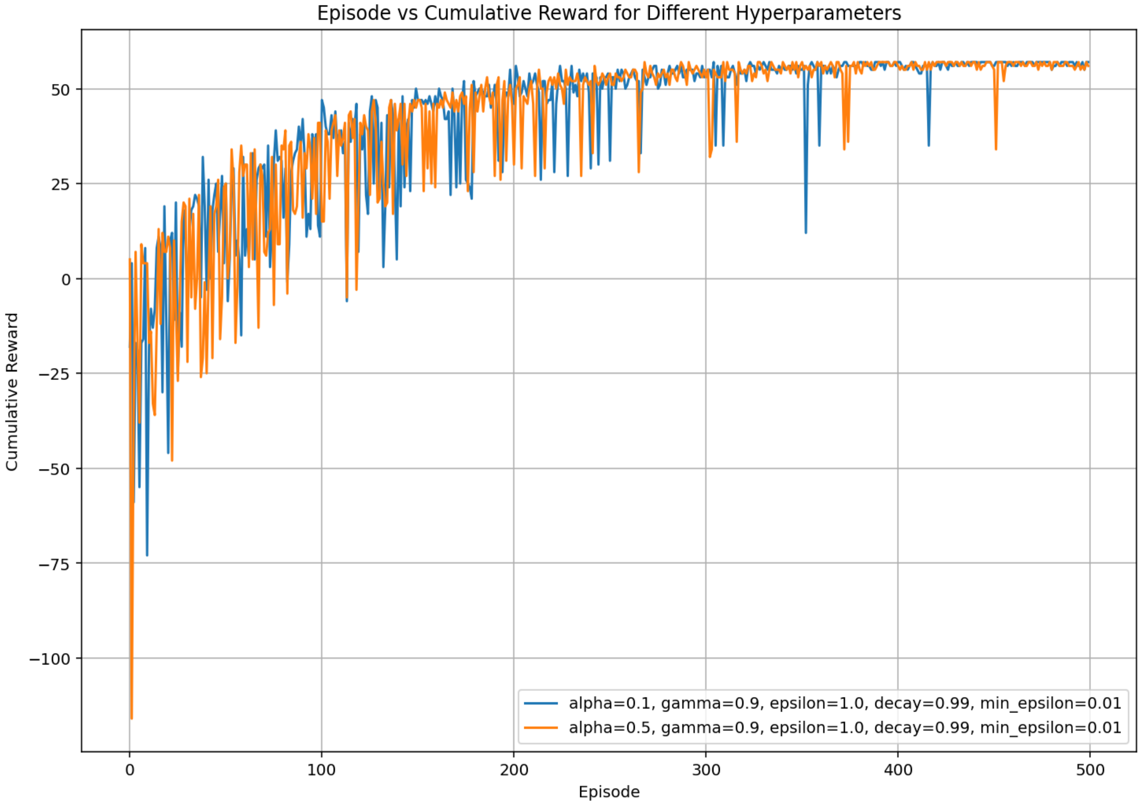


Figure 45 Episode vs Cumulative Reward for Different Hyperparameters in Q-learning

In the early learning phase, episodes 0 through 100, the blue line produces significant fluctuations in cumulative rewards; it means the agent is exploring and learning incrementally. On the other hand, the orange line also produces bumps but bounces back to a positive gradient quickly, which may suggest that the agent with a higher learning rate (α = 0.5) learns quicker. One can see that α = 0.5 learns quicker. In the phase of learning progression from episodes 100 to 300, the blue line increases steadily with reduced fluctuations, which means that this is a pretty consistent improvement, and there is a balance between exploitation and exploration. Meanwhile, the orange line shows how the updating of the Q-values with significant steps brings faster increases in returns but is again accompanied by instabilities reliably reflected in the continuing fluctuations. In the stabilization phase, which includes episodes 300-500, it converges to an equilibrium with a cumulative reward of 40, indicating that it has converged to some somewhat effective policy with minor refinements. In contrast, the orange line stabilizes at a higher order of reward—to the order of 50-60—with minor fluctuations, which means that the higher learning rate allows the agent to achieve better policies more quickly for a high and stable reward. The comparison of SARSA and Q-learning with sample-based planning under the same hyperparameter settings reveals that Q-learning with sample-based enhancements generally outperforms SARSA in terms of cumulative reward and learning efficiency. The higher performance and faster convergence of Q-learning make it a powerful tool for training the robot in this MDP. Although higher learning rates speed up learning, it is known to have possible issues with stability, mainly for SARSA. Q-learning with sample-based planning demonstrates higher robustness, performance for more significant learning rates, higher cumulative rewards, and faster convergence. Thus, tuning the learning rate should be appropriate to get maximum efficiency and stability for the reinforcement learning algorithms.

8.3 Reasons of Disturbances

8.3.1 Imbalance Between Rewards and Penalties

An important disturbance encountered during the simulation was the imbalance between rewards and penalties within the environment. In environments where the penalties significantly outweigh the rewards, or vice versa, the learning dynamics can become skewed, leading to suboptimal policies or even learning failure. The imbalance between rewards and penalties can disrupt the agent’s ability to strike an effective balance between exploration and exploitation. If the agent is overly penalized for exploratory behavior, it may avoid trying new actions that could lead to better long-term rewards. On the other hand, an excessive focus on rewards can lead the agent to become too exploitative, preventing it from effectively learning from the full range of experiences.

8.3.2 Slow Convergence

Another challenge was the slow convergence of the Q-learning algorithm. Early episodes showed little improvement in performance, and the reward graph often stagnated or fluctuated without showing clear signs of learning. This was particularly evident in environments with large state spaces, where the agent had to explore many states and actions before it could begin to discern optimal policies.

After several unsuccessful attempts with different parameters, adjustment for the learning rate (α) and the discount factor (γ) to facilitate faster learning. Additionally, the exploration rate (ϵ) is gradually decayed to ensure that the agent balanced exploration with exploitation more effectively, which contributed to improving the convergence rate. But at the end, they aren't the most optimal parameters tuned, that's why there is still and will remain disturbance at the beginning till the convergence.

Chapter 9: Conclusion and Future Work

This research aimed to develop a reinforcement learning (RL) framework capable of enabling robots to operate autonomously and adaptively in industrial environments. By leveraging Q-learning, SARSA, Dyna-Q, and sample-based planning within a structured Markov Decision Process (MDP) environment, the study sought to enhance the robot's resilience and efficiency across tasks such as navigation, assembly, quality control, and maintenance. Each RL method contributed distinct advantages that, collectively, empowered the robot to handle complex workflows and unforeseen disruptions with greater adaptability.

9.1 Key Findings:

1. **Q-learning** provided a straightforward yet effective approach for training the robot to maximize cumulative rewards through trial and error, emphasizing exploration and exploitation. Its off-policy nature allowed rapid convergence to an optimal policy, although it showed limitations in highly dynamic environments.
2. **SARSA's on-policy approach** yielded more stable learning, ideal for dynamic industrial settings where the robot needs to prioritize safety and consistent policy adherence. SARSA's conservative updates made it effective for tasks requiring stable, reliable actions over rapid policy shifts.
3. **Dyna-Q's integration of real and simulated experiences** accelerated learning by enabling planning updates, which proved particularly advantageous in environments where task sequences were complex, and real interactions were costly. The hybrid nature of Dyna-Q allowed the robot to reinforce learning from a mix of real and simulated experiences, enhancing adaptability and learning efficiency.

4. **Sample-based planning** allowed the robot to simulate potential outcomes and explore a broader range of actions without excessive real-world interactions. This approach reduced learning time and resource usage, especially beneficial in resource-intensive tasks like quality control and restocking.

The study demonstrated that these methods, when implemented in a structured MDP framework, could train robots to manage industrial tasks with improved efficiency and adaptability. By systematically refining Q-values through a combination of real and simulated interactions, the robot achieved a resilient policy that maximized productivity and minimized response time to unexpected events.

9.2 Future Work: Integrating Deep Q-Networks (DQN)

While this research highlighted the potential of traditional RL algorithms, future advancements can be realized by incorporating **Deep Q-Networks (DQN)** to address some of the limitations encountered with large state-action spaces. DQN, a neural network-based extension of Q-learning, approximates the Q-value function in high-dimensional or continuous environments, providing a robust solution for scenarios where traditional Q-tables are impractical.

9.3 Benefits of DQN:

- **Scalability to Complex Environments**: DQN enables the agent to generalize across similar states, a critical advantage for complex industrial applications where the robot must perform diverse tasks under varying conditions.
- **Improved Learning Efficiency through Experience Replay**: DQN utilizes experience replay to store past experiences in a buffer, allowing the agent to learn from a wider variety of interactions without redundancy. This could enhance stability and learning efficiency in dynamic environments.

- **Handling Continuous State Spaces**: For industrial tasks requiring precise control and adaptation (e.g., quality inspection with fine-grained actions), DQN's ability to approximate the Q-value function without discretization allows for more granular and effective decision-making.

Future work could explore combining DQN with **prioritized experience replay** to further enhance sample efficiency, where experiences with larger temporal-difference errors are prioritized for learning. This improvement would allow the robot to focus more on critical transitions, optimizing its learning and response times in complex tasks.

9.4 Potential Applications of DQN in Industrial Settings:

1. **Multi-Task Learning**: Using DQN, the robot could handle multiple tasks across different industrial areas (e.g., transitioning seamlessly from assembly to quality control) with improved policy generalization.
2. **Adaptive Resource Management**: DQN could help the robot dynamically manage resource constraints (e.g., battery levels, component availability) by learning more precise policies that optimize resource usage without sacrificing performance.
3. **Real-World Testing and Deployment**: By training with DQN in simulated environments that mimic real-world conditions, future studies could move closer to deploying the trained policies on actual robots, bridging the gap between simulation and deployment.

In conclusion, this research successfully demonstrated that a structured MDP framework, combined with RL methods like Q-learning, SARSA, Dyna-Q, and sample-based planning, enables robots to adapt to dynamic industrial settings. Integrating advanced methods such as DQN will further empower robots to learn complex tasks efficiently, setting the stage for highly adaptable and scalable industrial robotics in the future.

Bibliography

[1] J. Kober, J. A. Bagnell, and J. Peters, “Reinforcement Learning in Robotics: A Survey”.

[2] D. Silver, “Lecture 1: Introduction to Reinforcement Learning”.

[3] R. S. Sutton and A. Barto, *Reinforcement learning: an introduction*, Second edition. in Adaptive computation and machine learning. Cambridge, Massachusetts London, England: The MIT Press, 2020.

[4] N. Altuntaş, E. İMal, N. Emanet, and C. N. Öztürk, “Reinforcement learning-based mobile robot navigation,” *Turk J Elec Eng & Comp Sci*, vol. 24, pp. 1747–1767, 2016, doi: 10.3906/elk-1311-129.

[5] H. Ruan, S. Zhou, Z. Chen, and C. P. Ho, “Robust Satisficing MDPs”.

[6] L. Vordemann, “Safe reinforcement learning for human-robot collaboration : Shielding of a robotic local planner in an autonomous warehouse scenario,” master, KTH, School of Electrical Engineering and Computer Science (EECS) / KTH, School of Electrical Engineering and Computer Science (EECS), 2022.

[7] H. Lee and J. Jeong, “Mobile Robot Path Optimization Technique Based on Reinforcement Learning Algorithm in Warehouse Environment,” *Applied Sciences*, vol. 11, no. 3, p. 1209, Jan. 2021, doi: 10.3390/app11031209.

[8] V. N. Sichkar, “Reinforcement Learning Algorithms in Global Path Planning for Mobile Robot,” in *2019 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM)*, Sochi, Russia: IEEE, Mar. 2019, pp. 1–5. doi: 10.1109/ICIEAM.2019.8742915.

[9] Y. Jiang, F. Yang, S. Zhang, and P. Stone, “Task-Motion Planning with Reinforcement Learning for Adaptable Mobile Service Robots,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Macau, China: IEEE, Nov. 2019, pp. 7529–7534. doi: 10.1109/IROS40897.2019.8967680.

[10] J. Pizoń, Ł. Wójcik, A. Gola, Ł. Kański, and I. Nielsen, “Autonomous Mobile Robots in Automotive Remanufacturing: A Case Study for Intra-Logistics Support,” *Adv. Sci. Technol. Res. J.*, vol. 18, no. 1, pp. 213–230, Feb. 2024, doi: 10.12913/22998624/177398.

[11] N. M. Gomes, F. N. Martins, J. Lima, and H. Wörtche, “Reinforcement Learning for Collaborative Robots Pick-and-Place Applications: A Case Study,” *Automation*, vol. 3, no. 1, pp. 223–241, Mar. 2022, doi: 10.3390/automation3010011.

[12] D. Silver, “Lecture 2: Markov Decision Processes,” *Markov Processes*.

[13] S. W. Contributors, “Spyder | The Python IDE that scientists and data analysts deserve,” Spyder IDE. Accessed: Nov. 10, 2024. [Online]. Available: https://www.spyder-ide.org

[14] “Welcome to Python.org,” Python.org. Accessed: Nov. 10, 2024. [Online]. Available: https://www.python.org/

[15] “Anaconda | The Operating System for AI.” Accessed: Nov. 10, 2024. [Online]. Available: https://www.anaconda.com/

[16] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, “The NumPy Array: A Structure for Efficient Numerical Computation,” *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 22–30, Mar. 2011, doi: 10.1109/MCSE.2011.37.

[17] “random — Generate pseudo-random numbers,” Python documentation. Accessed: Nov. 16, 2024. [Online]. Available: https://docs.python.org/3/library/random.html

[18] J. D. Hunter, “Matplotlib: A 2D Graphics Environment,” *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, 2007, doi: 10.1109/MCSE.2007.55.

[19] W. McKinney, “Data Structures for Statistical Computing in Python,” presented at the Python in Science Conference, Austin, Texas, 2010, pp. 56–61. doi: 10.25080/Majora-92bf1922-00a.

[20] A. A. Hagberg, D. A. Schult, and P. J. Swart, “Exploring Network Structure, Dynamics, and Function using NetworkX,” presented at the Python in Science Conference, Pasadena, California, Jun. 2008, pp. 11–15. doi: 10.25080/TCWV9851.

[21] “Robot mobili per intralogistica,” IEN ITALIA - Scopri i nostri prodotti e servizi industriali. Accessed: Nov. 16, 2024. [Online]. Available: https://www.ien-italia.eu/articolo/robot-mobili-per-intralogistica/

[22] “ROS: Home.” Accessed: Nov. 10, 2024. [Online]. Available: https://www.ros.org/

[23] “ROS Architecture and Concepts,” Packt. Accessed: Nov. 10, 2024. [Online]. Available: https://www.packtpub.com/en-us/learning/how-to-tutorials/ros-architecture-and-concepts

[24] D. Silver, “Lecture 5: Model-Free Control”.

[25] “flowchart.js.” Accessed: Nov. 10, 2024. [Online]. Available: https://flowchart.js.org/

[26] D. Silver, “Lecture 8: Integrating Learning and Planning”.

[27] D. Silver, “Lecture 9: Exploration and Exploitation”.

Appendix:

MDP Navigation Code:

```
 import networkx as nx
import matplotlib.pyplot as plt

# Create a directed graph
G = nx.DiGraph()

# Add nodes (states) to the graph
states = ['S0', 'SA', 'SB', 'SC', 'HA', 'HB', 'HC', 'P']
actions = ['Perform', 'O', 'F']
G.add_nodes_from(states)

# Define transitions and corresponding actions
transitions = {
    ('S0', 'SA'): 'F',
    ('S0', 'HA'): 'O',
    ('S0', 'SB'): 'F',
    ('S0', 'HB'): 'O',
    ('S0', 'SC'): 'F',
    ('S0', 'HC'): 'O',
    ('HA', 'SB'): 'F',
    ('HA', 'SC'): 'F',
    ('HB', 'SA'): 'F',
    ('HB', 'SC'): 'F',
    ('HC', 'SA'): 'F',
    ('HC', 'SB'): 'F',
    ('HA', 'HB'): 'O',
    ('HA', 'HC'): 'O',
    ('HB', 'HA'): 'O',
    ('HB', 'HC'): 'O',
    ('HC', 'HA'): 'O',
    ('HC', 'HB'): 'O',
    ('SA', 'P'): '', #Perform
    ('SB', 'P'): '', #Perform
    ('SC', 'P'): '', #Perform
    ('P', 'SA'): '', #F
    ('P', 'SB'): '', #F
    ('P', 'SC'): '', #F
    ('P', 'HA'): 'O',
    ('P', 'HB'): 'O',
    ('P', 'HC'): 'O',
}

# Add edges with labels (actions)
for (u, v), action in transitions.items():
    G.add_edge(u, v, label=action)

# Set a layout for our nodes
layout = {
    'S0': (-2, 1),
    'SA': (-1, -1),
    'SB': (2, 2.5),
    'SC': (2, 0),
    'HA': (-1, 3),
    'HB': (1, 3),
    'HC': (2, 1.5),
    'P': (1, -1)
}

# Drawing parameters
node_size = 2000
font_size = 12
plt.figure(figsize=(12, 8))

# Draw the graph with the specific node positions we've set
nx.draw(G, pos=layout, node_size=node_size, node_color='lightblue',
edge_color='black',
        width=2, linewidths=1, font_size=font_size,
with_labels=True, arrowsize=20)

# Draw edge labels
edge_labels = nx.get_edge_attributes(G, 'label')
nx.draw_networkx_edge_labels(G, pos=layout,
edge_labels=edge_labels, font_color='red')

# Display the graph
plt.title('Markov Decision Process Visualization', size=20)
plt.axis('off')  # Turn off the axis
plt.show()
```

MDP Experiment Code:

```
import networkx as nx
import matplotlib.pyplot as plt

# Create a directed graph
G = nx.DiGraph()

#EOL: end of line inspection

# Add nodes (states) to the graph
states = ['charging_station', 'warehouse_kitting',
'assemble_station_t', 'assemble_station_p', 'EOL_t', 'EOL_p'
          , 'EOL', 'quality_control', 'final_warehouse']
actions = ['move_to_grap_the_component_parts',
'taking_components_to_assemble', 'moving_the_tool_box',
'move_to_grap_the_assembled_part', 'moving _the_assembled_product'
           , 'move_to_measure', 'move_to_check_the_quality',
'taking_the_product_to_the_shelf',
'returning_to_the_initial_position']
G.add_nodes_from(states)

# Define transitions and corresponding actions
transitions = {
    ('charging_station', 'warehouse_kitting'):
'move_to_grap_the_component_parts',
    ('warehouse_kitting', 'assemble_station_t'):
'taking_components_to_assemble',
    ('assemble_station_t', 'EOL_t'): 'moving_the_tool_box',
    ('EOL_t', 'assemble_station_p'):
'move_to_grap_the_assembled_product',
    ('assemble_station_p', 'EOL_p'): 'moving
_the_assembled_product',
    ('EOL_p', 'EOL'): 'move_to_measure',
    ('EOL', 'quality_control'): 'move_to_check_the_quality',
    ('quality_control', 'final_warehouse'):
'taking_the_product_to_the_shelf',
    ('final_warehouse', 'charging_station'):
'returning_to_the_initial_position',
}

# Add edges with labels (actions)
for (u, v), action in transitions.items():
    G.add_edge(u, v, label=action)

# Set a layout for our nodes
layout = {
    'charging_station': (-2, 1),
    'warehouse_kitting': (-1, 1),
    'assemble_station_t': (0, 1),
    'assemble_station_p': (0, 0),
    'EOL_t': (1,1),
    'EOL_p': (0,-3),
    'EOL': (1,-3),
    'quality_control': (2, -2),
    'final_warehouse': (2, 3),
}

# Drawing parameters
node_size = 9000
font_size = 10
plt.figure(figsize=(25, 12))

# Draw the graph with the specific node positions we've set
nx.draw(G, pos=layout, node_size=node_size, node_color='lightblue',
edge_color='black',
        width=2, linewidths=1, font_size=font_size,
with_labels=True, arrowsize=20)

# Draw edge labels
edge_labels = nx.get_edge_attributes(G, 'label')
nx.draw_networkx_edge_labels(G, pos=layout,
edge_labels=edge_labels, font_color='red')

# Display the graph
plt.title('Markov Decision Process Visualization', size=20)
plt.axis('off')  # Turn off the axis
plt.show()
```

MDP Experiment with Adversaries+R Code:

```
import networkx as nx
import matplotlib.pyplot as plt

# Create a directed graph
G = nx.DiGraph()

#EOL: end of line inspection

# Add nodes (states) to the graph
states = ['charging_station', 'warehouse_kitting',
'assemble_station_t', 'assemble_station_p'
          , 'EOL_t' , 'EOL', 'successed_quality_control',
'final_warehouse', 'warehouse_mismatch' , 'idle', 'restocking'
          , 'failed_quality_control', 'mismatch', 'EOL_corrected',
'urgent', 'system_failure'
          , 'maintenance', 'under_threshold', 'above_threshold']
actions = ['move_to_grap_the_component_parts',
'taking_components_to_assemble', 'moving_the_tool_box',
           'move_to_grap_the_assembled_product',
'moving_the_assembled_product_to_measure',
           'move_to_check_the_quality_s',
'move_to_check_the_quality_f', 'taking_the_product_to_the_shelf',
'returning_to_the_initial_position',
           'taking_the_wrong_component', 'returning_to_warehouse',
'wait', 'move', 'move_to_measure_again'
           , 'moving_to_measure', 'moving_to_replace',
'mandatory_measurement', 'mandatory_quality_control'
           , 'move_to_warehouse', 'unexpected_failure',
'continue_working', 'move_for_maintenance'
           , 'check_the_battery', 'return_to_work',
'move_for_charging']
G.add_nodes_from(states)

# Define transitions and corresponding actions
transitions = {
    ('charging_station', 'warehouse_kitting'):
'move_to_grap_the_component_parts',
    ('warehouse_kitting', 'assemble_station_t'):
'taking_components_to_assemble',
    ('assemble_station_t', 'EOL_t'): 'moving_the_tool_box',
    ('EOL_t', 'assemble_station_p'):
'move_to_grap_the_assembled_product',
    ('assemble_station_p', 'EOL'):
'moving_the_assembled_product_to_measure',
    ('final_warehouse', 'charging_station'):
'returning_to_the_initial_position',
    ('mismatch', 'warehouse_mismatch'): 'moving_to_replace',
    ('warehouse_mismatch', 'assemble_station_p'):
'taking_components_to_assemble',
    ('assemble_station_p', 'idle'): '', #move
    ('idle', 'assemble_station_p'): '', #wait
    ('EOL', 'failed_quality_control'): 'move_to_check_the_quality',
    ('EOL', 'successed_quality_control'):
'move_to_check_the_quality',
    ('idle', 'EOL'): 'moving_to_measure',
    ('warehouse_kitting', 'restocking'): '', #move
    ('restocking', 'warehouse_kitting'): '', #wait
    ('restocking', 'EOL'): 'moving_to_measure',
    ('successed_quality_control', 'final_warehouse'):
'taking_the_product_to_the_shelf',
    ('failed_quality_control', 'final_warehouse'):
'taking_the_product_to_the_shelf',
    ('mismatch', 'EOL'): 'moving_to_measure',
    ('assemble_station_p', 'mismatch'): 'move',
    ('mismatch', 'EOL'): 'moving_to_measure',
    ('failed_quality_control', 'EOL_corrected'):
'move_to_measure_again',
    ('EOL_corrected', 'final_warehouse'):
'taking_the_product_to_the_shelf',
    ('charging_station', 'urgent'): 'move',
    #('warehouse_kitting', 'urgent'): 'urgent_move',
    #('assemble_station_t', 'urgent'): 'urgent_move',
    ('urgent', 'EOL'): 'mandatory_measurement',
    ('urgent', 'failed_quality_control'):
'mandatory_quality_control',
    ('urgent', 'successed_quality_control'):
'mandatory_quality_control',
    ('urgent', 'warehouse_kitting'): 'move_to_warehouse',
    ('EOL', 'system_failure'): '', #unexpected_failure
    ('system_failure', 'maintenance'): 'move_for_maintenance',
    ('system_failure', 'EOL'): '', #continue_working
    ('maintenance', 'under_threshold'): 'check_the_battery',
    ('maintenance', 'above_threshold'): 'check_the_battery',
    ('under_threshold', 'urgent'): 'return_to_work',
    ('under_threshold', 'charging_station'): 'move_for_charging',
    ('above_threshold', 'urgent'): 'return_to_work',
    ('above_threshold', 'charging_station'): 'move_for_charging',
}

# Add edges with labels (actions)
for (u, v), action in transitions.items():
    G.add_edge(u, v, label=action)

# Set a layout for our nodes
layout = {
    'charging_station': (-3.5, 1.5),
    'warehouse_kitting': (-1.5, 1),
    'assemble_station_t': (0.2, 1.7),
    'assemble_station_p': (-1, -3.5),
    'idle': (-0.2, -4),
    'component_mismatch': (-2, -0.5),
    'mismatch': (-1.9, -2.2),
    'warehouse_mismatch': (-3, -3.5),
    'EOL_t': (1.3,3.5),
    'EOL': (1,-3),
    'quality_control': (2, -2),
    'failed_quality_control': (3, 1),
    'final_warehouse': (2, 3),
    'restocking': (-0.8, -1),
    'EOL_corrected': (1.5, -0.7),
    'urgent': (-2.5, -1),
    'successed_quality_control': (3, -3),
    'system_failure': (0.8, -0.7),
    'maintenance': (-1.2, 3),
    'under_threshold': (-5, 2),
    'above_threshold': (-5, -2),
}

# Drawing parameters
node_size = 10000
font_size = 10
plt.figure(figsize=(25, 12))

# Draw the graph with the specific node positions we've set
nx.draw(G, pos=layout, node_size=node_size, node_color='lightblue',
edge_color='black',
        width=2, linewidths=1, font_size=font_size,
with_labels=True, arrowsize=20)

# Draw edge labels
edge_labels = nx.get_edge_attributes(G, 'label')
nx.draw_networkx_edge_labels(G, pos=layout,
edge_labels=edge_labels, font_color='red')

# Display the graph
plt.title('Markov Decision Process Visualization', size=20)
plt.axis('off')  # Turn off the axis
plt.show()
```

Dyna-Q Code:

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

# states and actions
states = ['S0', 'SA', 'SB', 'SC', 'HA', 'HB', 'HC', 'P']
actions = ['Perform', 'O', 'F']
state_indices = {state: idx for idx, state in enumerate(states)}
action_indices = {action: idx for idx, action in
enumerate(actions)}

# Initialize rewards based on description
rewards = np.zeros((len(states), len(actions)))
rewards[state_indices['S0'], action_indices['F']] = 1
rewards[state_indices['SA'], action_indices['Perform']] = 1
rewards[state_indices['SB'], action_indices['Perform']] = 1
rewards[state_indices['SC'], action_indices['Perform']] = 1
rewards[state_indices['P'], action_indices['F']] = 1
rewards[state_indices['HA'], action_indices['F']] = 1
rewards[state_indices['HB'], action_indices['F']] = 1
rewards[state_indices['HC'], action_indices['F']] = 1
rewards[state_indices['S0'], action_indices['O']] = -5
rewards[state_indices['P'], action_indices['O']] = -5
rewards[state_indices['HA'], action_indices['O']] = -5
rewards[state_indices['HB'], action_indices['O']] = -5
rewards[state_indices['HC'], action_indices['O']] = -5

# Stochastic transitions
transitions_stochastic = {
    'S0': {'F': [('SA', 0.33), ('SB', 0.33), ('SC', 0.34)],
           'O': [('HA', 0.33), ('HB', 0.33), ('HC', 0.34)]},
    'HA': {'F': [('SB', 0.5), ('SC', 0.5)], 'O': [('HB', 0.5),
('HC', 0.5)]},
    'HB': {'F': [('SA', 0.5), ('SC', 0.5)], 'O': [('HA', 0.5),
('HC', 0.5)]},
    'HC': {'F': [('SA', 0.5), ('SB', 0.5)], 'O': [('HA', 0.5),
('HB', 0.5)]},
    'SA': {'Perform': [('P', 1.0)]}, 'SB': {'Perform': [('P',
1.0)]}, 'SC': {'Perform': [('P', 1.0)]},
    'P': {'F': [('SA', 0.33), ('SB', 0.33), ('SC', 0.34)], 'O':
[('HA', 0.33), ('HB', 0.33), ('HC', 0.34)]},
    }

# Function to simulate state transition
def simulate_transition(current_state, action,
transitions_stochastic):
    if action in transitions_stochastic[current_state]:
        possible_transitions =
transitions_stochastic[current_state][action]
        next_states, probabilities = zip(*possible_transitions)
        next_state = random.choices(next_states,
weights=probabilities)[0]
        return next_state
    return current_state  # Return current state if no action is
possible

# Dyna-Q parameters
alpha = 0.1  # Learning rate
gamma = 0.95  # Discount factor
epsilon_start = 1.0  # Initial exploration rate
epsilon_end = 0.1  # Final exploration rate
episodes = 500  # Number of episodes for training
epsilon_decay = (epsilon_start - epsilon_end) / episodes  # Linear
decay over episodes
n_planning_steps = 5  # Number of planning steps per real step

# Initialize Q-values and model
Q = np.zeros((len(states), len(actions)))
model = {}

# Function to choose action using epsilon-greedy policy
def choose_action(state, epsilon):
    if np.random.rand() < epsilon:
        return np.random.choice(actions)
    else:
        return actions[np.argmax(Q[state_indices[state]])]

# List to store cumulative rewards for each episode
cumulative_rewards = []

# Training loop
epsilon = epsilon_start
for episode in range(episodes):
    state = 'S0'
    cumulative_reward = 0
    while state != 'P':  # Continue until terminal state is reached
        action = choose_action(state, epsilon)
        next_state = simulate_transition(state, action,
transitions_stochastic)
        reward = rewards[state_indices[state],
action_indices[action]]
        cumulative_reward += reward

        # Update Q-values
        best_next_action = np.argmax(Q[state_indices[next_state]])
        Q[state_indices[state], action_indices[action]] += alpha *
(reward + gamma *
        Q[state_indices[next_state], best_next_action] -
Q[state_indices[state], action_indices[action]])

        # Update model
        if state not in model:
            model[state] = {}
        model[state][action] = (next_state, reward)

        # Planning steps
        for _ in range(n_planning_steps):
            if model:
                sampled_state = random.choice(list(model.keys()))
                sampled_action = 
random.choice(list(model[sampled_state].keys()))
                sampled_next_state, sampled_reward = 
model[sampled_state][sampled_action]

                best_sampled_next_action = 
np.argmax(Q[state_indices[sampled_next_state]])
                Q[state_indices[sampled_state], 
action_indices[sampled_action]] += alpha * (sampled_reward + gamma 
* 

Q[state_indices[sampled_next_state], 
                best_sampled_next_action] - 
Q[state_indices[sampled_state], action_indices[sampled_action]])

        state = next_state

    cumulative_rewards.append(cumulative_reward)

    # Decay epsilon
    if epsilon > epsilon_end:
        epsilon -= epsilon_decay

# Extract the optimal policy
optimal_policy = {state: 
actions[np.argmax(Q[state_indices[state]])] for state in states}

# Print the optimal policy
print("Optimal Policy:")
for state, action in optimal_policy.items():
    if state != 'P':
        print(f"State {state}: Best Action {action}")

# Print the Q-table
print("\nQ-table:")
Q_df = pd.DataFrame(Q, index=states, columns=actions)
print(Q_df)

# Plot the cumulative reward over episodes
plt.plot(cumulative_rewards)
plt.xlabel('Episodes')
plt.ylabel('Cumulative Reward')
plt.title('Cumulative Reward over Episodes')
plt.show()
```

SBP with Q-learning Code for Navigation MDP:

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

# states and actions
states = ['S0', 'SA', 'SB', 'SC', 'HA', 'HB', 'HC', 'P']
actions = ['Perform', 'O', 'F']
state_indices = {state: idx for idx, state in enumerate(states)}
action_indices = {action: idx for idx, action in
enumerate(actions)}

# Initialize rewards based on description
rewards = np.zeros((len(states), len(actions)))
rewards[state_indices['S0'], action_indices['F']] = 1
rewards[state_indices['SA'], action_indices['Perform']] = 1
rewards[state_indices['SB'], action_indices['Perform']] = 1
rewards[state_indices['SC'], action_indices['Perform']] = 1
rewards[state_indices['P'], action_indices['F']] = 1
rewards[state_indices['HA'], action_indices['F']] = 1
rewards[state_indices['HB'], action_indices['F']] = 1
rewards[state_indices['HC'], action_indices['F']] = 1
rewards[state_indices['S0'], action_indices['O']] = -5
rewards[state_indices['P'], action_indices['O']] = -5
rewards[state_indices['HA'], action_indices['O']] = -5
rewards[state_indices['HB'], action_indices['O']] = -5
rewards[state_indices['HC'], action_indices['O']] = -5

# Stochastic transitions
transitions_stochastic = {
    'S0': {'F': [('SA', 0.33), ('SB', 0.33), ('SC', 0.34)],
           'O': [('HA', 0.33), ('HB', 0.33), ('HC', 0.34)]},
    'HA': {'F': [('SB', 0.5), ('SC', 0.5)], 'O': [('HB', 0.5),
('HC', 0.5)]},
    'HB': {'F': [('SA', 0.5), ('SC', 0.5)], 'O': [('HA', 0.5),
('HC', 0.5)]},
    'HC': {'F': [('SA', 0.5), ('SB', 0.5)], 'O': [('HA', 0.5),
('HB', 0.5)]},
    'SA': {'Perform': [('P', 1.0)]}, 'SB': {'Perform': [('P',
1.0)]}, 'SC': {'Perform': [('P', 1.0)]},
    'P': {'F': [('SA', 0.33), ('SB', 0.33), ('SC', 0.34)], 'O':
[('HA', 0.33), ('HB', 0.33), ('HC', 0.34)]},
    }

# Function to simulate state transition
def simulate_transition(current_state, action,
transitions_stochastic):
    if action in transitions_stochastic[current_state]:
        possible_transitions =
transitions_stochastic[current_state][action]
        next_states, probabilities = zip(*possible_transitions)
        next_state = random.choices(next_states,
weights=probabilities)[0]
        return next_state
    return current_state  # Return current state if no action is
possible

num_samples_per_state = 10

def generate_samples(states, transitions_stochastic,
num_samples_per_state):
    samples = []

    for state in states:
        for _ in range(num_samples_per_state):
            if state not in transitions_stochastic:
                continue
            action =
random.choice(list(transitions_stochastic[state].keys()))
            next_state = simulate_transition(state, action,
transitions_stochastic)
            samples.append((state, action, next_state))

    return samples

# Generate a set of samples from the MDP
sampled_transitions = generate_samples(states,
transitions_stochastic, num_samples_per_state)

# Print some samples to verify
print("Sampled Transitions:")
for sample in sampled_transitions[:]:
    print(sample)

# Initialize Q-values
Q = np.zeros((len(states), len(actions)))

# Hyperparameters for Q-learning
alpha = 0.2  # Learning rate
gamma = 0.1  # Discount factor
initial_epsilon = 1.0  # Start with 100% exploration
decay_rate = 0.99  # Decay rate per episode
min_epsilon = 0.01  # Minimum value of epsilon

def update_q_value(prev_state, action, reward, next_state, Q,
alpha, gamma):
    prev_state_idx = state_indices[prev_state]
    action_idx = action_indices[action]
    next_state_idx = state_indices[next_state]

    # Get the maximum Q-value for the next state
    max_future_q = np.max(Q[next_state_idx])

    # Update the Q-value for the previous state and action
    Q[prev_state_idx, action_idx] += alpha * (reward + gamma *
max_future_q - Q[prev_state_idx, action_idx])

def epsilon_greedy_policy(state, epsilon, Q):
    # With probability epsilon, select a random action
    if random.random() < epsilon:
        return random.choice(actions)
    # With probability 1 - epsilon, select the action with the
maximum Q-value
    state_idx = state_indices[state]
    return actions[np.argmax(Q[state_idx])]

def train_q_learning(Q, sampled_transitions, alpha, gamma, epsilon,
decay_rate, min_epsilon, epochs):
    cumulative_rewards = []

    for epoch in range(epochs):
        total_reward = 0
        np.random.shuffle(sampled_transitions)  # Shuffle samples
each epoch for better learning

        for (prev_state, action, next_state) in
sampled_transitions:
            # Select an action using epsilon-greedy policy
            action = epsilon_greedy_policy(prev_state, epsilon, Q)
            reward = rewards[state_indices[prev_state],
action_indices[action]]
            total_reward += reward
            update_q_value(prev_state, action, reward, next_state,
Q, alpha, gamma)
        cumulative_rewards.append(total_reward)

        # Exponentially decay epsilon
        epsilon = max(min_epsilon, epsilon * decay_rate)

    return cumulative_rewards, epsilon

epochs = 500

# Train and get the rewards history
rewards_history, final_epsilon = train_q_learning(Q,
sampled_transitions, alpha, gamma, initial_epsilon, decay_rate,
min_epsilon, epochs)

def derive_policy(Q, states, actions):
    policy = {}
    for idx, state in enumerate(states):
        best_action_idx = np.argmax(Q[idx])
        policy[state] = actions[best_action_idx]
    return policy

# Derive the optimal policy
optimal_policy = derive_policy(Q, states, actions)
print("Optimal Policy:", optimal_policy)

def plot_rewards(rewards_history):
    plt.figure(figsize=(10, 5))
    plt.plot(rewards_history, label='Cumulative Reward per
Episode')
    plt.xlabel('Episode')
    plt.ylabel('Cumulative Reward')
    plt.title('Episode vs Cumulative Reward')
    plt.legend()
    plt.grid(True)
    plt.show()

# Plot the cumulative rewards
plot_rewards(rewards_history)

# Convert Q-table to a pandas DataFrame for better visualization
def print_q_table(Q, states, actions):
    df = pd.DataFrame(Q, index=states, columns=actions)
    return df


q_table_df = print_q_table(Q, states, actions)

# Adjust display settings to show all rows and columns if needed
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)
pd.set_option('display.colheader_justify', 'center')
pd.set_option('display.precision', 3)

print(q_table_df)
```

SBP with Q-learning Code for Task Sequence MDP:

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

# states and actions
states = ['charging_station', 'warehouse_kitting',
'assemble_station_t', 'assemble_station_p'
          , 'EOL_t' , 'EOL', 'successed_quality_control',
'final_warehouse', 'warehouse_mismatch' , 'idle', 'restocking'
          , 'failed_quality_control', 'mismatch', 'EOL_corrected',
'urgent', 'system_failure'
          , 'maintenance', 'under_threshold', 'above_threshold']
actions = ['move_to_grap_the_component_parts',
'taking_components_to_assemble', 'moving_the_tool_box',
           'move_to_grap_the_assembled_product',
'moving_the_assembled_product_to_measure',
           'move_to_check_the_quality',
'taking_the_product_to_the_shelf',
'returning_to_the_initial_position',
           'taking_the_wrong_component', 'returning_to_warehouse',
'wait', 'move', 'move_to_measure_again'
           , 'moving_to_measure', 'moving_to_replace',
'mandatory_measurement'
           , 'mandatory_quality_control', 'move_to_warehouse',
'unexpected_failure', 'continue_working'
           , 'move_for_maintenance', 'check_the_battery',
'return_to_work', 'move_for_charging']
state_indices = {state: idx for idx, state in enumerate(states)}
action_indices = {action: idx for idx, action in
enumerate(actions)}

# Initialize rewards based on description
rewards = np.zeros((len(states), len(actions)))
rewards[state_indices['charging_station'],
action_indices['move_to_grap_the_component_parts']] = 1
rewards[state_indices['charging_station'], action_indices['move']]
= 0
rewards[state_indices['urgent'],
action_indices['move_to_warehouse']] = -20
rewards[state_indices['urgent'],
action_indices['mandatory_quality_control']] = 1
rewards[state_indices['urgent'],
action_indices['mandatory_measurement']] = 1
rewards[state_indices['warehouse_kitting'], action_indices['move']]
= 0
rewards[state_indices['restocking'], action_indices['wait']] = -20
rewards[state_indices['restocking'],
action_indices['moving_to_measure']] = 1
rewards[state_indices['warehouse_kitting'],
action_indices['taking_components_to_assemble']] = 1
rewards[state_indices['assemble_station_t'],
action_indices['moving_the_tool_box']] = 1
```

```
rewards[state_indices['EOL_t'],
action_indices['move_to_grap_the_assembled_product']] = 1
rewards[state_indices['assemble_station_p'],
action_indices['move']] = 0
rewards[state_indices['mismatch'],
action_indices['moving_to_measure']] = -20
rewards[state_indices['mismatch'],
action_indices['moving_to_replace']] = 1
rewards[state_indices['warehouse_mismatch'],
action_indices['taking_components_to_assemble']] = 1
rewards[state_indices['assemble_station_p'],
action_indices['moving_the_assembled_product_to_measure']] = 1
rewards[state_indices['idle'], action_indices['wait']] = -20
rewards[state_indices['idle'], action_indices['moving_to_measure']]
= 1
rewards[state_indices['assemble_station_p'],
action_indices['move']] = 1
rewards[state_indices['EOL'],
action_indices['move_to_check_the_quality']] = 1
rewards[state_indices['successed_quality_control'],
action_indices['taking_the_product_to_the_shelf']] = 1
rewards[state_indices['failed_quality_control'],
action_indices['move_to_measure_again']] = 1
rewards[state_indices['failed_quality_control'],
action_indices['taking_the_product_to_the_shelf']] = -20
rewards[state_indices['EOL_corrected'],
action_indices['taking_the_product_to_the_shelf']] = 1
rewards[state_indices['final_warehouse'],
action_indices['returning_to_the_initial_position']] = 1
rewards[state_indices['EOL'], action_indices['unexpected_failure']]
= 0
rewards[state_indices['system_failure'],
action_indices['continue_working']] = -20
rewards[state_indices['system_failure'],
action_indices['move_for_maintenance']] = 1
rewards[state_indices['maintenance'],
action_indices['check_the_battery']] = 1
rewards[state_indices['under_threshold'],
action_indices['return_to_work']] = -20
rewards[state_indices['above_threshold'],
action_indices['move_for_charging']] = -20
rewards[state_indices['under_threshold'],
action_indices['move_for_charging']] = 1
rewards[state_indices['above_threshold'],
action_indices['return_to_work']] = 1

# Stochastic transitions
transitions_stochastic = {
    'charging_station': {'move_to_grap_the_component_parts':
[('warehouse_kitting', 1.0)]
                         , 'move':[('urgent', 1.0)]},
    'urgent': {'move_to_warehouse': [('warehouse_kitting', 1.0)],
'mandatory_quality_control': [('failed_quality_control', 0.1),
('successed_quality_control', 0.9)]
              , 'mandatory_measurement': [('EOL', 1.0)]},
    'warehouse_kitting': {'move': [('restocking', 1.0)],
'taking_components_to_assemble': [('assemble_station_t', 1.0)]},
    'restocking': {'wait': [('warehouse_kitting', 1.0)],
'moving_to_measure': [('EOL', 1.0)]},
    'assemble_station_t': {'moving_the_tool_box': [('EOL_t',
1.0)]},
    'EOL_t': {'move_to_grap_the_assembled_product':
[('assemble_station_p', 1.0)]},
    'assemble_station_p': {'move': [('mismatch', 0.5), ('idle',
0.5)], 'moving_the_assembled_product_to_measure': [('EOL', 1.0)]},
    'mismatch': {'moving_to_replace': [('warehouse_mismatch',
1.0)], 'moving_to_measure': [('EOL', 1.0)]},
    'warehouse_mismatch': {'taking_components_to_assemble':
[('assemble_station_p', 1.0)]},
    'idle': {'wait': [('assemble_station_p', 1.0)],
'moving_to_measure': [('EOL', 1.0)]},
    'EOL': {'move_to_check_the_quality':
[('failed_quality_control', 0.1), ('successed_quality_control',
0.9)], 'unexpected_failure': [('system_failure', 1.0)]},
    'failed_quality_control': {'taking_the_product_to_the_shelf':
[('final_warehouse', 1.0)], 'move_to_measure_again':
[('EOL_corrected', 1.0)]},
    'EOL_corrected': {'taking_the_product_to_the_shelf':
[('final_warehouse', 1.0)]},
    'final_warehouse': {'returning_to_the_initial_position':
[('charging_station', 1.0)]},
    'successed_quality_control':
{'taking_the_product_to_the_shelf': [('final_warehouse', 1.0)]},
    'system_failure': {'continue_working': [('EOL', 1.0)],
'move_for_maintenance': [('maintenance', 1.0)]},
    'maintenance': {'check_the_battery': [('under_threshold', 0.5),
('above_threshold', 0.5)]},
    'under_threshold': {'return_to_work': [('urgent', 1.0)],
'move_for_charging': [('charging_station', 1.0)]},
    'above_threshold': {'return_to_work': [('urgent', 1.0)],
'move_for_charging': [('charging_station', 1.0)]}
    }

# Function to simulate state transition
def simulate_transition(current_state, action,
transitions_stochastic):
    if action in transitions_stochastic[current_state]:
        possible_transitions =
transitions_stochastic[current_state][action]
        next_states, probabilities = zip(*possible_transitions)
        next_state = random.choices(next_states,
weights=probabilities)[0]
        return next_state
    return current_state  # Return current state if no action is
possible

num_samples_per_state = 3

def generate_samples(states, transitions_stochastic,
num_samples_per_state):
    samples = []

    for state in states:
        for _ in range(num_samples_per_state):
            if state not in transitions_stochastic:
                continue
            action =
random.choice(list(transitions_stochastic[state].keys()))
            next_state = simulate_transition(state, action,
transitions_stochastic)
            samples.append((state, action, next_state))

    return samples

# Generate a set of samples from the MDP
sampled_transitions = generate_samples(states,
transitions_stochastic, num_samples_per_state)

# Print some samples to verify
print("Sampled Transitions:")
for sample in sampled_transitions[:]:
    print(sample)

# Initialize Q-values
Q = np.zeros((len(states), len(actions)))

# Hyperparameters for Q-learning
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
initial_epsilon = 1.0  # Start with 100% exploration
decay_rate = 0.99  # Decay rate per episode
min_epsilon = 0.01  # Minimum value of epsilon

def update_q_value(prev_state, action, reward, next_state, Q,
alpha, gamma):
    prev_state_idx = state_indices[prev_state]
    action_idx = action_indices[action]
    next_state_idx = state_indices[next_state]

    # Get the maximum Q-value for the next state
    max_future_q = np.max(Q[next_state_idx])

    # Update the Q-value for the previous state and action
    Q[prev_state_idx, action_idx] += alpha * (reward + gamma *
max_future_q - Q[prev_state_idx, action_idx])

def epsilon_greedy_policy(state, epsilon, Q):
    # With probability epsilon, select a random action
    if random.random() < epsilon:
        return random.choice(actions)
    # With probability 1 - epsilon, select the action with the
maximum Q-value
    state_idx = state_indices[state]
    return actions[np.argmax(Q[state_idx])]

def train_q_learning(Q, sampled_transitions, alpha, gamma, epsilon,
decay_rate, min_epsilon, epochs):
    cumulative_rewards = []

    for epoch in range(epochs):
        total_reward = 0
        np.random.shuffle(sampled_transitions)  # Shuffle samples
each epoch for better learning

        for (prev_state, action, next_state) in
sampled_transitions:
            # Select an action using epsilon-greedy policy
            action = epsilon_greedy_policy(prev_state, epsilon, Q)
            reward = rewards[state_indices[prev_state],
action_indices[action]]
            total_reward += reward
            update_q_value(prev_state, action, reward, next_state,
Q, alpha, gamma)
        cumulative_rewards.append(total_reward)

        # Exponentially decay epsilon
        epsilon = max(min_epsilon, epsilon * decay_rate)

    return cumulative_rewards, epsilon

epochs = 500

# Train and get the rewards history
rewards_history, final_epsilon = train_q_learning(Q,
sampled_transitions, alpha, gamma, initial_epsilon, decay_rate,
min_epsilon, epochs)

def derive_policy(Q, states, actions):
    policy = {}
    for idx, state in enumerate(states):
        best_action_idx = np.argmax(Q[idx])
        policy[state] = actions[best_action_idx]
    return policy

# Derive the optimal policy
optimal_policy = derive_policy(Q, states, actions)
print("Optimal Policy:")
for state, action in optimal_policy.items():
    print(f"State: {state}, Best Action: {action}")

def plot_rewards(rewards_history):
    plt.figure(figsize=(10, 5))
    plt.plot(rewards_history, label='Cumulative Reward per
Episode')
    plt.xlabel('Episode')
    plt.ylabel('Cumulative Reward')
    plt.title('Episode vs Cumulative Reward')
    plt.legend()
    plt.grid(True)
    plt.show()

# Plot the cumulative rewards
plot_rewards(rewards_history)

# Convert Q-table to a pandas DataFrame for better visualization
def print_q_table(Q, states, actions):
    df = pd.DataFrame(Q, index=states, columns=actions)
    return df
```

```
q_table_df = print_q_table(Q, states, actions)

# Adjust display settings to show all rows and columns if needed
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)
pd.set_option('display.colheader_justify', 'center')
pd.set_option('display.precision', 3)

print(q_table_df)
```

SARSA for Task Sequence MDP:

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

# states and actions
states = ['charging_station', 'warehouse_kitting',
'assemble_station_t', 'assemble_station_p'
          , 'EOL_t' , 'EOL', 'successed_quality_control',
'final_warehouse', 'warehouse_mismatch' , 'idle', 'restocking'
          , 'failed_quality_control', 'mismatch', 'EOL_corrected',
'urgent', 'system_failure'
          , 'maintenance', 'under_threshold', 'above_threshold']
actions = ['move_to_grap_the_component_parts',
'taking_components_to_assemble', 'moving_the_tool_box',
           'move_to_grap_the_assembled_product',
'moving_the_assembled_product_to_measure',
           'move_to_check_the_quality',
'taking_the_product_to_the_shelf',
'returning_to_the_initial_position',
           'taking_the_wrong_component', 'returning_to_warehouse',
'wait', 'move', 'move_to_measure_again'
           , 'moving_to_measure', 'moving_to_replace',
'mandatory_measurement'
           , 'mandatory_quality_control', 'move_to_warehouse',
'unexpected_failure', 'continue_working'
           , 'move_for_maintenance', 'check_the_battery',
'return_to_work', 'move_for_charging']
state_indices = {state: idx for idx, state in enumerate(states)}
action_indices = {action: idx for idx, action in
enumerate(actions)}

# Initialize rewards based on description
rewards = np.zeros((len(states), len(actions)))
rewards[state_indices['charging_station'],
action_indices['move_to_grap_the_component_parts']] = 1
rewards[state_indices['charging_station'], action_indices['move']]
= 0
rewards[state_indices['urgent'],
action_indices['move_to_warehouse']] = -20
rewards[state_indices['urgent'],
action_indices['mandatory_quality_control']] = 1
rewards[state_indices['urgent'],
action_indices['mandatory_measurement']] = 1
rewards[state_indices['warehouse_kitting'], action_indices['move']]
= 0
rewards[state_indices['restocking'], action_indices['wait']] = -20
rewards[state_indices['restocking'],
action_indices['moving_to_measure']] = 1
rewards[state_indices['warehouse_kitting'],
action_indices['taking_components_to_assemble']] = 1
rewards[state_indices['assemble_station_t'],
action_indices['moving_the_tool_box']] = 1
```

```
rewards[state_indices['EOL_t'],
action_indices['move_to_grap_the_assembled_product']] = 1
rewards[state_indices['assemble_station_p'],
action_indices['move']] = 0
rewards[state_indices['mismatch'],
action_indices['moving_to_measure']] = -20
rewards[state_indices['mismatch'],
action_indices['moving_to_replace']] = 1
rewards[state_indices['warehouse_mismatch'],
action_indices['taking_components_to_assemble']] = 1
rewards[state_indices['assemble_station_p'],
action_indices['moving_the_assembled_product_to_measure']] = 1
rewards[state_indices['idle'], action_indices['wait']] = -20
rewards[state_indices['idle'], action_indices['moving_to_measure']]
= 1
rewards[state_indices['assemble_station_p'],
action_indices['move']] = 1
rewards[state_indices['EOL'],
action_indices['move_to_check_the_quality']] = 1
rewards[state_indices['successed_quality_control'],
action_indices['taking_the_product_to_the_shelf']] = 1
rewards[state_indices['failed_quality_control'],
action_indices['move_to_measure_again']] = 1
rewards[state_indices['failed_quality_control'],
action_indices['taking_the_product_to_the_shelf']] = -20
rewards[state_indices['EOL_corrected'],
action_indices['taking_the_product_to_the_shelf']] = 1
rewards[state_indices['final_warehouse'],
action_indices['returning_to_the_initial_position']] = 1
rewards[state_indices['EOL'], action_indices['unexpected_failure']]
= 0
rewards[state_indices['system_failure'],
action_indices['continue_working']] = -20
rewards[state_indices['system_failure'],
action_indices['move_for_maintenance']] = 1
rewards[state_indices['maintenance'],
action_indices['check_the_battery']] = 1
rewards[state_indices['under_threshold'],
action_indices['return_to_work']] = -20
rewards[state_indices['above_threshold'],
action_indices['move_for_charging']] = -20
rewards[state_indices['under_threshold'],
action_indices['move_for_charging']] = 1
rewards[state_indices['above_threshold'],
action_indices['return_to_work']] = 1

# Stochastic transitions
transitions_stochastic = {
    'charging_station': {'move_to_grap_the_component_parts':
[('warehouse_kitting', 1.0)]
                          , 'move':[('urgent', 1.0)]},
    'urgent': {'move_to_warehouse': [('warehouse_kitting', 1.0)],
'mandatory_quality_control': [('failed_quality_control', 0.1),
('successed_quality_control', 0.9)]
               , 'mandatory_measurement': [('EOL', 1.0)]},
    'warehouse_kitting': {'move': [('restocking', 1.0)],
'taking_components_to_assemble': [('assemble_station_t', 1.0)]},
    'restocking': {'wait': [('warehouse_kitting', 1.0)],
'moving_to_measure': [('EOL', 1.0)]},
    'assemble_station_t': {'moving_the_tool_box': [('EOL_t',
1.0)]},
    'EOL_t': {'move_to_grap_the_assembled_product':
[('assemble_station_p', 1.0)]},
    'assemble_station_p': {'move': [('mismatch', 0.5), ('idle',
0.5)], 'moving_the_assembled_product_to_measure': [('EOL', 1.0)]},
    'mismatch': {'moving_to_replace': [('warehouse_mismatch',
1.0)], 'moving_to_measure': [('EOL', 1.0)]},
    'warehouse_mismatch': {'taking_components_to_assemble':
[('assemble_station_p', 1.0)]},
    'idle': {'wait': [('assemble_station_p', 1.0)],
'moving_to_measure': [('EOL', 1.0)]},
    'EOL': {'move_to_check_the_quality':
[('failed_quality_control', 0.1), ('successed_quality_control',
0.9)], 'unexpected_failure': [('system_failure', 1.0)]},
    'failed_quality_control': {'taking_the_product_to_the_shelf':
[('final_warehouse', 1.0)], 'move_to_measure_again':
[('EOL_corrected', 1.0)]},
    'EOL_corrected': {'taking_the_product_to_the_shelf':
[('final_warehouse', 1.0)]},
    'final_warehouse': {'returning_to_the_initial_position':
[('charging_station', 1.0)]},
    'successed_quality_control':
{'taking_the_product_to_the_shelf': [('final_warehouse', 1.0)]},
    'system_failure': {'continue_working': [('EOL', 1.0)],
'move_for_maintenance': [('maintenance', 1.0)]},
    'maintenance': {'check_the_battery': [('under_threshold', 0.5),
('above_threshold', 0.5)]},
    'under_threshold': {'return_to_work': [('urgent', 1.0)],
'move_for_charging': [('charging_station', 1.0)]},
    'above_threshold': {'return_to_work': [('urgent', 1.0)],
'move_for_charging': [('charging_station', 1.0)]}
    }

# Function to simulate state transition
def simulate_transition(current_state, action,
transitions_stochastic):
    if action in transitions_stochastic[current_state]:
        possible_transitions =
transitions_stochastic[current_state][action]
        next_states, probabilities = zip(*possible_transitions)
        next_state = random.choices(next_states,
weights=probabilities)[0]
        return next_state
    return current_state  # Return current state if no action is
possible

num_samples_per_state = 3

def generate_samples(states, transitions_stochastic,
num_samples_per_state):
    samples = []

    for state in states:
        for _ in range(num_samples_per_state):
            if state not in transitions_stochastic:
                continue
            action =
random.choice(list(transitions_stochastic[state].keys()))
            next_state = simulate_transition(state, action,
transitions_stochastic)
            samples.append((state, action, next_state))

    return samples

# Generate a set of samples from the MDP
sampled_transitions = generate_samples(states,
transitions_stochastic, num_samples_per_state)

# Print some samples to verify
print("Sampled Transitions:")
for sample in sampled_transitions[:]:
    print(sample)

# Initialize Q-values
Q = np.zeros((len(states), len(actions)))

# Hyperparameters for SARSA
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
initial_epsilon = 1.0  # Start with 100% exploration
decay_rate = 0.99  # Decay rate per episode
min_epsilon = 0.01  # Minimum value of epsilon

def update_q_value_sarsa(prev_state, action, reward, next_state,
next_action, Q, alpha, gamma):
    prev_state_idx = state_indices[prev_state]
    action_idx = action_indices[action]
    next_state_idx = state_indices[next_state]
    next_action_idx = action_indices[next_action]

    # Get the Q-value for the next state and action
    future_q = Q[next_state_idx, next_action_idx]

    # Update the Q-value for the previous state and action
    Q[prev_state_idx, action_idx] += alpha * (reward + gamma *
future_q - Q[prev_state_idx, action_idx])

def epsilon_greedy_policy(state, epsilon, Q):
    # With probability epsilon, select a random action
    if random.random() < epsilon:
        return random.choice(actions)
    # With probability 1 - epsilon, select the action with the
maximum Q-value
    state_idx = state_indices[state]
    return actions[np.argmax(Q[state_idx])]

def train_sarsa(Q, sampled_transitions, alpha, gamma, epsilon,
decay_rate, min_epsilon, epochs):
    cumulative_rewards = []

    for epoch in range(epochs):
        total_reward = 0
        np.random.shuffle(sampled_transitions)  # Shuffle samples
each epoch for better learning

        for (prev_state, action, next_state) in
sampled_transitions:
            # Select an action using epsilon-greedy policy
            action = epsilon_greedy_policy(prev_state, epsilon, Q)
            reward = rewards[state_indices[prev_state],
action_indices[action]]
            total_reward += reward
            next_action = epsilon_greedy_policy(next_state,
epsilon, Q)
            update_q_value_sarsa(prev_state, action, reward,
next_state, next_action, Q, alpha, gamma)
        cumulative_rewards.append(total_reward)

        # Exponentially decay epsilon
        epsilon = max(min_epsilon, epsilon * decay_rate)

    return cumulative_rewards, epsilon

epochs = 500

# Train and get the rewards history
rewards_history, final_epsilon = train_sarsa(Q,
sampled_transitions, alpha, gamma, initial_epsilon, decay_rate,
min_epsilon, epochs)

def derive_policy(Q, states, actions):
    policy = {}
    for idx, state in enumerate(states):
        best_action_idx = np.argmax(Q[idx])
        policy[state] = actions[best_action_idx]
    return policy

# Derive the optimal policy
optimal_policy = derive_policy(Q, states, actions)
print("Optimal Policy:")
for state, action in optimal_policy.items():
    print(f"State: {state}, Best Action: {action}")

def plot_rewards(rewards_history):
    plt.figure(figsize=(10, 5))
    plt.plot(rewards_history, label='Cumulative Reward per
Episode')
    plt.xlabel('Episode')
    plt.ylabel('Cumulative Reward')
    plt.title('Episode vs Cumulative Reward')
    plt.legend()
    plt.grid(True)
    plt.show()

# Plot the cumulative rewards
plot_rewards(rewards_history)

# Convert Q-table to a pandas DataFrame for better visualization
```

```python
def print_q_table(Q, states, actions):
    df = pd.DataFrame(Q, index=states, columns=actions)
    return df

q_table_df = print_q_table(Q, states, actions)

# Adjust display settings to show all rows and columns if needed
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)
pd.set_option('display.colheader_justify', 'center')
pd.set_option('display.precision', 3)

print(q_table_df)
```

SARSA for Task Sequence MDP + Hyperparameter Tuning Comparison:

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

# States and actions
states = ['charging_station', 'warehouse_kitting', 'assemble_station_t',
'assemble_station_p', 'EOL_t', 'EOL',
          'successed_quality_control', 'final_warehouse',
'warehouse_mismatch', 'idle', 'restocking',
          'failed_quality_control', 'mismatch', 'EOL_corrected', 'urgent',
'system_failure', 'maintenance',
          'under_threshold', 'above_threshold']
actions = ['move_to_grap_the_component_parts',
'taking_components_to_assemble', 'moving_the_tool_box',
           'move_to_grap_the_assembled_product',
'moving_the_assembled_product_to_measure', 'move_to_check_the_quality',
           'taking_the_product_to_the_shelf',
'returning_to_the_initial_position', 'taking_the_wrong_component',
           'returning_to_warehouse', 'wait', 'move',
'move_to_measure_again', 'moving_to_measure', 'moving_to_replace',
           'mandatory_measurement', 'mandatory_quality_control',
'move_to_warehouse', 'unexpected_failure',
           'continue_working', 'move_for_maintenance',
'check_the_battery', 'return_to_work', 'move_for_charging']
state_indices = {state: idx for idx, state in enumerate(states)}
action_indices = {action: idx for idx, action in enumerate(actions)}

# Initialize rewards based on description
rewards = np.zeros((len(states), len(actions)))
rewards[state_indices['charging_station'],
action_indices['move_to_grap_the_component_parts']] = 1
rewards[state_indices['charging_station'], action_indices['move']] = 0
rewards[state_indices['urgent'], action_indices['move_to_warehouse']] = -
20
rewards[state_indices['urgent'],
action_indices['mandatory_quality_control']] = 1
rewards[state_indices['urgent'], action_indices['mandatory_measurement']]
= 1
rewards[state_indices['warehouse_kitting'], action_indices['move']] = 0
rewards[state_indices['restocking'], action_indices['wait']] = -20
rewards[state_indices['restocking'], action_indices['moving_to_measure']]
= 1
rewards[state_indices['warehouse_kitting'],
action_indices['taking_components_to_assemble']] = 1
rewards[state_indices['assemble_station_t'],
action_indices['moving_the_tool_box']] = 1
```

```
rewards[state_indices['EOL_t'],
action_indices['move_to_grap_the_assembled_product']] = 1
rewards[state_indices['assemble_station_p'], action_indices['move']] = 0
rewards[state_indices['mismatch'], action_indices['moving_to_measure']] =
-20
rewards[state_indices['mismatch'], action_indices['moving_to_replace']] =
1
rewards[state_indices['warehouse_mismatch'],
action_indices['taking_components_to_assemble']] = 1
rewards[state_indices['assemble_station_p'],
action_indices['moving_the_assembled_product_to_measure']] = 1
rewards[state_indices['idle'], action_indices['wait']] = -20
rewards[state_indices['idle'], action_indices['moving_to_measure']] = 1
rewards[state_indices['assemble_station_p'], action_indices['move']] = 1
rewards[state_indices['EOL'], action_indices['move_to_check_the_quality']]
= 1
rewards[state_indices['successed_quality_control'],
action_indices['taking_the_product_to_the_shelf']] = 1
rewards[state_indices['failed_quality_control'],
action_indices['move_to_measure_again']] = 1
rewards[state_indices['failed_quality_control'],
action_indices['taking_the_product_to_the_shelf']] = -20
rewards[state_indices['EOL_corrected'],
action_indices['taking_the_product_to_the_shelf']] = 1
rewards[state_indices['final_warehouse'],
action_indices['returning_to_the_initial_position']] = 1
rewards[state_indices['EOL'], action_indices['unexpected_failure']] = 0
rewards[state_indices['system_failure'],
action_indices['continue_working']] = -20
rewards[state_indices['system_failure'],
action_indices['move_for_maintenance']] = 1
rewards[state_indices['maintenance'], action_indices['check_the_battery']]
= 1
rewards[state_indices['under_threshold'],
action_indices['return_to_work']] = -20
rewards[state_indices['above_threshold'],
action_indices['move_for_charging']] = -20
rewards[state_indices['under_threshold'],
action_indices['move_for_charging']] = 1
rewards[state_indices['above_threshold'],
action_indices['return_to_work']] = 1

# Stochastic transitions
transitions_stochastic = {
    'charging_station': {'move_to_grap_the_component_parts':
[('warehouse_kitting', 1.0)], 'move': [('urgent', 1.0)]},
    'urgent': {'move_to_warehouse': [('warehouse_kitting', 1.0)],
'mandatory_quality_control': [('failed_quality_control', 0.1),
('successed_quality_control', 0.9)],
               'mandatory_measurement': [('EOL', 1.0)]},
    'warehouse_kitting': {'move': [('restocking', 1.0)],
'taking_components_to_assemble': [('assemble_station_t', 1.0)]},
    'restocking': {'wait': [('warehouse_kitting', 1.0)],
'moving_to_measure': [('EOL', 1.0)]},
    'assemble_station_t': {'moving_the_tool_box': [('EOL_t', 1.0)]},
    'EOL_t': {'move_to_grap_the_assembled_product':
[('assemble_station_p', 1.0)]},
    'assemble_station_p': {'move': [('mismatch', 0.5), ('idle', 0.5)],
'moving_the_assembled_product_to_measure': [('EOL', 1.0)]},
    'mismatch': {'moving_to_replace': [('warehouse_mismatch', 1.0)],
'moving_to_measure': [('EOL', 1.0)]},
    'warehouse_mismatch': {'taking_components_to_assemble':
[('assemble_station_p', 1.0)]},
    'idle': {'wait': [('assemble_station_p', 1.0)], 'moving_to_measure':
[('EOL', 1.0)]},
    'EOL': {'move_to_check_the_quality': [('failed_quality_control', 0.1),
('successed_quality_control', 0.9)], 'unexpected_failure':
[('system_failure', 1.0)]},
    'failed_quality_control': {'taking_the_product_to_the_shelf':
[('final_warehouse', 1.0)], 'move_to_measure_again': [('EOL_corrected',
1.0)]},
    'EOL_corrected': {'taking_the_product_to_the_shelf':
[('final_warehouse', 1.0)]},
    'final_warehouse': {'returning_to_the_initial_position':
[('charging_station', 1.0)]},
    'successed_quality_control': {'taking_the_product_to_the_shelf':
[('final_warehouse', 1.0)]},
    'system_failure': {'continue_working': [('EOL', 1.0)],
'move_for_maintenance': [('maintenance', 1.0)]},
    'maintenance': {'check_the_battery': [('under_threshold', 0.5),
('above_threshold', 0.5)]},
    'under_threshold': {'return_to_work': [('urgent', 1.0)],
'move_for_charging': [('charging_station', 1.0)]},
    'above_threshold': {'return_to_work': [('urgent', 1.0)],
'move_for_charging': [('charging_station', 1.0)]}
}

# Function to simulate state transition
def simulate_transition(current_state, action, transitions_stochastic):
    if action in transitions_stochastic[current_state]:
        possible_transitions =
transitions_stochastic[current_state][action]
        next_states, probabilities = zip(*possible_transitions)
        next_state = random.choices(next_states, weights=probabilities)[0]
        return next_state
    return current_state  # Return current state if no action is possible

num_samples_per_state = 3

def generate_samples(states, transitions_stochastic,
num_samples_per_state):
    samples = []

    for state in states:
        for _ in range(num_samples_per_state):
            if state not in transitions_stochastic:
                continue
            action =
random.choice(list(transitions_stochastic[state].keys()))
            next_state = simulate_transition(state, action,
transitions_stochastic)
            samples.append((state, action, next_state))

    return samples

# Generate a set of samples from the MDP
sampled_transitions = generate_samples(states, transitions_stochastic,
num_samples_per_state)

# Print some samples to verify
print("Sampled Transitions:")
for sample in sampled_transitions[:]:
    print(sample)

# Hyperparameters for SARSA
alpha_1, gamma_1, initial_epsilon_1, decay_rate_1, min_epsilon_1 = 0.1,
0.9, 1.0, 0.99, 0.01
alpha_2, gamma_2, initial_epsilon_2, decay_rate_2, min_epsilon_2 = 0.5,
0.9, 1.0, 0.99, 0.01

epochs = 500

def update_q_value_sarsa(prev_state, action, reward, next_state,
next_action, Q, alpha, gamma):
    prev_state_idx = state_indices[prev_state]
    action_idx = action_indices[action]
    next_state_idx = state_indices[next_state]
    next_action_idx = action_indices[next_action]

    # Get the Q-value for the next state and action
    future_q = Q[next_state_idx, next_action_idx]

    # Update the Q-value for the previous state and action
    Q[prev_state_idx, action_idx] += alpha * (reward + gamma * future_q -
Q[prev_state_idx, action_idx])

def epsilon_greedy_policy(state, epsilon, Q):
    # With probability epsilon, select a random action
    if random.random() < epsilon:
        return random.choice(actions)
    # With probability 1 - epsilon, select the action with the maximum Q-
value
```

```
    state_idx = state_indices[state]
    return actions[np.argmax(Q[state_idx])]

def train_sarsa(Q, sampled_transitions, alpha, gamma, epsilon, decay_rate,
min_epsilon, epochs):
    cumulative_rewards = []

    for epoch in range(epochs):
        total_reward = 0
        np.random.shuffle(sampled_transitions)  # Shuffle samples each
epoch for better learning

        for (prev_state, action, next_state) in sampled_transitions:
            # Select an action using epsilon-greedy policy
            action = epsilon_greedy_policy(prev_state, epsilon, Q)
            reward = rewards[state_indices[prev_state],
action_indices[action]]
            total_reward += reward
            next_action = epsilon_greedy_policy(next_state, epsilon, Q)
            update_q_value_sarsa(prev_state, action, reward, next_state,
next_action, Q, alpha, gamma)
        cumulative_rewards.append(total_reward)

        # Exponentially decay epsilon
        epsilon = max(min_epsilon, epsilon * decay_rate)

    return cumulative_rewards, epsilon

def derive_policy(Q, states, actions):
    policy = {}
    for idx, state in enumerate(states):
        best_action_idx = np.argmax(Q[idx])
        policy[state] = actions[best_action_idx]
    return policy

# Train with the first set of hyperparameters
Q_1 = np.zeros((len(states), len(actions)))
rewards_history_1, final_epsilon_1 = train_sarsa(Q_1, sampled_transitions,
alpha_1, gamma_1, initial_epsilon_1, decay_rate_1, min_epsilon_1, epochs)
optimal_policy_1 = derive_policy(Q_1, states, actions)

# Train with the second set of hyperparameters
Q_2 = np.zeros((len(states), len(actions)))
rewards_history_2, final_epsilon_2 = train_sarsa(Q_2, sampled_transitions,
alpha_2, gamma_2, initial_epsilon_2, decay_rate_2, min_epsilon_2, epochs)
optimal_policy_2 = derive_policy(Q_2, states, actions)

# Plot the cumulative rewards for comparison
plt.figure(figsize=(12, 8))
plt.plot(rewards_history_1, label=f'alpha={alpha_1}, gamma={gamma_1},
epsilon={initial_epsilon_1}, decay={decay_rate_1},
min_epsilon={min_epsilon_1}')
plt.plot(rewards_history_2, label=f'alpha={alpha_2}, gamma={gamma_2},
epsilon={initial_epsilon_2}, decay={decay_rate_2},
min_epsilon={min_epsilon_2}')
plt.xlabel('Episode')
plt.ylabel('Cumulative Reward')
plt.title('Episode vs Cumulative Reward for Different Hyperparameters')
plt.legend()
plt.grid(True)
plt.show()

# Print the optimal policy for both sets of hyperparameters
print("Optimal Policy for first set of hyperparameters:")
for state, action in optimal_policy_1.items():
    print(f"State: {state}, Best Action: {action}")

print("\nOptimal Policy for second set of hyperparameters:")
for state, action in optimal_policy_2.items():
    print(f"State: {state}, Best Action: {action}")
```

SQL for Task Sequence MDP + Hyperparameter Tuning Comparison:

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

# States and actions
states = ['charging_station', 'warehouse_kitting', 'assemble_station_t',
'assemble_station_p', 'EOL_t', 'EOL',
          'successed_quality_control', 'final_warehouse',
'warehouse_mismatch', 'idle', 'restocking',
          'failed_quality_control', 'mismatch', 'EOL_corrected', 'urgent',
'system_failure', 'maintenance',
          'under_threshold', 'above_threshold']
actions = ['move_to_grap_the_component_parts',
'taking_components_to_assemble', 'moving_the_tool_box',
           'move_to_grap_the_assembled_product',
'moving_the_assembled_product_to_measure', 'move_to_check_the_quality',
           'taking_the_product_to_the_shelf',
'returning_to_the_initial_position', 'taking_the_wrong_component',
           'returning_to_warehouse', 'wait', 'move',
'move_to_measure_again', 'moving_to_measure', 'moving_to_replace',
           'mandatory_measurement', 'mandatory_quality_control',
'move_to_warehouse', 'unexpected_failure',
           'continue_working', 'move_for_maintenance',
'check_the_battery', 'return_to_work', 'move_for_charging']
state_indices = {state: idx for idx, state in enumerate(states)}
action_indices = {action: idx for idx, action in enumerate(actions)}

# Initialize rewards based on description
rewards = np.zeros((len(states), len(actions)))
rewards[state_indices['charging_station'],
action_indices['move_to_grap_the_component_parts']] = 1
rewards[state_indices['charging_station'], action_indices['move']] = 0
rewards[state_indices['urgent'], action_indices['move_to_warehouse']] = -
20
rewards[state_indices['urgent'],
action_indices['mandatory_quality_control']] = 1
rewards[state_indices['urgent'], action_indices['mandatory_measurement']]
= 1
rewards[state_indices['warehouse_kitting'], action_indices['move']] = 0
rewards[state_indices['restocking'], action_indices['wait']] = -20
rewards[state_indices['restocking'], action_indices['moving_to_measure']]
= 1
rewards[state_indices['warehouse_kitting'],
action_indices['taking_components_to_assemble']] = 1
rewards[state_indices['assemble_station_t'],
action_indices['moving_the_tool_box']] = 1
```

```
rewards[state_indices['EOL_t'],
action_indices['move_to_grap_the_assembled_product']] = 1
rewards[state_indices['assemble_station_p'], action_indices['move']] = 0
rewards[state_indices['mismatch'], action_indices['moving_to_measure']] =
-20
rewards[state_indices['mismatch'], action_indices['moving_to_replace']] =
1
rewards[state_indices['warehouse_mismatch'],
action_indices['taking_components_to_assemble']] = 1
rewards[state_indices['assemble_station_p'],
action_indices['moving_the_assembled_product_to_measure']] = 1
rewards[state_indices['idle'], action_indices['wait']] = -20
rewards[state_indices['idle'], action_indices['moving_to_measure']] = 1
rewards[state_indices['assemble_station_p'], action_indices['move']] = 1
rewards[state_indices['EOL'], action_indices['move_to_check_the_quality']]
= 1
rewards[state_indices['successed_quality_control'],
action_indices['taking_the_product_to_the_shelf']] = 1
rewards[state_indices['failed_quality_control'],
action_indices['move_to_measure_again']] = 1
rewards[state_indices['failed_quality_control'],
action_indices['taking_the_product_to_the_shelf']] = -20
rewards[state_indices['EOL_corrected'],
action_indices['taking_the_product_to_the_shelf']] = 1
rewards[state_indices['final_warehouse'],
action_indices['returning_to_the_initial_position']] = 1
rewards[state_indices['EOL'], action_indices['unexpected_failure']] = 0
rewards[state_indices['system_failure'],
action_indices['continue_working']] = -20
rewards[state_indices['system_failure'],
action_indices['move_for_maintenance']] = 1
rewards[state_indices['maintenance'], action_indices['check_the_battery']]
= 1
rewards[state_indices['under_threshold'],
action_indices['return_to_work']] = -20
rewards[state_indices['above_threshold'],
action_indices['move_for_charging']] = -20
rewards[state_indices['under_threshold'],
action_indices['move_for_charging']] = 1
rewards[state_indices['above_threshold'],
action_indices['return_to_work']] = 1

# Stochastic transitions
transitions_stochastic = {
    'charging_station': {'move_to_grap_the_component_parts':
[('warehouse_kitting', 1.0)], 'move': [('urgent', 1.0)]},
    'urgent': {'move_to_warehouse': [('warehouse_kitting', 1.0)],
'mandatory_quality_control': [('failed_quality_control', 0.1),
('successed_quality_control', 0.9)],
               'mandatory_measurement': [('EOL', 1.0)]},
    'warehouse_kitting': {'move': [('restocking', 1.0)],
'taking_components_to_assemble': [('assemble_station_t', 1.0)]},
    'restocking': {'wait': [('warehouse_kitting', 1.0)],
'moving_to_measure': [('EOL', 1.0)]},
    'assemble_station_t': {'moving_the_tool_box': [('EOL_t', 1.0)]},
    'EOL_t': {'move_to_grap_the_assembled_product':
[('assemble_station_p', 1.0)]},
    'assemble_station_p': {'move': [('mismatch', 0.5), ('idle', 0.5)],
'moving_the_assembled_product_to_measure': [('EOL', 1.0)]},
    'mismatch': {'moving_to_replace': [('warehouse_mismatch', 1.0)],
'moving_to_measure': [('EOL', 1.0)]},
    'warehouse_mismatch': {'taking_components_to_assemble':
[('assemble_station_p', 1.0)]},
    'idle': {'wait': [('assemble_station_p', 1.0)], 'moving_to_measure':
[('EOL', 1.0)]},
    'EOL': {'move_to_check_the_quality': [('failed_quality_control', 0.1),
('successed_quality_control', 0.9)], 'unexpected_failure':
[('system_failure', 1.0)]},
    'failed_quality_control': {'taking_the_product_to_the_shelf':
[('final_warehouse', 1.0)], 'move_to_measure_again': [('EOL_corrected',
1.0)]},
    'EOL_corrected': {'taking_the_product_to_the_shelf':
[('final_warehouse', 1.0)]},
    'final_warehouse': {'returning_to_the_initial_position':
[('charging_station', 1.0)]},
    'successed_quality_control': {'taking_the_product_to_the_shelf':
[('final_warehouse', 1.0)]},
    'system_failure': {'continue_working': [('EOL', 1.0)],
'move_for_maintenance': [('maintenance', 1.0)]},
    'maintenance': {'check_the_battery': [('under_threshold', 0.5),
('above_threshold', 0.5)]},
    'under_threshold': {'return_to_work': [('urgent', 1.0)],
'move_for_charging': [('charging_station', 1.0)]},
    'above_threshold': {'return_to_work': [('urgent', 1.0)],
'move_for_charging': [('charging_station', 1.0)]}
}

# Function to simulate state transition
def simulate_transition(current_state, action, transitions_stochastic):
    if action in transitions_stochastic[current_state]:
        possible_transitions =
transitions_stochastic[current_state][action]
        next_states, probabilities = zip(*possible_transitions)
        next_state = random.choices(next_states, weights=probabilities)[0]
        return next_state
    return current_state  # Return current state if no action is possible

num_samples_per_state = 3

def generate_samples(states, transitions_stochastic,
num_samples_per_state):
    samples = []

    for state in states:
        for _ in range(num_samples_per_state):
            if state not in transitions_stochastic:
                continue
            action =
random.choice(list(transitions_stochastic[state].keys()))
            next_state = simulate_transition(state, action,
transitions_stochastic)
            samples.append((state, action, next_state))

    return samples

# Generate a set of samples from the MDP
sampled_transitions = generate_samples(states, transitions_stochastic,
num_samples_per_state)

# Print some samples to verify
print("Sampled Transitions:")
for sample in sampled_transitions[:]:
    print(sample)

# Hyperparameters
alpha_1, gamma_1, initial_epsilon_1, decay_rate_1, min_epsilon_1 = 0.1,
0.9, 1.0, 0.99, 0.01
alpha_2, gamma_2, initial_epsilon_2, decay_rate_2, min_epsilon_2 = 0.5,
0.9, 1.0, 0.99, 0.01

epochs = 500

def update_q_value(prev_state, action, reward, next_state, Q, alpha,
gamma):
    prev_state_idx = state_indices[prev_state]
    action_idx = action_indices[action]
    next_state_idx = state_indices[next_state]

    # Get the maximum Q-value for the next state
    max_future_q = np.max(Q[next_state_idx])

    # Update the Q-value for the previous state and action
    Q[prev_state_idx, action_idx] += alpha * (reward + gamma *
max_future_q - Q[prev_state_idx, action_idx])

def epsilon_greedy_policy(state, epsilon, Q):
    # With probability epsilon, select a random action
    if random.random() < epsilon:
        return random.choice(actions)
    # With probability 1 - epsilon, select the action with the maximum Q-
value
    state_idx = state_indices[state]
    return actions[np.argmax(Q[state_idx])]

def train_q_learning(Q, sampled_transitions, alpha, gamma, epsilon,
decay_rate, min_epsilon, epochs):
    cumulative_rewards = []

    for epoch in range(epochs):
        total_reward = 0
        np.random.shuffle(sampled_transitions)  # Shuffle samples each
epoch for better learning

        for (prev_state, action, next_state) in sampled_transitions:
            # Select an action using epsilon-greedy policy
            action = epsilon_greedy_policy(prev_state, epsilon, Q)
            reward = rewards[state_indices[prev_state],
action_indices[action]]
            total_reward += reward
            update_q_value(prev_state, action, reward, next_state, Q,
alpha, gamma)
        cumulative_rewards.append(total_reward)

        # Exponentially decay epsilon
        epsilon = max(min_epsilon, epsilon * decay_rate)

    return cumulative_rewards, epsilon

def derive_policy(Q, states, actions):
    policy = {}
    for idx, state in enumerate(states):
        best_action_idx = np.argmax(Q[idx])
        policy[state] = actions[best_action_idx]
    return policy

# Train with the first set of hyperparameters
Q_1 = np.zeros((len(states), len(actions)))
rewards_history_1, final_epsilon_1 = train_q_learning(Q_1,
sampled_transitions, alpha_1, gamma_1, initial_epsilon_1, decay_rate_1,
min_epsilon_1, epochs)
optimal_policy_1 = derive_policy(Q_1, states, actions)

# Train with the second set of hyperparameters
Q_2 = np.zeros((len(states), len(actions)))
rewards_history_2, final_epsilon_2 = train_q_learning(Q_2,
sampled_transitions, alpha_2, gamma_2, initial_epsilon_2, decay_rate_2,
min_epsilon_2, epochs)
optimal_policy_2 = derive_policy(Q_2, states, actions)

# Plot the cumulative rewards for comparison
plt.figure(figsize=(12, 8))
plt.plot(rewards_history_1, label=f'alpha={alpha_1}, gamma={gamma_1},
epsilon={initial_epsilon_1}, decay={decay_rate_1},
min_epsilon={min_epsilon_1}')
plt.plot(rewards_history_2, label=f'alpha={alpha_2}, gamma={gamma_2},
epsilon={initial_epsilon_2}, decay={decay_rate_2},
min_epsilon={min_epsilon_2}')
plt.xlabel('Episode')
plt.ylabel('Cumulative Reward')
plt.title('Episode vs Cumulative Reward for Different Hyperparameters')
plt.legend()
plt.grid(True)
plt.show()

# Print the optimal policy for both sets of hyperparameters
print("Optimal Policy for first set of hyperparameters:")
for state, action in optimal_policy_1.items():
    print(f"State: {state}, Best Action: {action}")

print("\nOptimal Policy for second set of hyperparameters:")
for state, action in optimal_policy_2.items():
    print(f"State: {state}, Best Action: {action}")
```