PATH PLANNING IN ROBOTICS USING HYBRID Q-LEARNING APPROACH

Research Phase-2 Report

Submitted by

Mr. Sachin John Thomas

REG NO : TKM22MEAI15

the APJ Abdul Kalam Technological University in partial fulfillment for the award of the degree of

MASTER OF TECHNOLOGY
IN
Artificial Intelligence

Under the guidance of
Dr. Imthias Ahamed T P

Centre for Artificial Intelligence

TKM College of Engineering, Kollam

JUNE 2024

Thangal Kunju Musaliar College of Engineering
Centre for Artificial Intelligence

C E R T I F I C A T E

This is to certify that, this report titled ***PATH PLANNING IN ROBOTICS USING HYBRID Q-LEARNING APPROACH*** is a bonafide record of the **Research Phase 1** presented by **Sachin John Thomas (TKM22MEAI15)**, under our guidance and supervision, in partial fulfillment of the requirements for the award of the degree, **M. Tech in Artificial Intelligence** in **APJ Abdul Kalam Technological University**.

Internal Supervisor	Project Coordinator	Head of the Department
Dr. Imthias Ahamed T P Professor Centre for AI	Dr. Sumod Sundar Associate Professor Centre for AI	Dr. Imthias Ahamed T P Professor Centre for AI

ACKNOWLEDGEMENT

A successful project is a fruitful culmination of efforts by many people, some directly involved and some others indirectly, by providing support and encouragement. Firstly I would like to thank the almighty for giving me the wisdom and grace for making my project a memorable one. I thank him for steering me to the shore of fulfillment under his protective wings.

I express my sincere gratitude to **Dr. T Shahul Hameed**, Principal of T.K.M College of Engineering . I would like to thank my guide **Dr. Imthias Ahamed T P**, Professor and Head of the Department, Centre for Artificial Intelligence, TKM College of Engineering, Kollam, for his constant support and encouragement throughout the work.

With a profound sense of gratitude, I would like to thank my Project Coordinator, **Dr. Sumod Sundar**, Associate Professor,**Prof. Chinnu Jacob**, Assistant Professor and **Prof. Christy D Ponnan**, Assistant Professor, Centre for Artificial Intelligence(AI), TKM College of Engineering, Kollam for their expert guidance, cooperation, and immense encouragement. I also extend my thanks to the entire faculty and staff members of the Centre for AI, TKMCE, who have encouraged me throughout this work.

I also express my thanks to my loving parents and friends, for their support and encouragement in the successful completion of this work.

Sachin John Thomas

Abstract

Reinforcement learning is a technique that enables agents to learn optimal behaviors through interactions with their environment, using rewards and penalties to shape their actions. In this project, we address the challenge of enabling a mobile robot to navigate through environments such as a factory layout or a hospital setting while avoiding collisions with obstacles. The main objective of the agent is to navigate through these environments without colliding with any of the static or dynamic obstacles in its way. The robot, or the agent, is equipped with three proximity sensors that determine the proximity of the obstacles in their respective directions. The learning is achieved through the combination of a reinforcement learning approach known as the Q-learning algorithm, which is a value-based reinforcement learning technique, and the A* algorithm, a heuristic-based search algorithm. Q-learning is notable for its ability to handle problems with various state and action spaces, as well as its simplicity and versatility in various applications, including robotics, game playing, and autonomous systems. However, Q-learning faces the difficulty of large state spaces. To address this, we consider a heuristic approach to handle both large spaces and testing in unknown and dynamic environments. For testing and visualization, Python's Pygame library is involved. The agent undergoes training within a grid-based environment. The hybrid approach of combining Q-learning with the A* algorithm ensures faster learning and lesser computational time. This combination leverages the strengths of both methods, with Q-learning providing robust policy learning and A* offering efficient pathfinding through heuristic search. This ensures that the agent learns to efficiently navigate complex environments while minimizing computational overhead, ultimately enhancing its ability to operate autonomously and safely in real-world scenarios.

Contents

List of Figures

Chapter 1

Introduction

Autonomous navigation holds immense potential in various domains, particularly in environments where precision, efficiency and safety are paramount. The ability to navigate autonomously within any settings can revolutionize logistics, patient care and facility management offering opportunities for streamlined operations and enhanced outcomes.

Traditionally, navigation in such environments has relied on methods like search-based algorithms, which have proven effective to a certain extent but come with inherent constraints. Search-based approaches often struggle with complex environments, real-time decision-making and dynamic obstacle avoidance. Moreover, these methods may lack adaptability to diverse scenarios and may require extensive computational resources, limiting their practicality in real-world applications.

In light of these challenges, the adoption of reinforcement learning techniques, specifically Q-learning, emerges as a promising alternative for autonomous navigation. Reinforcement Learning is a subset of machine learning that focus on enabling agents to learn optimal behaviours through interaction with environment. Q-learning, a fundamental algorithm in reinforcement learning, offers a principled approach to learn optimal policies by iterative updating action values based on received rewards. However Q-learning have faced other challenges such as large state spaces and testing in an unknown environment.

In this project, we propose an innovative approach that combines the strengths of Q-learning with the efficiency of the A* algorithm for autonomous navigation in complex environments. By leveraging the adaptability and learning capabilities of Q-learning alongside the path finding ability of the A* algorithm, we aim to overcome the limitations of traditional navigation methods. Our goal is to empower mobile robots to navigate safely and efficiently through dynamic environment settings, adeptly avoiding obstacles and optimizing path planning in real-time.Through comprehensive analysis and experimental validation, we demonstrate the potential of our hybrid approach to deliver robust and adaptive solutions for autonomous navigation in healthcare facilities. By harnessing the complementary strengths of Q-learning and the A* algorithm, we strive to achieve total efficiency across various state spaces, marking a significant advancement in autonomous navigation technology.

Chapter 2

Literature Survey

Pei et al. [1] propose an enhanced Dyna-Q algorithm for mobile robot path planning in environments with both static and dynamic obstacles. Their method incorporates heuristic search strategies, simulated annealing, and reactive navigation principles into Q-learning based on the Dyna architecture. An innovative action-selection strategy combining the epsilon-greedy policy with cooling schedule control is introduced to address the exploration-exploitation dilemma, improving global search, convergence, and learning efficiency. The proposed method demonstrates superior performance compared to classical Q-learning and Dyna-Q algorithms in unknown static environments and effectively manages dynamic obstacles in simulations. Practical experiments conducted using MATLAB and ROS confirm the method's success in real-world autonomous navigation tasks.

Bianchi et al. [3] introduce Heuristically Accelerated Q-Learning (HAQL), which enhances Q-learning by incorporating heuristics to influence action selection. The heuristic function H guides action selection, automatically extracted during the learning process. Experimental results show that even simple heuristics can significantly boost the performance of the reinforcement learning algorithm, highlighting the effectiveness of the HAQL approach.

Shi et al. [7] introduce an improved Q-learning algorithm that integrates prior knowledge to enhance the path planning of mobile robots with partial environmental knowledge. By initializing the Q-value with prior knowledge, the algorithm guides the agent towards the target direction early on, reducing invalid iterations. Additionally, the algorithm dynamically adjusts the greedy factor epsilon based on successful target reaches to balance exploration and exploitation, leading to faster convergence and higher learning efficiency compared to traditional methods. Simulation results demonstrate the practical significance of the improved algorithm in enhancing the autonomous navigation efficiency of mobile robots.

Surmann et al. [6] present a method that involves the actor-critic algorithm in unknown static environments and successfully addresses uncertain environments with multiple dynamic obstacles in simulations. Sensor data fusion with RGB-D camera technology enables the robot to navigate in real environments with 3D obstacle avoidance without requiring environmental modifications to align with the robot's sensory capabilities. Although proved to be efficient in results, immoderate training time is proved to be imminent. The paper does not utilize prior information. However, in the case of Lazhar et al. [2], the paper

utilizes fuzzy logic for individual behavior design to reduce the complexity of the navigation problem by dividing them into smaller actions which is easier for design and implementation.

Pei et al. [1] also utilize prior information such as the known goal state through heuristic search strategies. They introduce an improved Dyna-Q algorithm augmented with simulated annealing and a heuristic function.By incorporating a novel action-selection strategy that combines epsilon-greedy policy with cooling schedule, the algorithm effectively balances exploration-exploitation trade-offs.

In our project, we utilize the path generated by A* algorithm as the prior information, which makes the learning much faster without the need to explore the entire environment.

Chapter 3

Methodology

3.1 Objective(s)

- To build an environment where the agent can interact with obstacle efficiently
- To enable the agent to traverse through the environment utilizing prior information

3.2 Proposed Framework

The flowchart depicted in Figure 3.1 adheres to a standard framework characteristic of many reinforcement learning paradigms. Within this framework, the agent's decision-making process is crucially influenced by its spatial relationship to obstacles within the environment. Specifically, the agent faces the choice between employing either the A* algorithm or the Epsilon Greedy Action approach to navigate its surroundings and interact with the environment. Subsequently, the environment provides feedback to the agent in the form of state information and corresponding rewards. This iterative process persists until the culmination of an episode.

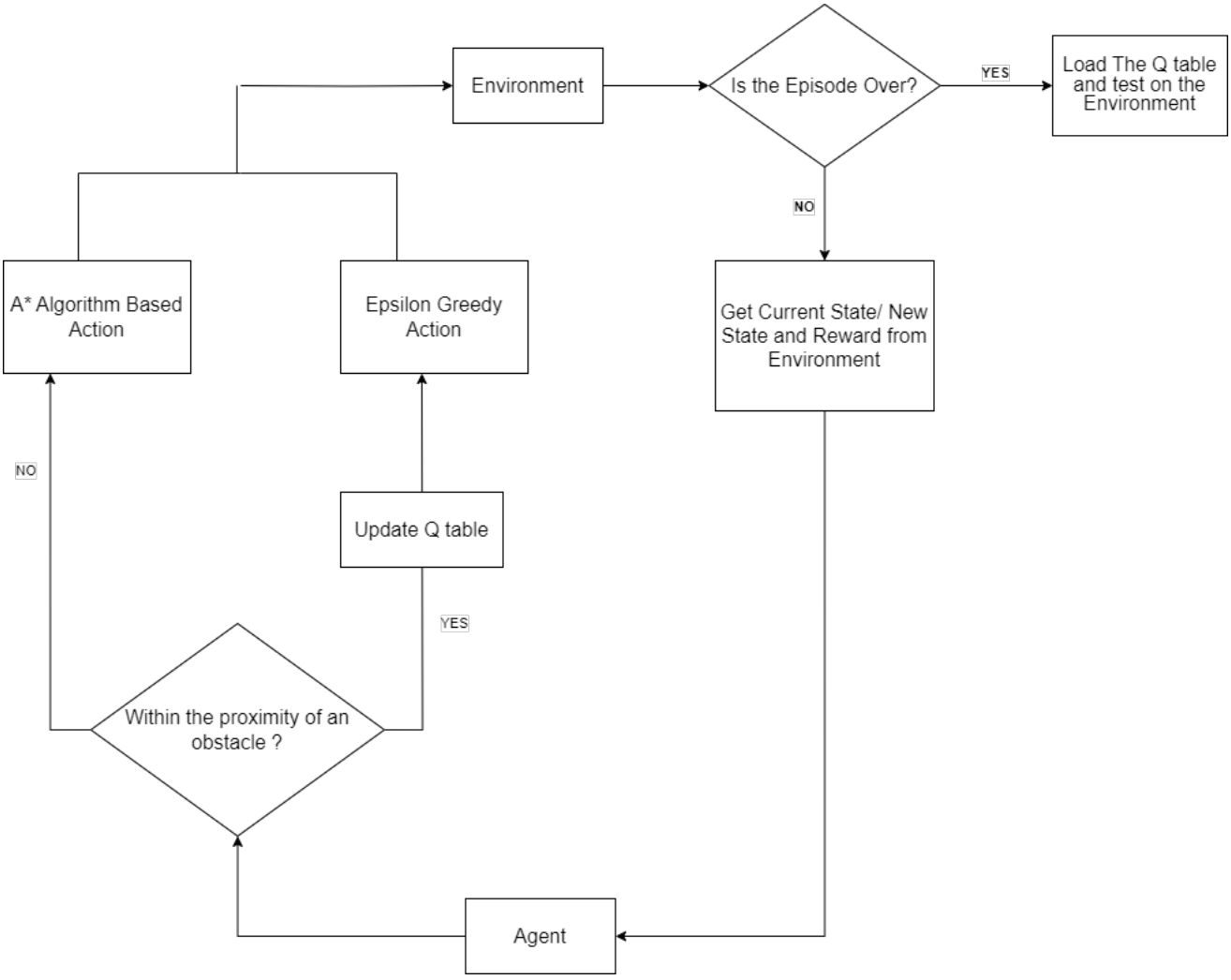


Figure 3.1: Proposed Framework

3.3 A* Algorithm

The A* algorithm stands as a cornerstone in computer science, revered for its ability to determine the shortest path between two points within a graph or grid structure. This method finds wide application across diverse fields such as navigation systems, robotics, and video game development, where efficient pathfinding is indispensable. Central to the effectiveness of A* is its utilization of two fundamental components: the 'g-value,' representing the cost incurred to reach a particular point from the starting position, and the 'h-value,' indicating an estimate of the cost to traverse from that point to the destination. By combining both the 'h-value' and 'g-value', A* strategically prioritizes nodes for exploration, channeling its efforts toward identifying the most promising route.

By integrating both historical path cost and heuristic estimations of future costs, A* effectively balances between the exploration of potential pathways and the exploitation of existing knowledge, facilitating the discovery of optimal routes. This approach ensures that

A* exhibits both accuracy and efficiency in determining paths, making it a preferred choice for scenarios demanding robust pathfinding solutions. Fig. 3.2 shows the flowchart for the A* algorithm. First, initialize the starting node, 'n', and place it on the open list. Then, calculate the heuristic function, h(n), which represents the Manhattan distance between 'n' and the goal state. The cost function, f(n), is defined as the sum of g(n) and h(n). Remove 'n' from the open list and add its index with the smallest 'f' value. Is 'n' in the goal state? If yes, terminate the algorithm; if not, detect all neighbors of 'n' not in the closed list and calculate 'f' for each neighbor. Repeat this process until the goal state is reached.

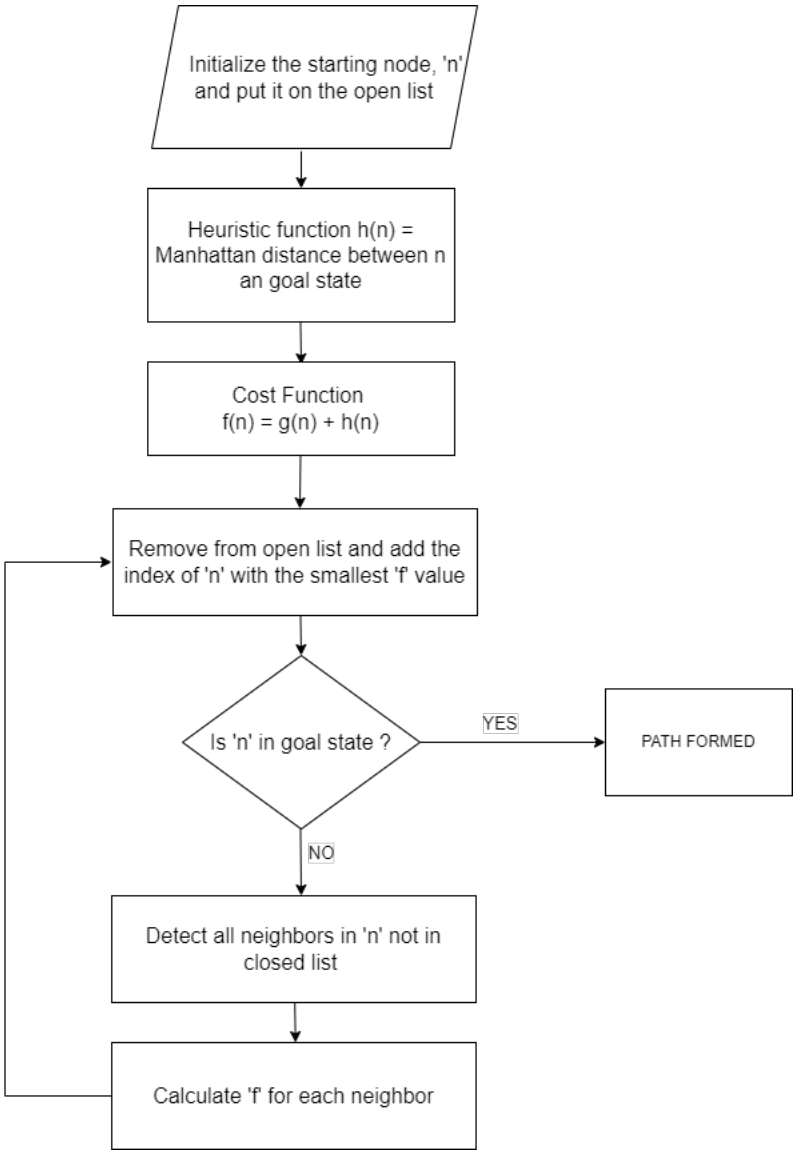


Figure 3.2: A* Algorithm Flowchart

The A* algorithm is a widely used pathfinding and graph traversal algorithm, known for its efficiency and accuracy in finding the shortest path from a starting node to a goal node. The process begins by initializing the starting node, denoted as 'n', and placing it on the

open list. This open list is a priority queue that helps in selecting the most promising node to explore next. The algorithm then uses a heuristic function $h(n)$, which in this case is the Manhattan distance between node 'n' and the goal state. The Manhattan distance is a common heuristic in grid-based pathfinding that calculates the sum of the absolute differences of their Cartesian coordinates.

The cost function in A* is defined as $f(n) = g(n) + h(n)$, where $g(n)$ represents the actual cost from the start node to the current node 'n', and $h(n)$ is the heuristic estimate from node 'n' to the goal. The algorithm proceeds by removing the node with the smallest f value from the open list and examining its index. This node is considered the most promising candidate at this stage.Next, the algorithm checks if this node 'n' is in the goal state. If 'n' is indeed the goal, the algorithm successfully forms a path from the start to the goal node, and the search terminates. However, if 'n' is not the goal state, the algorithm detects all the neighbors of 'n' that are not in the closed list. The closed list keeps track of nodes that have already been evaluated to prevent redundant processing.

For each neighbor, the algorithm calculates the cost function f. These calculations determine the most efficient path by considering both the known cost to reach the neighbor and the estimated cost from the neighbor to the goal. Each neighbor's f value is assessed, and they are added to the open list if they have not been evaluated before or if a cheaper path to them is found. This process repeats iteratively: selecting nodes from the open list based on their f values, expanding them, and checking for the goal state. The algorithm continues to refine the path by exploring nodes in a manner that balances the actual distance traveled and the estimated distance remaining. Through this method, A* ensures that the path formed is both optimal and efficient, successfully guiding from the starting node to the goal node using the defined cost and heuristic functions.

3.4 Epsilon-Greedy Algorithm

The Epsilon-Greedy approach is a fundamental technique in reinforcement learning that helps an agent strike a balance between exploring new actions (exploration) and exploiting known actions that have yielded high rewards (exploitation). This balance is crucial for effective learning, as it allows the agent to gather information about the environment while also capitalizing on the best-known strategies.

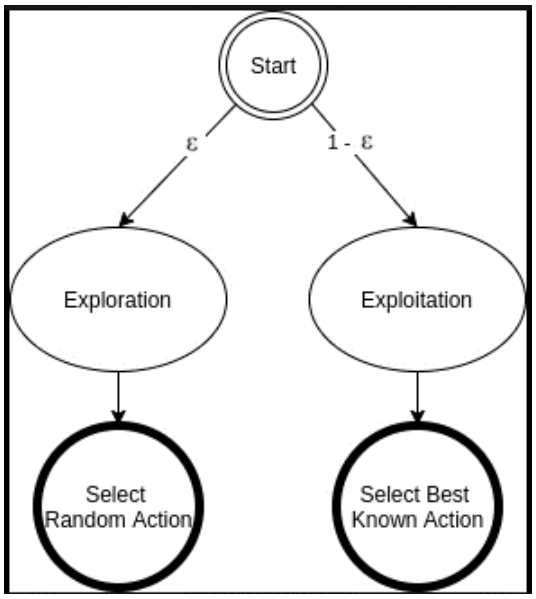


Figure 3.3: Epsilon Greedy Algorithm

As illustrated in Fig. 3.3, the Epsilon-Greedy algorithm operates within the context of exploration and exploitation. The exploration aspect involves randomness, governed by the value of Epsilon (ϵ). This value determines the probability of selecting a random action, allowing the agent to explore new and potentially better actions that have not yet been thoroughly evaluated.

On the other hand, the exploitation aspect is engaged with a probability of $1 - \epsilon$. In this mode, the agent follows a greedy strategy, selecting the action that currently has the highest expected value based on the Q-function, which represents the Q-values from the Q-table. These Q-values measure the expected utility of actions in given states, guiding the agent towards choices that historically have led to better outcomes.

The following algorithm outlines the Epsilon-Greedy action selection process:

Algorithm 1 Epsilon-Greedy Action Selection

Require: ϵ, Q-values for state s
Ensure: action a
initialize $random\_value$ between 0 and 1
if $\epsilon > random\_value$ **then**
    $a \leftarrow$ random action
else
    $a \leftarrow \text{argmax}_a(Q(s, a'))$
end if

Initially, the algorithm generates a random value between 0 and 1. If this value is less than ϵ, the agent selects a random action, embracing exploration. If the value is greater than ϵ, the agent exploits its knowledge by choosing the action with the highest Q-value for the current state, thereby following the greedy strategy.

To enhance the effectiveness of the epsilon-greedy algorithm over time, an epsilon decay

mechanism is often employed. This mechanism gradually reduces the exploration probability ϵ as learning progresses. The decay helps the agent shift from a high exploration phase in the early stages of learning to a more exploitation-focused approach as it gains more information about the environment. The decay formula is given by:

$$\text{decay} = \left(\frac{e_f}{e_i}\right)^{\frac{1}{N}} \tag{3.1}$$

where ϵ_i is the initial epsilon value, ϵ_f is the final epsilon value, and N is the total number of episodes over which the decay takes place. This formula ensures a smooth and gradual transition from exploration to exploitation, allowing the agent to make increasingly informed decisions as it learns. At each episode, the value of ϵ is updated using the decay rate to progressively reduce the likelihood of exploring random actions. This equation will ensure a consistent exploration vs exploitation graph with respect to the number of episodes as shown in Fig. 3.4.

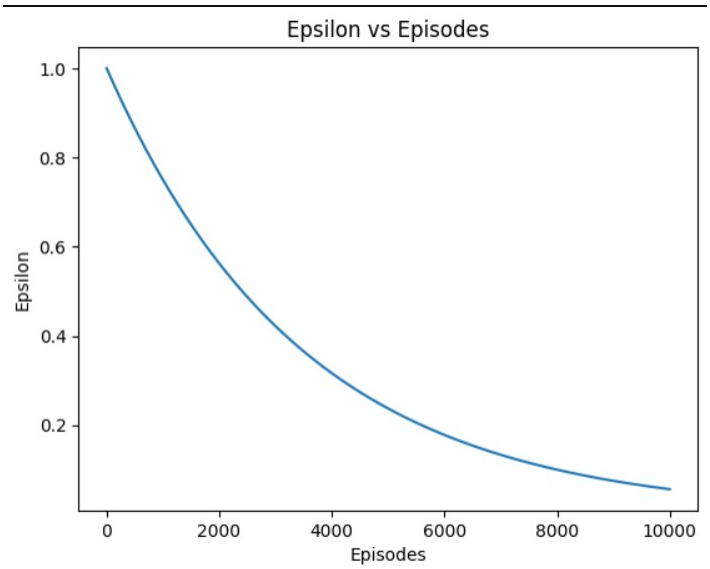


Figure 3.4: Epsilon rate over episodes

3.5 Q-learning

Q-learning is an off-policy reinforcement learning algorithm which includes iterative learning for an agent by guiding it to take optimal actions in a given environment.The algorithm's primary goal is to learn a policy, updated in Q-values, guiding agents to take actions that maximize cumulative rewards in a given environment. With the help of epsilon-greedy strategy, Q-learning balances exploration and exploitation, allowing agents to experiment with new actions exploiting those with the highest known Q-values. Q-values are updated through the Bellman equation, articulating the link between a state-action pair's value and its potential successor states. This iterative updating mechanism ensures agents improves their decision-making strategies over time, making Q-learning a powerful tool for optimizing ac-

tions and maximizing rewards in dynamic environments.

The components of Q-learning include:

- Agent : An entity that acts and operates within an environment.
- Actions Space: The possible action is the agent's operation when it is in a specific state.
- State Space Representation : Possible states an agent can have in its environment
- Transition Function : Transition function accepts current state and action by the agent and generates the resulting "new state"
- Reward Function : The reward function takes in the current state , action and the new state and generates the reward based upon these variables

In this project we consider two cases with two variations.

- Part 1 : Manually Setting the Regions
- Part 2 : Heuristic Based Action

3.5.1 Agent

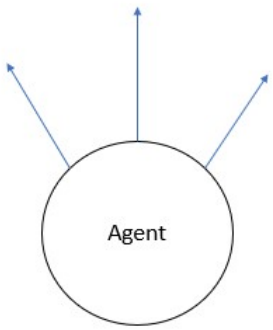


Figure 3.5: Agent with proximity sensors facing in the forward direction

The agent has three proximity sensors in this case. The proximity sensors helps the agent to understand the distance between itself and the environment.

- $Ps[i] = 0$ indicates the agent is within a safe zone, free from immediate obstacles.
- $Ps[i] = 1$ signifies the detection of an obstacle at a moderate distance from the agent's position.
- $Ps[i] = 2$ denotes the presence of an obstacle in closer proximity to the agent.
- $Ps[i] = 3$ indicates that the obstacle is dangerously close to the agent's location.

Here, $Ps[i]$ represents one of the three proximity sensors.

3.6 Q-learning - Part 1: Manually Setting the Regions

3.6.1 Problem Statement

Navigating robots through complex environments, such as hospital corridors, presents a significant challenge due to the presence of dynamic and stationary obstacles. In this study, we focus on a corridor scenario with realistic dimensions: 30m in length and 4m in width for one corridor, and 20m in length for another. Considering a robot with a diameter of 25cm and individuals occupying circles with a diameter of 50cm, the task at hand is to guide the robot effectively through these corridors while avoiding collisions with obstacles.

The context of hospital navigation underscores the critical importance of efficient traversal without compromising safety or disrupting the flow of human traffic. This requires the development of robust navigation strategies capable of dynamically adapting to the environment's changing conditions. Our primary objective is to devise a state-dependent action strategy that empowers the robot to make informed decisions based on its current region within the corridor. Our primary object of the robotic agent is to enable the robot to traverse through the corridor effectively, avoiding collisions with any encountered obstacles. To address this challenge, we propose the implementation of state-dependent action strategy. This approach involves the robot making decisions based on its current region within the corridor.

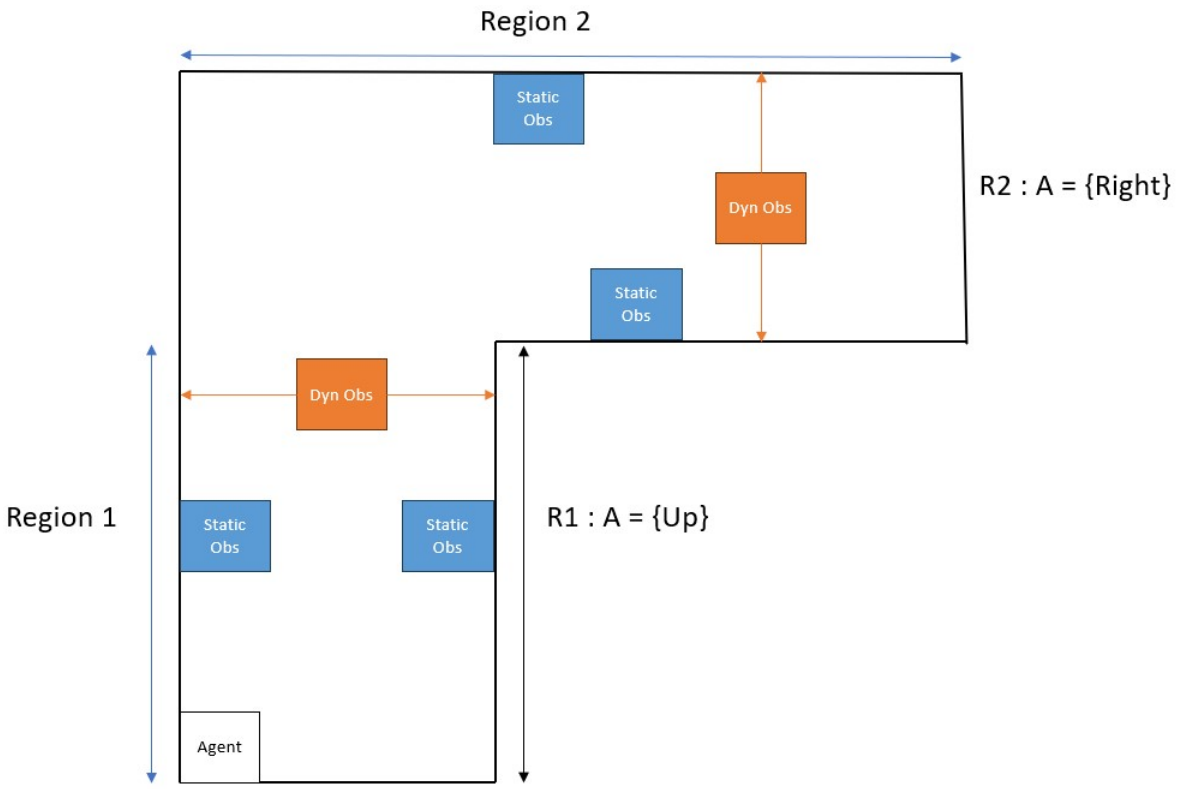


Figure 3.6: Grid World with agent(blue) , obstacle(Black) and goal state(Green)

As shown in fig. 3.6, if the robot finds itself within Region 1, the appropriate action to take is to move upwards, unless the agent detects the presence of an obstacle within its proximity.

3.6.2 Agent

The agent consists of three proximity sensors to detect the proximity of the sensor ahead of its direction. The range is discretized from 0-3. The higher the value of the proximity sensor, the closer it is to an obstacle. Fig. 3.13 shows the placement of the three proximity sensors(denoted by blue). A single sensor facing horizontal direction and the other two sensors facing opposite diagonal directions.

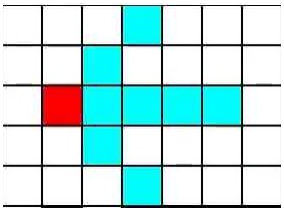


Figure 3.7: Agent(red) with proximity sensor(blue) placement in the environment used

3.6.3 Action Space

Action Space is defined as the set of all possible actions that can be taken by the agent in each state. Actions are the choices available to influence the system's behavior.

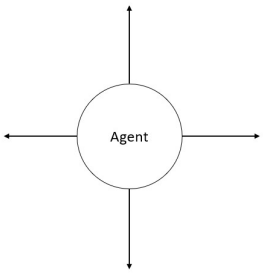


Figure 3.8: Agent with four possible orientation

The agent has 6 movements: up, down, left, right, turn left and turn right to navigate the grid world environment. The agent's transition depends on its orientation, which can be in one of these four directions. Since the main focus is on state dependant actions, it's important to note that the agent's actions are restricted based on its current location within the environment. This means that the actions are restricted with respect to the regions the agent is in. In R1 the state dependant action of the agent is moving towards "UP" direction . In R2 the state dependant action of the agent is moving towards "RIGHT" direction . State-dependant action selection enables the agent to adapt its movement strategy to the prevailing environmental conditions . When the agent resides in region R0 , its action selection undergo modifications with respect to the algorithm to accommodate the same.

3.6.4 State Space Representation

The state contains the vital information crucial for agent's navigation and decision-making process. The grid world environment is characterized by a 20x20. grid, where each cell represents a unique state. Assume the size of the robot to be one singular state. The same can be said for the dynamic obstacle. The grid is partitioned into regions to facilitate regions specific actions. Within this structure, the presence of obstacles and the goal state impose constraints on agent movement and path planning strategies. The grid world is divided into three primary regions, each defined by specific dimensions and obstacle configurations.

- **R1** : $\{(r, c) \mid r_a < r < r_b, c_a < c < c_b : (r, c) \notin \textit{obstacle position}\}$

- **R2** : $\{(r, c) \mid r_c < r < r_d, c_c < c < c_d : (r, c) \notin \textit{obstacle position}\}$
- **R0** : $\{X \mid X$ is within the vicinity of the obstacle$\}$

Regions 1 and 2 encompasses states with their specified coordinates. But these regions excludes obstacle positions or within the specific area of the same. Region 0 represents states within the vicinity of the dynamic obstacles. This regions dynamically changes as the obstacle moves within the grid.

Obstacles within the grid world environment are dynamic and exhibit constant movement at a predefined pace.In addition to obstacles, the grid contains a goal state that agents aim to reach. The location of the goal state remains static throughout the simulation.

The State of the agent consists of Grid Coordinates, Orientation of the agent, Current Proximity Sensors values and Previous Proximity Sensors values. The grid coordinates and orientation represented by the x and y coordinates and 'o', indicates the current position of the agent within the grid world environments. The current proximity sensor values denotes by Ps_{current}, provide real-time feedback on the proximity of obstacle in the agent's immediate surroundings. It consist of 3 separate values with respect to the current position of the agent. The previous proximity sensor readings , designated as Ps_{previous}, consist of earlier proximity measurements obtained from the immediate preceding state. These readings serve as historical data, offering insights into the temporal evolution of obstacle proximity and facilitating predictive analysis for future movements

Hence the state space is represented as follows :

State : (x coordinate, y coordinate, orientation, $Ps_{\text{current}}(1)$, $Ps_{\text{current}}(2)$, $Ps_{\text{current}}(3)$, $Ps_{\text{previous}}(1)$,$Ps_{\text{previous}}(2)$, $Ps_{\text{previous}}(3)$)

3.6.5 Transition Function

The transition function in a dynamic environment specifies the probability or likelihood of transitioning from one state to another based on a selected action. However, in certain scenarios, such as navigating around obstacles, the actions available to an agent may be contingent upon the state in which it finds itself. For instance, let's consider two distinct regions: Region 0 (R0) and Region N (RN). Within the vicinity of an obstacle or hazardous area, denoted as R0, the agent's actions are determined by an epsilon-greedy algorithm. This approach allows the agent to explore and exploit its surroundings effectively, balancing between taking random actions to discover new paths and exploiting known strategies to achieve its goals. When the agent is situated within a safe distance in Region N, denoted as RN, the actions are predefined based on the specific characteristics or requirements of that region.

The transition between these action regimes is often governed by certain conditions. For example, in the context of R0, the transition to this state occurs when the proximity sensor of the agent detects an obstacle within a defined threshold, such as a distance of 2 units or more. This transition signifies the need for the agent to adopt a more exploratory approach to navigate around obstacles effectively.

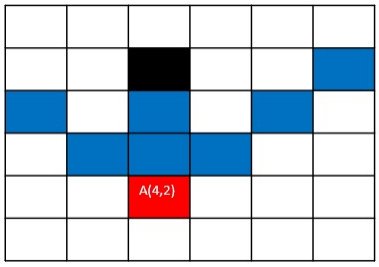


Figure 3.9: An example gridworld of size 6x6

Take this example of a grid world scenario(fig 3.9, the state is represented as (4, 2, 1, [1, 1, 0], [1, 0, 0]). The state includes information such as the agent's position, orientation, and proximity sensor readings. In this scenario, the proximity sensor reading is less than 2, which means it is within a safe distance from the obstacle. This will result in agent selecting state dependant action. Assume the agent's state dependant action for that particular region to move upward.

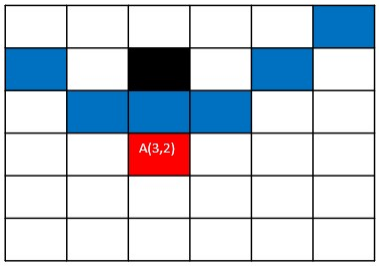


Figure 3.10: The agent transitioning to the next state

In fig 3.10 the next state as (3, 2, 1, [2, 1, 0], [1, 1, 0]), where one of the main proximity sensors reads a value of 2, the action for the next state will now change based on the epsilon greedy algorithm.

3.6.6 Reward Function

The reward function $r(s, a, s')$ is defined as follows:

$$r(s, a, s') = \begin{cases} 10 & \text{if } s' \text{ is a goal state} \\ -10 & \text{if } s' \text{ is an obstacle} \\ -1 & \text{otherwise (default reward)} \end{cases}$$

The reward function $r(s, a, s')$ provides feedback based on the action a taken in state s leading to the resulting state s'. The reward values are determined by specific conditions:

- **Goal State**: If the resulting state s' is a goal state, the reward is 10. This positive reward encourages the agent to reach the goal state, reinforcing that this is a desirable outcome.

- **Obstacle**: If the resulting state s' is an obstacle, the reward is -10. This negative reward penalizes the agent for moving into an obstacle, discouraging such actions and guiding the agent away from harmful states.
- **Default Reward**: In all other cases, the reward is -1. This default reward provides a slight penalty for each action, which can encourage the agent to reach the goal state efficiently rather than wandering aimlessly.

These reward values guide the agent's learning process by providing feedback that helps the agent distinguish between good and bad actions and states, ultimately shaping its policy towards achieving goals while avoiding obstacles.

3.6.7 Q-Learning Algorithm

Algorithm 2 Q-Learning Algorithm

Require: Number of episodes N, learning rate α, exploration probability ϵ, decay factor *decay*

Initialize Q-values table Q with zeros
Initialize epsilon-greedy algorithm with ϵ
Initialize current state s
for $episode = 1$ to N **do**
  Initialize current state s
  repeat
    if $s(ps) \geq 2$ **then**
      Get action based on epsilon-greedy algorithm
    else
      if s in Region 1 **then**
        $action = UP$
      else if s in Region 2 **then**
        $action = RIGHT$
      end if
    end if
    Take action a, observe reward $r(s, a)$ and next state s'
    if s' in terminal state **then**
      $Q(s, a) \leftarrow Q(s, a) + \alpha[r(s, a) - Q(s, a)]$
    else
      $Q(s, a) \leftarrow Q(s, a) + \alpha[r(s, a) + \max_{a'} Q(s', a') - Q(s, a)]$
    end if
    $s \leftarrow s'$
  until s is terminal
  $\epsilon \leftarrow \epsilon \times decay$
end for

Q-learning stands as a robust off-policy machine learning paradigm, furnishing agents with the capacity to perpetually refine their decision-making prowess. This iterative learning mechanism centers on discerning the optimal action predicated on the agent's prevailing state, irrespective of whether it adheres to a prescribed policy or devises its own. Unlike certain other reinforcement learning methods, Q-learning doesn't mandate a preordained

policy; instead, it autonomously explores and learns. This autonomy is facilitated through the utilization of Q-values, also referred to as action values, which encapsulate the anticipated future rewards for executing specific actions in defined states and are stored within a Q-table. The agent's odyssey toward mastering its environment entails exploration, reward acquisition, and continual model refinement.

The provided algorithm 3 outlines the Q-Learning technique, a pivotal approach in reinforcement learning for training agents to make decisions in environments with uncertain dynamics. Initially, the algorithm initializes the Q-values table, which stores expected rewards for state-action pairs, along with parameters such as the number of episodes, learning rate, exploration probability, and decay factor. In each episode, the agent explores the environment while balancing exploration and exploitation through an epsilon-greedy strategy, adapting its actions based on proximity sensor readings and region-specific constraints. The agent iteratively updates Q-values using observed rewards and transitions, adjusting its policy towards maximizing cumulative rewards. This iterative process gradually refines the agent's decision-making abilities, culminating in learned policies that optimize performance over time.

3.7 Q-learning - Part 2: Heuristic Based Actions

3.7.1 Problem Statement

Navigating robots through complex environments, presents a significant challenge due to the presence of dynamic and stationary obstacles. Autonomous navigation holds immense potential in various domains, particularly in environments where precision, efficiency and safety are paramount.

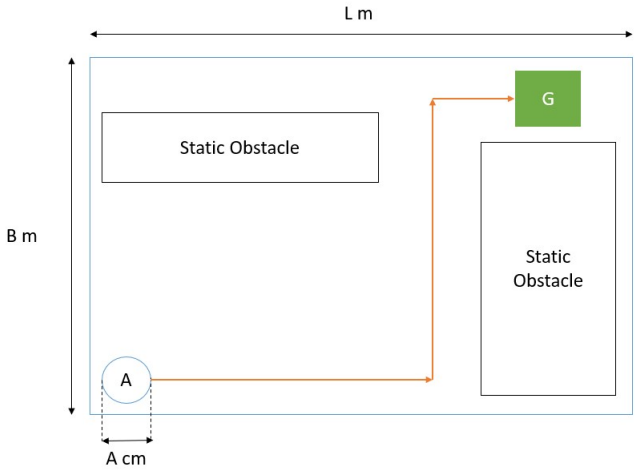


Figure 3.11: A Layout with agent(A) , obstacle and goal state(G)

Consider the layout depicted in fig 3.11, comprising a rectangular space measuring 20m x 20m. Within this space, there exists an agent, or in this context, a robot with a diameter of 50cm, alongside several stationary obstacles representing some sort of equipments. A designated goal point is indicated within the layout. The primary aim of the agent is to navigate

the environment safely, avoiding collisions with the obstacles. However, the challenges faced by the agent extend beyond these stationary obstacles. Through Simple search methods, it is plausible to compute the path from the starting state to the goal

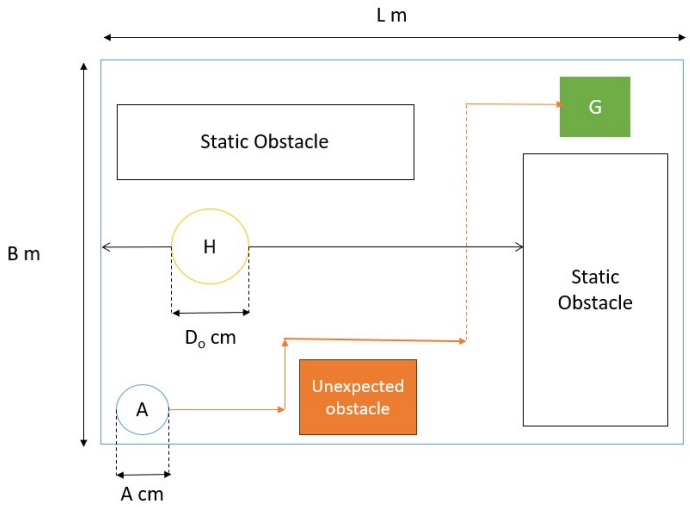


Figure 3.12: The Layout with agent(A), dynamic obstacle and static obstacles, goal state(G)

However computing dynamic obstacles or unexpected may rely on the need for complex algorithms. As illustrated in fig 3.12, the agent may encounter dynamic obstacles, such as moving individuals within the factory setting. Additionally, there is a possibility of encountering unforeseen obstacles deliberately placed by workers. These dynamic obstacles are denoted as 'H' in the figure. The dashed lines signify the need for the agent to determine the optimal path to navigate through these dynamic obstacles safely.

3.7.2 Agent

The agent, as mentioned in the section problem statement, consists of three proximity sensors to detect the proximity of the sensor ahead of its direction. The range is discretized from 0-3. The higher the value of the proximity sensor, the closer it is to an obstacle. Fig. 3.13 shows the placement of the three proximity sensors(denoted by blue) in three vertical directions in environment to be used for our experiment.

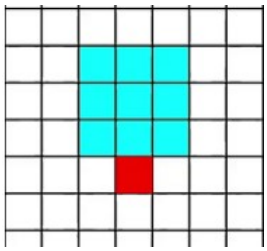
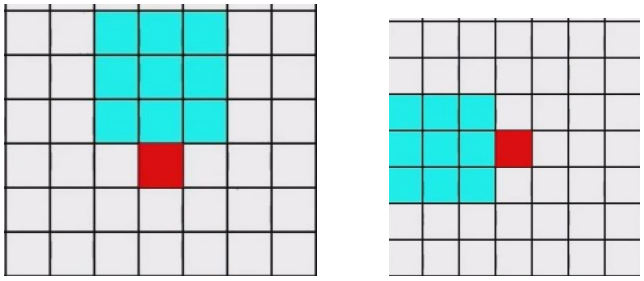


Figure 3.13: Agent(red) with proximity sensor(blue) placement in the environment used

3.7.3 Action Space



(a) State 1 (b) State 2

Figure 3.14: Agent (black) performing left action from State 1 to State 2. Note the sensors (blue) changing with respect to its orientation.

Action space defines the set of actions an agent can take in the environment.
The following are the actions the agent can take:

- Move Forward: Lets the agent traverse forward.
- Move Backward: Lets the agent go back.
- Turn Left: This helps the agent to turn left. The orientation changes as it moves (see Fig. 3.14).
- Turn Right: This helps the agent to turn right.
- Wait: This will let the agent stay in its current position, useful when there is a dynamic obstacle nearby.

The actions are defined with respect to the agent's orientation.

3.7.4 State Space Representation

The state contains the vital information crucial for agent's navigation and decision-making process. The grid world environment is characterized into different cells, where each cell represents a unique state. Assume the size of the robot to be one singular state. The same can be said for the dynamic obstacle. Within this structure, the presence of obstacles and the goal state impose constraints on agent movement and path planning strategies.

The State of the agent consists of Orientation of the agent, Current Proximity Sensors values, Previous Proximity Sensors values, goal direction.
The orientation represents the position of the agent in the environment.The current proximity sensor values denotes by Ps_{current}, provide real-time feedback on the proximity of obstacle in the agent's immediate surroundings. It consist of 3 separate values with respect to the current position of the agent. The previous proximity sensor readings , designated as Ps_{previous}, consist of earlier proximity measurements obtained from the immediate preceding state. These readings serve as historical data, offering insights into the temporal evolution of obstacle proximity and facilitating predictive analysis for future movements.

The goal direction denotes the direction which the goal is at with respect to the current position of the agent(ref Fig.3.15). The values are denoted as follows:

- Top Left = 0
- Top = 1
- Top right = 2
- Right = 3
- Bottom Right = 4
- Bottom = 5
- Bottom Left = 6
- Left = 7

Figure 3.15: The possible directions of the goal with respect of the agent

Hence the state space is represented as follows :
State : (orientation, $Ps_{\text{current}}(1)$, $Ps_{\text{current}}(2)$, $Ps_{\text{current}}(3)$, $Ps_{\text{previous}}(1)$,$Ps_{\text{previous}}(2)$, $Ps_{\text{previous}}(3)$, goal direction, distance completed)

3.7.5 Transition Function

The transition function in a dynamic environment specifies the probability or likelihood of transitioning from one state to another based on a selected action. However, in certain scenarios, such as navigating around obstacles, the actions available to an agent may be contingent upon the state in which it finds itself. Within the vicinity of an obstacle, the agent's actions are determined by an epsilon-greedy algorithm. This approach allows the agent to explore and exploit its surroundings effectively, balancing between taking random actions to discover new paths and exploiting known strategies to achieve its goals. When the agent is situated within a safe distance, the actions are predefined based on the specific characteristics or requirements of that region.

Here are three cases based upon which action is chosen:

- Within the nodes of A* algorithm
- Within the vicinity of any obstacle
- In a free space

Case 1 : Within the nodes of A* algorithm

When the agent's current coordinate lies within any node of the path generated by the A* algorithm and is not within the range of an obstacle, the action is determined based on the upcoming path and the agent's orientation.

For instance(Fig 3.16, consider an 8x5 grid world. If the agent is situated at coordinate [2,2] and the upcoming path leads to coordinate [2,3], the action is generated according to the agent's orientation.

- If the agent is facing downward, the action to be taken is to move forward.
- If the agent is facing right, the action will be to turn right.

This dynamic decision-making process ensures that the agent aligns its actions with the anticipated path while considering its current orientation, thereby optimizing navigation efficiency within the environment.

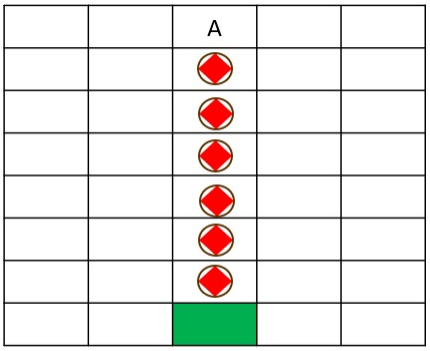


Figure 3.16: Path(red circles) generated by A* algorithm

Case 2 : Within the nodes of A* algorithm

When the current coordinate falls within the range of an obstacle, the agent resorts to selecting the optimal action from the Q-table.For instance, let's consider a scenario (Fig. 3.17) where there's an obstacle located at coordinate [2,4] within the grid. In such a situation, the agent will determine the appropriate action to take based on Q-learning. This approach ensures that the agent can navigate around obstacles efficiently by leveraging learned strategies encoded within the Q-table.

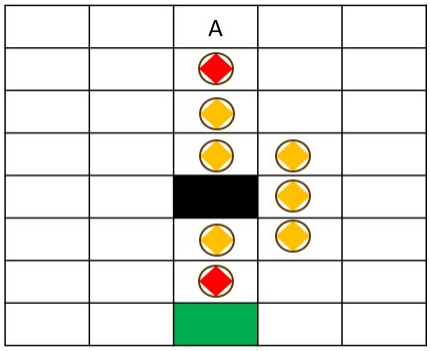
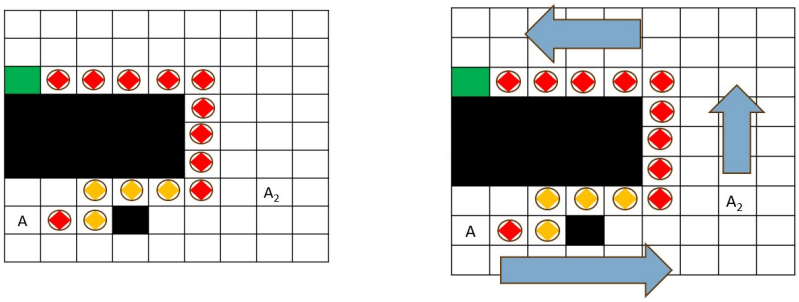


Figure 3.17: Path(yellow circles) generated by Qlearning

3.7.6 Case 3 : In a free space

In the given scenario(Fig. 3.18), where the agent is situated in free space (A2), neither within the nodes of the path nor in the vicinity of any obstacles, the action can be generated by initially identifying the closest coordinate from the path generated by A* algorithm to the current state of agent based on Euclidean distance. Subsequently, the action recommended for the closest coordinate is selected.This methodology ensures that the agent dynamically adapts its actions based on its immediate surroundings, effectively navigating the environment while prioritizing proximity to nearby coordinates.



(a) Combination of A* algorithm and Qlearning

(b) Generalized Regions generated by A* algorithm

Figure 3.18: Path generation with respect to its coordinates

3.7.7 Reward Function

The reward function provided governs the reinforcement learning process, dictating the consequences of the agent's actions within its environment. It operates through a series of conditions aimed at encouraging desirable behavior and penalizing unfavorable outcomes.

The reward function $r(s, a, s')$ is defined as follows:

$$r(s, a, s') = \begin{cases} 0 & \text{if } a \text{ is a preferred action} \\ 0 & \text{if } s' \text{ is a goal state} \\ -(\text{rowsize} \times \text{colsize}) & \text{if } s' \text{ is an obstacle} \\ -(\text{rowsize} \times \text{colsize}) + \text{occupied space} & \text{otherwise (default reward)} \end{cases}$$

The reward function $r(s, a, s')$ provides feedback based on the action a taken in state s leading to the resulting state s'. The reward values are determined by specific conditions:

- **Preferred Action**: If the action a is a preferred action (an action taken from case 2 or case 3), the reward is 0. This implies that taking a preferred action neither provides a positive nor negative incentive directly from the reward function itself, however it is still encouraged to take this action since its a lesser positive reward.
- **Goal State**: If the resulting state s' is a goal state, the reward is 0. This is because, in the case of Hybrid Q-learning, the agent cannot differentiate between an obstacle and a goal state. Additionally, the reward is not highly positive because the goal state might be near an obstacle, and the agent might mistake it for an obstacle.
- **Obstacle**: If the resulting state s' is an obstacle, the reward is $-(\text{rowsize} \times \text{colsize})$. This negative reward penalizes the agent for moving into an obstacle, discouraging such actions and guiding the agent away from harmful states. The use of $-(\text{rowsize} \times \text{colsize})$ ensures that hitting an obstacle yields a significantly higher negative reward, as the product of rowsize and colsize will provide the appropriate magnitude of penalty.
- **Default Reward**: In all other cases, the reward is $-(\text{rowsize} \times \text{colsize}) + \text{occupied space}$. This default reward provides a variable penalty based on the occupied spaces, which include spaces occupied by obstacles and the goal state. The addition of occupied space ensures a lesser negative reward than hitting an obstacle directly, encouraging the agent to take preferred actions and reach the goal state efficiently rather than wandering aimlessly.

These reward values guide the agent's learning process by providing feedback that helps the agent distinguish between good and bad actions and states, ultimately shaping its policy towards achieving goals while avoiding obstacles.Overall, this reward function serves as a critical mechanism for shaping the agent's learning process, guiding its behavior towards optimal decision-making within the environment.

3.8 Modified Qlearning Algorithm

The algorithm as shown in Algorithm 3 refers to the modified Q learning algorithm. It requires several parameters: The number of training episodes (N) the agent will experience. The learning rate (alpha) that controls how quickly the agent adapts its knowledge based on new experiences.The exploration probability (epsilon) that determines the balance between trying new actions (exploration) and exploiting what it already knows (exploitation). This value usually decays over time (decay factor).The algorithm starts with a Q-values table (Q)

initialized with zeros. This table will store the estimated value (Q-value) associated with taking a specific action (a) in a particular state (s). An epsilon-greedy strategy is used for action selection. With a probability of epsilon, the agent explores by choosing a random action. Otherwise, it exploits its current knowledge by taking the action with the highest Q-value in the current state.

The main loop iterates for a specified number of episodes (N). Within each episode, the agent starts from a given state (s). It enters a loop that continues until the agent reaches a terminal state (goal or failure). The algorithm checks a specific condition ((ps) ¿= 1). This condition is related to the proximity to the obstacle and determines if the agent should rely more on the A* path or Qlearning.Based on this condition and the current state, the agent selects an action (a). It might use the epsilon-greedy strategy, follow a pre-defined path from an A* algorithm, or explore towards the path based on its proximity.The agent takes the chosen action (a) and observes the resulting reward (r(s,a)) and the next state it lands in (s'). The Q-value for the previous state-action pair (s,a) is updated based on the Bellman equation. This update considers the immediate reward (r(s,a)), the estimated value of the best action in the new state (max(a') Q(s',a')), and the previously learned Q-value (Q(s,a)). The current state is updated to the next state (s = s'). This loop continues until the agent reaches a terminal state. After each episode, the exploration probability (epsilon) is decayed using the decay factor, making the agent more likely to exploit learned knowledge in future episodes.

By iteratively experiencing the environment, taking actions, and receiving rewards, the Q-learning algorithm gradually populates the Q-table with better estimates of the value of each action in each state. This allows the agent to learn the optimal policy for achieving its goals within the environment.

Algorithm 3 Q-Learning Algorithm

Require: Number of episodes N, learning rate α, exploration probability ϵ, decay factor *decay*

Initialize Q-values table Q with zeros
Initialize epsilon-greedy algorithm with ϵ
Initialize current state s
for $episode = 1$ to N **do**
  Initialize current state s
  repeat
    if $s(ps) \geq 1$ **then**
      Get action based on epsilon-greedy algorithm
      Take action a, observe reward $r(s, a)$ and next state s'
      if s' in terminal state **then**
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r(s, a) - Q(s, a)]$
      else
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r(s, a) + \max_{a'} Q(s', a') - Q(s, a)]$
      end if
    else
      if s in Path generated by A* algorithm **then**
        Get action based upon the coordinates of the upcoming states
      else if s in Free Space **then**
        Get action based upon the closest coordinate of path generated by A* algorithm to the current state
      end if
      Take action a, Let reward $r(s, a) = 0$ and get next state s'
    end if
    $s \leftarrow s'$
  until s is terminal
  $\epsilon \leftarrow \epsilon \times decay$
end for

Chapter 4

Experimental Analysis and Results

4.1 Experimental Setup

The experimentation process was conducted on a computational platform powered by an AMD Ryzen 7 4800H processor with Radeon Graphics clocked at 2.90 GHz, complemented by 16.0 GB of RAM. Additionally, the system is equipped with a Nvidia GeForce GTX 1660 Ti graphics card boasting 6 GB of dedicated memory. The development environment utilized for coding and experimentation was JupyterLab, providing a versatile and interactive interface for algorithm implementation and evaluation.

The visualization of the grid environment was facilitated by the Python library Pygame, renowned for its robust capabilities in rendering graphics and interactive applications. Leveraging Pygame's features, a grid world was constructed, comprising essential elements such as a goal state, a dynamic obstacle, and an agent. The dynamic obstacle, synchronized with the agent's movements, traversed horizontally along its axis, mirroring the agent's speed

4.2 Result Part A : Manual Setting Regions

4.2.1 Environment Setup

The experimental environment comprises a structured gridworld arranged in a 20x20 configuration, as depicted in Fig. 4.1. Implemented using the pygame framework, the setup features an agent equipped with three proximity sensors. Notably, there are two dynamic

Table 4.1: Parameter configurations of simulations

Parameter	Value
Learning Rate – α	0.1
Initial Epsilon – ϵ	1
Final Epsilon	0.2
Default reward (Reward per step)	-1
Goal state reward	20
Obstacle reward	-10
Episode Number – N	5000

obstacles in which one moves horizontally and the the other vertically. The parameters outlined in Table 4.1 will remain consistent for this approach.

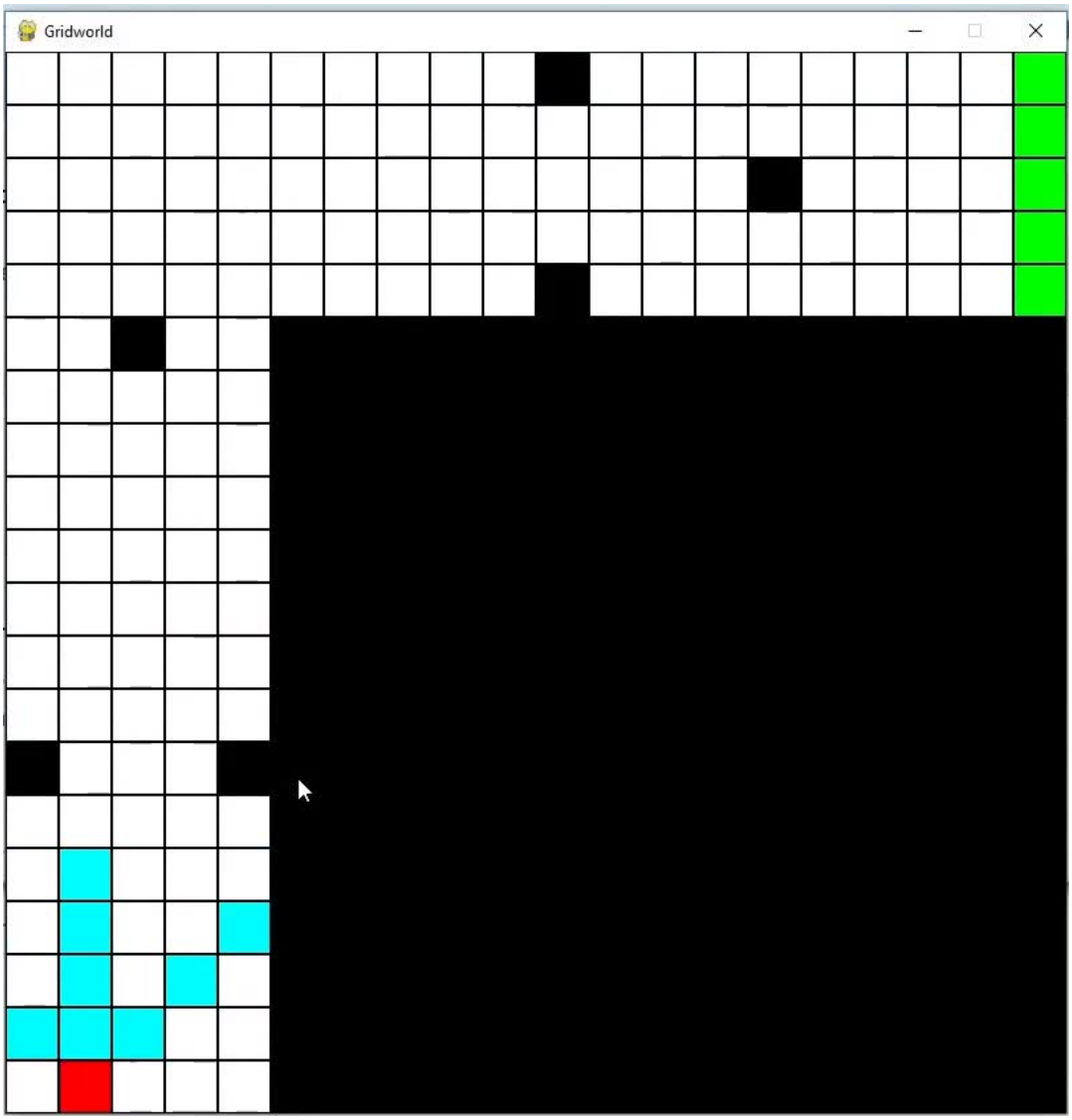
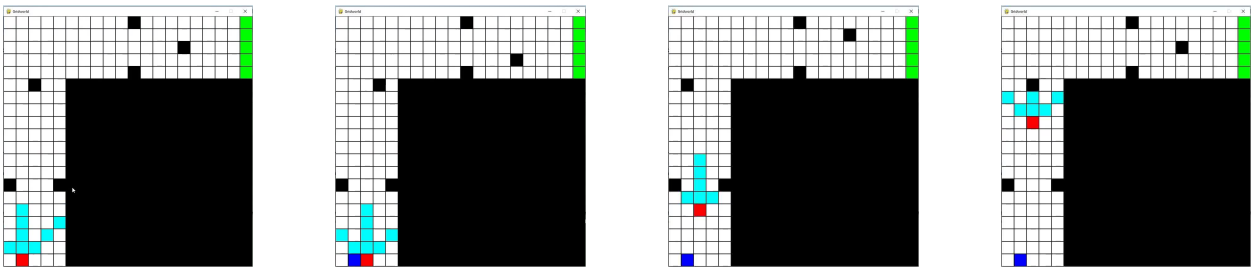


Figure 4.1: Environment Setup in pygame. Agent(red
with three proximity sensors(blue). The three obstacles(black) moves up and down .

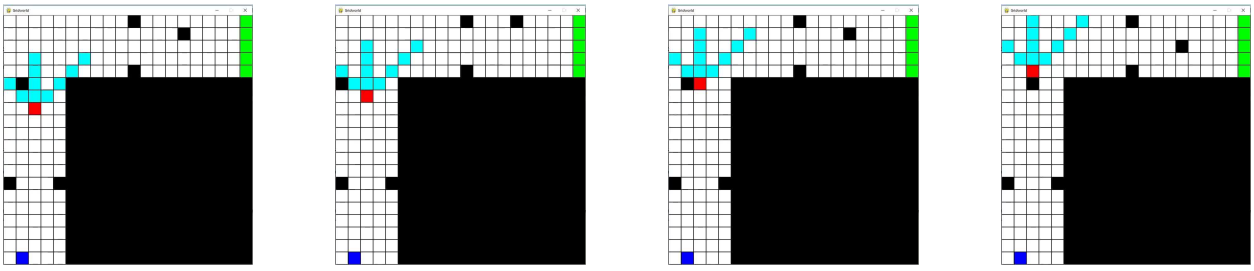
4.2.2 Result

The simulation experiment was conducted using Pygame. With the help of Qlearning the agent traverses through the dynamic environments. In this experiment, the agent is tasked with navigating through a hospital corridor, starting from one corner and reaching the opposite end. The corridor is divided into distinct regions, each region constitution the state dependant actions.

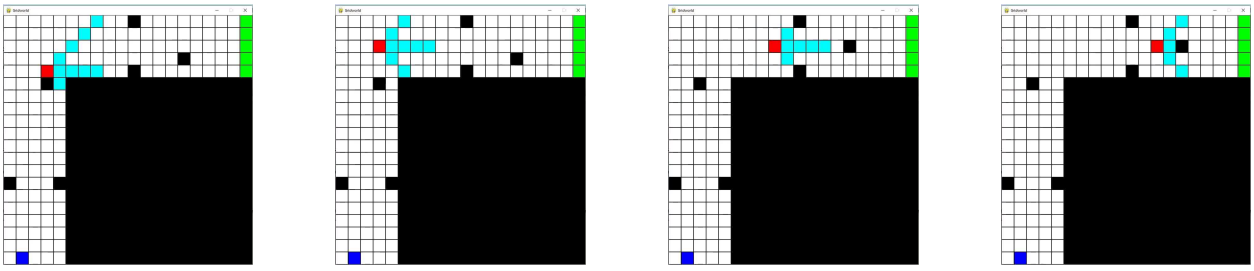
Within each region of the corridor, a dynamic obstacle is placed. Region 1 of the corridor requires the agent to navigate forward with orientation 1(face upward), while in region 2,



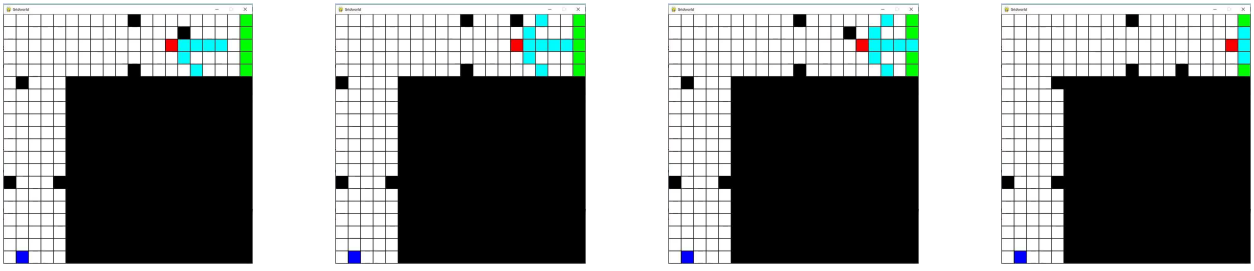
(a) Starting State (b) Step 1 (c) Step 5 (d) Step 12



(e) Step 13 (f) Step 14 (g) Step 15 (h) Step 16



(i) Step 17 (j) Step 19 (k) Step 24 (l) Step 28



(m) Step 29 (n) Step 30 (o) Step 31 (p) Step 34

Figure 4.2: Combined results

the agent must adjust its orientation to 0(face right) to progress effectively. When the agent detects proximity to an obstacle in region 0, it derives the best policy from the Q-table for the given state.

The visual representation provided in Figure 4.2 captures various stages of the agent's journey through the corridor. Sub figure (a) depicts the starting state of the agent as it embarks on its path finding task. From (b) to (d) it takes a straight path .As the agent encounters obstacles in the corridor, depicted in sub figures (d) , it moves straight forward as it understands the trajectory of the the moving obstacle because of the proximity sensors. Here the agent traverses forward with assurance that it wont engage with the obstacle in the next step.

As the agent enters the next region as shown in (h), it turns right and moves straight ((i) to (k)). This is due to the state dependant action is determined to move towards right. The agent again encounters the moving obstacles and moves in a straight path understanding the trajectory of the dynamic obstacle ((l) to (o)).

Through this experiment, we gain valuable insights into the efficacy of Q-learning algorithms in dynamic environments, demonstrating their potential to address complex pathfinding challenges commonly encountered in real-world scenarios.

4.3 Result Part B : Heuristic Based Action

4.3.1 Part 1 : Simple Static Grid World

The experimental environment comprises a structured gridworld arranged in a 8x8 configuration, as depicted in Fig. 4.3. The gridworld consist of static obstacles. The main goal of this experiment is to compare Simple Q learning and the Hybrid Q learning

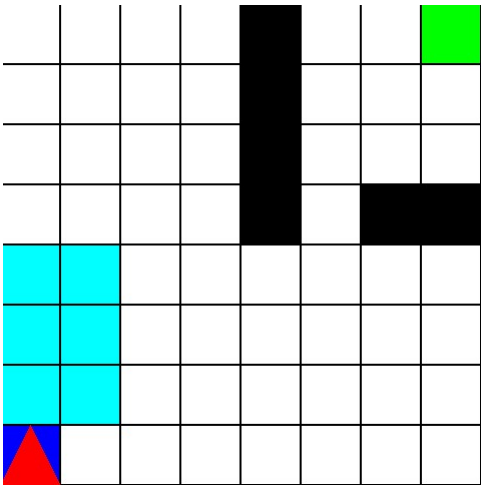


Figure 4.3: Environment Setup in pygame. Agent(red
with three proximity sensors(blue). The agent must traverse through the obstacles(Black) .

Table 4.2: Parameter configurations of Simple Q learning simulations

Parameter	Value
Learning Rate – α	0.1
Initial Epsilon – ϵ	1
Episode Number – N	10000
Default Reward – R	-1
Obstacle Reward – R	-10
Goal Reward – R	+20

4.3.2 Q-learning vs Modified Q-learning

The environment mentioned earlier will serve as the testing ground for both traditional Q-learning and our modified version. The decay rate of epsilon will be uniform for both methods, as illustrated in Figure 4.4.

4.3.3 Comparative Analysis

Simple Q-learning

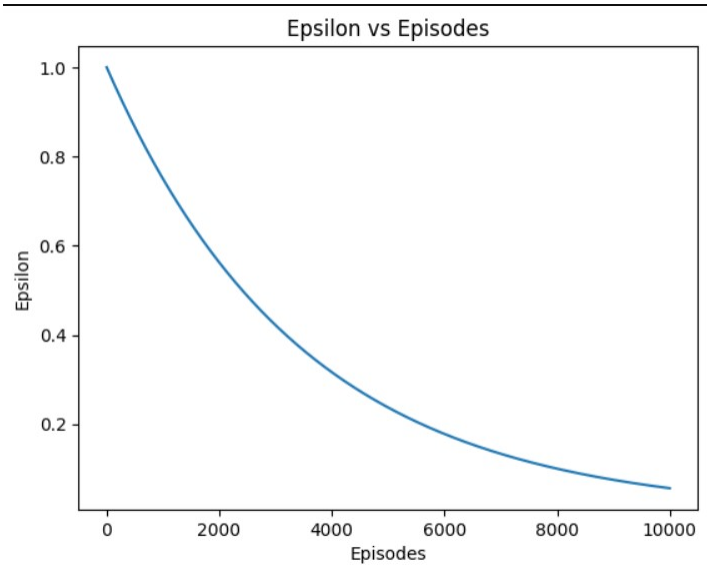
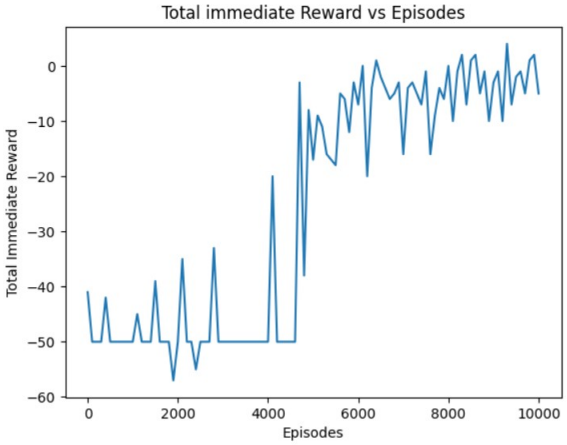
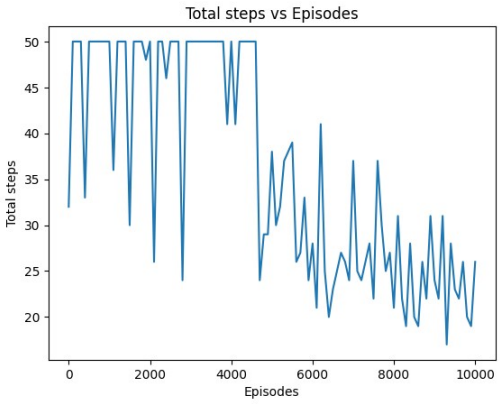


Figure 4.4: Epsilon rate over episodes

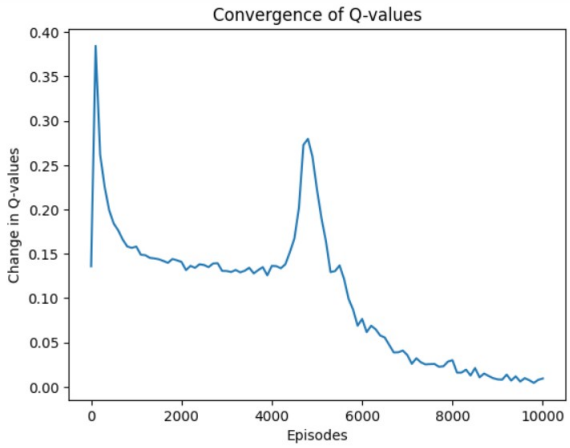
In evaluating Simple Q-learning, two key metrics were considered. Firstly, the total reward for each episode was calculated to assess the stability of immediate rewards over time. Consistent rewards indicate convergence in the learning process. Secondly, Q-value convergence was examined by tracking the average absolute change in Q-values between consecutive episodes. This metric offers insights into whether Q-values stabilize or fluctuate during training. The parameters outlined in Table 4.2 will be used for this approach.



(a) Total immediate reward per episode



(b) Total steps over episodes



(c) Convergence of Q-values over episodes

Figure 4.5: Various metrics for Simple Q-learning in a static grid world.

We get stable rewards and Q-value convergence as expected, the results, depicted in Figures 4.5a , 4.5b and 4.5c, respectively, reveal decent rewards and Q-values, suggesting ongoing learning dynamics with the learning process.

Modified Q-learning

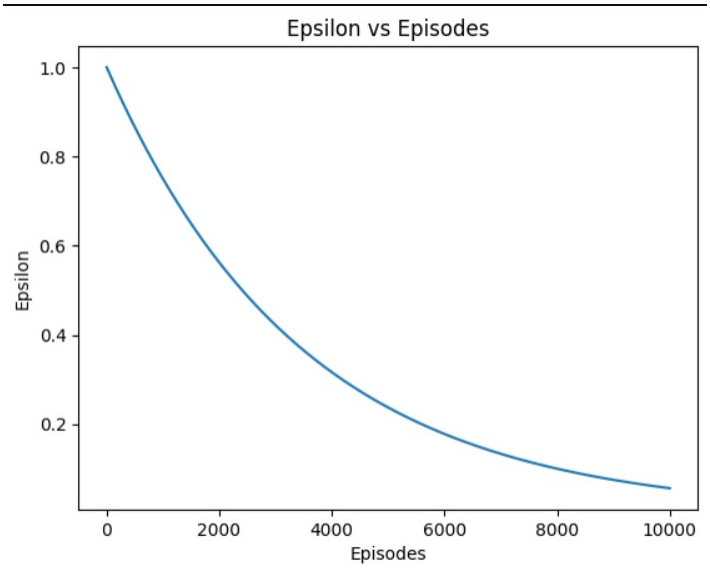
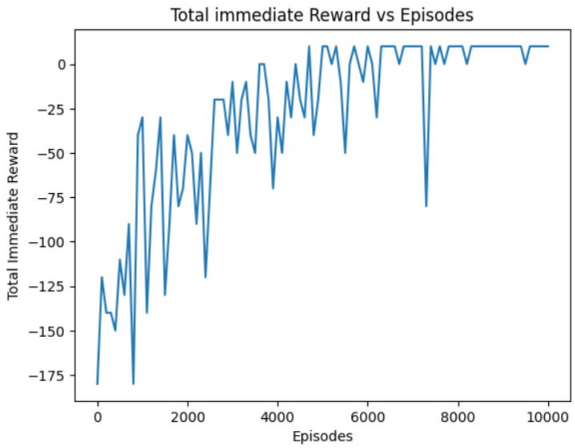


Figure 4.6: Epsilon rate over episodes

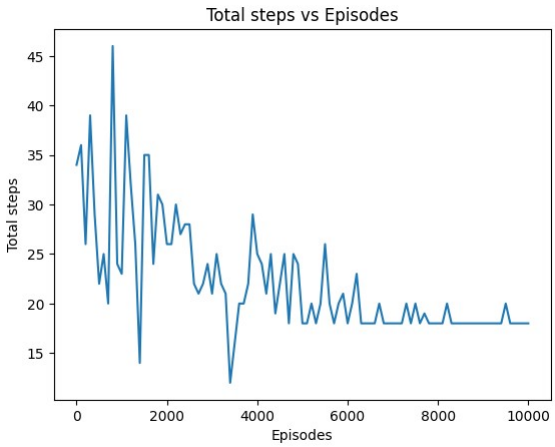
The hybrid approach will utilize the table 4.3 for the required parameters. The epsilon equation will be the same as the simple q learning as shown in fig. 4.6. Our evaluation demonstrated significant improvements with the hybrid approach as shown in fig. 4.7.

Table 4.3: Parameter configurations of Simple Q learning simulations

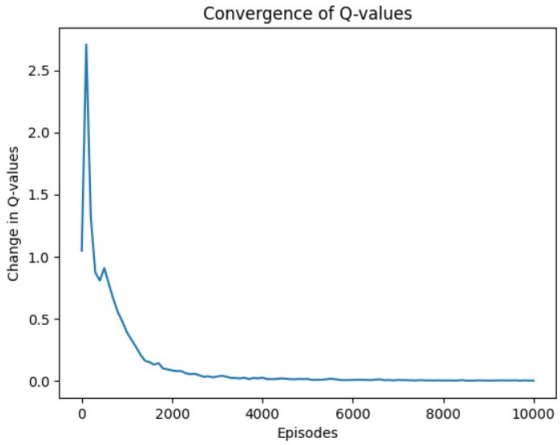
Parameter	Value
Learning Rate – α	0.1
Initial Epsilon – ϵ	1
Episode Number – N	10000
Preferred Action Reward – R	0
Default Reward – R	-1
Obstacle Reward – R	-10
Goal Reward – R	0



(a) Total immediate reward per episode



(b) Total steps over episodes



(c) Convergence of Q-values over episodes

Figure 4.7: Various metrics for Modified Q-learning.

Fig. 4.7a and 4.7b shows better improvements in both total reward over time and total steps over time. Furthermore, Fig.4.7c clearly illustrates convergence. The near-flat line indicates that the Q-values are stabilizing as the number of episodes increases

4.3.4 Part 2 : Larger Static Grid World

The second part of the experimental environment comprises of a larger structured gridworld arranged in a 35x35 configuration, as depicted in Fig. 4.8. The gridworld consist of static obstacles. The main goal of this experiment is to compare Simple Q learning and the Hybrid Q learning

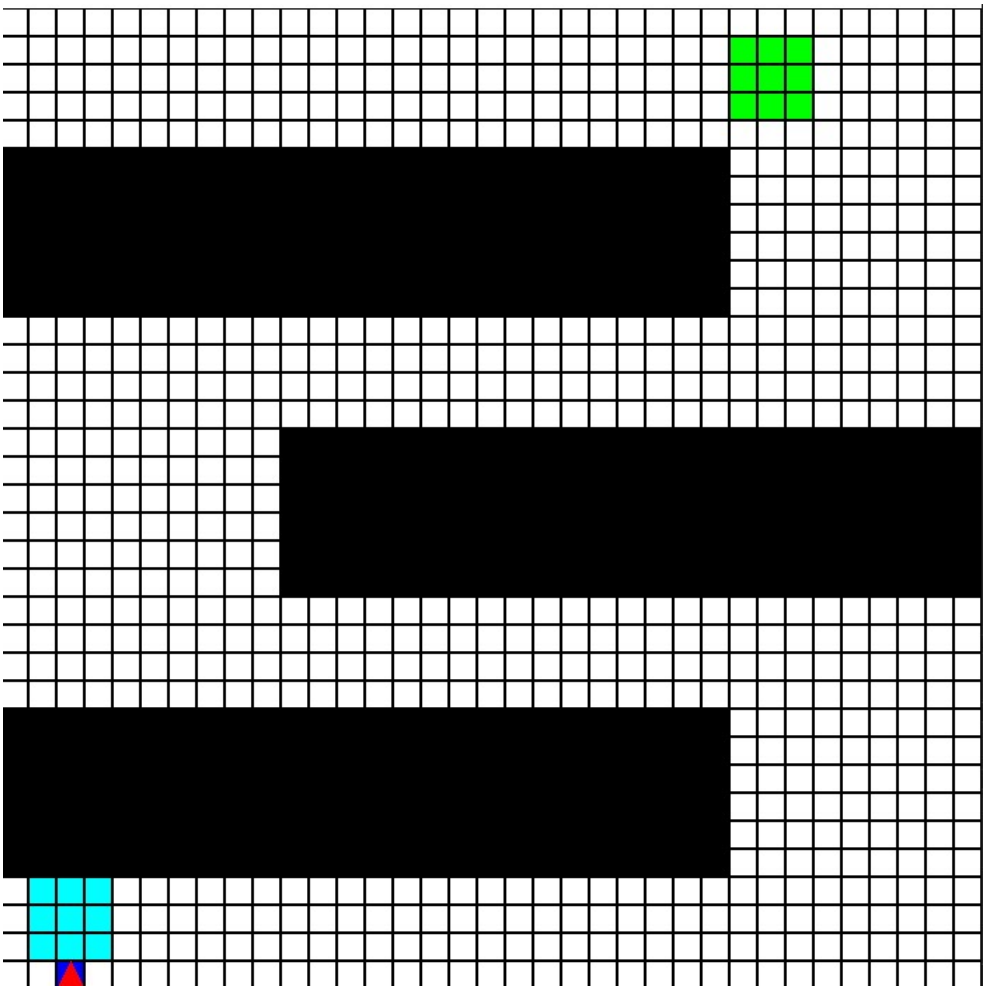


Figure 4.8: Environment Setup in pygame. Agent(red
with three proximity sensors(blue). The agent must traverse through the obstacles(Black) .

4.3.5 Q-learning vs Modified Q-learning

The environment mentioned earlier will serve as the testing ground for both traditional Q-learning and our modified version.

4.3.6 Comparative Analysis

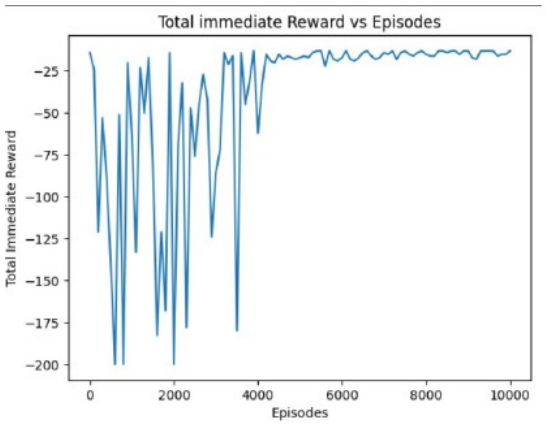
Here We find Q-learning and the hybrid Q learning have similar rewards and both got the required result, however the difference in time taken is immense

- Time Taken for Simple Q learning - 984.11 seconds
- Time Taken for Hybrid Q learning - 296.23 seconds

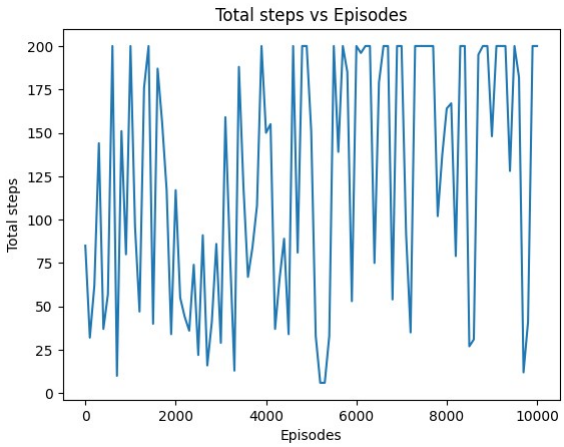
This shows the hybrid Q learning has a lower computational time which contributes to faster learning.

Simple Q-learning

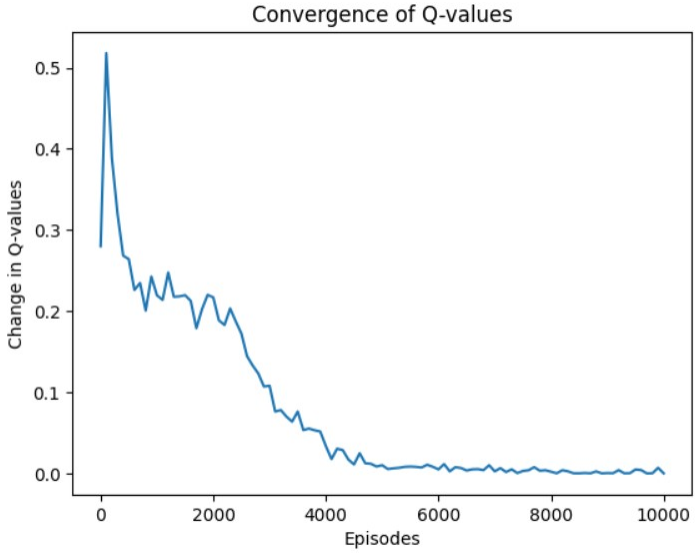
The parameters outlined in Table 4.2 will be used for this approach and a balanced epsilon graph as illustrated in 4.4 will be utilized. We do get stable rewards and Q-value convergence as expected, the results, depicted in fig. 4.9a and 4.9c, respectively, reveal excellent total reward over time and convergence of Q-values. However the total steps over episodes seems to fluctuate by a large margin as shown in fig. 4.9b



(a) Total immediate reward per episode



(b) Total steps over episodes



(c) Convergence of Q-values over episodes

Figure 4.9: Various metrics for Simple Q-learning in a larger static grid world.

Table 4.4: Parameter configurations of Simple Q learning simulations

Parameter	Value
Learning Rate – α	0.1
Initial Epsilon – ϵ	1
Episode Number – N	10000
Preferred Action Reward – R	0
Default Reward – R	-(rowsize*colsize) + occupied space
Obstacle Reward – R	-(rowsize*colsize)
Goal Reward – R	0

Modified Q-learning

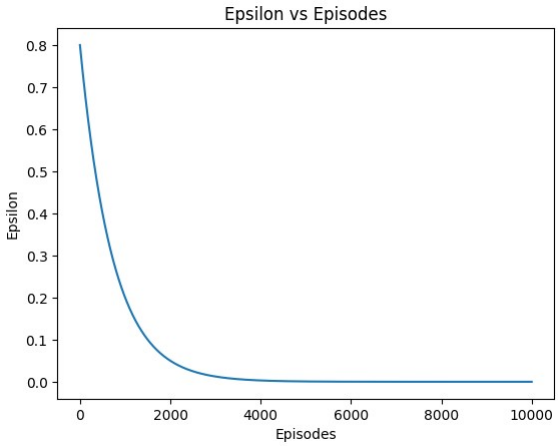
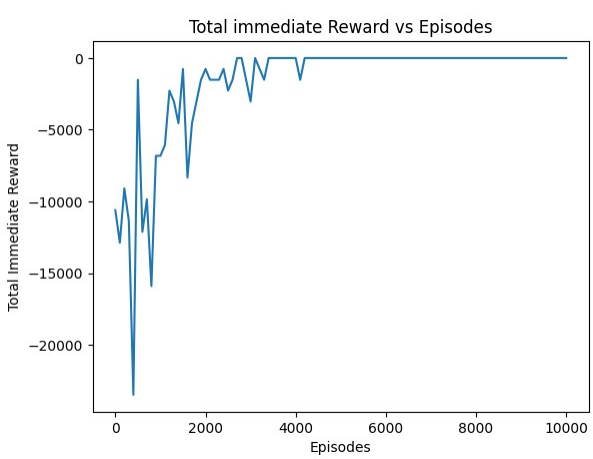
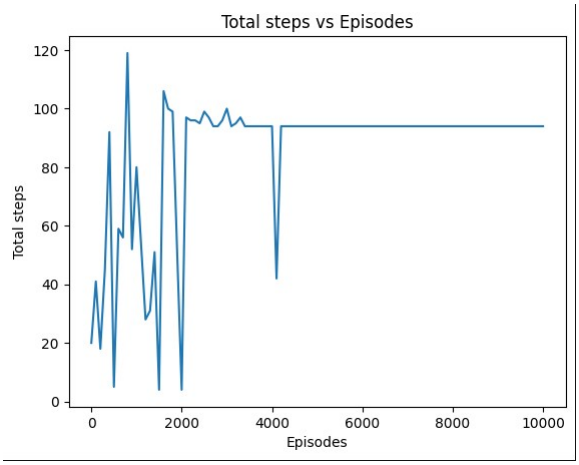


Figure 4.10: Epsilon rate over episodes

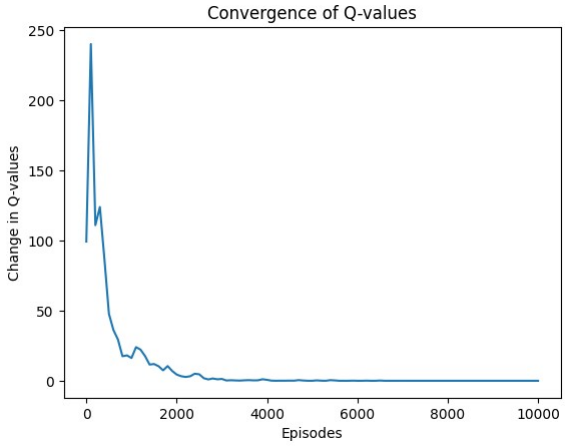
On evaluating hybrid approach, we realised, minimum exploration is required for the agent to compute as immediate learning is not necessary in the environment. Fig. 4.10 shows increase in exploitation rate in comparison the exploration rate..



(a) Total immediate reward per episode



(b) Total steps over episodes



(c) Convergence of Q-values over episodes

Figure 4.11: Various metrics for Modified Q-learning in a larger static grid world.

Fig 4.11 shows very high improvements in all of the graphs. This shows that learning in the case of hybrid, less learning is required regardless of the size of the environment.

4.3.7 Part 3 : Simple Dynamic Grid World

The experimental environment comprises a structured gridworld arranged in a 5x16 configuration, as depicted in Fig. 4.12. Implemented using the pygame framework, the setup features an agent equipped with three proximity sensors. Notably, three obstacles dynamically traverse the grid vertically, adding a dynamic element to the environment.

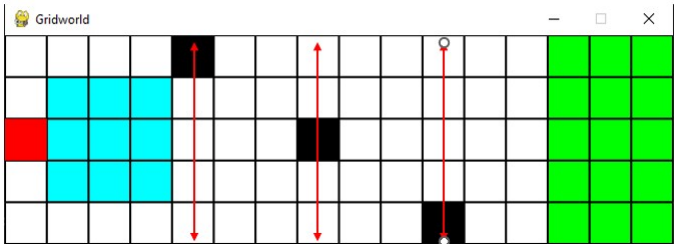


Figure 4.12: Environment Setup in pygame. Agent(red
with three proximity sensors(blue). The three obstacles(black) moves up and down .

4.3.8 Q-learning vs Modified Q-learning

The environment shown earlier will serve as the testing ground for both traditional Q-learning and our modified version.

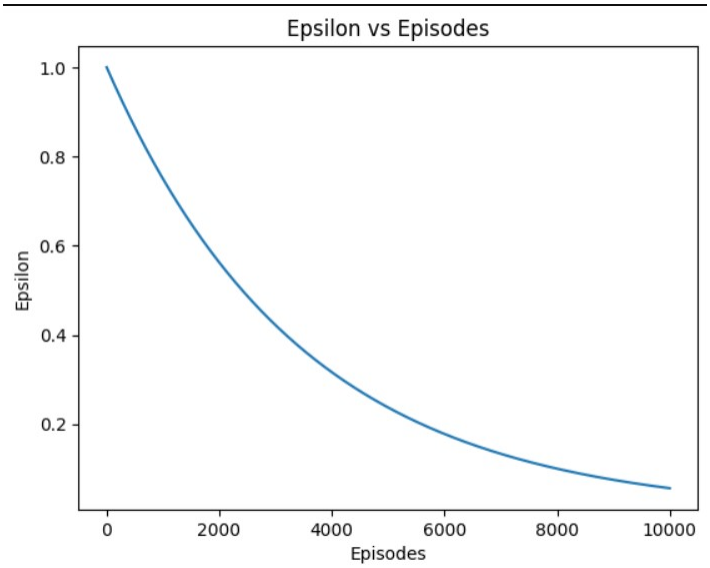


Figure 4.13: Epsilon rate over episodes

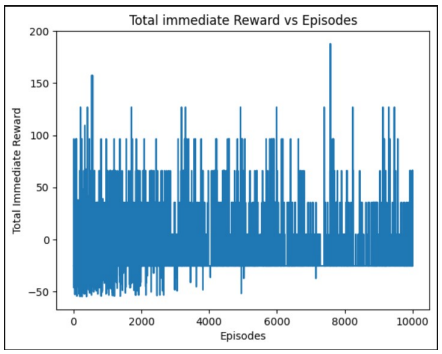
4.3.9 Comparative Analysis

Simple Q-learning

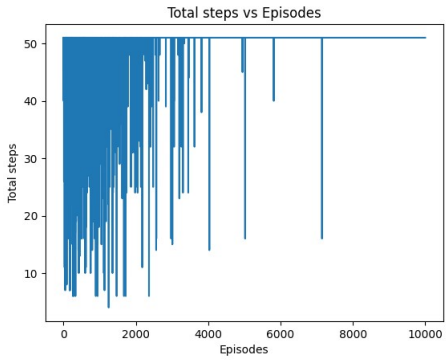
The epsilon graph(fig 4.13) will be used for the case of exploration-exploitation. The parameters from Table 4.5 will be utilized.

Table 4.5: Parameter configurations of Simple Q learning simulations

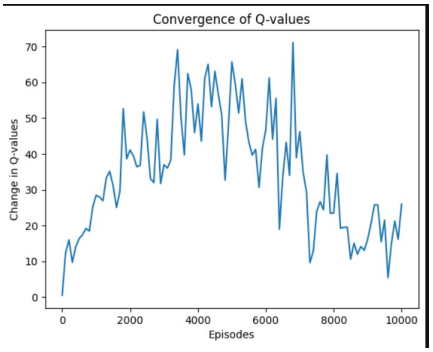
Parameter	Value
Learning Rate – α	0.1
Initial Epsilon – ϵ	1
Episode Number – N	10000
Default Reward – R	-1
Obstacle Reward – R	-10
Goal Reward – R	+20



(a) Total immediate reward per episode



(b) Total steps over episodes



(c) Convergence of Q-values over episodes

Figure 4.14: Various metrics for Simple Q-learning in a dynamic grid world.

Despite the anticipation of stable rewards and Q-value convergence, the outcomes, as shown in Figures 4.14a and 4.14c, respectively, unveil inconsistent rewards and fluctuating Q-values, implying a lack of effective learning. Additionally, 4.14b highlights apparent convergence in the number of steps over time, albeit falling short of the desired learning outcome.

Table 4.6: Parameter configurations of Simple Q learning simulations

Parameter	Value
Learning Rate – α	0.1
Initial Epsilon – ϵ	0.8
Episode Number – N	10000
Preferred Action Reward – R	0
Default Reward – R	-(rowsize*colsize) + occupied space
Obstacle Reward – R	-(rowsize*colsize)
Goal Reward – R	0

Modified Q-learning

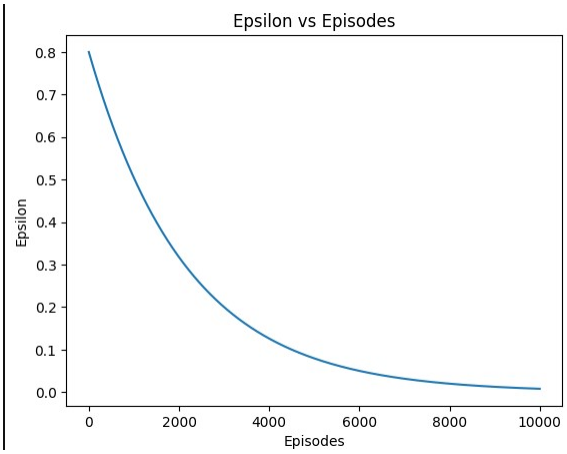
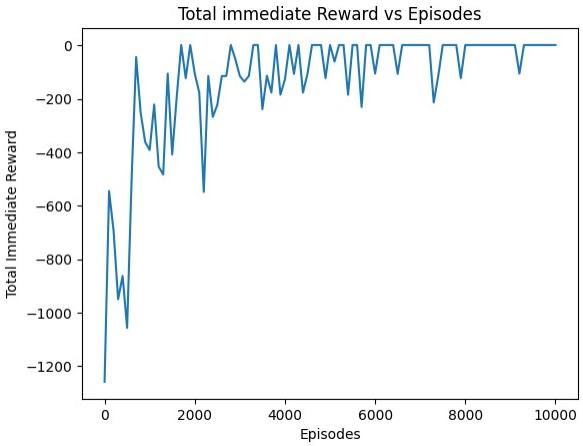
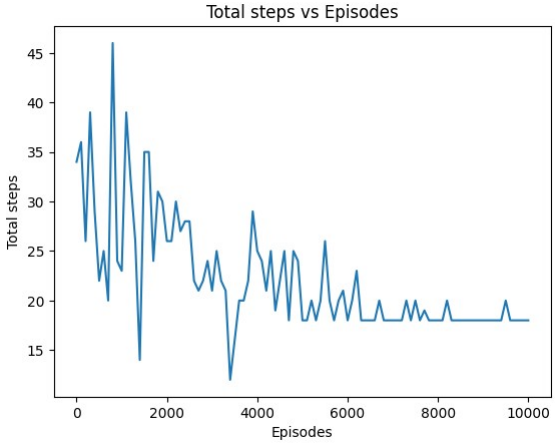


Figure 4.15: Epsilon rate over episodes

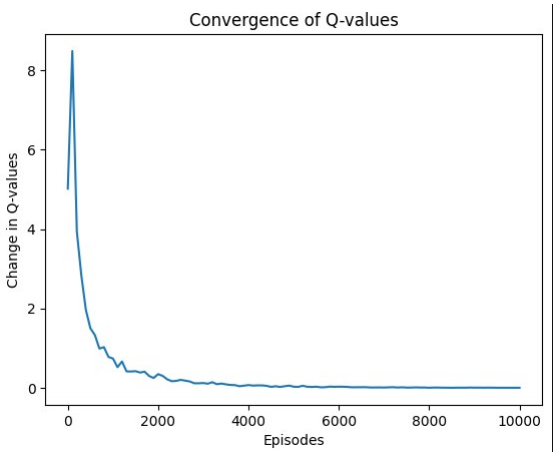
The epsilon graph(fig 4.15) will be used for the case of exploration-exploitation where the exploration rate is reduced . The parameters from Table 4.6 will be utilized.



(a) Total immediate reward per episode



(b) Total steps over episodes



(c) Convergence of Q-values over episodes

Figure 4.16: Various metrics for Modified Q-learning.

Our evaluation demonstrated significant improvements with the hybrid approach. Fig. 4.16 shows best results in all cases in reward,steps and convergence over episodes.

4.3.10 Part 4 : Complex Dynamic Grid World

The experimental environment comprises a structured Environment arranged in a 35x35 configuration, as depicted in Fig. 4.17. Implemented using the pygame framework, the setup features an agent equipped with three proximity sensors. Notably, six obstacles dynamically traverse the grid vertically and horizontally, adding a dynamic element to the environment. Here we will only be utilizing the hybrid approach.

Table 4.7: Parameter configurations of Simple Q learning simulations

Parameter	Value
Learning Rate – α	0.1
Initial Epsilon – ϵ	0.8
Episode Number – N	10000
Preferred Action Reward – R	0
Default Reward – R	-(rowsize*colsize) + occupied space
Obstacle Reward – R	-(rowsize*colsize)
Goal Reward – R	0

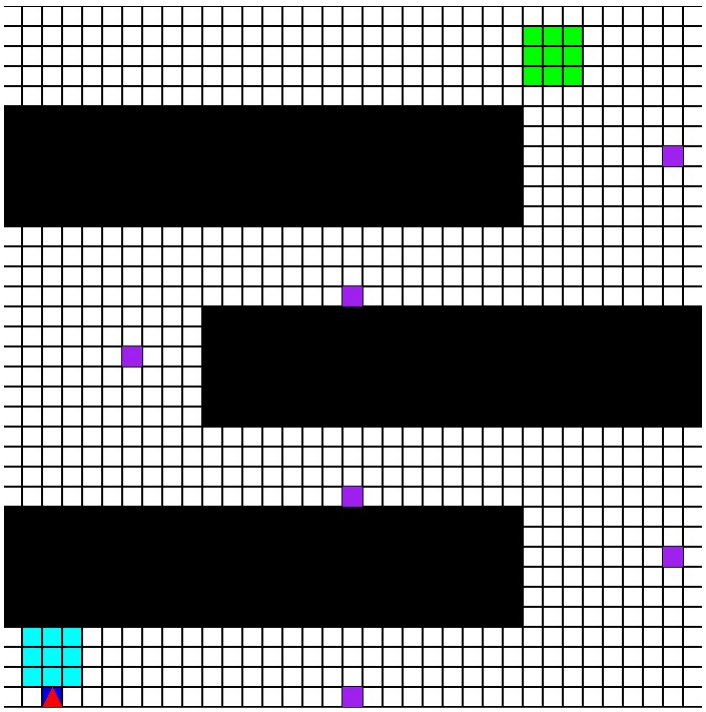
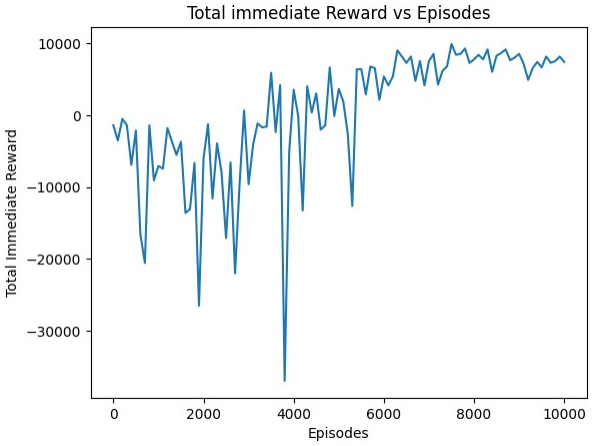


Figure 4.17: Environment Setup in pygame. Agent(red with three proximity sensors(blue). Dynamic Obstacles (purple) moves either horizontally or vertically .

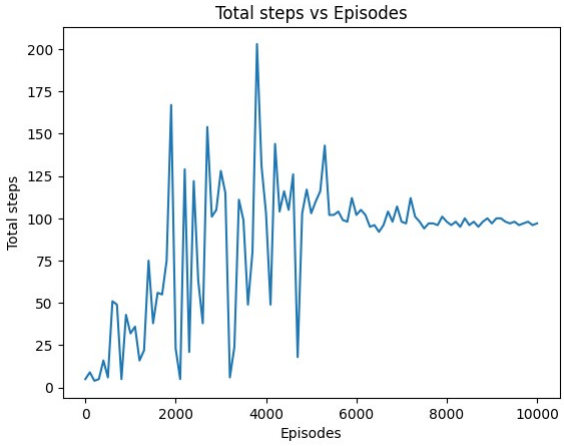
Modified Q-learning

The epsilon graph(fig 4.15) will be used for the case of exploration-exploitation where the exploration rate is reduced . The parameters from Table 4.6 will be utilized.

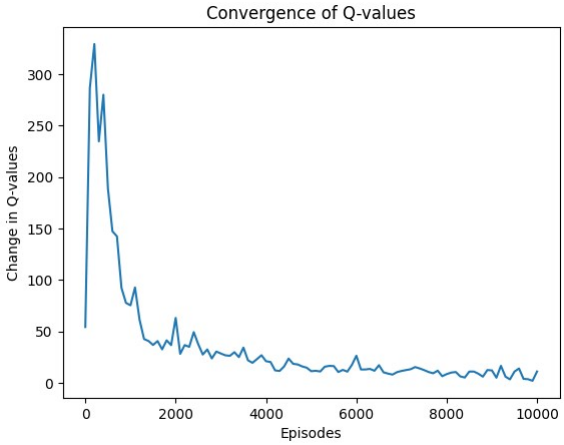
Our evaluation demonstrated decent results with the hybrid approach. Fig. 4.18 shows best results in all cases in reward,steps and convergence over episodes.



(a) Total immediate reward per episode



(b) Total steps over episodes



(c) Convergence of Q-values over episodes

Figure 4.18: Various metrics for Modified Q-learning.

Chapter 5

Conclusion and Future Scope

In this project we explored the potential of combining A* and Q-learning algorithms for autonomous robot navigation in complex environments. The proposed approach borrows the efficiency of A* for pathfinding while incorporating the adaptability of Q-learning for real-time decision-making and dynamic obstacle avoidance. This fusion has the potential to significantly enhance the navigation capabilities of mobile robots, enabling them to operate safely and efficiently in dynamic settings.In comparison with Q-learning, the hybrid was proved to be quite efficient with respect to the rewards over time and the convergence of Q-values.

Future research will focus on evaluating the performance of the proposed fusion approach in real-world environments with varying complexities. Additionally, we aim to explore advanced Q-learning techniques to address the challenges associated with high-dimensional state spaces. Furthermore, investigating alternative fusion methods, such as deep reinforcement learning, could be a promising avenue for further improvement.

References

[1] Pei, Muleilan, Hao An, Bo Liu, and Changhong Wang. "An improved dyna-q algorithm for mobile robot path planning in unknown dynamic environment." IEEE Transactions on Systems, Man, and Cybernetics: Systems 52, no. 7 (2021): 4415-4425.

[2] Khriji, Lazhar, Farid Touati, Kamel Benhmed, and Amur Al-Yahmedi. "Mobile robot navigation based on Q-learning technique." International Journal of Advanced Robotic Systems 8, no. 1 (2011): 4.

[3] Bianchi, Reinaldo AC, Carlos HC Ribeiro, and Anna HR Costa. "Heuristically Accelerated Q–Learning: a new approach to speed up Reinforcement Learning." In Brazilian Symposium on Artificial Intelligence, pp. 245-254. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[4] Altuntaş, Nihal, Erkan Imal, Nahit Emanet, and Ceyda Nur Öztürk. "Reinforcement learning-based mobile robot navigation." Turkish Journal of Electrical Engineering and Computer Sciences 24, no. 3 (2016): 1747-1767.

[5] Yun, Soh Chin, Subramaniam Parasuraman, and Velappa Ganapathy. "Mobile robot navigation: neural Q-learning." In Advances in Computing and Information Technology: Proceedings of the Second International Conference on Advances in Computing and Information Technology (ACITY) July 13-15, 2012, Chennai, India-Volume 3, pp. 259-268. Springer Berlin Heidelberg, 2013.

[6] Surmann, Hartmut, Christian Jestel, Robin Marchel, Franziska Musberg, Houssem Elhadj, and Mahbube Ardani. "Deep reinforcement learning for real autonomous mobile robot navigation in indoor environments." arXiv preprint arXiv:2005.13857 (2020).

[7] Shi, Zhen, Keyin Wang, and Jianhui Zhang. "Improved reinforcement learning path planning algorithm integrating prior knowledge." Plos one 18, no. 5 (2023): e0284942.

[8] Sichkar, Valentyn N. "Reinforcement learning algorithms in global path planning for mobile robot." In 2019 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), pp. 1-5. IEEE, 2019.

[9] Bianchi, Reinaldo AC, Carlos HC Ribeiro, and Anna HR Costa. "Heuristically Accelerated Q–Learning: a new approach to speed up Reinforcement Learning." In Brazilian Symposium on Artificial Intelligence, pp. 245-254. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.