DLR-IB-FA-SD-2021-128

Deep reinforcement learning and graph-based approaches for multi-robot collision control

Masterarbeit

Nihal Acharya Adde

Deutsches Zentrum für Luft- und Raumfahrt

MH-FA-56-FB03_v1.3

Institut für Faserverbundleichtbau und Adaptronik

DLR-IB-FA-SD-2021-128

Deep reinforcement learning and graph-based approaches for multi-robot collision control

Zugänglichkeit:

Stufe 1 Allgemein zugänglich: Der Interne Bericht wird elektronisch ohne Einschränkungen in ELIB abgelegt. Falls vorhanden, ist je ein gedrucktes Exemplar an die zuständige Standortbibliothek und an das zentrale Archiv abzugeben.

Stade, *August, 2021*

Der Bericht umfasst: 84 Seiten

Abteilungsleiter:
Dr. Ing. Daniel Stefaniak

Autor:
Nihal Acharya Adde

Autor 2 / Betreuer:
Dr. Christoph Brauer

Deutsches Zentrum für Luft- und Raumfahrt

Institute for Mathematical Optimization
Institute of Analysis and Algebra
German Aerospace Center (DLR)

Master's Thesis

Deep reinforcement learning and graph-based approaches for multi-robot collision control

Nihal Acharya Adde

4941635

August 31, 2021

Institute for Mathematical Optimization
Institute of Analysis and Algebra
German Aerospace Center (DLR)

Prof. Dr. Sebastian Stiller, Prof. Dr. Timo de Wolff

Supervisor:
Dr. Christoph Brauer

Statement of Originality

This thesis has been performed independently with the support of my supervisor/s. To the best of the author's knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text.

Braunschweig, August 31, 2021

_____

Abstract

Recently, multi-robot systems, in which several robots coordinate along with each other to achieve clearly defined goals, have become increasingly popular. But the challenge is to schedule them efficiently to avoid collisions and execute their respective jobs in the least time possible. The simplicity of multi-robots led to a wide range of potential applications, and we will focus on applications in the process of fiber placement. Fiber placement refers to a fabrication process for composite materials where reinforcing fibers are placed along a predetermined path in the component. The present thesis proposes various approaches like reinforcement learning and graph-based methods for optimal collision control in a multi-robot system and to schedule the robots to execute their tasks in the least possible time. These can be compared and later employed to significantly reduce the lead time in the multi-robot fiber placement process.

Keywords: Multi-robot system, Collision control, Reinforcement learning, Graph-based shortest path algorithms

Contents

List of Figures

List of Tables

Abbreviations

AI Artificial Intelligence

BFS Breadth-first search

CFRP Carbon fiber reinforced plastic

CNN Convolutional Neural Network

DDQN Double DQN

DFS Depth-first search

DLR Deutsches Zentrum für Luft- und Raumfahrt

DQN Deep Q-Network

DRL Deep Reinforcement Learning

GroFi Großbauteile in Fibreplacementtechnologie

MDP Markov decision process

MRS Multi-Robot System

POC Proof of Concept

RL Reinforcement learning

SARSA State-action-reward-state-action

TD Temporal difference

ZLP Zentrum für Leichtbauproduktionstechnologie

Nomenclature

A Set of all the possible actions

α Learning rate

a Action taken in reinforcement learning

δ TD error

E Set of edges of a graph

G Representation of a graph

R Action's return or cumulative rewards

γ Discount factor

μ Distance of the shortest path yet seen

n_1 Total number of jobs for robot 1

n_2 Total number of jobs for robot 2

n_{max} Maximum number of jobs per robot

n_{min} Minimum number of jobs per robot

π Policy in reinforcement learning

P Path in a graph

ρ Potential for goal directed Dijkstra's algorithm

r Reward achieved in reinforcement learning

s Source vertex of a graph

s State observation in reinforcement learning

t Target vertex of a graph

V Set of vertices of a graph

w Edge weight

1 Introduction

Fiber placement refers to a fabrication process for composite materials where reinforcing fibers are placed along a predetermined path in the component. Usually, a Multi-Robot System (MRS) where several robots coordinate among themselves is employed to achieve this process. One of the examples of an MRS for fiber placement is GRoFi operated by the Center for Lightweight-Production-Technology in Deutsche Zentrum für Luft- und Raumfahrt (DLR), Stade. GroFi is developed to manufacture several highly integral components of fiber composite materials in an automated fiber placement process. But the challenge here is to ensure that these robots efficiently coordinate among themselves and take the optimal path with the least time. This thesis focuses on solving this challenge and finding the optimal paths in a multi-robot system to schedule the robots to execute their tasks as early as possible without any collisions. The findings of the thesis are based on the work carried out by me as a student assistant at Zentrum für Leichtbauproduktionstechnologie (ZLP), DLR, Stade.

To approach the challenge, we need the data from a multi-robot system. For this purpose, we could use the data generated by a simulation software VNCK provided by Siemens that simulates the working of multiple robots to check for possible collisions when they function simultaneously. But this software has limited data set from the real world. Hence we will create a small-scale collision control environment that randomly generates the data set with collision points similar to the actual simulation data. The data set generated randomly will be represented in a matrix form that is called a collision matrix. Later, we will apply deep Q learning, a type of model-free reinforcement learning using the open-source software of the TF-agents library by Google to find the fastest path of robots in the collision matrix. For a random collision matrix configuration, we will validate the performance of this method by using several metrics for evaluation and represent only the best hyperparameters and high accuracy models. In the next step, for the same collision matrix, we will be representing the robot scheduling task as a graph and consequently querying it with various algorithms to find the shortest path from the source to the target. We present a few speed-up techniques to reduce the total execution time of these graph-based methods. Additionally, this thesis discusses learning-based speed-up for large graphs as a literature survey. We then compare the results of several algorithms in graph-based methods against each other and also against the results from reinforcement learning. Furthermore, we will also evaluate the graph-based method by applying it to data generated by VNCK software. As the last step, we will explore the advantages and disadvantages of all these methods in finding the optimal fastest path in the multi-robot system.

The report will be divided into six main chapters, the first being this introductory chapter. The overview of upcoming chapters is as follows. The Chapter 2 elaborates the problem, data set, and the motivation of the thesis. Chapters 3 and 4 brief on reinforcement learning and graph-based methods used to find the optimal fastest path. Here, we will discuss the fundamental of different algorithms used and explain the implementation of these methods to solve our task. In Chapter 5, these methods are applied to the data sets, and the results are interpreted. Finally, the concluding Chapter 6 contains a review of the findings and the valuable takeaways from the thesis.

2 Problem Description

This chapter briefly describes the structure of Großbauteile in Fibreplacementtechnologie (GroFi) and presents the manufacturing process at the DLR plant in Stade. For the multi-robot system described in the manufacturing process, an efficient method is to be developed such that the multi-robot system is scheduled in a way that the robots execute their respective jobs in the least possible time excluding the risk of collision. The thesis focuses on a two robot system that can be easily expanded to a n-robot system in many cases. This is explained in detail in the further chapters. This chapter is based on the observation of the production process during my work at ZLP Stade, as well as explanations from various employees, the thesis of Markus Schreiber [Sch15] and the final documentation of the GroFi project [KB14].

2.1 GRoFi system

GRoFi is a large-scale facility operated by the Center for Lightweight Production Technology in German Aerospace Center (DLR), Stade. GroFi is developed with an aim to manufacture large, highly integral components made of fibre composite materials in an automated fibre placement process. Therefore, a system of coordinated robots was developed which enables simultaneous fiber placement which is used flexibly for production tasks. In addition to the robotic platform, the system's technology also includes a new generation of fiber placement and tape laying heads. Through the use of coordinated and simultaneous working layup, a highly flexible research platform is realized. The combination of different layup technologies, namely fiber placement (dry) and tape laying enable the development and validation of new technologies and manufacturing processes for large-area composite parts. This allows the investigation of new materials, technologies and processes on both, small coupons, but also large components such as wing covers or fuselage skins [Lufi6].

The facility is developed based on several coordinated robot-based layup units that can be moved on the rail system. Up to 8 layup units are located on the rail network. The rail system is outlined in Figure 2.1. The rail system is divided into a manufacturing loop, which allows circumferential movement around the double-sided moulding tool, and an additional loop for maintenance. It consists of straight, continuous sections connected by rotating platforms, two of which are designated as working areas. In the area between these two rails, workpieces can be suspended onto which the depositing units apply the carbon fiber reinforced plastic (CFRP). In addition, there are six maintenance rails on which the depositing units are refitted and retooled. The remaining rails are only intended as a

connection between these areas.

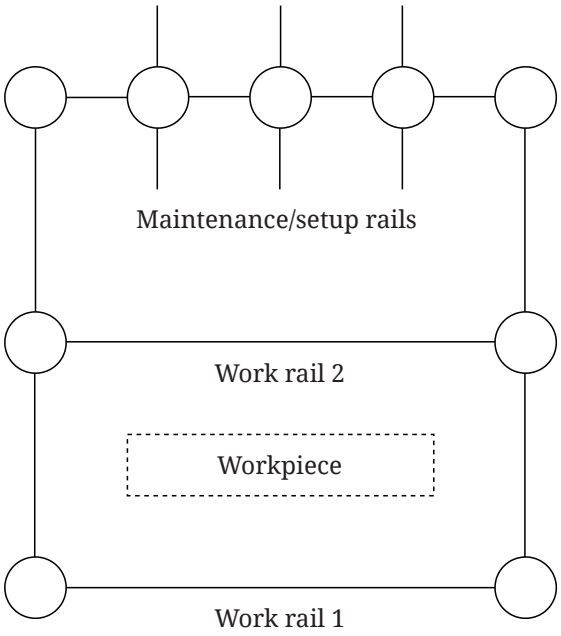


Figure 2.1: Schematic representation of the rail network. Straight lines represent rails and circles represent rotating platforms.

A layup unit consists of a shuttle platform on which a 6-joint articulated robot is placed. A tool head is mounted on the flange of the robot. These units can move independently along the rails. To reach an adjacent rail, the units have to cross the rotating platforms between the rails. If necessary, they are rotated by these according to the orientation of the rail to be approached.

As discussed above, numerous robots work on the same workpiece simultaneously. The limitation of the present system is that the robot jobs are not scheduled optimally. The lead time for the multi-robot system can be significantly reduced if the jobs are scheduled efficiently. The next sections form the baseline of the research, which motivates the use of different methods to efficiently schedule the robots.

2.2 Robot Job Scheduling

Presently, the multiple robots are scheduled in such a way that they always maintain a certain distance from each other during execution to avoid collisions. For efficient scheduling of the multi-robot system, we first need a data set that provides information about possible collisions. The data set can be achieved by a simulation that simulates the working of multiple robots to check for possible collisions when they function simultaneously.

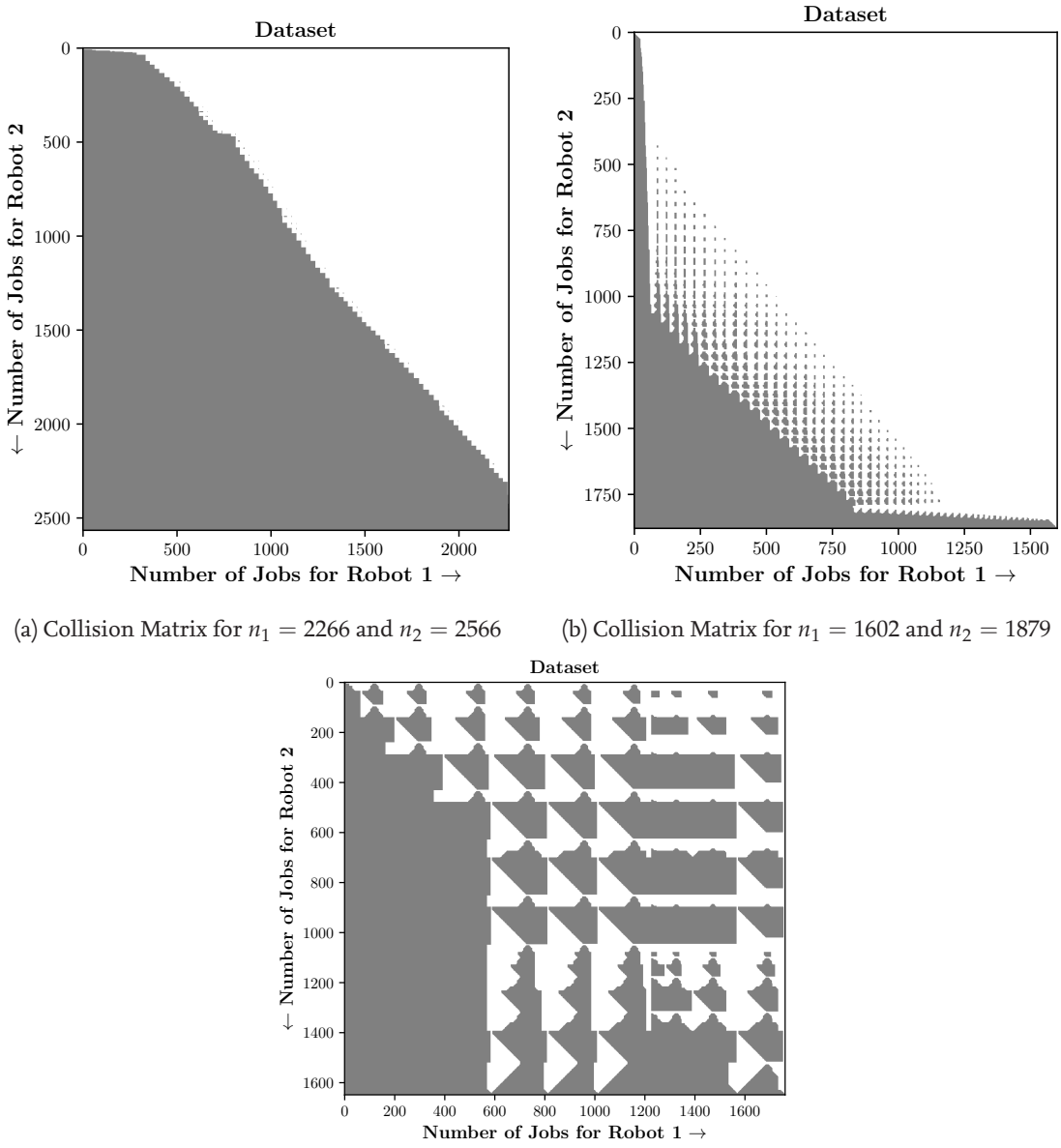
Simulation software VNCK provided by Siemens is used to carry out such simulations. It simulates all the possible sequences in which the jobs can be carried out by a single robot. Each time step corresponds to a 4 millisecond. This simulation is carried out for every robot that works simultaneously on the workpiece. By comparing the data from every robot that work simultaneously, a matrix (for a 2D case) or a tensor (for a higher dimensional case) is formed that represents the various possible collisions at different timesteps. This data set can be called a collision matrix/tensor. The shape of this data set depends on the number of robots that work simultaneously, i.e., 2D matrix for 2 robot system, 3D tensor for 3 robot system and so on. This thesis mainly deals with the 2 robot system. The assignment of jobs to robots takes place before the collision matrix is established. Based on that, the VNCK simulates parallel execution of all pairs of jobs $(j_1, j_2) \in J_1 \times J_2$, where J_i is the set of jobs assigned to robot i. So, finding a shortest collision-free path through the matrix can be considered a subsequent scheduling step. The upcoming section gives a clear insight into the representation of data in the collision matrix.

2.2.1 Collision Matrix

The collision matrix is a binary matrix where the collision points are represented by 1, and the others are represented by 0. A 0-point indicates the timestamps of non-collision between the jobs carried out by different robots simultaneously. We can also interpret this matrix as a graph in which each axis represents the jobs of a particular robot. The goal is to find an efficient traceable path from the first entry (source) of the matrix to the diagonally opposite last entry (target) of the matrix. To accomplish this goal, we can trace several different paths from first entry to last entry without encountering a single collision point (1). But an optimal path is such that the task completion takes the least possible time. To illustrate this, the thesis considers a 2 robot system that generates a 2d matrix to find the optimal path. These procedures can be further extended to find the optimal fastest path in an n-robot system.

For a 2 robot system, if n_1 and n_2 are the total number of jobs for robot 1 and robot 2, respectively, the resulting size of the collision matrix will be $n_1 \times n_2$. The job execution starts at $(0, 0)$. If robot 1 executes its respective job while the robot 2 stays idle, the job pointer is moved towards the right to $(0, 1)$. Similarly, if robot 2 executes its respective job while the robot 1 stays idle, the job pointer has moved a step downwards to $(1, 0)$. Finally, if both the robots execute their respective jobs, the job pointer moves 1 step diagonally downwards to $(1, 1)$. As explained before, each job step represents the execution of the robot for $4ms$. Figure 2.2 shows three collision matrices for a 2 robot system obtained by simulation. In all the 3 matrices considered here, the robots are assigned at most 2500 jobs each. In the worst-case scenario where the two robots work individually, each robot completes its job in around $10s$ ($2500 \times 4ms = 10s$). Therefore, the datasets considered here are for relatively small workpieces. In the figure, the grey points represent the colli-

sion points 1. As seen in the figure, there exist various paths through the traceable points from the source $(0, 0)$ to the target (n_1, n_2) and we aim to find the shortest one.



(a) Collision Matrix for $n_1 = 2266$ and $n_2 = 2566$

(b) Collision Matrix for $n_1 = 1602$ and $n_2 = 1879$

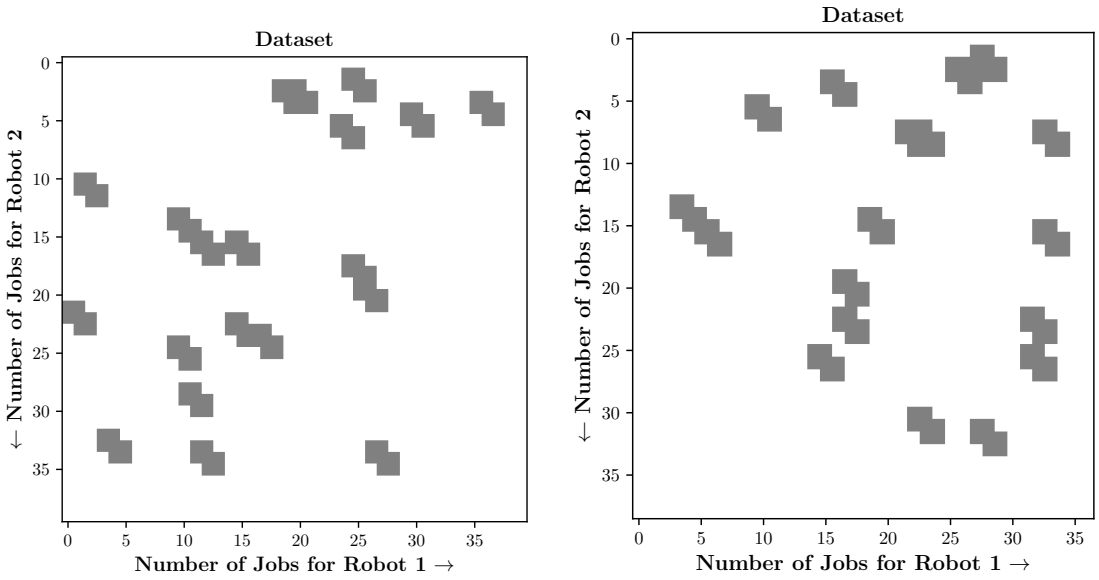
(c) Collision Matrix for $n_1 = 1761$ and $n_2 = 1649$

Figure 2.2: Collision matrix dataset generated by simulation

However, due to the limited availability of data and also to simplify the problem, we try to first solve the task in a small scale regime. The algorithms can be later extended to the actual data. To do so, datasets are randomly generated with various collision points. The creation of such datasets for a small scale regime will be explained in the upcoming Chapter 3. Figure 2.3 shows a couple of examples of such dataset for (a) $n_1 = 40$, $n_2 = 40$

and (b) $n_1 = 37$, $n_2 = 39$. Similar to the previous case, grey points represent the collision points. We follow this colour representation in the rest of the thesis to represent the collision points.

Since the motivation of the task is to find an efficient path with minimal total execution time, we can use optimal methods to find the shortest path. Although various methods can be used to find an efficient path, we consider deep Q learning and graph-based methods to solve such problems, which will be explained in Chapters 3 and 4.



(a) Collision Matrix for $n_1 = 40$ and $n_2 = 40$

(b) Collision Matrix for $n_1 = 37$ and $n_2 = 39$

Figure 2.3: Randomly generated collision matrix dataset for a small scale regime

3 Methodology 1 - Reinforcement learning

Human beings do not learn from a concrete set of data, rather through continuous experience-driven trial and error processes in which decisions are made. Each decision has its outcome which can be positive or negative, and these feedbacks received from the environment guide the learning process for further decisions. In reinforcement learning, the feedback from the environment is called reward/punishment. Almost all biological intelligence is due to an interactive trial and error process with its environment, and all living organisms are greedy reward-driven entities. The reward-driven trial-and-error process in which a system learns to interact with a complex environment to achieve rewarding outcomes is referred to in machine learning parlance as *reinforcement learning* [Agg18, p. 373].

Reinforcement learning (RL) is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning. In reinforcement learning, the agent (e.g., a real or simulated robot) takes an action to maximize the cumulative reward to solve a sequential decision task. The agent is a mimic of the human brain that can understand the coupling between action, state of the system, and reward function. RL seeks an optimal policy that dictates the learning agent's behaviour in each state to maximize the total expected reward (or minimize the punishment) by trial and error interaction with the environment. The RL problem is defined by three features, namely, agent-environment interface, function for evaluating rewards, and Markov property of the learning process [DS13, p. 547]. In Deep Reinforcement Learning (DRL), the learning of taking an action based on feedback is achieved by deep learning. In a larger scope, DRL can be a gateway to the journey of creating truly intelligent systems and revolutionize the field of Artificial Intelligence (AI) by building autonomous systems with a high-level understanding of the visual world.

3.1 Basic Framework of Reinforcement Learning

In Reinforcement Learning, the agent interacts with its environment through various actions. The environment is typically the set of states the agent tries to alter by taking certain actions. These actions modify the environment resulting in a new state, and rewards are awarded or deducted as a consequence of these actions. The awarded rewards are based on how well the goals of the learning applications are achieved. For example, in a video

game, the movement of the playing character in a certain direction is considered as an action and the player who controls the movement is the agent. The environment can be considered as the entire set-up of the video game itself where the agent brings in an action. All the variables describing the current position of the player at a particular point are represented by a state.

Each reward is associated with a certain action in isolation. However, how the rewards are considered in the system depends on the particular problem at hand. For example, in a video game for which the initial state and the state transitions are deterministic, a reward is not earned based on a particular move rather it depends on all the moves done in the past. The credit assessment for a self-driving car is different as the reward for rapidly steering the car in the normal state would be different from performing similar action which would increase the risk of collision. Therefore these rewards are problem-specific and hence it is required to quantify the reward of each action in a way that is specific to a particular system state.

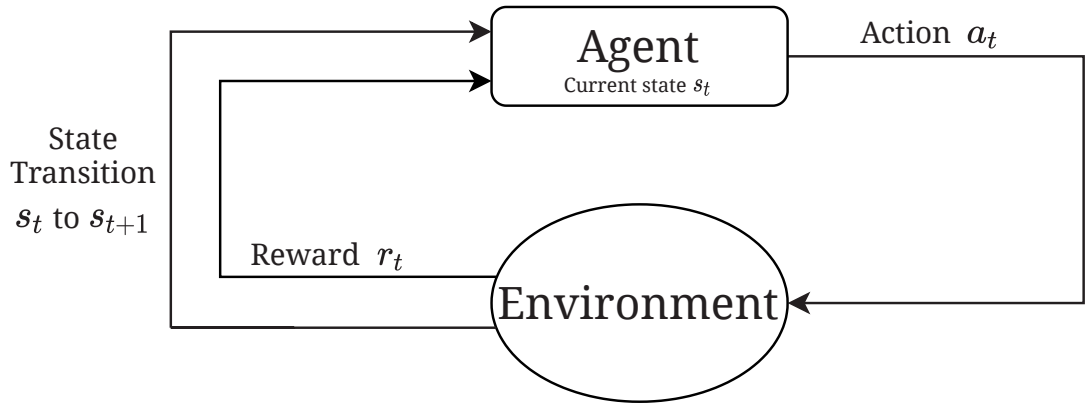


Figure 3.1: Framework of reinforcement learning

One of the primary goals of reinforcement learning is to identify the inherent values of actions in different states, irrespective of the timing and stochasticity of the reward. The learning process helps the agent choose actions based on the inherent values of the actions in different states [Agg18, p. 378]. In deep reinforcement learning, neural networks are used to predict the actions from the sensory inputs. Figure 3.1 shows the interaction of the agent with its environment. RL learns the mapping from states s to actions a by maximizing the rewards r. The agent receives sensory input as a state s_t from its environment. After performing a specific action a_t, the agent receives a reward r_t. This loop stabilizes the algorithm. Trial and error (exploration) help find a better possible action, whereas the memory of high reward winning actions (exploitation) helps keep good solutions. This is called the exploration-exploitation trade-off and is explained in detail in Section 3.4.1. In every step, the agent receives the feedback of the next environment state s_{t+1} and the

reward r_t for its action a_t. This process is continued till it terminates. The entire set of states, actions and transitions from one state to another is called the Markov decision process. The state in a particular time step encodes all the information required by the environment to make a state transition by carrying out a specific action. The finite Markov decision process terminates after a certain number of steps, which is referred to as an episode. An episode of this process is a finite sequence of states, actions and rewards. An episode of length $(n+1)$ is as follows:

$$s_0 a_0 r_0 s_1 a_1 r_1 \ldots\ldots s_t a_t r_t \ldots\ldots s_n a_n r_n$$

On the other hand, Infinite Markov decision (continuous reinforcement learning) processes do not have finite episode length and are referred to as non-episodic. Function approximators are used to approximate the value function. Although longer exploration leads to a better approximation of the action-value function in discrete reinforcement learning, it is not the case in continuous reinforcement learning. In the latter, the approximation accuracy of the function depends on the distribution of the data [DS13, p. 550].

3.1.1 Model-Based vs Model-Free

The model has a very specific meaning in reinforcement learning. It refers to the different dynamic states of the environment and how the rewards are achieved from these states. Reinforcement learning can be either model-based or model-free. Irrespective of it being model-based or model-free, agents may use value functions or direct policy search. The terms model-based and model-free do not refer to the use of the neural network or other statistical learning models. Rather, the term refers to whether the agent uses predictions of the environmental response during learning. The agent can use a single prediction from the model of the next reward and next state (a sample), or it can ask the model for the expected next reward. These predictions can be provided entirely outside the learning agent or can be learned by the agent (approximation). In model-free approaches, the optimal policy is obtained by directly mapping the states to the actions. They learn to take different actions based on the situation (state) but do not learn the effect of the actions. On the other hand, model-based methods try to construct a model of the environment, typically in the form of a Markov decision process (MDP). They acquire optimal behaviour by learning the model of the environment by taking actions and observing their outcomes. Therefore, model-based methods often use limited experience and mostly achieve a better policy with fewer interactions with the environment. However, model-free methods are simple and computationally less expensive compared to model-based methods. Algorithms that sample from the experience such as Q-learning and SARSA are examples of model-free algorithms. They rely on samples from the environment and never use generated predictions of the next state and next rewards to alter behaviours. However, they might sample from experience memory which is close to being a model.

3.1.2 On-policy vs Off-policy

On-policy learning algorithms evaluate and improve the same policy that is used to select the action. That is, they use the policy that the agent is already using for selecting an action. State-action-reward-state-action (SARSA) is an example of an on-policy algorithm that estimates the policy being followed. In this algorithm, the agent grasps the optimal policy and uses the same to act. The policy that is used for updating and the policy used for acting is the same, unlike in Q-learning. In contrast to this, off-policy algorithms evaluate and improve a policy that is different from the policy that is used to select an action. It is independent of the agent's action and it finds the optimal policy regardless of the agent's motivation. Q-learning is an example of an off-policy algorithm where the agent learns the optimal policy with the help of a greedy policy during exploration and behaves using a different policy. The $Q(s, a)$ function is learned from actions that we took using our current policy. In Q learning, the best possible action is chosen to update the parameters even though the policy that is actually executed is greedy. Most commonly, an ϵ-greedy policy is used which acts randomly with the probability ϵ or greedily with the probability $1 - \epsilon$. It is explained further in Section 3.4.1. If we set the value of the greedy policy to 0, Q-Learning and SARSA would specialize to the same algorithm. However, due to no exploration, such a method will not function well. SARSA is used when learning cannot be separated from prediction. Q learning is used when offline learning is possible. However, it should be noted that ϵ-greedy policy must not be used at inference as the policy never pays for its exploratory component and therefore does not learn how to keep exploration safe. For example, a Q learning based robot will attempt to find the shortest path from source to destination though it is along the edge of a cliff, whereas a SARSA trained robot will not [Agg18, p. 388].

3.1.3 Actor-Critic Model

An agent tries to find an optimal policy π that maximizes the expected value function of immediate rewards while following that policy. A policy can be defined as the algorithm an agent uses to determine its action [Ger19, p. 612]. The reinforcement learning algorithm can be divided into three groups [WS92, p. 469]: actor(policy)-only, critic(value function)-only, and actor-critic methods. Actor only methods that work with a parameterized policy over its optimization can be used to get an optimal policy. The optimization method used sometimes suffers from high variance in the gradient estimation. Due to its gradient descent nature, the actor only methods have strong convergence but results in slow learning. The advantage of the actor only methods over critic only methods are that they allow the policy to generate action on complete continuous action space.

In critic-only methods, the estimates of expected returns have lower variance [Sut88]. A policy can be obtained by selecting greedy actions [SB98]. Hence, critic-only methods usually discretize the continuous action space, in which the optimization can be carried

out. Q-learning [WD92], Temporal difference (TD) and SARSA [SB98] are examples of critic only methods where they use a state action-value function and no other explicit function for the policy. For continuous state-action space, this function is approximated and used. The Q value for the state-action pair is denoted as $Q(s_t, a_t)$ which is the measure of long term value of performing certain action a_t in the state s_t. Q function represents the best possible actions taken by the agent in the state till the end of the learning. Therefore $Q(s, a)$ can be written as $E_\pi[R_t|a_t = a, s_t = s]$, where R is the cumulative rewards or otherwise called action's return. Therefore, if A is the set of all possible actions, then the chosen action at time t is given by the action a_t^* that maximizes $Q(s_t, a_t)$. In other words, we have:

$$a_t^* = argmax_{a \in A} Q(s_t, a_t) \tag{3.1}$$

The predicted action is combined with an exploratory method like ϵ-greedy policy to improve long-term learning [Agg18, p. 383]. This approach is used in our project for multi-robot collision control which is explained in detail in Section 3.4.

Finally, in the actor-critic methods, advantages of both actor-only and critic-only methods are combined where the critic evaluates the quality of the policy described by the actor. Based on the rewards received, the critic approximates and updates the value function. The actor's policy parameters are then updated using the state value function for best control action.

3.2 Environment

In reinforcement learning, the task which needs to be solved by an agent is described as an environment. An environment interacts with the agent by returning its state and reward. As mentioned before, in the video game example, the environment can be considered as the entire set-up of the game where changes are made to the state observation by bringing in an action. It is responsible for the calculation of the reward. In many cases, systems are required to exercise a deep understanding of the situation and analyse the different choices that they have to return an accurate reward or penalty. In a nutshell, the environment contains the entire simulation of the problem to be addressed. Therefore, the environment can be considered as the heart of reinforcement learning. In order to train the agent, one of the biggest challenges of RL is to have a working environment. Therefore, for an agent to learn a game or complete any other task, a simulator is to be built or programmed. For a robot agent, the actual working environment can be used as a simulator but it has its limits. Since the agent learns by its failures, it increases component damages and hence the total learning cost. For many video games, the OpenAI Gym toolkit provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so one can train agents, compare them, or develop new RL algorithms. However, for a robot agent, a virtual simulator of the robot

is to be created that replicates the actual functioning of the agent and must account for the allocation of rewards based on the actions. Hence, a collision control environment is created to simulate the job sequence of a multi-robot system.

3.2.1 Collision Control Environment

In this thesis, a replica of state observations is used to create a collision control environment for a small scale regime. The environment is created based on two important parameters, namely, the minimum number of jobs per robot n_{min} and the maximum number of jobs per robot n_{max}. The number of jobs per robot for a two robots system is randomly drawn from discrete uniform distribution from the range $[n_{min}, n_{max}] \cap \mathbb{N}$. The total number of jobs for robot 1 is represented by n_1 and the total number of jobs for robot 2 is represented by n_2. Based on their respective jobs, the state observation matrix is generated which includes randomly generated clusters of collision points. The state matrix is further zero-padded so that the resulting shape of $n_{max} \times n_{max} \times 2$ is achieved. The state matrix can be considered as an image where the 1st channel represents the collision points and the 2nd channel represents the position of the robot. The creation of the state observation is further described in the Section 3.2.2.

Three actions are allowed for the multi-robot system: a step in the rightward direction which indicates completion of a job for robot 1, a step in the downward direction which indicates completion of a job for robot 2 and finally, a diagonal step in the down-right direction which indicates that both the robots 1 and 2 completed their respective jobs (1 job each). When the action towards the right direction is taken, only robot 1 executes its respective job while the other robot waits till robot 1 finished its job. Similarly, in case of a downward step, only the job of robot 2 is executed while robot 1 waits for the other robot to complete its job. Finally, the ideal scenario is a step in the straight down-right direction where both the robots execute their respective jobs. During this process, if an action leads to a collision, the episode is terminated and is reset with a new state observation based on the new randomly chosen values n_1 and n_2. The environment can be reset to a new random configuration of state observation at any given point. However, the environment is usually reset when encountered with a collision point or on the completion of the assigned total task. These features of the environment are exclusively handled by the learning algorithm. Collision is determined based on the position of the job path of the multi-robot system in channel 2 and the collision matrix of channel 1. If the position of the job path is a collision point in channel 1, it is considered a collision and hence negative reward is awarded. The penalty is given to the system every time a wrong action is taken. The goal of the system is to find an optimal path such that both the robots complete their respective total number of jobs n_1 and n_2. Therefore a high reward is awarded when the job path reaches the final position. In our case, the final position is when the job path position reaches the bottom right end of the state observation matrix. The complete

formulation of the reward function is described in detail in Section 3.2.4.

Finally, a trajectory is the entirety of all state-action-reward-next state tuples of one episode, where the collision matrix representation is the state, the robot motion is the action, and the resulting outcomes of the motion are the reward. In other words, the complete simulation can be called a trajectory. The simulation for each action is provided by the collision control environment.

3.2.2 State Observation

As mentioned before, the state is the observation of the current world or the environment at a particular time. Initially, the state matrix is created as the replica of the real collision matrix in a small scale regime. The state matrix is generated as a 3-dimensional array in which the 1st dimension represents the total number of jobs for robot 1 n_1 and the 2nd dimension represents the total number of jobs for robot 2 n_2. The total number of jobs n_1 and n_2 are randomly drawn from the interval $[n_{min}, n_{max}] \cap \mathbb{N}$. A random configuration is selected so that the algorithm is scalable and robust. At inference, it can be applied for any number of jobs within the interval. The 3rd dimension however has a fixed index of 2 making the resulting shape $n_{max} \times n_{max} \times 2$. The state matrix could be considered as an image with two channels, where channel 2 represents the actual position of the job path and channel 1 represents the collision points. The whole state matrix could have been generated as a 2-dimensional matrix having only one channel which represents both the job position and the collision matrix. For example, each collision point could be represented as 1 and job position as 2. However, since the state observations are fed into the convolutional neural network (CNN), considering the position of the job path and the collision points as two different input channels results in better generalization and quicker learning.

The state observation matrix is a binary array. In the second slice, the current job position is indicated by 1 and the rest by 0. In the first slice, clusters of 7 collision points are generated in random locations of the array. The random locations are integers sampled from discrete uniform distribution from the range $[n_{max} - n_1 + 1, n_{max} - 2] \cap \mathbb{N}$ for robot 1 and $[n_{max} - n_2 + 1, n_{max} - 2] \cap \mathbb{N}$ for robot 2. The total number of collision clusters are decided by the formula $\lceil \frac{n_1+n_2}{5} \rceil$, where the operator $\lceil \rceil$ rounds the quantity to the next largest integer. The collision matrix is again a binary matrix where 1's represent the collision points and 0's represent the traceable path. When considered as a grid, each sequence of grid points represents a traceable path. For example, if the job sequence is at index $(0, 0)$, one movement to the right represents the execution of a job for robot 1 (while robot 2 waits), one movement in the downward direction represents the execution of a job for robot 2 (while robot 1 waits) and finally, one movement in the diagonally downward direction represents that both the robots execute their respective jobs simultaneously as

mentioned earlier. The collision points denote a collision between the two robots during the execution of their respective jobs. The collision matrix is generated in a way such that there always exists a traceable path so that both the robots completed their respective jobs. The plot of the collision matrix with $n_1 = n_2 = 36$ is shown in Figure 3.2. In this particular

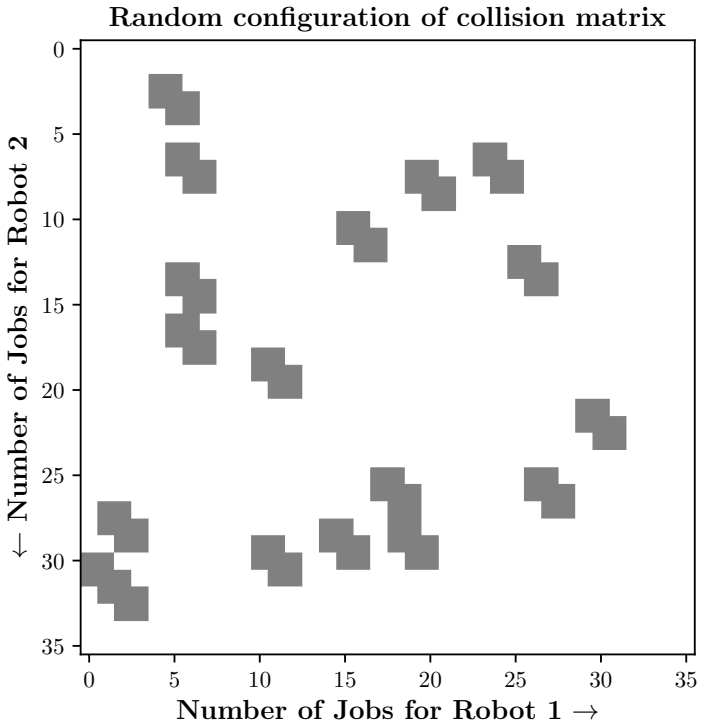


Figure 3.2: Visualisation of a random configuration of collision matrix for $n_1 = 36$ and $n_2 = 36$

example, we consider the $n_1 = 36$ and $n_2 = 36$. As seen in the figure, the x-axis represents the job sequence of robot 1 and the y-axis represents the job sequence of robot 2. Therefore, the task is considered complete when both the robots execute their respective jobs and reach the final position of the path. The collision points are obstructions to the job path while tracing the path. As we see, numerous job paths are possible to complete the job sequence, and the task persists to obtain the best path. It is clear from the figure that the fastest possible path would be around the diagonal where both the robots complete their respective jobs simultaneously.

Finally, both the slices of state observation are further zero-padded in the top and the left direction such that the resulting shape of $n_{max} \times n_{max} \times 2$ is achieved. Since the state matrix is inputted into the Neural network, a consistent shape must be maintained to make the model scalable up to a maximum number of robot jobs n_{max}. Therefore, the state observation is represented as a combination of both collision and position matrix. Figure 3.3 shows the overview of state observation. As seen in the figure, irrespective of total number of jobs for robot 1 and 2, the state observation is 0-padded to have a final

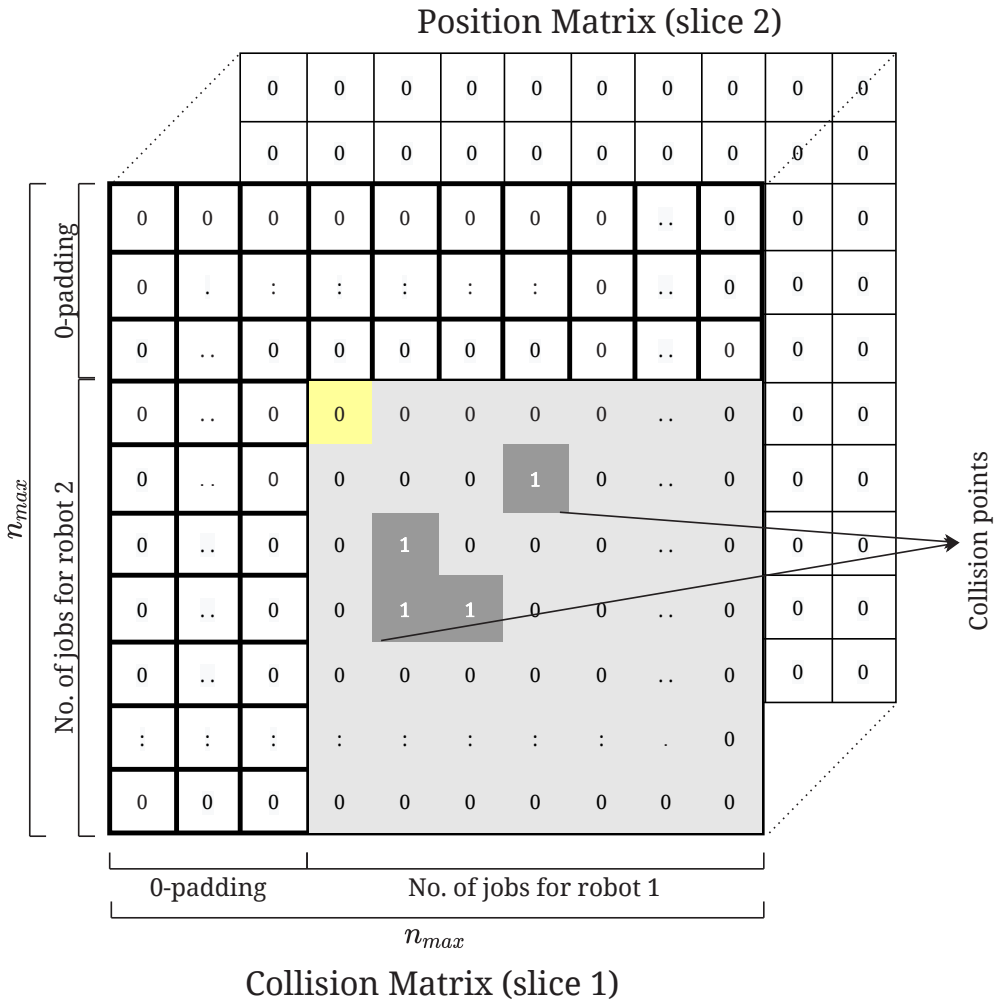


Figure 3.3: Overview of state observation

shape of $n_{max} \times n_{max} \times 2$. The area coloured in light grey represents the collision matrix for a specific number of jobs per robot: n_1, n_2 before zero-padding. As usual, dark grey points represent the collision point. Finally, the yellow point in the figure is the initial position of the job pointer in the position matrix. Figure 3.4 shows an example of such state observation for $n_1 = 7$, $n_2 = 7$ and $n_{max} = 8$ where collision matrix and position matrix are separately visualized for better understanding.

3.2.3 Step Action

In principle, the agent can move in three directions: towards the right where the job of robot 1 is executed, towards the downward direction where the job of robot 2 is executed and finally the diagonally downwards directions where both the robot jobs are executed simultaneously. To measure the quality of the actions, a well-defined reward function is to be formulated. Each action and its consequences are examined based on the changes it

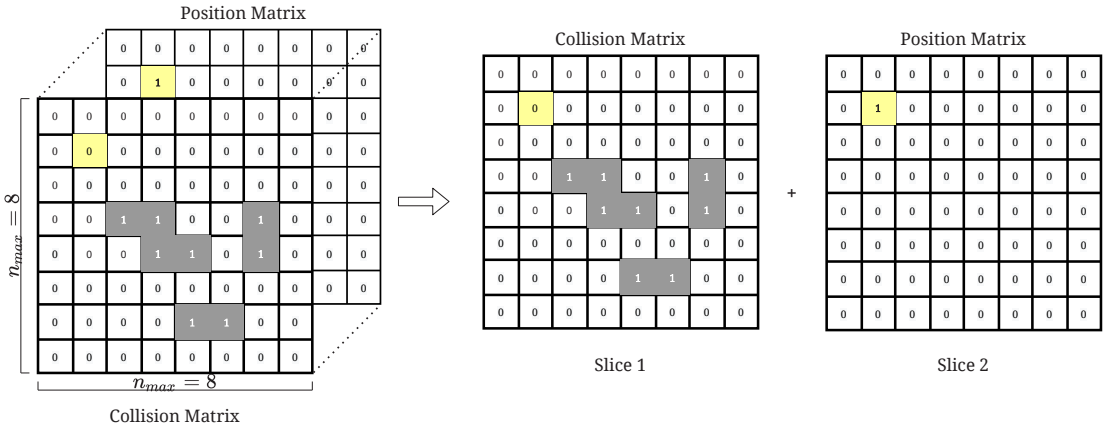


Figure 3.4: An example of state observation for $n_1 = 7$, $n_2 = 7$ and $n_{max} = 8$

brings to the environment. The formulation of the reward function is explained below in detail.

3.2.4 Reward Function

A reward in reinforcement learning is the feedback to the system from the environment. When interacting with the environment, changes are brought in the state observations by performing actions. This reward signal can be positive for a good action and negative (then called a penalty) for a bad action. A negative reward penalizes the agent informing it about the consequence of a bad action, and a positive reward informs the agent to take more such actions to reach the goal. In goal-oriented problems, a very high reward is given to the system on completion of the task. The goal, in general, is to solve a given task with maximum rewards and minimal time. That is why many algorithms have a small negative reward for each action taken by the agent to minimize the total time taken to solve the problem.

In our task, the goal of the agent is to find an optimal path without collision. The optimal path would take the least possible time compared to any other possible job path. Therefore small negative rewards are given to the system every time an action is taken. We want to encourage the agent to move diagonally so that both the robots can execute their jobs simultaneously. Hence the negative reward of the diagonal step is less than the rightward/downward step. For every diagonally downward action, -0.01 reward is given since both the robots perform their respective jobs simultaneously. However, for the rightward action (execution of the job by robot 1) and downward action (execution of the job by robot 2), -0.02 reward is awarded respectively.

Since avoiding the collision is one of our most important goals, a very high penalty is given to the system when the job path encounters a collision point. Therefore, if an

action leads to a collision, a negative reward of -1 is given to the system, the episode is terminated and the environment is reset with a freshly initialized state observation. Since it is safer when the two robots maintain a certain distance from each other to avoid the chance of collision, a small negative reward of -0.1 is given to the system whenever the robots come closer. In other words, a small negative reward is given when the job path is in close vicinity to the collision points. Finally, on completion of all the respective jobs of the robots, a positive reward of $+1$ is awarded.

Credit Assessment Problem

The RL algorithms guide the agent to learn by rewards, and these rewards are usually sparse and delayed. For example, if the agent meets a collision point after 50 steps, it becomes difficult to say which of these actions were good and which were not. It cannot be known which of the actions led to the collision of the robots. When the agent gets a reward, it is hard for it to identify which actions must be credited or blamed. To tackle this problem, a common strategy is to evaluate an action based on the discounted sum of all the rewards that come after it. This is called the discount factor γ. This sum of discounted rewards is called the action's return. The discount factor determines how much the agent cares about rewards in the distant future relative to those in the immediate future. The total rewards at the end of the trajectory are computed as the discounted sum of rewards collected from every time step. For the finite-horizon trajectory (trajectory that terminates after a certain number of time steps due to failure or on reaching the goal), the action's return R_t is computed as shown in equation:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...\gamma^{T-1} r_{t+(T-1)} \tag{3.2}$$

$$= \sum_{k=0}^{T-1} \gamma^k r_{t+k} \tag{3.3}$$

where T is the length of the trajectory/episode. In our study, the length of the trajectory depends on collision points and the goal.

For example if the agent takes 4 steps: 2 diagonal steps followed by 2 steps in the rightward direction, then considering a discount factor γ of 0.9, the first action will have return of $-0.01 + \gamma \times (-0.01) + \gamma^2 \times (-0.02) + \gamma^3 \times (-0.02) = -0.019$. Discount factors can vary between 0 and 1. If the discount factor is close to 0, the future rewards won't count much compared to immediate rewards and if the discount factor is close to 1, then the rewards far in the future will count as much as the immediate results. Most often, a discount factor between 0.9 to 0.99 is used. With a discount factor of 0.95, rewards 13 steps into the future count roughly for half as much as immediate rewards, while with a discount factor of 0.99, rewards 69 steps into the future count for half as much as immediate rewards [Ger19, p. 619].

3.3 Temporal Difference Learning

Reinforcement learning with discrete actions can be modelled as Markov decision processes (MDPs). However, when a reward is received, a major problem that arises is how to distribute the rewards among the decisions that led to it. This is known as the temporal credit assessment problem. Temporal Difference (TD) [Sut88] learning is a model-free RL method to solve these problems. TD is an incremental learning procedure that uses past experiences with a partly known system to predict future actions. The step in a sequence is evaluated and adjusted based on their immediate or near immediate successor rather than their final outcome. In TD learning, the agent has partial knowledge of the MDP, i.e, in the beginning, we assume that the agent knows only a few possible states and actions. The agent uses an exploration policy (example ϵ-greedy) to explore the MDP as it progresses. TD methods assign credit by the means of the difference between temporally successive predictions and the learning occurs when there is a change in prediction over time. In TD learning, reward estimates at successive times are compared. The algorithm updates the estimates of state values based on transitions and rewards that are observed. TD methods require peak computation but less memory and produce accurate outcomes.

We consider (s_t, a_t, r_t, s_{t+1}) as state-action-reward-next state experience tuple summarizing the transition in the environment at time t, where the state s_t changes to s_{t+1} after a transition due to an action a_t, and r_t is the instantaneous reward it receives. The value V of a policy is learned using the TD algorithm

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + V(s_{t+1}) - V(s_t)], \tag{3.4}$$

where α is the learning rate and $r_t + V(s_{t+1})$ is the TD target. $\alpha[r_t + V(s_{t+1}) - V(s_t)] = \delta(s_t, r_t, s_{t+1})$ is called the TD error. If the learning rate α is adjusted properly and the policy is held fixed, TD is guaranteed to converge to the optimal value function.

3.4 Q-learning

Q-learning is an off-policy algorithm in which approximation to optimal action takes place independently of the evaluation policy by using the path with the greatest action value to calculate the one periodic difference. Q learning is the most widely used algorithm with good converges control. If we had the observed value of the Q-function, we could easily set up a loss function in terms of $Q(s,a) - \hat{Q}(s,a)$ to learn after each action, where $\hat{Q}$ is the target. The problem here is that the Q-function represents the maximum discounted reward over all the future actions and it is impossible to observe it at the current time. However, we do not really need the observed Q-values to set up a loss function as long as an improved estimate of the Q-values can be calculated by using partial knowledge about the future. A surrogate observed model can be created using this improved

estimate. This observed value is defined by the Bellman equation [Bel57], which is a dynamic programming relationship satisfied by the Q-function and the partial knowledge of the rewards. Assuming that the best action is taken initially, we get the optimal policy $\pi = argmaxQ(s,a)$, which chooses an action having the maximum Q value for the current state. Q learning works by watching the agent explore the environment and gradually improve its Q-value estimates. Once accurate Q value estimates are achieved, the optimal policy is then obtained by choosing the action that has the highest Q-Value. Based on the Bellman equation, we set the ground truth by looking ahead one step and predicting the next state s_{t+1}:

$$Q(s_t, a_t) \leftarrow r_t + \gamma \operatorname*{argmax}_{a \in A} \hat{Q}(s_{t+1}, a). \tag{3.5}$$

The correctness of this equation comes from the fact that the Q-function is designed to maximize the discounted future rewards. We look for an action 1 step ahead to create an improved estimate $\hat{Q}(s_{t+1}, a)$. This term must be set to 0 when the episode terminates.

The Q-value estimate is essentially similar to TD learning. It is important to mention the update rule in Q-learning. The new Q-value is the sum of the old Q-value and TD-error. Q-values can be estimated online by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)). \tag{3.6}$$

where α is the learning rate and γ is the discount factor. For each state-action pair, the agent keeps track of the running average of immediate rewards plus the sum of discounted rewards it expects to achieve. For doing this, the maximum Q-value estimate for the next state is considered assuming that the target would act optimally.

If every action in each state is executed an infinite number of times and α is decayed appropriately, on an infinite run, the Q-values will converge to the optimal values with probability 1 to Q^* [WD92], independent of how the agent behaves while the data are being collected. Hence, it is an effective model-free algorithm with delayed rewards that are widely used. However, it sometimes does not generalize well over large state-action space. It may also converge quite slowly to a good policy. When it almost converges, the greedy action with the highest Q-value is taken. However, it is difficult to make an exploitation-exploration trade-off during learning.

3.4.1 Exploration vs Exploitation

When we train an agent, the challenges of exploration and exploitation immediately arise. To maximise its rewards, the agent repeats the best actions based on its learnt knowledge. This is called exploitation, where the agent takes advantage of the learnt information and takes actions that lead to favourable long term rewards. However, in order to find these optimal actions, the agent has to sample from a set of actions and try out different actions

which it has not previously experienced. For this, random actions are taken to explore the environment, and this randomness in the output is called exploration. Therefore, exploration is when an agent has to sample different actions from a set of actions to obtain better rewards. The key challenge that arises is to balance the trade-off between the two. All learning algorithms aim at solving the exploration-exploitation dilemma, meaning achieving the best performance at a minimum learning cost. This is decisive, as too much randomness leads to an increase in learning cost, and less randomness reduces the overall performance of the agent due to overfitting. Thus finding a right balance between the two becomes crucial.

ϵ-greedy

Q-learning only functions when the exploration policy explores the MDP thoroughly. A random policy can be used to visit every state and transition. However, such an approach would take a long time and would be computationally expensive. Therefore, ϵ-greedy is usually used to explore different states. The goal of the greedy algorithm is to use the best strategy as soon as possible without wasting a significant number of trials. At each step, it acts randomly with the probability ϵ or greedily with the probability $1 - \epsilon$ (i.e. choosing the action corresponding to the highest Q-value) [Ger19, p. 632]. The advantage of this approach is that one is assured to not be trapped in a bad strategy forever. Furthermore, as the exploration starts in the early stage, one is likely to use the best strategy for a large fraction of time. In other words, it will spend more time exploring the interesting parts of the environment, as the estimates of Q-values improve, while still exploring unknown regions of the MDP. The most common practice is to use annealing, in which, a high value of ϵ is used in the beginning and is then gradually reduced to as low as 0.01 as the learning progresses. Hence, in the beginning, the agent takes many random actions to explore the environment. As the learning progresses, the ϵ value decreases and the agent exploits the learnt behaviour and takes actions based on its knowledge. The value of ϵ is required to be reasonably small towards the end to gain significant advantages from the exploitation portion of the approach.

3.5 Deep Q learning

The major problem of Q-learning is that it does not scale well to large MDPs with many states and actions, and it gets impossible to keep track of an estimate for every single Q-Value. This is the case for our problem since the number of possible states and their associated actions are large because of the size and randomness of the collision matrix. The solution for this is to use a deep neural network for such complex problems to estimate the Q-values of the respective states. This section explains in detail the functioning of the Deep Q-Network (DQN) as described by [Agg18, p. 384]. We assume that the state representation s_t is denoted as X_t. Therefore, the neural network inputs the state X_t and

outputs the Q-values $Q(s_t, a)$ for all the possible actions a. Let the set A be the set of all the possible actions a. We consider that the network is parameterized by weights W and has $|A|$ outputs containing Q-values that correspond to various actions in A. Therefore the network computes the function $F(X_t, W, a)$ which is the learnt estimate of $Q(s_t, a)$:

$$F(X_t, W, a) = \hat{Q}(s_t, a) \tag{3.7}$$

$\hat{Q}$ indicates the predicted value of the network. Therefore, learning the weights W is the key to decide the different possible actions. In our case, the algorithm passes the state observation of shape $n_{max} \times n_{max} \times 2$ as an input to a Conventional Neural Network (CNN) to output the Q-value estimates corresponding to the 3 actions. The methodology is described in detail in Section 3.7.2. This network used to estimate the Q-values are called Q-network. Figure 3.5 shows the general architecture of a Q-network which takes state observation as an input to output Q-values corresponding to the actions.

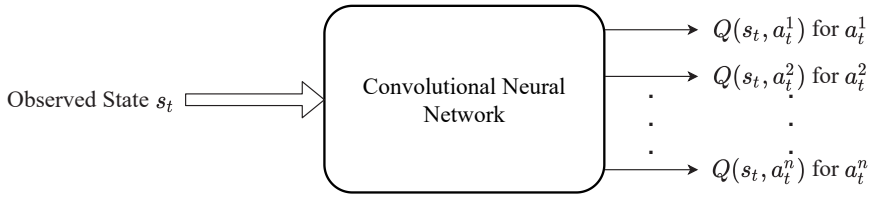


Figure 3.5: General architecture of Q-network which takes state observation as an input to output Q-values corresponding to the respective actions.

We can now reformulate the equation (3.5) in terms of a neural network as:

$$F(X_t, W, a_t) = r_t + \gamma \max_{a \in A} F(X_{t+1}, W, a) \tag{3.8}$$

To compute this observed value at time-step t, one has to wait to observe the state X_{t+1} and reward r_t by forming the action a_t. With this, we can express it as a neural network loss L_t between the surrogate observed value and the predicted value at the time-step t:

$$L_t = [r_t + \gamma \max_{a \in A} F(X_{t+1}, W, a) - F(X_t, W, a_t)]^2 \tag{3.9}$$

The above equation is similar to that of equation (3.6) but is written in terms of a loss. Although squared loss is used in this equation, any loss can be used based on the problem. Now we can update the weights W by backpropagating the loss function. The target value estimate $r_t + \gamma \max_{a \in A} F(X_{t+1}, W, a)$ using inputs at $(t+1)$ are considered as constant ground truths by the backpropagation algorithm. It can also be called the target Q value. Therefore, while computing the gradient, this term is treated as a constant though they were obtained from a parameterized neural network with input X_{t+1}. We consider the prediction at $t+1$ as an improved estimate of this ground truth. Hence, the weight update using the backpropagation algorithm will compute the following:

$$W \leftarrow W + \alpha\{[r_t + \gamma \max_{a \in A} F(X_{t+1}, W, a)] - F(X_t, W, a_t)\} \frac{\partial F(X_t, W, a_t)}{\partial W} \tag{3.10}$$

At the beginning of the learning process, the Q-value estimate of the neural network is random because of the random initialization of the weights. However, the estimate gradually improves with time as the weights are constantly updated to maximize the rewards.

Therefore, if the action a_t, reward r_t is observed at any given time-step t, the following process is used to update the weights W:

Algorithm 1: Deep Q-learning Outline

1. Perform forward pass through the network with input X_{t+1} to compute $\hat{Q}_{t+1} = \max_{a \in A} F(X_{t+1}, W, a)$. Set value to 0 in case of termination. $r_t + \gamma \max_{a \in A} F(X_{t+1}, W, a)$ is the surrogate for the target value at t and is considered as an observed value.
2. Perform forward pass through the network with input X_t to compute $F(X_t, W, a_t)$.
3. Set up loss function $L_t = [r_t + \gamma \max_{a \in A} F(X_{t+1}, W, a) - F(X_t, W, a_t)]^2$.
4. Backpropagate the loss to update the weights W. Consider surrogate value as constant.

Since the value of the present action is used to update the weight and select the next action, training and prediction are performed simultaneously in deep Q-learning. The optimality prediction is coupled with a exploration policy such as ϵ-greedy as explained in Section 3.4.1. Instead of training the Deep Q-Network (DQN) agent purely based on its latest experience, a better strategy is to store a set of experiences in a repay buffer or replay memory and sample random training batch from it at each training iteration. This intern helps in reducing the correlations between the experiences in training batch and remarkably helps in training. Before starting the training iterations, the replay buffer is filled with random experiences. If it is not filed sufficiently, there will not be enough diversity in the replay buffer.

3.6 Variants of DQN

In this section, we will look into different variants of Deep Q-learning to stabilize and speed-up the learning process.

3.6.1 Fixed Q Target

The basic deep learning algorithm uses the same network to make predictions and set its target as explained in the previous section. This might sometimes make the network unstable resulting in divergence or oscillation. To solve such a problem, 2 DQNs can be used instead of a single network [Mni+13]. The first network is called the online network which learns at each step and controls the agent to take actions. The other network is

called the target network which is used to only define the target or the ground truth. The target model is created as a clone of the online model. During learning, the weights of the online model are copied to the target model at regular intervals. Since the target model is updated less often compared to that of the online model, the Q-targets are more stable. This dampens the feedback loop making its effects less severe.

3.6.2 Double DQN (DDQN)

It was observed that the target network was sometimes prone to overestimate the Q-values. This is mostly the case when all the actions are equally good. In such a case, the target Q network must estimate identical Q-values, but since they are merely approximations, some values tend to be slightly higher than the rest, purely by chance. The target model will hence choose the largest Q-value overestimating the true Q-value. To solve this problem, DeepMind tweaked the DQN algorithm, increasing its performance and stabilizing the training [HGS15]. Instead of the target model, the online model was used to select the best actions for the next states. The target model was only used to estimate the Q-values for these best actions. This variant is called Double DQN.

3.6.3 Prioritized Experience Replay

In the generic algorithm, the experiences are sampled uniformly from the replay buffer. However, instead of sampling uniformly, sampling important experiences more frequently would lead to a better result. This approach is called importance sampling or prioritized experience sampling [Sch+16]. Experiences are considered of higher priority if they speed up the learning process. One approach to prioritize the experiences is based on the TD error associated with it. A higher TD error indicates a bad action and thus is not worth learning. Therefore, when recording the experiences, the ones with a low TD error is associated with a high priority so that it is sampled at least once. Every time it is sampled from the buffer, the TD error δ is computed and the priority of that experience is reset to $|\delta| + c$, where c is a small constant to ensure that every experience has a non-zero probability of being sampled. The probability of sampling an experience P with a certain priority p is proportional to p^{ζ}, where ζ is the probability that controls how greedy an importance sampling must be. $\zeta = 0$ gives an uniform sampling, whereas $\zeta = 1$ results in a high importance sampling. The optimal value of this hyperparameter is problem specific and depends on the respective task. However, since the samples are biased towards priority experiences, the bias must be compensated during training by down weighting the experiences according to their priority. If the experiences are not down-weighted, the model will overfit to important experiences. It is done so by defining each experience's training weight as $w = (nP)^{-\beta}$, where n is the total number of experiences in the replay buffer and β is the hyperparameter that controls the compensation for the importance sampling bias. Here, $\beta = 0$ would result in no compensation and $\beta = 1$ would result in high compensation [Ger19, p. 640]. Again, the optimal value of this hyperparameter is problem

specific.

3.6.4 Dueling - DQN

Dueling DQN is yet another algorithm that was presented by DeepMind [Wan+16]. The algorithm splits the Q-values into 2 different parts, the value function and the advantage function. The Q-values of the state-action pair can be represented as follows:

$$Q(s,a) = V(s) + A(s,a) \tag{3.11}$$

where the value function $V(s)$ tells about the rewards collected from the state s and the advantage function $A(s,a)$ tells the advantage of taking an action a over all other actions in state s. Moreover, the value of a state is equal to the Q-value of the best action a^* for that state, which implies $A(s,a^*) = 0$. In Dueling DQN, the model estimates both the value and the advantage of each action. It splits the last layer of the neural network into two parts to estimate the value function and the advantage function respectively. Since the best action should have an advantage of 0, the model subtracts the maximum predicted advantage from all predicted advantages. At the end it combines the two into a single output to estimate the Q-values:

$$Q(s,a) = V(s) + (A(s,a) - \max_{a \in A} A(s,a)) \tag{3.12}$$

The key motivation behind this architecture is that, for some problems, it is unnecessary to know the value of each action at every timestep. By explicitly separating two estimators, the dueling algorithm can learn which states are valuable (or not) without having to learn the effect of each action for each state. The rest of the algorithm is the same as before. This algorithm can be combined with different other algorithms like prioritized experience replay and Double DQN (Dueling-DDQN) to get improved results. For example, DeepMind combined 6 different architectures into an agent called rainbow, which outperformed the state of the art [Hes+17].

3.7 Implementation of Deep Q-Learning for robot scheduling

Implementation of the deep Q-learning model is carried out using the Tf-agents library provided by Google. It is an open-source software based on Tensorflow used for reinforcement learning. It has the feature to build custom environments along with different wrappers and implements many RL algorithms like DQN, DDQN and many more, as well as various components of reinforcement learning such as replay buffers, metrics and step drivers. Since it is based on Tensorflow, the neural networks of Tf-agents can be easily constructed similar to TensorFlow. The library is fast and customizable and hence can be easily used to solve various problems. We use Tf-agents to train the agent to find an

efficient path using a custom collision control environment with the Double DQN algorithm. After preliminary trials, the DDQN variant was prefered due to its performance. Here, we aim to find an optimal path in a small scale regime for $n_{min} = 30$ and $n_{max} = 40$.

3.7.1 TF-agent environment

The first task in reinforcement learning is to have a well-defined environment with state observations, reward functions and step actions. Here, state observation returns the current state, step action takes a particular action and the reward function calculates the reward based on the step taken. Each episode is initiated with a random configuration of the state observation. The total number of jobs for robots 1 and 2 are randomly chosen from the considered interval $[n_{min} = 30, n_{max} = 40] \cap \mathbb{N}$. The clusters of collision points are generated at random locations of the state observations. The random numbers are drawn from a discrete uniform distribution between the given range. The number of clusters of collision points to be considered is given by the formula: $\lceil \frac{n_1+n_2}{5} \rceil$, where the operator $\lceil \rceil$ rounds the quantity to the next largest integer. The state observations and the reward functions are formulated exactly as described in section 3.2.1.

The environment is further wrapped with a TimeLimit wrapper which terminates the episode if it runs longer than the maximum number of steps provided. We considered the maximum number of steps for the timelimit wrapper to be 80. Since $n_{max} = 40$, the path must be found within 80 steps even in the worst-case scenario. The environment is further wrapped with a RunStas wrapper which stores the statistics of the learning which can be later used for metric calculations. Finally, the environment must be wrapped inside the TFPyEnvironment wrapper. This makes the environment usable within a TensorFlow graph. After creating a useable environment, DDQN agent and other essential components need to be created before training. In the upcoming sections, the training architecture of the built system is discussed.

3.7.2 Deep Q learning training Architecture

The training program is split into two parts that run in parallel, namely, the collection of the trajectories and the training. Deep Q-learning training architecture can be seen in the Figure 3.6. Initially, a collect driver explores the environment using a collect policy to chose actions for a certain number of steps. These collected trajectories are saved into the replay buffer (replay memory) by the observer. The agent pulls small batches of these trajectories from the replay buffer and trains the neural network. This process helps reduce the correlations between the experiences in a training batch, which tremendously helps in training. The trajectories created by the actions taken by the network are again saved back into the replay buffer. In short, the collect policy explores the environment and collects trajectories and the network learns and updates the collect policy. Random

actions are taken to explore the environment before exploiting the predictions of the network. The amount of exploration and exploitation is governed by the ϵ - greedy method, as explained in Section 3.4.1. The common practice of annealing is used in which a high value of epsilon is used at the beginning which is then gradually reduced to a very low value as the learning progresses. Therefore, in the initial stages, the agent explores the environment by taking random action. As the learning progresses, fewer random actions are taken and the agent starts to exploit the learnt behaviour of the network. Towards the end, the agent only exploits the learnt behaviour and takes actions based on the prediction of the network.

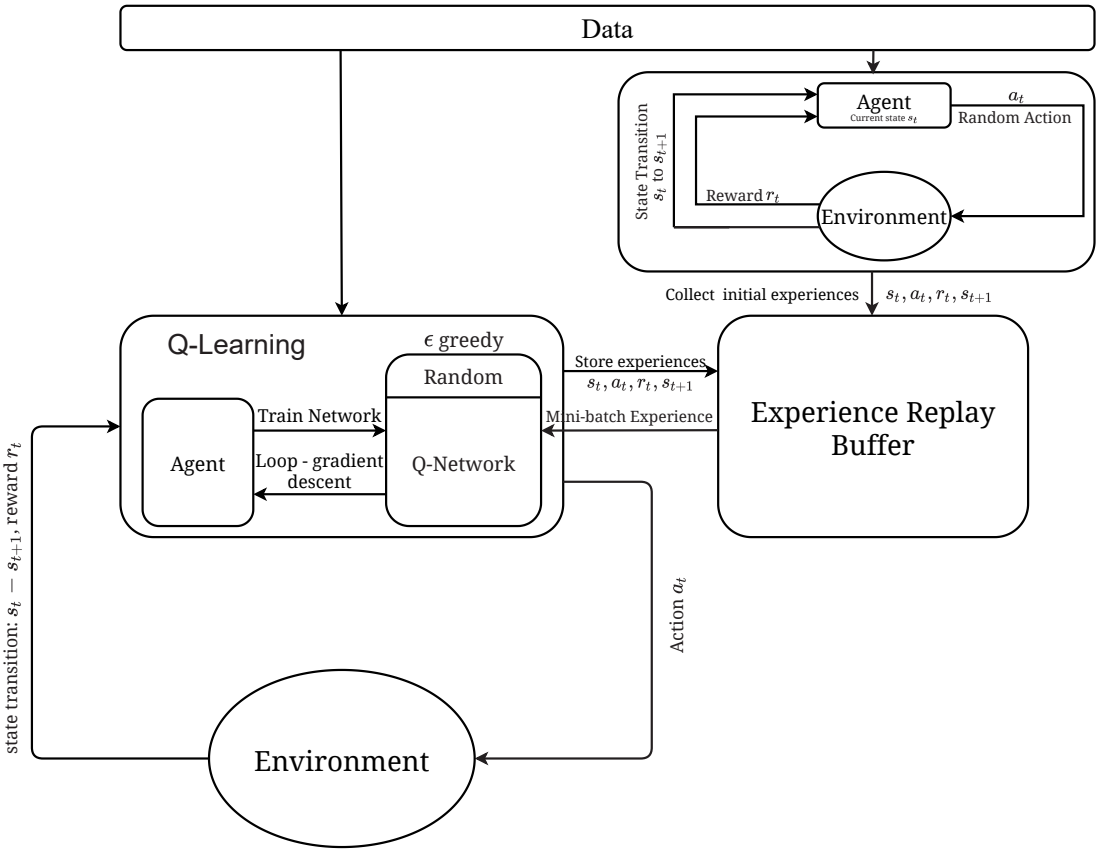


Figure 3.6: Deep Q Learning training architecture.

Deep Q-learning training depends on several components and hyper-parameters. With an optimal choice of these, the agent learns the optimal behaviour which can be then used at inference. As discussed in the architecture, we now create all the required components: the DQN network, DDQN agent, the replay buffer, the collect driver and finally the dataset to train the network. After creating all the components, we populate the replay buffer with some initial trajectories and then continue with the main training loop.

Deep Q Network and Agent

The Q-network takes the state observation as an input and outputs the Q-value for every action. The Q-network for the small scale regime of input size $n_{max} \times n_{max} \times 2 = 40 \times 40 \times 2$ is explained in this section. It starts with a prepossessing layer which casts the observations to a 32-bit float and normalizes it (in our case, the values are already between 0 and 1). The observations are initially integer. It is not cast to 32-bit float before to save the RAM space in the replay buffer. After the prepossessing layer, the Q-network contains 2 convolutional layers, the first having 32 filters of size 4×4 with a stride of 2, followed by a layer having 64 filters of size 2×2 with a stride of 1. Next, the flatten output of the convolutions layers is passed through a fully connected dense layer with 256 units. Finally, it applies a dense output layer with 3 neurons, each to represent the Q-values of the respective actions. All the convolutional and the dense layers except the last output layer uses the ReLU activation function. No activation is used for the output layer. The architecture of the neural network can be seen in the Figure 3.7. In this section, the working of the convolutional neural networks is not explained. For further details refer [GBC16] and [Ger19].

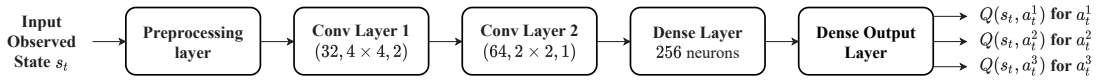


Figure 3.7: Architecture of the neural network used to train the agent.

Now that we already have the network, we need to now build the DDQN agent. The DDQN agent provided by TFagents considers a lot of hyperparameters. We first create a variable to count the number of training steps. We then build the optimizer based on the problem. All the different optimisers provided by tensorflow or even custom optimisers can be used for the problem. In our thesis, we considered RMSprop optimiser with a initial learning rate $= 1 \times 10^{-3}$, decay $= 0.90$, momentum $= 0$ and a small value of 0.000001 to avoid vanishing derivative. We also create a PolynomialDecay object that computes the ϵ value for ϵ-greedy collect policy, given the current training step. The value decays from 1.0 to 0.01 in 250000 steps. This means that, after 250000 steps, the agent rarely takes a random action and exploits the network. We then build the DDQN agent which takes the following arguments: time step, action spec, the Q network, the optimiser, number of training steps between target model update, the loss function, the discount factor, the train step counter and finally a function that returns the ϵ value. Various models were trained with different combinations of hyperparameters. However, the best results were obtained when using the following parameters: target update period $=$ 1000 training steps, loss function $=$ Huber loss without reduction and discount factor $\gamma = 0.98$. We want the loss function to return an error per instance and not the mean. That is the reason we set the reduction to none. Discount factor close to 1 is used as we want the future rewards to count as much

as the immediate results at least for 30 steps. Since we only have 3 movements possible without a possibility to move backwards, we need the agent to take optimal actions from the beginning so that the agent reaches the goal with maximum rewards without getting stuck between the collision points. Lastly, we need to initialize the agent before using it for training.

Metric Calculation

Unlike supervised learning, the quality of learning cannot be decided based on the loss function. The agent must be evaluated repeatedly after certain iterations to check the performance of the agent until then. TF-agents provides implementations of several RL metrics that can be directly used to keep track of the number of steps, episodes, average return, average episode length etc. Additionally, a custom metric evaluation function can be written which can be repeatedly called after a certain number of iterations to see the progress. In this thesis, a custom metric function was created which calculates the average reward, max reward and accuracy. The function is repeatedly called after every 1000 iteration. The agent evaluates its policy on observations from the environment for a certain number of episodes, say 100 episodes, and returns the considered metrics to log the progress. Here, accuracy refers to the job completion accuracy in percentage, i.e. the number of times the agent successfully reached the goal without encountering the collision points for every 100 episode. The average reward is the average cumulative reward for 100 episodes. Finally, as the name suggests, maximum reward represents the maximum episode reward received among the 100 evaluated episodes.

Replay Buffer

For saving the experiences, we need to create a replay buffer of sufficient size. Tf-agents provides a high-performance implementation of the replay buffer with uniform sampling which takes the following arguments: data specification, batch size and the max length. It saves the specification of the data in the replay buffer. Therefore the agent knows how the data will look like. Here, the batch size specifies the number of trajectories that will be added into the replay buffer at each step. In our case, the value is just one as the driver just executes one action per step and collects one trajectory. If we considered a batched environment (returns batched observations), then the driver would have to save a batch of trajectories at each step. The maximum length argument specifies the maximum size of the replay buffer. In our thesis, we created a large replay buffer that can store up to 100000 trajectories.

Additionally, we also need to create an observer that writes the trajectories into the replay buffer. An observer is a function that takes a trajectory as an argument and saves it in the replay buffer.

Collect Driver and Dataset creation

A driver explores the environment using the given policy, collects experiences, sends them to the observer which saves them in the replay buffer. The driver forwards the current time step to the collect-policy, which chooses an action based on the current time step and returns an action step object. This action is then passed into the environment by the driver, which returns the next time step. Lastly, it creates a trajectory object to represent the transition and sends it to the observer. In the case of a batched environment, all the above operations are carried out in batches. Tf-agents mainly provides two drivers: Dynamic Step Driver which collects experiences for the given number of steps and Dynamic Episode Driver which collects experiences for a given number of episodes. In this thesis, a dynamic step driver was used to collect experiences for every step. The step driver takes the following arguments: the environment, agents collect-policy, list of observers, and finally the number of steps (1 in our case). The driver is run in every iteration to collect experience.

As discussed before, it is a good practice to fill the replay buffer with certain initial experiences using a purely random policy. To implement this, we use the random policy class to create another driver which runs for certain initial steps. In our thesis, the step driver with random policy is run for 5000 steps, in the beginning, to collect initial experience before training.

The final step before training is to create a data set sampled from the replay buffer. To do so, it is important to understand how the trajectories are saved. Each trajectory is a concise illustration of a chain of consecutive time steps and action steps, designed to avoid redundancy. Transition n consists of time step n, action step n, and time step $n+1$, while transition $n+1$ consists of time step $n+1$, action step $n+1$, and time step $n+2$. If we save them directly into the replay buffer, the $n+1^{th}$ time step will be duplicated. To avoid this redundancy, the n^{th} trajectory step includes only the type and observation from time step n (not its reward and discount), and it does not contain the observation from time step $n+1$. However, it does contain a copy of the next time step's type which is the only duplication [Ger19, p. 659].

For training our main loop, we use the Dataset method provided by TFagent. In this way, we benefit from the power of Data API in terms of parallelization and prefetching. In our thesis, a batch size of 128 trajectories is sampled at each training step, each having 2 steps (1 full transition including next step's observation). The dataset processes 3 elements in parallel and prefetches 3 batches. Now that all the components are created, the agent can be trained to solve the collision control system.

3.7.3 Training

Before jumping into the final training, all the hyperparameters used are summarized below in Table 3.1. Several different combinations of parameters were tried and the best-known set of parameters are listed here.

Hyperparameters	**Value**	**Hyperparameters**	**Value**
Iterations	1000000	Loss Function	Huber
Replay buffer size	100000	Optimizer	RMSProp
Initial collect steps	5000	Learnrate, Decay	1×10^{-3}, 0.90
Target update period	1000	Discount Factor	0.98
Evaluation interval	1000	ϵ-greedy decay steps	250000
Evaluation episodes	100	Batch size for dataset	128

Table 3.1: List of hyperparameters used to train the agent.

In the training, the agent first calls the collect-policy for its initial state. Since the policy is stateless, it returns an empty tuple. Next, we iterate over the dataset and run the training loop. At each iteration, the collect driver calls the run method by passing the current time step and the current policy state. It runs the collect policy and collects experience for the step and broadcasts it to the replay buffer. Next, we sample a batch of trajectories from the dataset and pass it to the agent to train it. After every 1000 iteration, we log all the metrics that are calculated by evaluating the policy (or provided directly by Tf-agents). This training iteration needs to be finally run for a certain number of iteration. In this thesis, the agent was trained for 1000000 iterations and it could be seen that the agent gradually learnt the system by maximizing its rewards over time. The iterations are computationally expensive and require a lot of computational effort to learn the system. Moreover, different random seeds result in different results and therefore, the algorithm needs to be run several times with different random seeds to achieve an excellent result. However, once the training is complete, the agent will efficiently find the path from the source to the goal. Also, at inference, we just handle the neural network and its weights and hence it is computationally inexpensive compared to training. Unlike supervised learning, the model weights returned at the end of the learning need not be always the best possible results. The results of Deep Q learning keep fluctuating and therefore the policy with the best accuracy and best average reward must be saved for evaluation. This is carried out by comparing the metrics after every 1000 iteration. The policy with the best metrics (highest accuracy and max return) is saved and can be used at inference. Also, the loss is not a good metric to measure the quality of learning which can be seen in Chapter 5.

The results of the Deep Q learning are thereby compared with different shortest path algorithms which are based on graph theory. The next chapter brushes through the graph

theory and explains the working of different shortest path algorithms. There, we attempt to represent the state observation as a graph and try to search the shortest path using various graph-based algorithms.

4 Methodology 2 - Graph based Approaches

In the previous chapter, we represented the robot scheduling task as a collision control environment in order to schedule the robots efficiently by using reinforcement learning. In this chapter, we attempt to solve that problem by representing the multi-robot system as a graph. We then exploit different graph-based shortest path algorithms for the efficient scheduling of the robots. We further discuss a few speed-up techniques to solve the task in the least time possible.

4.1 Graphs and paths

A graph is a non-linear data structure consisting of nodes and edges. The nodes are also referred to as vertices and the edges as arcs or arrows that connect two nodes of a graph. A graph can be mathematically represented as $G = (V, E)$ which consists of a set V of vertices and a set $E \subset V \times V$ of edges. An edge is the ordered pair $(u, v) \in E$ of nodes $u, v \in V$, where u and v are source and destination nodes of an Edge set E for an uni-directed graph. The size of the graph is represented in terms of the number of vertices and edges of a graph and this number of vertices and uni-directed edges is represented by $|V|$ and $|E|$ respectively. A weighted graph $G(V, E, w)$ is a graph $G(V, E)$ with an edge distance/length/weight: $w \rightarrow \mathbb{R}^+$. In weighted graphs, non-negative weight $w(u, v)$ or $dist(u, v)$ is assigned to each edge (u, v).

Directed graphs are a category of graphs that don't presume symmetry or reciprocity in edges that connects the vertices. In a directed graph, if u and v are two vertices connected by an edge, then there necessarily does not exist an edge connected in the opposite direction (v, u). However, the undirected graph assumes reciprocity in the relationship between the pair of the vertices connected by an edge. In this case, if an edge (u, v) exists between two vertices, it guarantees that the edge (v, u) also exist [AMO93, p. 24]. Figure 4.1 gives an example of both directed and undirected weighted graphs. As we see in the Figures 4.1(a) and 4.1(b), the only difference between the two is the relationship between the pair of the vertices connected by an edge. All the examples considered in this chapter are not merely random, but represents the graph or portions of the graph used to represent our problem.

A path P in a graph G from vertices u to v is a finite sequence of vertices $\{u_1, u_2, u_3, ..., u_k\}$,

Figure 4.1: An example for directed and undirected graph.

such that $u = u_1$, $v = u_k$ and $(u_{i-1}, u_i) \in E$ for all $1 \leq i \leq k$. The hoplength $hop(P)$ is the total number of edges in the path P: $hop(P) = k$. For a weighted graph, the length or total weight $w(P)$ of path P is the sum of weights of all the consecutive edges of the path, i.e. $w(P) = \sum_{i=1}^{k}(u_{i-1}, u_i)$. P is the shortest path if there is no other path P' from u to v with $w(P') < w(P)$. A similar definition is possible in terms of hoplength for a unweighted graph. In the weighted graph, the distance $dist(u, v)$ of two vertices is the total weight/length of the shortest path $P = (u, .., v)$: $dist(u, v) = min\{w(P) : P = (u, .., v)\}$. If no path exists between the source u and target v, then the distance $dist(u, v) = \infty$. If all the vertices on a path P are pairwise distinct, then the path P is called simple. However a path $P = (u, ..., v)$ is cyclic, if the target $v = u$ and the $hop(P) > 0$ [Col19]. Since our problem deals with completion of the robot jobs and finding an efficient job path, we deal with simple graphs to find the shortest paths.

4.1.1 Representation of Graphs

A graph can be represented in two standards ways: a collection of adjacency lists or a adjacency matrix. Although both representations can be used interchangeably, adjacency list representation is usually preferred for sparse graphs as it provides a compact way to represent a sparse graph. A sparse graph is a graph in which the number of edges is close to the minimal number of edges. For a sparse graph $|E|$ is much less than $|V|^2$. However, a adjacency matrix representation is preferred for a dense graph where $|E|$ is close to $|V|^2$. A dense graph is opposite to a sparse graph and can be defined as a graph in which the number of edges is close to the maximal number of edges.

The adjacency list representation consists of a list or array 'Adj_list' containing $|V|$ lists, one for each vertex in V. For every $u \in V$, the adjacency list Adj_list consists of all the vertices adjacent to u. If there exists two edges (u_1, u_2) and (u_1, u_3), then $Adj\_list[u_1]$ contains all the connected vertices u_2 and u_3. For a directed graph, the total sum of lengths

of all the adjacency lists is $|E|$ and for an undirected graph, it is $2|E|$. The total amount of memory it requires is $\mathcal{O}(V + E)$. This representation can be easily extended to represent weighted graphs. Weights are represented alongside the adjacent vertices that the source u is connected to. That is, the weight $w(u, v)$ of the edge $(u, v)$is simple stored with vertex v in u's adjacency list. This representation is robust, memory efficient and can be used to represent a variety of sparse graphs. However, the potential disadvantage is that there is no quick way to determine the presence of the edge (u, v) in the graph, other than searching for the vertex v in the adjacency list $Adj\_list[u]$ [Cor+01, p. 528]. An example for adjacency list representation can be seen in Figure 4.3.

For the adjacency matrix representation [Cor+01, p. 529], if we assume that the vertices are numbered $1, 2, ..., |V|$ in some arbitrary manner, then the adjacency matrix consists of $|V| \times |V|$ matrix $A = (a_{ij})$ such that:

$$a_{ij} = \begin{cases} 1, & \text{if } (i, j) \in E, \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

An example for adjacency matrix representation can be seen in Figure 4.4. The adjacency matrix representation requires a memory of $\mathcal{O}(V^2)$ and does not depend on the number of edges in the graph. For an undirected graph, $A = A^T$ and therefore in some applications, one of the values above the diagonal are stored. The adjacency matrix representation can be easily adapted to weighted graphs. For a weighted graph G, the weight $w(u, v)$ is simply stored as the entry in row u and column v of the adjacency matrix. If an edge does not exists, it can be stored as 0 or ∞ representing no or infinity distance. In other words, all the 1's in the adjacency list can be replaced with the respective weights, i.e. for a adjacency matrix $A = (a_{ij})$, we have

$$a_{ij} = \begin{cases} w(i, j), & \text{if } (i, j) \in E, \\ \infty, & \text{otherwise} \end{cases} \tag{4.2}$$

Since it is memory intensive, adjacency list representation is usually preferred for reasonably small and sparse graphs.

4.1.2 Robot scheduling task as a Graph

The robot scheduling task can be easily represented as a graph and can thereafter be queried with various algorithms to find the shortest path from the source to the target. This shortest-path would then represent the optimal job scheduling path without encountering collision points. The problem can be represented as a graph with the number of nodes equal to the total number of job positions possible. So, if n_1 and n_2 are the total number of jobs for the robot 1 and 2 respectively, the generated graph must have $n_1 \times n_2$ nodes in total. In the robot scheduling task, as discussed earlier, each job position can be

moved in 3 directions at a given time: right, down and diagonally right. Therefore each vertex is connected to 3 other vertices representing right, down and diagonal movements, except when the job sequence is at n_1 or n_2. In such cases, only 1 directional movement is possible, i.e when robot 1 has already completed its set of jobs, the rest of the jobs are executed by robot 2 and vice versa. Since each edge represents the completion of a job, the edges are unidirectional. The generated graph is therefore a directed graph. The total number of edges can be calculated as $3 \times (n_1 - 1) \times (n_2 - 1) + (n_1 - 1) + (n_2 - 1)$. When the job position reaches the final node, the job sequence is deemed to be complete. Figure 4.2 shows the general representation of a graph for n_1 and n_2 number of jobs for

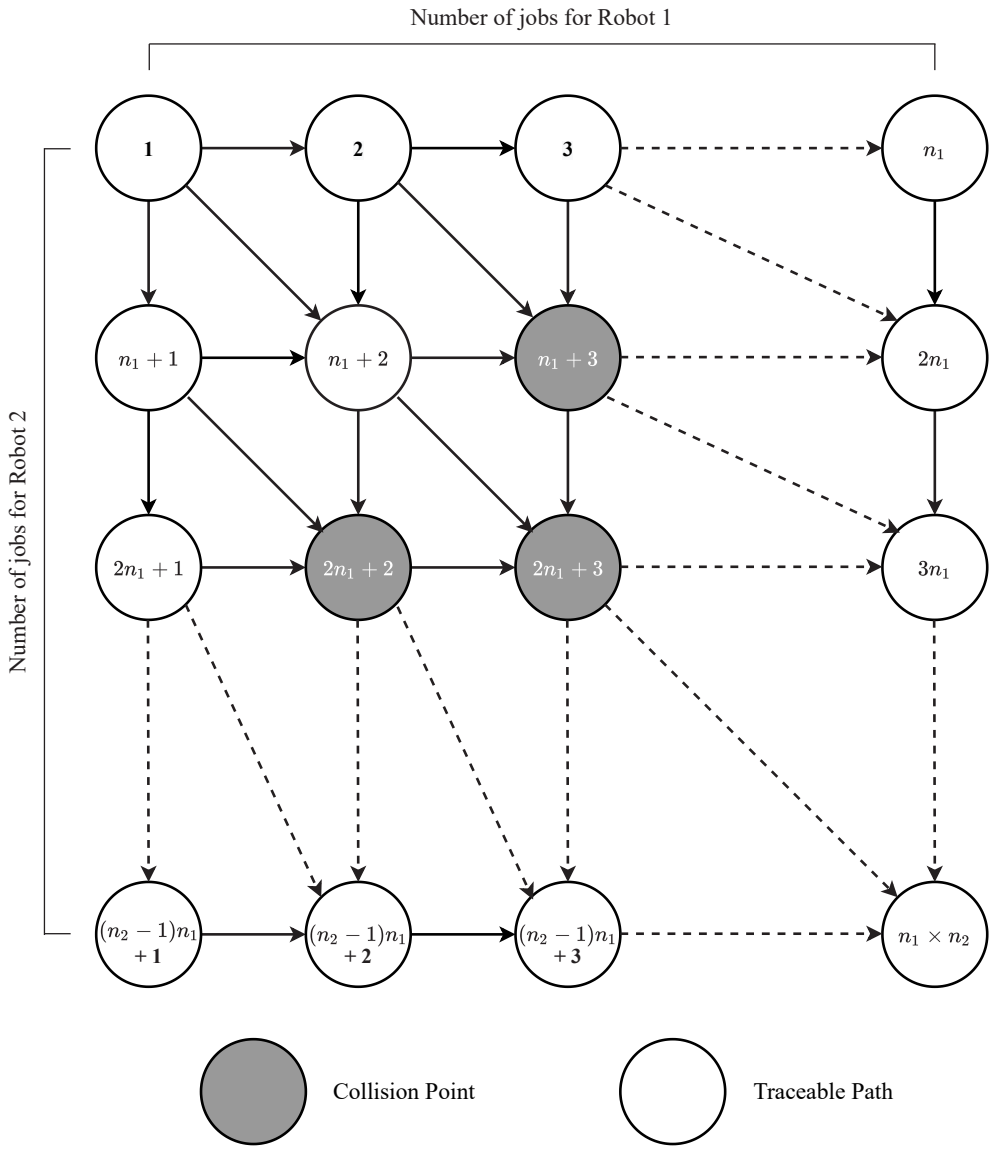


Figure 4.2: General representation of the scheduling task as a graph for a two robot system with n_1 and n_2 number of jobs.

robots 1 and 2 respectively. The collision matrix is converted to a graph where each entry is represented by vertices and the connection from one entry to the next is represented by edges. The collision points are coloured grey similar to the previous chapters.

Adjacency list representation is preferred over the matrix representation, as the generated graph is sparse. For example, in a task having total number of jobs $n_1 = 5$ and $n_2 = 5$, the total number of edges $|E|$ can be calculated using the formula described earlier as $3 \times (5-1) \times (5-1) + (5-1) + (5-1) = 56$ which is much less than the total possible edges $|V|^2 = |5 \times 5|^2 = 625$. To find the optimal shortest path, the edge weight must be carefully formulated based on the problem knowledge. As already discussed, taking a diagonal step is optimal since both the robot jobs are executed simultaneously. Therefore, the diagonal edge is weighted less compared to the edge towards the right and the downward direction. The edge weight of $w = 1$ is assigned to the diagonal edge and $w = 2$ is assigned to the other two edges connecting the adjacent vertices. Since a path is not possible through the collision points, the edges connecting the collision vertices are assigned the weight $w = \infty$. Since we aim to find the shortest path, considering similar

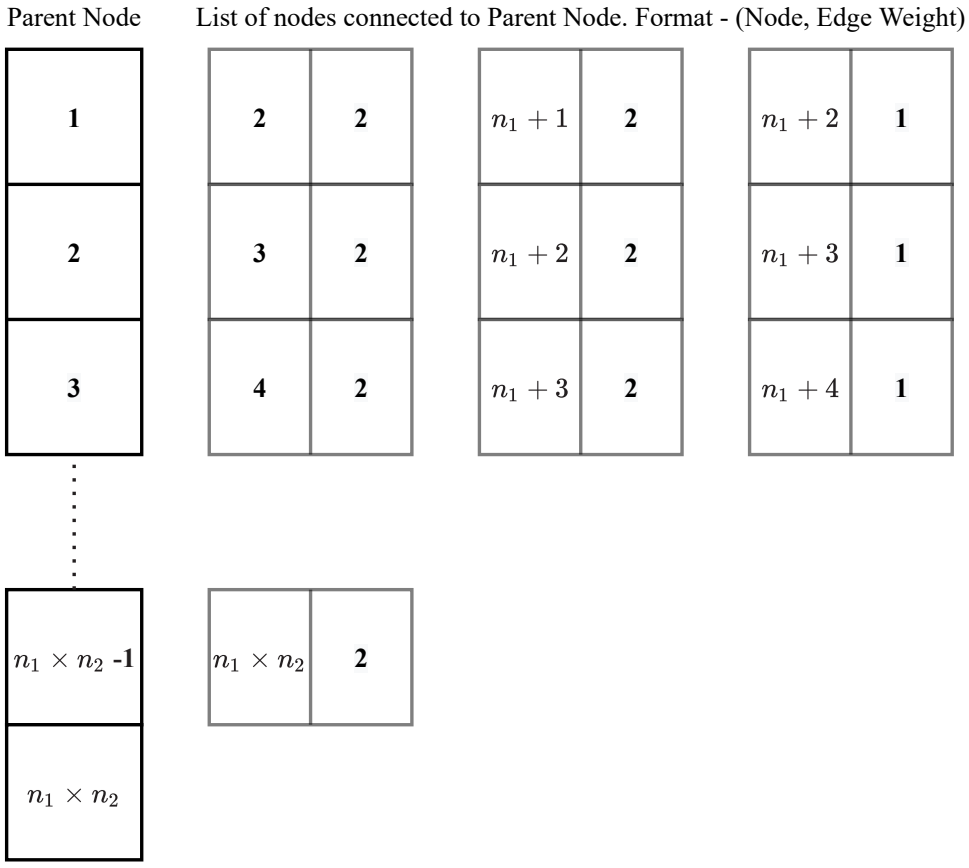


Figure 4.3: Adjacency list graph representation for total number of jobs per robot: n_1 and n_2.

edge weights for diagonal and non-diagonal movement would also give optimal results. However, assigning edge weight as mentioned would speed up the search algorithms as taking a diagonal step is always favoured unless not possible. Moreover, we assign the edge weights similar to rewards achieved by the agent for taking a certain action for ease of comparison. For every problem, the graph must be created before querying with different algorithms. Therefore, the method is scalable and can be used to solve any configuration of a n-robot system irrespective of its size. Figure 4.3 shows the adjacency list representation of the graph for total numbers of robot jobs n_1 and n_2.

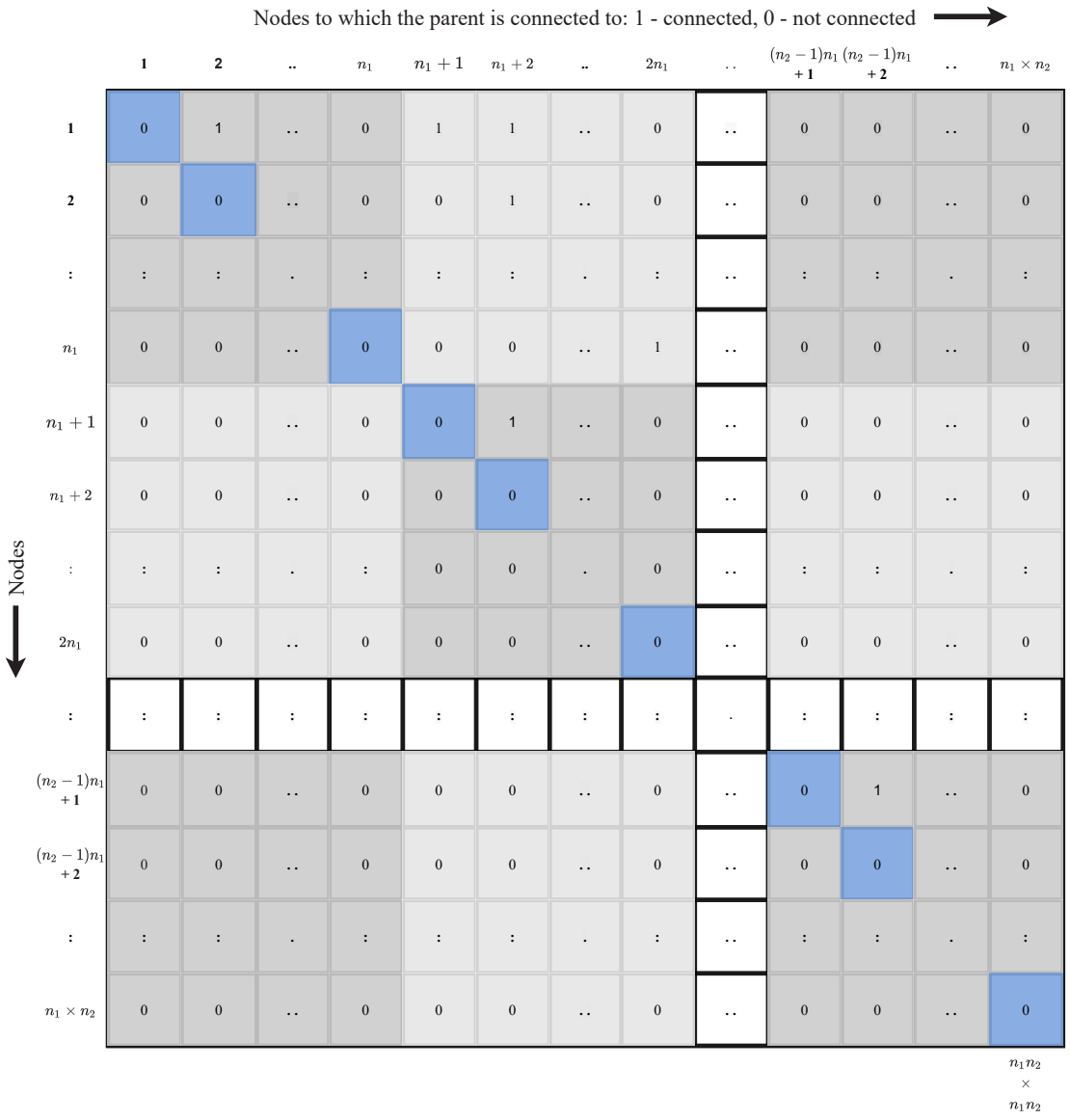


Figure 4.4: Adjacency matrix graph representation for total number of jobs per robot: n_1 and n_2.

Although we use adjacency list representation, adjacency matrix representation is also considered for a single case to compare the time taken to create and search the shortest path. Figure 4.4 shows the adjacency matrix representation of our task for n_1, n_2 number of jobs for robot 1 and robot 2 respectively.

Different graph representations are chosen to speed up the search process. For example, the collision vertices and their respective edges can be omitted before searching the shortest path. Since a considerable amount of collision points exist in the data set, removing these vertices will reduce the total number of vertices considered in the graph, which in turn reduces the computational efforts. In the upcoming sections, we discuss different algorithms to find an optimal path through the graph from the starting vertex $s = 0$ to the target $t = n_1 \times n_2$. Implementation of the algorithms and various speed-up techniques are explained later in this chapter in Section 4.8.

4.2 Breadth-first search vs Depth-first search

Before diving into the shortest path algorithm, we first discuss the two major search algorithms: Breadth-first search (BFS) and Depth-first search (DFS) [Cor+01, p. 531-549]. These are one of the simplest algorithms for searching a graph. As the name suggests, DFS searches deeper in the graph whenever possible. The algorithm starts from the source node and explores as far as possible along each branch before backtracking. The edges are explored from the most recently discovered vertex v that still has undiscovered edges leaving it. When all the edges of vertex v are discovered, the algorithm backtracks to explore the vertex from which v was discovered. This process continues till all the reachable nodes are visited. If any of the vertices remain undiscovered, then one of them is selected as the new source and the search is repeated till all the vertices are scanned. DFS uses a stack (First in Last out) to store the visited nodes.

BFS is the opposite of DFS. It starts at the root source node and explores all the neighbouring nodes at a present dept/layer before moving to nodes present in the next depth. It is moved to the next node when all the neighbours of the current layer are traversed. For a given Graph G and a source s, BFS systematically explores the edges of the graph to traverse every vertex which is in reach of the s. It computes the distance from each reachable node (smallest number of edges) to s. BFS uses a Queue (First in First out) to implement the search algorithm. It starts from the source nodes and visits the unvisited adjacent nodes (shortest numbered). Once scanned, it marks it visited and stores it in the Queue. When no adjacent vertex is found in that layer, the first vertex of the queue is popped and the process is continued till the queue gets empty. Each time a node is visited, it is also stored in a 'discovered' list to keep a track of all the visited nodes. In general, we use the Breadth-first algorithm when the vertex we search is relatively close to the root as the algorithm searches across the breadth of the root. We use a depth-first algorithm when the

vertex is deep in the graph as it traverses deeper quickly. BFS also produces a breadth-first tree from source s that contains all the reachable nodes. If a vertex v is reachable from s, the path in the breadth-first tree corresponds to the shortest path from s to v. When a vertex v is discovered will scanning from the vertex u, the edge (u, v) is added to the tree. The vertex u is called the *predecessor/parent* of v in the breadth-first tree. Every vertex has at most one predecessor as every vertex is discovered only once. BFS is not affected by the weight of the edge. Therefore, if the edge weight of a weighted graph is identical (or has no weight) and is not cyclic, it can be used to find the shortest path. If not, it must be slightly modified which gives rise to Dijkstra's shortest path algorithm.

4.3 Dijkstra's Algorithm

Computing the shortest path from the source s to target t in a weighted graph is a classic problem in computer science. Dijkstra's algorithm finds the shortest path in a graph $G = (V, E)$ by solving the single-source shortest-paths problem on a weighted graph for the case in which all edge weights are non-negative. In this section, we therefore assume that the weights $w(u, v) \geq 0$ for each edge $(u, v) \in E$. The algorithm was formulated by computer scientist Edsger W. Dijkstra in 1956 and published a few years later [Dij59]. Many variants of Dijkstra's algorithms exists of which few are discussed in our study.

Dijkstra's algorithm originally found the shortest path between two given nodes [Dij59], but a more common variant fixes a node as a source node and finds the shortest path from this node to all other nodes by analysing the graph. The algorithm keeps track of the already known shortest distance from the present node to the source node and thereby updates it if a shorter path is found. Once the shortest path between the source and the present node is computed, this node and its corresponding distance from the source node are added to the path. This process continues until the shortest path for all the traceable nodes is computed. Hence, it returns a path that connects the source node to all the other nodes with the shortest possible distance/weight from the source node.

Dijkstra's algorithm starts from a given source vertex $s \in V$ and visits all the other vertices $u \in V$ sorted by ascending order of distances $dist(s, u)$. The algorithm keeps a tentative distance array $d(u)$ to store the distances of all the vertices from the source vertex s. The distance is initially set to ∞ for all vertices u except the starting vertex, i.e for $u \neq s$.

Starting with the source node s having distance $d(s) = 0$, in each iteration, it repeatedly chooses a reachable vertex u having minimal distance estimate for which $dist(s, u)$ is still unknown and settles it. A vertex is settled if its shortest distance from the sources is already found. Therefore, it sets the shortest distance $dist(s, u)$ to $d(u)$, adds u to the visited list and relaxes all the edges leaving u. Relaxing all the outgoing edges $(u, v) \in E$ refers to as comparing and possibly updating all the estimated distances $d(v)$ with

$d(u) + dist(u, v)$, if smaller. A vertex is marked visited if its distance has been updated at least once. It thus updates the estimate $d(v)$ and the predecessor $pred(v)$, if the shortest path to v can be improved by going through u. The choice of vertex u with minimal estimated distance $d(u)$ is commonly implemented using a minimum priority queue Q keyed by their d values that keep track of all the vertices that are discovered but not settled yet. A minimum priority queue is an abstract data type that provides 3 basic operations: *insert*(), *decrease_priority*() and *extract_min*(). Using such a data structure increases computational speed compared to a basic queue. Generally, a minimum priority queue is initialized in the beginning with all the vertices and then extracted in each iteration. The priority is then decreased every time a vertex is settled. Instead of initializing the priority queue with all the vertices and then decreasing priority after each relaxation, it is also possible to initialize it to contain only the source vertex along with its distance. During relaxation, the new vertex is inserted into the queue if it is already not present in the queue. An implementation of Dijkstra's algorithm in pseudo-code can be seen in the Algorithm 2.

Algorithm 2: Dijkstra's Algorithm

```
1  Function DIJKSTRA(Graph, Source):
2      Create Vertex priority queue Q
3      d[Source] ← 0
4      for vertex v in Graph do
5          if v ≠ Source then
6              d[v] ← Infinity
7          pred[v] ← None
8      end
9      Q.Insert(0, Source)
10
11     while Q is not Empty do
12         u ← Q.Extract_min()
13         for neighbour v of u do
14             tentative_distance ← d[u] + dist(u, v)
15             if tentative_distance < d[v] then
16                 d[v] ← tentative_distance
17                 pred[v] ← u
18                 Q.Insert(tentative_distance, v)
19         end
20     end
21
22     return d, pred
```

To compute the shortest path tree, it is important to remember that u is the predecessor of v if a shorter path to v is found. As seen above, Dijkstra's algorithm computes the

shortest path to all the vertices in the graph. However, if only one shortest path is required to a target vertex t, the algorithm can be terminated when the vertex t is settled.

Dijkstra's algorithm is correct because the vertex $u \in V$ gets settled only after all the vertices with smaller distances are already settled, i.e after all the vertices that could possibly lie on the path shorter than the current distance $d(u)$ are taken into consideration. Furthermore, each possible path is taken into consideration while finding the shortest path. All the outgoing edges uv of any already settled vertex u are relaxed so that any vertex reachable from s is eventually inserted into the queue and hence eventually settled [Col12].

4.3.1 Proof of Correctness

The proof of correctness of Dijkstra's algorithm is constructed by using induction on the number of visited nodes.

In the base case, there is just one visited node, namely the source node itself. In this case, the shortest distance is 0 and the empty path to the source node is optimal.

For the inductive case, we need to prove that if the Dijkstra's algorithm finds the k - shortest path correctly, it must also find the $k+1$ - shortest path. We prove this by contradiction. Suppose the algorithm fails to find the shortest path for $k+1^{th}$ case correctly, then there must be a shortest path to a vertex other than the one found by Dijkstra's algorithm that is not in the set of vertices reached by k-shortest paths. We can then show that this shorter path does not exist, and hence the path given by Dijkstra's algorithm is the shortest.

Assume that Dijkstra's algorithm picks a vertex v to add to the shortest path tree having the shortest path distance $d(u) + dist(u, v)$, where the algorithm has already found the shortest path to vertex u. By the inductive hypothesis, it is already found in previous iterations that the shortest path to u is correct. We need to now show that this is the shortest path of all the different paths from source s to vertex u.

Now, we assume that there exists a shorter path P_2, from source s to vertex v that has not yet been discovered. Any such path from s to v must cross the frontier between vertices whose shortest paths are known and vertices for which the shortest paths are not known. Let the path P_2 pass through the edge (x, y) such that there is a known shortest path from s to x, followed by the edge (x, y) and finally a path from y to v. As per the assumption of the algorithm, the distance of the last section of the path P_2 from y to v must be greater than or equal to 0 (since all the edge weights must be non-negative).

Finally consider the distance of the path P_2 from vertex s to y given by $d(x) + dist(x, y)$. But the proposed better path from s to y must be lesser than or equal to the shortest path from s to v, i.e. $d(x) + dist(x, y) \leq d(v)$ or $d(y) \leq d(v)$. Also, since the Dijkstra's algorithm tries to choose the vertex with minimum distance from the source, we have $d(v) \leq d(y)$. Therefore, it contradicts that the vertex chosen by Dijkstra's algorithm is the closest vertex to the source s among all the vertices one edge away from the vertices for which the shortest path is already known. Thus it proves the correctness of Dijkstra's algorithm.

4.3.2 Complexity

Run time bounds of the Dijkstra's algorithm on a graph with edges E and vertices V is expressed as a function of the number of edges and vertices, denoted by $|E|$ and $|V|$ respectively, using the big - $\mathcal{O}$ notation. The time complexity bounds mainly depend on the data structure used to represent the queue Q. The simplest version of Dijkstra's algorithm stores the vertices in an array or a list and the minimum is extracted using a linear search. In such cases, the time complexity is $\mathcal{O}(|V|^2)$. For sparse graphs, Dijkstra's algorithm can be implemented more efficiently with adjacency list representation. The binary heap and the Fibonacci heap is used as a priority queue to extract minimum distance efficiently. With a self-balancing binary search tree or binary heap, the algorithm complexity is reduced to $\mathcal{O}((|E| + |V|)log|V|)$ in the worst case. For a connected graph, this reduces to $\mathcal{O}(|E|log|V|)$. Finally, by using the Fibonacci heap, the time complexity can be further reduced to $\mathcal{O}(|E| + |V|log|V|)$. The minimum priority queue used here uses the Fibonacci heap to return the minimum distance which has a complexity of $\mathcal{O}(1)$ compared to the binary heap which takes $\mathcal{O}(log(n))$ time.

4.4 Best first search

As we saw in the previous section, BFS and DFS both consider an adjacent vertex and they explore paths without considering the cost function. However, the idea of the best first search is that the algorithm explores the graph by expanding the most promising vertex chosen based on a cost evaluation function, i.e. it explores the graph by expanding the node with the least cost first. A priority function is used to assign a cost to each candidate node in a decreasing order of desirability. Efficient selection of the current best candidate for extension is typically implemented using a priority queue. The key component of the best first search is the cost evaluation function $f(v)$ which is defined in terms of heuristic evaluation function $h(v)$ which may depend on the description of vertex v, the distance to the goal t, information gather up-to that point etc [RN10, p. 94]. Heuristic functions are the most common form in which additional knowledge of the problem is passed to the search algorithm. The heuristic evaluation function could be generic or problem-specific. If it uses the tentative distance up to a point, then it becomes similar to Dijkstra's algo-

rithm. Best-First Search algorithms constitute a large family of algorithms, with different evaluation functions having a variety of heuristic functions. The two special cases of the best first search are greedy best-first search and A-star search.

Most commonly, the heuristics evaluation function that attempts to estimate the distance from the current vertex v to goal t is used, so that the path which is close to the goal t are extended first. This type of search is called pure heuristic or greedy best-first search. Greedy best-first search expands the node that appears to be nearest to the goal. Straight line distance heuristics $h_{SLD}(v)$ from the vertex to the goal t could be used as a cost estimate. However, the greedy best-first search is neither complete (can get stuck in loops) nor optimal (does not guarantee a solution with the lowest cost).

4.5 Goal Directed Dijkstra's or A*

A-star is an optimization of Dijkstra's and best first algorithm. It is used for a case when the source and target are known, and at any given point, the distance from the given point to the target can be estimated. The estimation need not be precise, but the algorithm speeds up the search and gives an optimal solution when a good estimate is chosen. The technique modifies the priority of the unvisited nodes to alter the order in which the vertices are processed.

A goal-directed Dijkstra's algorithm adds a potential ρ_t to the priority $d(v)$ based on the goal t. These potential functions are functions that map vertices to real numbers. Given the potential ρ, we can define the new edge weights for every vertex with a reduced cost:

$$dist'(u,v) = dist(u,v) - \rho(u) + \rho(v) \tag{4.3}$$

With a suitable potential, the search may be driven closer to the goal, thereby decreasing the running time even as the algorithm returns an identical shortest path [Koz10]. In other words, running Dijkstra's algorithm on the transformed graph with the new edge weights $dist'$ results in the same shortest path as running Dijkstra's algorithm on the original graph. However, for the search to return the shortest path, the new edge weight $dist'$ must be feasible.

For a weighted graph $G(V,E)$, a potential $\rho : V \to \mathbb{R}$ is called feasible, if $dist(u,v) - \rho(u) + \rho(v) \geq 0$ for all edges $e \in E$

Potential $\rho(v)$ is the estimated distance from vertex v to target t. With such an estimate, the search avoids taking the incorrect direction and directs towards the target. It can be shown that the feasible potential ρ is the lower bound of the distance to the target. For example, a straight line distance from the vertex v to the target t is always lesser than or equal to the actual distance from vertex v to target t. With a tighter bound, the search is

attracted more towards the target. To be precise, goal-directed Dijkstra's algorithm visits nodes only on the shortest path, if the potential is the exact distance to the target [Koz10] [WW07].

A-star algorithm is essentially a different representation of Dijkstra's algorithm with potential. In each step, it picks a minimizing distance of vertex v, which is $dist(s,v) - \rho(s) + \rho(v)$, where $dist(s,v)$ is the distance of the path from source s to vertex v . Here we notice that the potential $\rho(s)$ is same for all the vertices. Therefore, the vertex which minimizes this expression is same as the vertex that minimizes $dist(s,v) + \rho(v)$. The potential $\rho(v)$ can be considered as a heuristic function $h(v)$ that estimates the cheapest path from vertex v to target t.

The above equation can be rewritten in terms of cost evaluation function $f(v)$ as follows:

$$f(v) = g(v) + h(v) \tag{4.4}$$
$$= dist(s,v) + h(v,t) \tag{4.5}$$

where $g(v)$ is the exact cost/distance of the path from source to vertex v and $h(v)$ is the heuristic function that estimates the distance to the target t. The algorithm balances the two as it searches from source to the target. At each step in the loop, it chooses a vertex v which has the minimum cost $f(v)$. Typically, the algorithm uses a priority queue to select the minimum cost vertices to expand. The heuristic function used is problem specific. If the function never overestimates the actual cost to the target and is the lower bound to the actual distance $dist(v,t)$, i.e. $h(v) \leq dist(v,t)$, it is called admissible. When the heuristic function is admissible, the algorithm is guaranteed to return a minimum cost/distance from source to target. A* algorithm terminates when the path it tries to extend is the target or if there are no paths eligible to be extended. Moreover, applying Dijkstra's algorithm on the transformed graph with edge weights $dist'(u,v)$ is the same as applying A* on the original graph with a suitable potential/heuristics. Therefore, if the heuristic is admissible, the algorithm guarantees a shortest path, and in such cases, it can be called *A-star with admissible heuristic* or *Goal-directed Dijkstra's algorithm* interchangeably. In equation (4.5), if $g(v) = 0$, then the algorithm becomes greedy best fit search and if $h(v) = 0$, then it becomes original Dijkstra's algorithm.

An implementation of the A-star algorithm in pseudo-code can be seen in the Algorithm 3. The implementation is specific to our problem. Here, the function *Loc* returns the grid location of a particular vertex. The dimension of the grid location depends on the number of robots considered in the system, i.e. 2D for 2 robot systems, 3D for 3 robot systems and so on. In the thesis, the Manhattan distance (straight line distance) from the present vertex to the target is taken as a heuristic/potential. Since the straight-line heuristic never overestimates the actual distance, the heuristic is admissible and hence guarantees an optimal path. With an efficient choice of heuristic function, the A* algorithm works on

any graph structure.

Algorithm 3: A-star Algorithm

```
1  Function heuristic(vertex : Grid Location, Target : Grid Location):
2      return abs(sum(Target − vertex))
3
4  Function A-star(Graph, Source, Target):
5      Create Vertex priority queue Q
6      d[Source] ← 0
7      for vertex v in Graph do
8          if v ≠ Source then
9              d[v] ← Infinity
10         pred[v] ← None
11     end
12     Q.Insert(0+ heuristic(Loc(Source), Loc(Target)), Source)
13
14     while Q is not Empty do
15         u ← Q.Extract_min()
16         if u == Target then
17             break
18         for neighbour v of u do
19             tentative_distance ← d[u] + dist(u, v)
20             if tentative_distance < d[v] then
21                 d[v] ← tentative_distance
22                 pred[v] ← u
23                 priority ← tentative_distance+ heuristic(Loc(v), Loc(Target))
24                 Q.Insert(priority, v)
25         end
26     end
27
28     return d, pred
```

4.5.1 Complexity

The time complexity of the A* or goal-directed algorithm depends on the choice of the heuristic function. Here we use different notations. In the worst case of unbounded space, the number of vertices expanded is exponential in the dept of the solution d: $\mathcal{O}(b^d)$, where b is the branching factor [RN10]. We assume that the goal is reachable as is at a dept d from the source. Good heuristics are those having minimum branching factors. The optimal branching factor is $b^* = 1$. Therefore in the best-case scenario, the computational time complexity reduces to $\mathcal{O}(1)$. In the worst case, it has the same complexity as Dijkstra's

algorithm.

4.6 Bi-directional Search

Bi-directional search is a search algorithm that is used to find the shortest point from the source s to target t in a weighted directed graph. As the name suggests, it searches in the forward direction from source s towards target t and in the backward direction from target t towards source s. For the backward variant, the algorithm is applied to the reversed graph, i.e., a graph with the same vertex set V as that of the original graph G, and the reverse edge set $E = (u, v)|(v, u) \in E$ [WW07]. If a path between the vertices exists, the search terminates when the two vertices meet. One degree of freedom of bidirectional search is the order in which forward and backward search is executed. Different bidirectional search algorithms can use different priority functions and can use different strategies for alternating between the forward and backward search. Few forward and backward search strategies are to choose the direction with a smaller priority queue, select based on the smaller minimal distance in the queue, simultaneously implement in both directions or simply alternate between the searches. Most commonly, the algorithm alternates between the forward and the backward search. The intuition behind bi-directional search is as follows. Suppose we consider the search space of the algorithm as a growing ball towards the goal. In bidirectional search, the growing ball around each end s and t would terminate when they meet. From this, a path can be found from the source to the target with a reduced search space. The bidirectional approach is usually considered when both the source and target states are unique and completely defined and the branching factor is the same in both directions. For the bidirectional search to return shortest paths, different shortest path algorithms can be used in a bi-directional manner with specific stopping criteria. In this section, bi-directional Dijkstra's (see Section 4.6.1) and bi-directional search guided by a heuristic (A*) (see Section 4.6.2) is discussed in details.

4.6.1 Bi-directional Dijkstra's Algorithm

Bi-directional Dijkstra's runs Dijkstra's algorithm in both forward and backward directions. It optimises the Dijkstra's algorithm and returns the shortest path in a reduced time. The directed graph must be initially reserved to facilitate backward search. In the considered task, the source and the target vertices are unique and the branching factor is same in both the direction as the graph representation is symmetric. Therefore, alternating the two searches will reduce the overall speed up to half the speed of unidirectional Dijkstra's algorithm. Therefore, in this implementation, forward and backward searches are performed alternatively.

The most important factor in bidirectional Dijkstra's algorithm is the stopping condition. If we consider S_f and S_b to be the set of vertices already discovered in forward and

backward search respectively, then there is no guarantee that a newly detected edge (u, v) between S_f and S_b would contribute to the shortest path. Therefore, stopping the search as soon as finding the edge (u, v) between the detected vertices in the forward and the backward search with total distance of $d_f[u] + dist(u, v) + d_b[v]$ or $d_f[v] + dist(v, u) + d_b[u]$ is incorrect. Here d_f and d_b are the distance labels of forward and backward search respectively.

The algorithm can be terminated when one of the nodes is permanently designated by both forward and backward search. If x is the designated permanent vertex, then the shortest path is determined by the vertex x with the least distance $d_f(x) + d_b(x)$ and it composes the shortest path from source s to intermediate vertex x found by the forward search and shortest path from vertex x to the target t found by the backward search. The vertex x itself needn't be settled by both searches [WW07]. An implementation of Bi-directional Dijkstra's algorithm in pseudo-code can be seen in the Algorithm 4. Here, only the implementation of the main iteration is explained since the initialization is the same as the original Dijkstra's algorithm. Since we are dealing with a directed graph, the graph must be initially reversed for the backward search. Similar to the basic Dijkstra's algorithm, we start with forward estimations d_f from source s and backward estimations d_b from target t. These are initialized to infinity except $d_f[s] = 0 = d_b[t]$. Similarly, two minimum priority queues: forward Queue Q_f and backward Queue Q_b are considered which are both initialized with the first vertex and corresponding distance. We also maintain two sets S_f and S_b to store processed vertices in forward and backward search respectively, which is initially empty. Also, a number μ is maintained which is initially set to infinity. μ stores the distance of the shortest path $s \rightsquigarrow t$ yet seen. In forward and backward search, when an edge connects the two sets of processed vertices S_f and S_b, and when the distance of the found path is lower than μ, μ is updated to the present path distance. Therefore, in each iteration, μ is updated to a shorter path, if any, until the shortest path is found. Therefore, when the algorithm exists the search, the value of μ is exactly equal to the distance of the shortest path from source s to target t. To recover the actual shortest path, a vertex must be maintained which gets updated to the considered vertex x every time μ is updated. The shortest path is then $s \rightarrow x$ from the forward search followed by the shortest path $x \rightarrow t$ from the backward search.

Now we discuss the correctness of the termination condition for bidirectional Dijkstra's algorithm. Suppose there exists a path P from $s \rightarrow t$ with a distance lesser than μ. It cannot consist of a vertex x outside $Q_f \cup Q_b$. As per the proof of Dijkstra's algorithm, the considered vertex is at least $d_f[u]$ from source s and at least $d_b[v]$ from target t. Therefore, path P would have a distance of at least μ. Thus P is contained in the set of processed vertices $S_f \cup S_b$. Here, it is considered when one of its edges was first found to connect the two sets of processed vertices and μ was updated to something at most the length of P [Tow20] [Gol+]. Therefore, bi-directional Dijkstra's is correct and gives the optimal shortest path. When

multiple shortest path exists, the shortest path given by bidirectional search might vary.

Algorithm 4: Bi-directional Dijkstra's Algorithm

```
1  Function Bi_Directional_DIJKSTRA(Graph, Source, Target):
2      RGraph = Graph.reverse()
3      ...
4      Q_f.Insert(0, Source)
5      Q_b.Insert(0, Target)
6
7      μ ← infinity
8      while Q_f and Q_b is not Empty do
9          u ← Q_f.Extract_min();    v ← Q_b.Extract_min()
10         S_f.add(u);               S_b.add(v)
11
12         if d_f[u] + d_b[v] ≥ μ then
13             break
14
15         for neighbour x of u do
16             tentative_distance ← d_f[u] + dist(u, x)
17             if tentative_distance < d_f[x] then
18                 d_f[x] ← tentative_distance
19                 pred_f[x] ← u
20                 Q_f.Insert(tentative_distance, x)
21             if x in S_b and d_f[u] + dist(u, x) + d_b[x] < μ then
22                 μ ← d_f[u] + dist(u, x) + d_b[x]
23         end
24         for neighbour x of v do
25             tentative_distance ← d_b[v] + dist(v, x)
26             if tentative_distance < d_b[x] then
27                 d_b[x] ← tentative_distance
28                 pred_b[x] ← v
29                 Q_b.Insert(tentative_distance, x)
30             if x in S_f and d_b[v] + dist(v, x) + d_f[x] < μ then
31                 μ ← d_b[v] + dist(v, x) + d_f[x]
32         end
33     end
34
35     Calculate shortest_path
36     distance ← μ
37
38     return shortest_path, distance
```

4.6.2 Bi-directional A-star Algorithm

Similar to the previous case, bi-directional A-star runs a heuristic search in both directions. The algorithm works similar to bi-directional Dijkstra's as discussed in Section 4.6.1, but with a added heuristic or potential to guide towards the target. In the beginning, as we are dealing with a directed graph, the graph must be reserved to facilitate a backward search. In the forward search, the potential leads to the target and in the backward search the potential leads to the source. The algorithm must be terminated when both the searches meet based on a certain termination condition. The termination is very similar to bi-directional Dijkstra's with an added consideration of the potential or heuristic estimation function. A heuristic estimate h is called consistent if h obeys the inequality $h(u) - h(v) \leq dist(u, v)$ for any two nodes $u, v \in V$, where $dist(u, v)$ is the actual weight of the edge. As the straight-line distance to the goal is considered, the heuristic estimate is consistent. Similar to the bi-directional Dijkstra's, the algorithm maintains and updates the parameter μ which is the current best estimate of the shortest path from source to target. The algorithm terminates when the estimated distance $s \rightarrow t$ or $t \rightarrow s$ is at least μ. For instance, the forward search is terminated when it is about to scan the vertex u (from the queue Q_f) having a distance: $dist(s, v) + heuristic(v, t) \geq \mu$. An implementation of the bi-directional A star algorithm in pseudo-code can be seen in the Algorithm 5. Similar to the previous section, only the implementation of the main iteration is explained, since the initialization is the same as the original A-star algorithm. Here, $priority_f$ and $priority_b$ contains the distance estimate of the respective vertex u and v which were inserted into the queues in the previous iterations. After termination, the shortest path can be retrieved similar to the bidirectional Dijkstra's algorithm, as explained in Section 4.6.1.

As discussed in Section 4.5, the choice of the heuristic function decides the sequence of the search. Since we use the Manhattan distance as the heuristics, the search is directed towards the target, or in other words, the search leans towards the depth. The straight line distance estimate is almost equal to the actual distance and therefore, the uni-directional A-star should return the shortest path in the least possible time. It should be noted that bi-directional A* is only as good as A* when the heuristic leans towards the depth. Therefore running bi-directional A* on an exactly symmetrical graph having source and target vertices at two opposite ends with the same branching factor and heuristic function should take the same time as A* since the number of processed vertices by the bidirectional and unidirectional variant remains the same. In cases where the number of jobs carried out by one robot is more than the other (unsymmetrical), the search might not terminate in the middle and thereby increases the computational time in such cases. Using a bidirectional variant is a good way to speed up A* when the heuristic leans towards the breath. Therefore it is clear that the bidirectional A star does not necessarily speed up our search. However, for certain problem statements like road networks, the bidirectional variant with a good choice of heuristic function can produce an efficient result.

Algorithm 5: Bi-directional A* Algorithm

```
1  Function Bi_Directional_Astar(Graph, Source, Target):
2      RGraph = Graph.reverse()
3      ...
4      Q_f.Insert(0+ heuristic(Loc(Source), Loc(Target)), Source)
5      Q_b.Insert(0+ heuristic(Loc(Target), Loc(Source)), Target)
6      μ ← infinity
7      while Q_f and Q_b is not Empty do
8          priority_f, u ← Q_f.Extract_min();    priority_b, v ← Q_b.Extract_min()
9          S_f.add(u);                           S_b.add(v)
10
11         if priority_f ≥ μ or priority_b ≥ μ then
12             break
13
14         for neighbour x of u do
15             tentative_distance ← d_f[u] + dist(u, x)
16             if tentative_distance < d_f[x] then
17                 d_f[x] ← tentative_distance
18                 pred_f[x] ← u
19                 priority ← tentative_distance+ heuristic(Loc(x), Loc(Target))
20                 Q_f.Insert(priority, x)
21             if x in S_b and d_f[u] + dist(u, x) + d_b[x] < μ then
22                 μ ← d_f[u] + dist(u, x) + d_b[x]
23         end
24         for neighbour x of v do
25             tentative_distance ← d_b[v] + dist(v, x)
26             if tentative_distance < d_b[x] then
27                 d_b[x] ← tentative_distance
28                 pred_b[x] ← v
29                 priority ← tentative_distance+ heuristic(Loc(x), Loc(Source))
30                 Q_b.Insert(priority, x)
31             if x in S_f and d_b[v] + dist(v, x) + d_f[x] < μ then
32                 μ ← d_b[v] + dist(v, x) + d_f[x]
33         end
34     end
35     Calculate shortest_path
36     distance ← μ
37
38     return shortest_path, distance
```

4.7 Learning based speed-up for large graphs

In the previous sections, we considered conventional speed up techniques modified for our specific problem at hand to find the shortest path in the least time. This section discusses the learning-based speed-up techniques for finding the shortest path for large graphs based on the paper *Shortest Path Distance Approximation using Deep learning Techniques* by [RSG18]. The implementation uses node2vec [GL16] to find feature vector embeddings for the graph. Here we only discuss the speed-up technique as a literature survey and therefore, this thesis does not include the implementation of the algorithm. Moreover, the method is useful for huge graphs like social media data or road networks. The multi-robot scheduling task can be solved with alterations to conventional techniques to receive excellent results, and therefore using machine learning-based speed up for our problem becomes irrelevant.

Traditional algorithms like Dijkstra's and A-star are very slow for large graphs and consume huge memory to store the precomputed distance. For a graph with millions of nodes and edges, the computation of a single node distance can take up to a minute. For most problems, computing approximate distance is good enough for getting accurate results. This motivates the use of vector embeddings generated by deep learning techniques for approximating the shortest path distance between the different nodes. Moreover, computing the distance at neural network inference is computationally inexpensive with time complexity of $\mathcal{O}(1)$. So calculating the approximate shortest path distance from a starting node to all other nodes takes $\mathcal{O}(|V|)$.

The general implementation is as follows. The node2vec algorithm is used to find the embeddings $\phi(v) \in \mathbb{R}^d$ for each node $v \in V$. For a pair of vertices $u, v \in V$, the goal is to build a deep neural network to approximate the distance $dist(u, v)$. The approximate distance $\hat{d}$ can be defined as a function

$$\hat{d} : \phi(u) \times \phi(v) \to \mathbb{R}^+$$

that maps the vector embeddings to the real-valued shortest path distance $dist(u, v)$. For training the network, training pairs must be extracted. A certain number of smartly chosen landmark nodes are selected, and the actual shortest path distance is computed from the landmark nodes to the rest of the nodes. Thereafter, the corresponding node embeddings of the landmarks and the nodes are fetched. These are further combined by applying suitable binary operations like subtraction $(\phi(u) \ominus \phi(v))$, concatenation $(\phi(u) \oplus \phi(v))$, average $(\frac{\phi(u) \oplus \phi(v)}{2})$ and point-wise multiplication $(\phi(u) \odot \phi(v))$. This gives the samples with input-output pair: (embedding - actual distance). From this input-output pair, a neural network with smartly chosen hyperparameters can be trained. On successfully training, the approximate distance can be computed at inference. For large graphs, different node embedding techniques like Poincare, HARP [Che+17], etc. could be tried in an aim

for optimal results.

The work of [BAS08] proposed yet another learning-based approach to compute the shortest path. Here, a genetic algorithm was used to solve the shortest path problem. The results showed that the algorithm returned the shortest path considerably faster than Dijkstra's algorithm.

4.8 Implementation of Graph-based approaches for robot scheduling

In the previous sections, we have discussed the working of different shortest path algorithms in detail. This section deals with the implementation of the above algorithm for our specific problem. As explained in Section 4.1.2, for every n-robot system, the collision matrix data set must be represented in terms of a graph. To make the model comparable to the deep Q-learning approach, the edge weights are assigned similar to that of the reward function used in the learning algorithm. A diagonal edge (most favourable) is assigned $w = 1$ and the other two edges towards the right and the downwards directions are assigned $w = 2$. Initially, all the edges connecting the collision vertices are assigned the weight $w = \infty$ so that the algorithm completely avoids these vertices during the search. As explained before, the graph-based approaches are scalable and can be used to solve any n-robot system irrespective of the robot jobs. However, in this thesis, we deal with a 2 robot system with n_1 and n_2 jobs for the robots 1 and 2 respectively. As we need to compare the results of the graph-based approaches to the deep Q-learning approach, the same randomly generated collision matrix configuration with $n_{max} = 40$ used at the inference of the Q learning approach is used to query the shortest path. In the graph-based approach, the collision matrix need not be zero-padded and therefore the total number of jobs n_1 and n_2 for robots 1 and 2 are considered to create the graph. The vertices are represented in a serial manner from source $s = 0$ to target $t = n_1 \times n_2$. Each vertex in the graph represents the job position at a given time. As we are dealing with a sparse graph, adjacency list representation is preferred over adjacency matrix representation. However, the adjacency matrix representation is also used to find the shortest path using Dijkstra's algorithm to compare the computational speed with Dijkstra's algorithm with adjacency list and Deep Q-learning at inference.

After creating the graph, it can be easily queried using different shortest path algorithms. The source and the target vertices are always the first and the last vertex respectively. With this information, Dijkstra's algorithm (Algorithm 2) can be readily used to get the shortest path. Dijkstra's algorithm doesn't directly return the shortest path but returns the distance and the predecessor list. From the predecessor list, the shortest path from the source to the target can be easily deduced as explained before. This shortest-path

represents the efficient scheduling of the robot system.

Since the source and target are exactly opposite to each other, the shortest path must ideally be around the diagonal of the graph. This motivates the use of goal-directed search or the A* algorithm. Based on an appropriate heuristic function or potential, the algorithm can be pushed towards the target by scanning only the required vertices. This considerably decreases the total computational time to find the shortest path. For example, in the symmetric graph with $n_1 = n_2$ having no collision points through the diagonal, the A-star algorithm with an admissible heuristic (Manhatten distance) scans only the diagonal vertices till it reaches the target, therefore solving the problem in $O(1)$ (best case). For searching the graph using the A* algorithm, the heuristic function and a function to find the grid location of the respective vertex must be considered additionally. In our thesis, we used the Manhattan distance or the straight line distance to find the heuristic estimate. With this estimate, the algorithm is guaranteed to return the shortest path as explained in Section 4.5. Since we represent the vertices serially, the exact grid location is required for the heuristic estimation. The grid location can be considered as the location of the vertex in a plot with the number of jobs for robots 1 and 2 considered as x and y axis respectively. Therefore, the grid location of the source vertex would be $(0, 0)$, followed by the second vertex at $(0, 1)$ and so on. The vertex connected to the source vertex in the downward direction would be $(1, 0)$. Finally the grid location of the target vertex would be (n_1, n_2). With this information, the search can be sped up by directing it towards the target. Similar to the Dijkstra's case, the shortest path must be deduced from the distance and predecessor list.

Finally, the bidirectional search algorithms are used to further speed up the search process. For the bidirectional search, the graph is first reversed to enable backward search. Bidirectional Algorithms 4 and 5 are used to find the shortest path. The implementation is straight forward and it is guaranteed to return the shortest path. Unlike the unidirectional method, the shortest path cannot be directly deduced from the predecessor. To recover the actual shortest path, a vertex must be maintained which gets updated to the considered vertex x every time the current estimate μ is updated. With this information, the shortest path $s \to x$ can be easily deduced from the predecessor of the forward search $pred_f$ and shortest path $x \to t$ using the predecessor of the backward search $pred_b$.

For comparison, the total run time to create and solve the shortest path problem, the total number of vertices scanned to solve the problem and the shortest path distance are recorded for all the implemented algorithms. Furthermore, as the algorithms are scalable, they are also used to find and compare the shortest paths for the actual data obtained from the simulation.

4.8.1 Speed-up based on problem knowledge

Apart from using different algorithms to speed up the search process, the graph representation itself is altered to speed up the process. The idea behind the speed up is to either reduce the total number of vertices considered for the search or to reduce the effort of creating a new graph for every specific problem. Few techniques to speed up the total process are explained below. Here, the algorithms considered remain the same and only the graph representation is changed to speed up the process.

Eliminating collision weights

In the small scale regime, the number of collision points considered are comparatively sparse and scattered. However, in the actual dataset, the number of collision points constitute around 30 - 50% of the total job positions. Therefore, omitting these collision vertices considerably reduces both the effort to create the graph and the total number of edges considered, thereby reducing the computational effort. That is, for a vertex u, if the adjacent vertex is a collision vertex v, the edge (u, v) is not considered when creating the graph. In principle, the collision vertices themselves are not connected to any other vertices. Apart from this, the edge weights for all the other edges remain the same. Therefore, the final graph only contains the traceable job positions represented by the vertices along with their edge weights. It must also be noted that, in rare cases, it is possible for certain parts of the graph to be completely disjointed from the main graph. However, this does not affect our search process and in principle, it must also reduce the computational time as the number of edges considered in the search further reduces. All the above-mentioned algorithms can be readily used on the new graph without collision points to find the shortest path.

Restricted Graph

The second approach to speed up the entire process is to create a restricted graph. As we know, the shortest path lies around the major diagonal, and therefore, scanning vertices away from the diagonal results in unnecessary computational efforts. This motivates the use of the restricted graph which only considers a certain number of vertices around the diagonal. The number of vertices around the diagonal can be called the scope. One needs to specify the number of vertices to be considered, and this value is very problem-specific. For example, for a graph with very few collision points around the diagonal, a lower number of vertices around the diagonal can be considered. On the other hand, when a huge cluster of collision points exist around the diagonal, a larger section must be considered (higher scope). The only condition is that there must exist a path from the source to the target. In other words, the graph must not be disjointed. The following formula is used to calculate the range of vertices around the diagonal

$$2 \times (|n_2 - n_1| + c), \qquad c > 0, c \in \mathbb{N}$$

, where c represents the number of vertices considered around the diagonal. The first term $|n_2 - n_1|$ is required for a non-symmetric graph where $n_1 \neq n_2$. For a rectangular graph, there exists no perfect diagonal from source to target passing through all the diagonal vertices. Therefore, extra vertices must be considered along the diagonal in addition to c, which acts as a correction factor. For the actual data set, a much higher value of c is desired. It again depends on the dataset and the position of the collision point. With a right choice of c, the creation of the graph and the search process can both be considerably sped up. Additionally, the two speed up approaches namely, eliminating collision weights and Restricted Graph approaches can be combined to further reduce the computation time. On the new graph, all the explained search algorithms are used to find and compare the shortest paths.

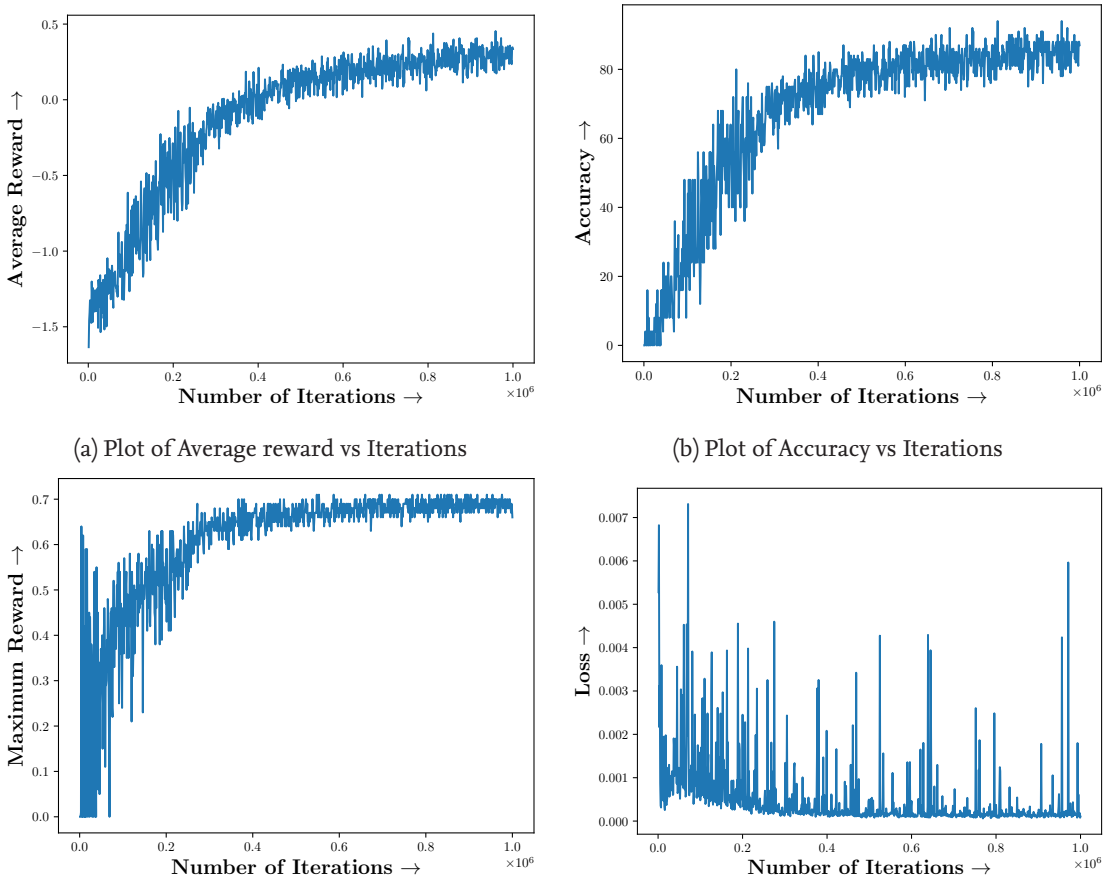
5 Results

This chapter summarises all the significant findings of the different approaches used to find the shortest path. We individually discuss the results of the two methods used, namely, the deep Q learning and Graph-based approaches, and finally compare the two methods based on specific parameters such as computational speed, accuracy, and many more. Although various configurations of the collision matrix are possible in the small-scale regime, we used the same configuration for all the approaches to have an equivalent comparison. Therefore, for validating the approaches, we considered the total numbers of jobs for robot 1 and robot 2 as $n_1 = 37$ and $n_2 = 40$ respectively. Furthermore, the approaches (except deep Q-learning) were also evaluated on the real data from the simulation. Due to the limited availability of such data, training of Deep Q-learning was unfortunately not possible, and therefore, we evaluated only the graph-based methods using actual data.

5.1 Deep Reinforcement learning

In this section, we first discuss the results of training the agent to solve the shortest path problem using the collision control environment, followed by the evaluation of a random configuration of state observation with $n_1 = 37$ and $n_2 = 40$ on the trained model. As seen in Section 3.7, the agent was trained with various configurations of collision matrices for 1000000 iterations. To validate the performance, the metrics were calculated after every 1000 iterations by evaluating the learnt policy for 100 episodes and then averaging them. The following metrics were considered to evaluate the performance: average reward, accuracy, maximum reward and finally the loss. Figure 5.1 shows the plot of these metrics with respect to the number of iterations.

As seen in Figures 5.1(a) and 5.1(b), the learning progresses is similar to that of a negative exponential curve. 50% of the learning took place in the first 20% of the iterations. The algorithm then progressed slowly and reached an accuracy up to 98% near the end of the iterations. The average reward increased gradually from -1.7 to around 0.4 over time. However, as we see, the learning performance was not smooth and showed a fluctuating tendency. For example, the accuracy decreased from 98% to around 75% in just over 1000 iterations. In our thesis, since we only evaluated after 1000 iterations, we only have data for every multiple of 1000 iterations. In reality, there is a possibility for the performance of the learning to decrease after every single iteration. This is called catastrophic forgetting and is one of the major problems in reinforcement learning. When the agent explores the environment, it updates its policies, but what it learns in one part of the environment



(a) Plot of Average reward vs Iterations

(b) Plot of Accuracy vs Iterations

(c) Plot of Maximum reward vs Iteration

(d) Plot of Average Loss vs Iteration

Figure 5.1: Plot of (a)Average reward, (b)Accuracy, (c)Max reward, (d)Average Loss with respect to the number of iterations.

may conflict with what it learns in other parts of the environment. Since the experiences are highly correlated and the learning environment keeps changing, it is not very ideal for gradient descent. Increasing the replay buffer, reducing the learning rate or playing along with the different hyperparameters sometimes helps in reducing catastrophic forgetting. As explained before, different hyperparameters were tried in our study and the results of the best ones are documented here. Therefore, even after using optimal hyperparameters, catastrophic forgetting is sometimes inevitable. A method to get the best model is by saving the policy or the weights of the network during training that corresponds to the highest accuracy. Such a model with 98% accuracy was used in our study at inference for evaluation.

In Figure 5.1(c), it can be seen that the maximum rewards were comparatively high in the initial iterations. This is because the agent initially explores randomly due to the ϵ greedy method. The initial max rewards are hence not because of the learnt behaviour

but because of the random action taken. For learning to be successful, the max rewards must be high initially as seen in our problem. This means that the agent has explored the environment well. It leads to a lot of diversity in the replay buffer that in turn facilitates good learning. As the learning progresses, the ϵ value gradually decreases. The plot is then very similar to that of accuracy and average return plot.

Finally, we can see from the loss plot (Figure 5.1(d)) that it is a poor indicator of the model's performance. The loss might go down over time and still, the agent might perform worse. On the other hand, an increasing loss does not necessarily mean that the performance of the model is decreasing. Therefore, the loss is usually not plotted as it gives no information about the performance.

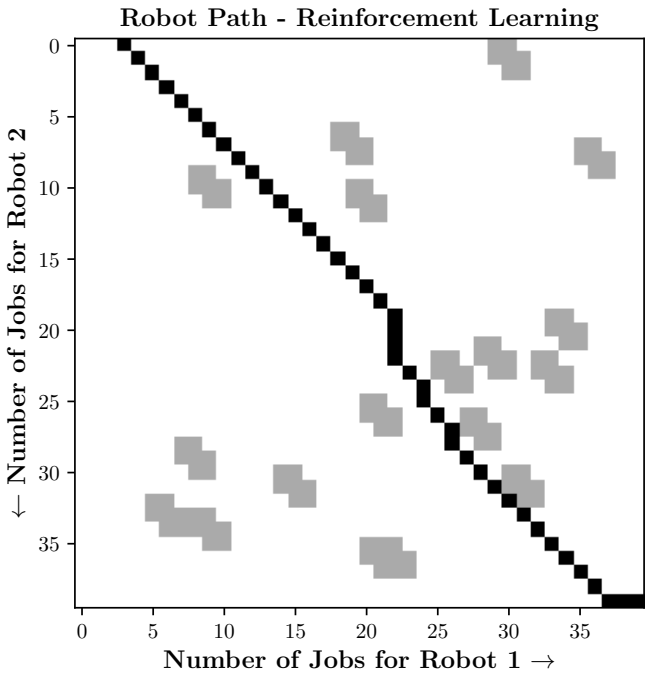


Figure 5.2: Robot Path found using reinforcement learning.

For further evaluation, the policy with the accuracy of 98% and an average reward of 0.47 was used. The policy was evaluated on a random configuration of collision matrix. The total number of jobs considered for robot 1 and robot 2 was $n_1 = 37$, $n_2 = 40$. Figure 5.2 shows the robot path found by the reinforcement learning. Here, grey squares represent the collision points and the black squares represent the robot path. In this particular case, the total steps taken by the agent to reach the target was 41 and the total reward achieved was 0.48. On average, the agent found a path 98% of times. The reason it failed 2% of times was probably that the agent was unable to analyse a certain distribution of data. As we know, the agent was trained with various collision matrix configurations and

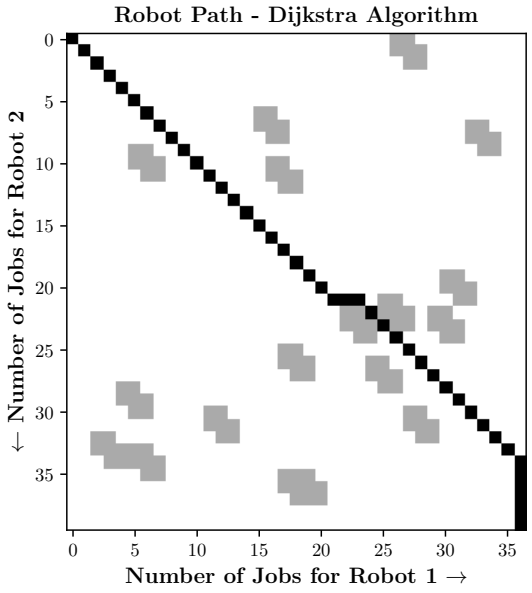
varying n_1 and n_2 and hence, there are over million such configurations possible. However, with the right set of hyperparameters and random seeds, the algorithm will perform excellently with an accuracy of 100%. One of the major drawbacks of this method is the computational effort it takes to learn a certain problem. Since different neural network configurations, hyperparameter sets and random seeds are to be tried repeatedly, the algorithm takes a lot of time to give optimal results. Another drawback is the convolutional neural network that we use. This makes the problem not scalable. The learnt model cannot be extended to evaluate the actual data having the total number of jobs over 2500. The only way to make the agent function for a higher range of configurations is by training the model with all the different configurations within a particular range. For example, if we choose $n_{min} = 40$ and $n_{max} = 2500$, the agent will then function on any configuration having total number of jobs per robot between n_{min} and n_{max}.

Now that we have evaluated the agent on a certain configuration, we try to solve the same configuration with different graph-based methods in Section 5.2. Furthermore, the performance of reinforcement learning is compared to that of graph-based methods in the Section 5.3.

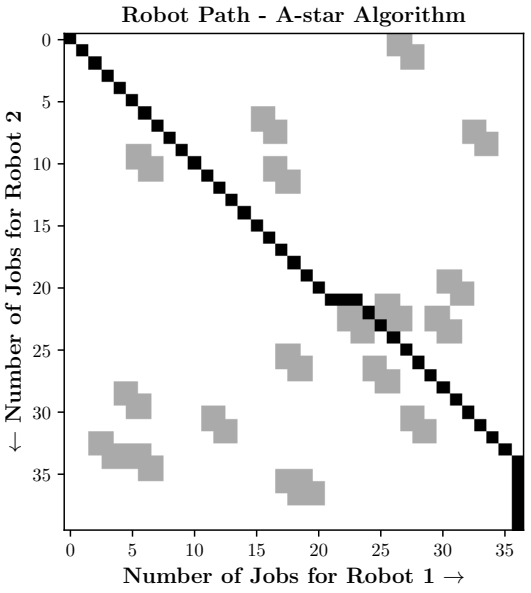
5.2 Graph Based Approaches

The above-discussed collision matrix configuration was solved using all the four shortest path algorithms discussed in Chapter 4. As discussed earlier, Adjacency list representation was used to represent the graph as we deal with a sparse graph. Although we use adjacency list representation, adjacency matrix graph representation was also used to search using Dijkstra's algorithm for the sake of comparing the total execution time with that of the reinforcement learning method. The robot path found using Dijkstra's algorithm for a graph which is represented by adjacency matrix gives the same result as that of the one found using adjacency list representation which is shown in Figure 5.3(a). This section mainly focuses on the individual results. The execution time comparison is carried out in Section 5.3.

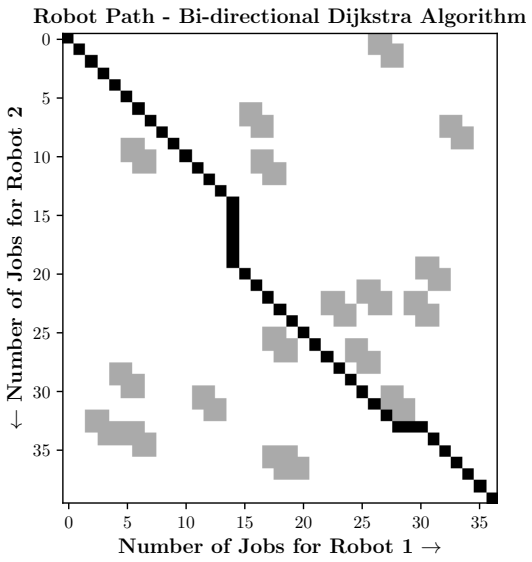
As mentioned before, in all the searches except one, adjacency list representation was used. Figure 5.3 shows the shortest path found by all the following algorithms: (a) Dijkstra's Algorithm, (b) A-star Algorithm, (c) Bi-directional Dijkstra's Algorithm, (d) Bi-directional A-star Algorithm. As mentioned before, there exists the possibility of having numerous shortest paths with the same total edge weight. For the configuration considered here, Dijkstra and A-star returned the same results. However, this might not be the case for every problem. The algorithms might give different paths, all being the shortest paths. This gets more clear in the bi-directional search case as seen in Figures 5.3(c) and 5.3(d). Bi-directional Dijkstra's algorithm runs Dijkstra's search in both the direction and meets in the center. Similarly, bi-directional A-star runs A-star algorithm is both the direction



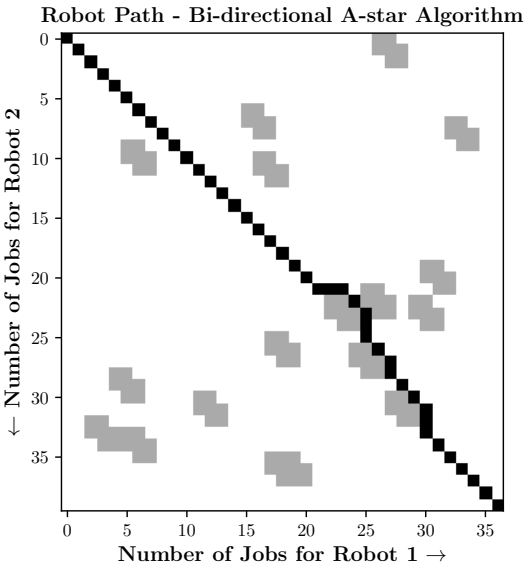
(a) Robot path found using Dijkstra's Algorithm



(b) Robot path found using A-star Algorithm



(c) Robot path found using Bi-directional Dijkstra's Algorithm



(d) Robot path found using Bi-directional A-star Algorithm

Figure 5.3: Plot of robot path found by the following graph based algorithms: (a)Dijkstra's Algorithm, (b)A-star Algorithm, (c)Bi-directional Dijkstra's Algorithm, (d)Bi-directional A-star Algorithm.

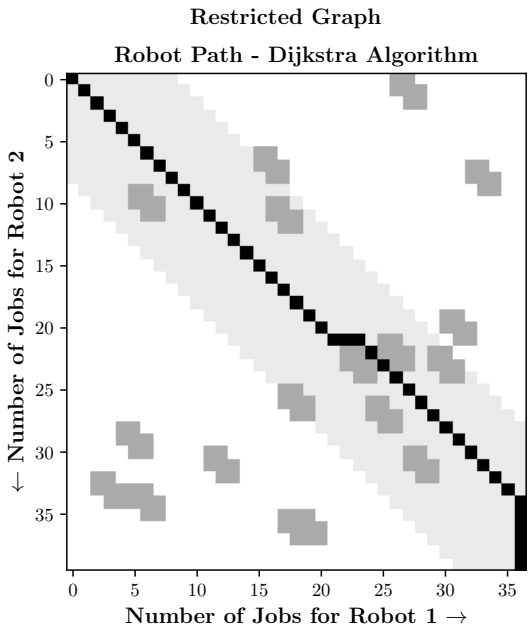
in an aim to meet in the middle. As seen in the graph, we obtained 3 different shortest paths using 4 algorithms, each having the same total step of 41 and a total edge wait of 48. It can also be seen that the reinforcement learning approach found yet another shortest path that is completely different from the ones obtained by these algorithms.

The advantage of these graph-based approaches is that they are scalable. Since these are not learning-based methods, the collision matrices can be easily represented as a graph and then queried using any of the approaches mentioned above. Also, these approaches are guaranteed to give optimal solutions. These are relatively very quick and there exists numerous speed up techniques to speed them up further. Therefore the most important task here is to represent the collision matrix as a graph with appropriate edge weights.

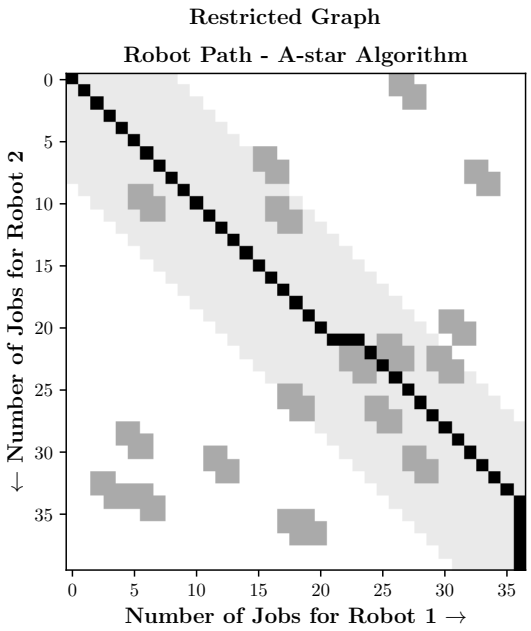
We now try to solve the same task with a restricted graph as explained in Section 4.8. As seen above, the shortest path passes through or around the diagonal that connects the source s and the target t. For this example, we chose the value of c to be 5. Therefore the number of vertices considered around the diagonal was $2 \times |n_2 - n_1| + c = 2 \times |40 - 37| + 5 = 16$. That is 16 vertices were considered around the diagonal such that the diagonal passes through the center of the considered vertices. Figure 5.4 shows the shortest path found by all the four algorithms on the restricted graph. As we see in the plots, the shortest paths found was exactly the same as previous graphs. The only difference here is that we consider a restricted graph which reduces the computational efforts significantly. Here, light grey represents the vertices considered in the restricted graph. Therefore, in the restricted graph, only a small portion of the required region is considered. Grey and black square represents collision points and robot path as explained before. The advantage of considering a restricted graph is that it significantly reduces the computational time. This is because of two reasons: 1. A graph with fewer nodes is created; 2. Fewer nodes are considered in the search process. Apart from this, the algorithms function similarly and return the shortest path having an edge weight of 48.

We also tried to speed up the process by eliminating collision weights from the initial and the restricted graph. This again does not change the shortest path as the edge weights considered are still the same (except for the collision weights). Therefore, plotting these would be redundant. Furthermore, these methods were also extended to the n-robot system. The only difference here is the creation of the graph. Once we have the correct representation of the graph for the n-robot system, the rest of the process remains the same. Since it is harder to visualize a system with more than 2 robots (greater than 2d), we have skipped the evaluation on higher-dimensional data. In the upcoming section, we compare the performance, computational speed and few other metrics of different graph-based and reinforcement learning algorithms.

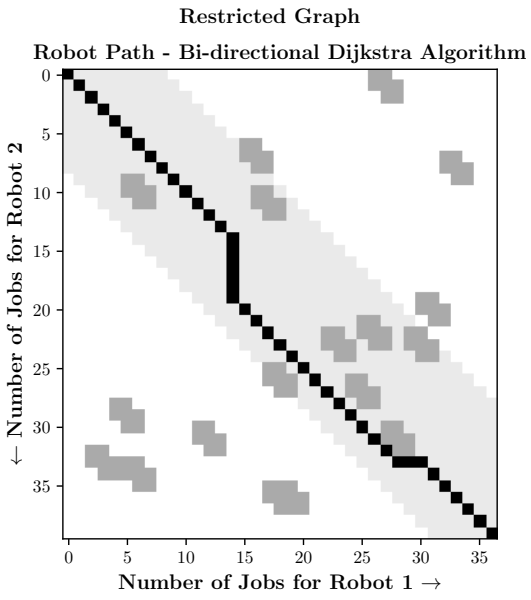
As we know these algorithms are scalable, these can be used to find the shortest path



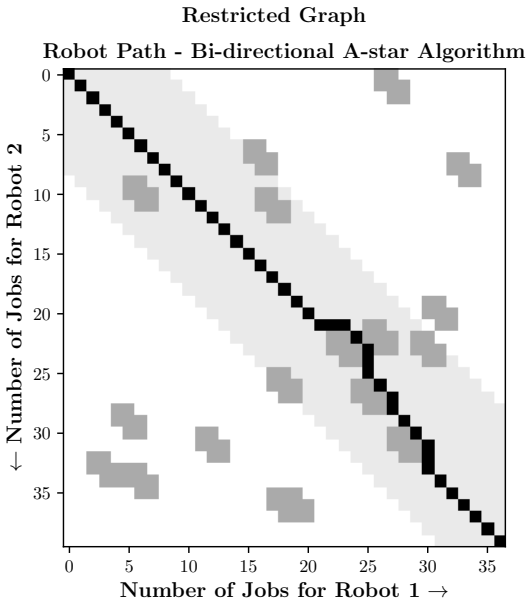
(a) Robot path found using Dijkstra's algorithm on a restricted graph



(b) Robot path found using A-star Algorithm on a restricted graph



(c) Robot path found using Bi-directional Dijkstra's Algorithm on a restricted graph



(d) Robot path found using Bi-directional A-star Algorithmon a restricted graph

Figure 5.4: Plot of robot path found by the following graph based algorithms on the restricted graph: (a)Dijkstra's Algorithm, (b)A-star Algorithm, (c)Bi-directional Dijkstra's Algorithm, (d)Bi-directional A-star Algorithm.

for any graph size. Therefore, the algorithms were used to search for the shortest path on the actual data. Figure 5.5 shows the robot path found using Dijkstra's algorithm on a collision matrix obtained by the simulation. The total number of robot jobs for robot 1 and robot 2 are $n_1 = 1761$ and $n_2 = 1649$ respectively. Here, the results of only Dijkstra's algorithm is shown. Any of the above algorithms can be used to find the shortest path. The algorithm efficiently found the shortest path in around 23s. This time includes the creation and querying of the graph. Finally, the comparison of the computational time against different algorithms for this collision matrix is documented in the next section.

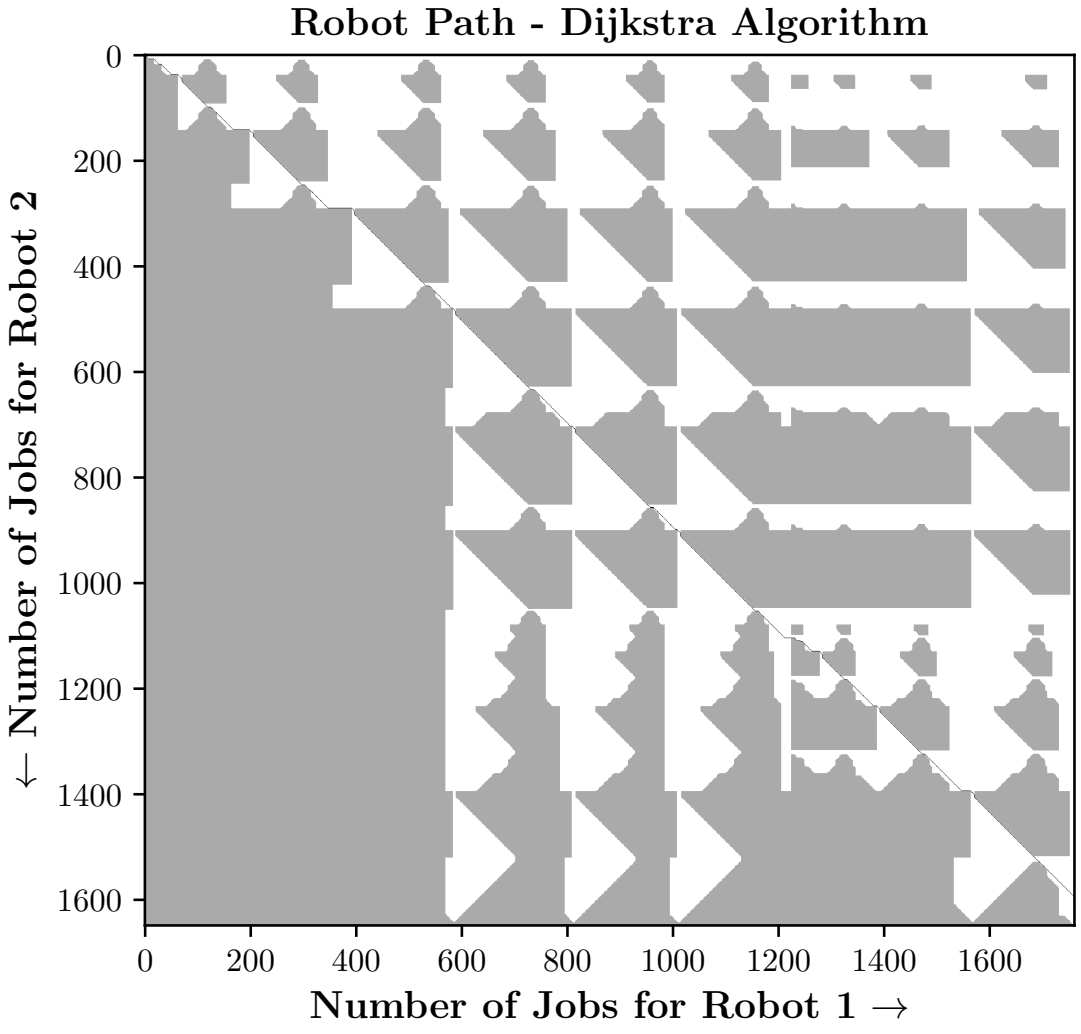


Figure 5.5: Robot Path found using Dijkstra's algorithm on the real data from simulation.

5.3 Comparison

As seen above, all the methods used above were successful in finding an efficient path from the source to the target. Now the task is to choose the best method among all the discussed methods above. In production, we need a method that can search the shortest path in the least possible time and computational effort. In this section, we compare the results of various algorithms used in terms of the following metrics: average time taken, accuracy, average steps taken to reach the goal, average edge weight or reward achieved and the average number of vertices searched (for graph-based methods) for a fixed configuration. We not only compare the learning-based method with the graph-based method, but we also compare the performance of the different algorithms used in graph-based methods with each other. Although we used adjacency list representation, we also considered adjacency matrix representation to search using Dijkstra's algorithm to compare the execution time. The comparison is carried out by evaluating these algorithms on 100 random configuration of collision matrices with $n_1 = 37$ and $n_2 = 40$ and then averaging the metrics. We first compare the reinforcement learning with Dijkstra's algorithm with adjacency matrix and Dijkstra's algorithm with adjacency list representation based on the following metrics: accuracy, average time taken, average steps taken and the average edge weight or reward achieved. The comparison can be found in Table 5.1.

Metrics	**Reinforcement learning**	**Dijkstra with adjacency matrix**	**Dijkstra with adjacency list**
Accuracy	98%	100%	100%
Average time taken	0.1984s	1.5423s	0.01652s
Average steps taken	41.15	39.44	39.44
Average edge weight or reward	48.45	43.32	43.32

Table 5.1: Comparison of Reinforcement learning, Dijkstra's algorithm with adjacency matrix and Dijkstra's algorithm with adjacency list based on the evaluation metrics.

As discussed before, Dijkstra's algorithm is correct and guarantees a shortest path as long as a shortest path exists. Therefore, it is 100% accurate. The policy used at the inference of RL was only 98% accurate which could be further improved by trying different hyperparameters and random seeds. This makes the entire process computationally expensive. However, it is remarkable that a learning-based algorithm learnt a complex problem having different configurations of jobs and collision matrix just by visualizing different datasets. It can also be seen that the reinforcement learning inference was considerably faster than Dijkstra's algorithm represented in terms of the adjacency matrix. However, since we deal with sparse graphs, Dijkstra's algorithm represented by the adjacency list gave the shortest path much faster than both RL and Dijkstra with adjacency matrix representation. Also, on average, reinforcement learning found a path successfully that was close enough to the shortest path. As we see in the table, the RL algorithm took

a couple of more steps to find the target compared to the others. It is also reflected in the average rewards achieved. Hence, it can be concluded that reinforcement learning performed exceptionally well with an accuracy of 98% returning paths close enough to the shortest paths. However, for our problem statement, Dijkstra's algorithm with adjacency list representation outperformed the rest. Another important factor to note is that Dijkstra's algorithm is scalable and can be used to solve the n-robot system with any given number of jobs. This is not the case for Reinforcement learning-based algorithms. The only way to make it work for a higher number of job sets is by training the model with a higher n_{max}. We now compare different Dijkstra's algorithm and their speed up to choose the best performing algorithm.

As we have already seen before, all the algorithms are guaranteed to return the shortest path. Also, even though the shortest paths returned by these algorithms might differ, all the graph-based algorithms return the shortest path with the same edge weight and total steps. Therefore, the average steps, average edge weight and accuracy metrics can be dropped from the comparison. Hence, we only compare these algorithms based on the average execution time and the average number of vertices scanned by them. Since the execution time is too low for a small scale regime, and because these algorithms are scalable, we chose a higher collision matrix configuration of $n_1 = 500$, $n_2 = 480$. Similar to the previous case, the algorithms were evaluated on 100 randomly generated samples and then the considered metrics were averaged and compared. Table 5.2 compares the metrics, namely, average execution time and the average number of visited vertices, for all four algorithms considered in our study. Moreover, the metrics are also compared with the speed up techniques used. For the restricted graph case, $c = 10$ was chosen. Hence $2 \times (|480 - 500| + 10)) = 60$ vertices were chosen around the diagonal such that the diagonal passes through the center. On average, the graph-based algorithms found the shortest path from source to the target by taking 500 steps with a total edge weight of 520.

It can be inferred from the Table 5.2 that the A-star or Goal-directed Dijkstra's algorithm outperformed all the other algorithms based on the considered metrics. The bi-directional A-star takes double the time as compared to the unidirectional case, as the algorithm doesn't terminate in the middle. Bi-directional search can be tweaked such that it meets in the middle but doing so does not improve the computational speed compared to A-star as explained in Section 4.6.2. Therefore, in our case, the bi-directional A-star can be at most as fast as the A-star algorithm. In certain cases, the total scanned vertices could be slightly lower in the bi-directional case when the scan meets in the middle. However, this does not significantly reduce the execution time of the algorithm. Moreover, since we deal with a unidirectional graph, the graph needs to be reversed before searching in the backward direction. This operation takes additional time, making the whole process time-consuming.

The execution time mentioned here depends on the performance of the computer and therefore might vary depending on the machine used. The computational efforts can be expressed better based on the number of vertices scanned by the algorithms to find the shortest path. The bi-directional Dijkstra's algorithm scanned 28.5% of the total vertices to return the shortest path in the original graph. On the other hand, the A-star algorithm only scanned 0.73% of the total vertices to return the shortest path. In other words, it only scanned 1262 extra vertices other than those already considered for the shortest path. Therefore, A-star would be the right choice for finding the shortest path in the least possible time.

Original Graph				
Metrics	**Dijkstra**	**A-star**	**Bi-directional Dijkstra**	**Bi-directional A-star**
Average execution time	3.1386s	1.1856s	2.6395s	2.1213s
Average vertices visited	239938	1762	68615	3359
Graph without collision weights				
Metrics	**Dijkstra**	**A-star**	**Bi-directional Dijkstra**	**Bi-directional A-star**
Average execution time	3.1096s	1.1770s	2.5885s	2.1075s
Average vertices visited	238666	1743	68145	3321
Restricted Graph				
Metrics	**Dijkstra**	**A-star**	**Bi-directional Dijkstra**	**Bi-directional A-star**
Average execution time	0.3661s	0.1968s	0.3657s	0.3228s
Average vertices visited	28315	1762	26996	3359

Table 5.2: Comparison of different graph-based algorithms for the original graph, graph without collision weights and the restricted graph based on the evaluation metrics.

We now discuss the speed up techniques used. For the graph constructed without the collision points, only slight improvements in the execution time were seen compared to the original graph. It is because we considered a limited number of collision points in the collision matrix. For a collision matrix with much higher collision points, a significant reduction in the execution time can be witnessed. The reason behind the speed-up is due to the reduced effort to create the graph and the reduced number of vertices considered

for the search. Therefore, this approach becomes useful for the collision matrix having a significantly large number of collision points. An example of such a data set can be seen in Figure 5.5. We discuss the execution time for this graph later in this section.

In the last case where we use a restricted graph, a huge reduction in the execution time could be seen. The reduced execution time is because of two main reasons: one, fewer efforts are put to create the graph, and two, fewer vertices are considered while searching. As explained before, the major part of the execution time depends on the creation of the graph. Hence, considering a smaller graph helps in producing quicker results. Apart from that, the trend of the evaluation metrics is similar to that of the original graph. One important point to note is that the A-star algorithm found the shortest path by scanning the exact same number of vertices as that of the original graph. This is because of the fact that the A-star algorithm with an admissible potential, such as ours, finds the shortest path by scanning as few vertices as possible. Hence considering a smaller section does not affect the speed or vertices scanned by the A-star algorithm. The two considered speed-up techniques can be further combined to achieve quicker results. Since we have considered a limited number of collision points in the random configuration, no significant improvements in execution time could be seen. Therefore, the comparison on the collision matrix of the small scale regime is not documented in the thesis. However, later in this section, we discuss the effect of combining the two speed-up techniques for querying the actual data having a significantly large number of collision points. All in all, it can be concluded that the A-star algorithm on the restricted graph with no collision weights produces the most promising results provided that a good heuristic estimate is used.

Finally, we also compare the computational time for finding the shortest path using different graph-based algorithms on the actual data discussed in the previous section (see Figure 5.5). Here we document the evaluation metrics for one single collision matrix. For the restricted graph, $c = 100$ was chosen. That is, 424 vertices were considered around the diagonal such that the diagonal passes through the center of the graph. As seen in Figure 5.5, around 50% of the vertices are collision points. Therefore, the effect of eliminating the collision weights could be clearly observed. The comparisons with different speed-up techniques can be seen in Table 5.3. The reduction in the execution time and the number of vertices visited could be clearly observed in the speed-up techniques used. The overall trend of the Table 5.3 is similar to that of the Table 5.2. In the table, we see that the bi-directional search required more execution time compared to its uni-directional variant. This is however not because of the search algorithm itself. Reversing the graph for the background search is computationally expensive because of the large number of vertices present in the graph. Therefore, the execution speed can be better compared based on the number of vertices scanned. As claimed before, the A-star algorithm with the combined speed up gave the most promising results. In very few cases, such as this, the bi-directional A-star outperforms its uni-directional variant. The bi-directional A-star gives the best

result only when it meets in the middle of the graph. Even so, the bi-directional variant only scanned 404 fewer vertices as compared to the other. Moreover, an extra effort must be put to reverse the graph which takes the same effort as creating a new graph. Therefore, for our problem, using a unidirectional A-star with the combined speed up would be the optimal choice to solve the shortest path problem. This shortest-path can be used in production to schedule the two robots efficiently. Moreover, due to its scalable property, it can be further used to schedule any n-robot system irrespective of the number of jobs.

Original Graph				
Metrics	**Dijkstra**	**A-star**	**Bi-directional Dijkstra**	**Bi-directional A-star**
Execution time	23.5191s	13.1491s	27.1244s	24.2478s
Vertices visited	1056917	10891	288855	11269
Graph without collision weights				
Metrics	**Dijkstra**	**A-star**	**Bi-directional Dijkstra**	**Bi-directional A-star**
Execution time	17.0112s	7.2813s	15.3152s	12.0255s
Vertices visited	1039276	10049	278114	9645
Restricted Graph				
Metrics	**Dijkstra**	**A-star**	**Bi-directional Dijkstra**	**Bi-directional A-star**
Execution time	5.4040s	3.6583s	7.1745s	6.2175s
Vertices visited	244744	10891	148329	11269
Combined speed-up: Restricted Graph without collision weights				
Metrics	**Dijkstra**	**A-star**	**Bi-directional Dijkstra**	**Bi-directional A-star**
Execution time	4.1924s	2.5746s	4.5573s	3.7477s
Vertices visited	239186	10049	141556	9645

Table 5.3: Comparison of different graph-based algorithms queried on the actual data for the original graph, graph without collision weights, restricted graph and the restricted graph without collision weights based on the evaluation metrics.

6 Conclusion and Outlook

A Multi-robot system (MRS) refers to any process where several robots coordinate among themselves to achieve a definite goal. GRoFi is a multi-robot system operated by the Center for Lightweight-Production-Technology in German Aerospace Center (DLR), Stade for fibre placement where reinforcing fibers are placed along a predetermined path on any component. The problem statement in this thesis was to find the optimal paths in this multi-robot system and to schedule the robots to execute their tasks as early as possible without any collisions. The challenge was to ensure that these robots coordinate among themselves efficiently and take the optimal path with the least time. The data used in the research was created in a small-scale collision control environment that randomly generated the data set with collision points similar to the actual simulation data from a simulation software VNCK provided by Siemens. This simulation software simulated the working of multiple robots to check for possible collisions when they function simultaneously. But as this software had limited data set from the real world at the time of conducting this research, we generated the data randomly. The data set was called a collision matrix. We then applied deep Q learning, a type of model-free reinforcement learning to find the fastest path of robots through the collision matrix. The performance of this method was validated using several metrics of evaluation for a random collision matrix configuration. The reinforcement learning model was successful in finding an efficient path by learning through various random generated data in the small scale regime. A job completion accuracy of 98% was recorded when evaluated on the best-saved policy. An advantage of reinforcement learning is that it can be used to solve diverse tasks which might not be related to each other. Moreover, it learns from its previous experiences and hence can be improved over time. However, it is data-hungry and requires a lot of training with optimal hyperparameters to achieve superior results. These models are not scalable due to the use of neural networks and hence can only be used to solve tasks similar to the ones it has already seen before.

After that, for the same collision matrix, we represented the robot scheduling task as a graph and consequently queried it with various algorithms to find the shortest path from the source to the target. All the graph based algorithms were successful in finding the optimal paths. The most important task in these methods was to represent the task efficient as a graph. We further investigated various speed up techniques to obtain quicker results. Although all the algorithms found the shortest path, the goal directed Dijkstra's algorithm queried on the restricted graph without the collision weights gave the best results in terms of the execution time. Furthermore, we evaluated the graph-based method by applying it on the data generated by VNCK software. As these algorithms were scalable

and similar results were achieved.

The results of all these different algorithms in graph-based methods were then compared against each other and also against the results from reinforcement learning. For a fair comparison and for validating all of the previously mentioned approaches, we considered only two robot systems in a small scale regime. The reinforcement learning at inference was significantly faster than Dijkstra's algorithm queried on the graph represented by the adjacency matrix. However, as we deal with a sparse graph, Dijkstra's algorithm on adjacency list representation outperformed the latter two. This thesis serves as a Proof of Concept (POC) for reinforcement learning. RL can be used to solve various tasks just by visualizing the data and gives a state of the art results. The RL agent was successful in finding a path close to the shortest path in a time much slower than the adjacency matrix representation. Among the graph-based methods, the A-star algorithm was observed to return the shortest path in the least possible time. After using further speed up techniques on the graph-based methods, the A-star algorithm used on the restricted graph without the collision points showed the best performance in terms of execution time. In the production, as we need an algorithm that schedules robots in the least possible time, an A-star algorithm with an admissible potential (Manhattan distance to the target) can be used to schedule the robots in the least time. The advantage of graph-based methods over reinforcement learning is that these algorithms are scalable and hence can be used to schedule any n-robot system irrespective of the job size. Moreover, these algorithms are complete and guarantee a shortest path. They also do not require extensive training before finding the shortest path as in the case of reinforcement learning.

Further studies can be done by applying all these algorithms to the real data generated by the simulation software VNCK which were unavailable at the time of research. We can check if we will observe any difference in the performance of all the algorithms on the actual data. We can also do further research exclusively on addressing the problem of catastrophic forgetting that we observed in the course of this research to further improve the learning. The proposed solutions can be further scaled to multi robot systems involving many more robots for both RL and graph based methods. Further research could be carried out on using machine learning to speed up the graph based approaches.

Bibliography

[Agg18] Charu C. Aggarwal. *Neural Networks and Deep Learning. A Textbook*. Cham: Springer, 2018, p. 497. ISBN: 978-3-319-94462-3. DOI: 10.1007/978-3-319-94463-0.

[AMO93] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.

[BAS08] Ayoub Bagheri, Mohammad-R Akbarzadeh-T, and Mohamad Saraee. "Finding shortest path with learning algorithms". In: *International Journal of Artificial Intelligence [electronic only]* 1 (Jan. 2008).

[Bel57] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957. ISBN: 9780486428093.

[Che+17] Haochen Chen et al. "HARP: Hierarchical Representation Learning for Networks". In: (June 2017).

[Col12] Tobias Columbus. *Search Space Size in Contraction Hierarchies*. Institute for Theoretical Informatics, KIT. 2012.

[Col19] Tobias Columbus. *On the Complexity of Contraction Hierarchies*. Institute for Theoretical Informatics, KIT. 2019.

[Cor+01] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937.

[Dij59] Edsger W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[DS13] K.-L Du and M.N.s Swamy. *Neural Networks and Statistical Learning*. Oct. 2013. ISBN: 978-1-4471-5570-6. DOI: 10.1007/978-1-4471-5571-3.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. Cambridge, MA, USA: MIT Press, 2016.

[Ger19] Aurelien Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Second edition. Sebastopol, CA: O'Reilly Media, 2019.

[GL16] Aditya Grover and Jure Leskovec. "node2vec: Scalable feature learning for networks". In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 855–864.

[Gol+] Andrew V. Goldberg et al. *Efficient Point-to-Point Shortest Path Algorithms*. Lecture notes: Princeton University.

[Hes+17] Matteo Hessel et al. *Rainbow: Combining Improvements in Deep Reinforcement Learning*. 2017. arXiv: 1710.02298 [cs.AI].

[HGS15] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].

[KB14] Christian Krombholz and Jens Bölke. *Abschlussbericht ZLP Stade: GrOnQA*. -Confidential-. Aug. 2014.

[Koz10] Alexander Kozyntsev. *Bidirectional search and Goal-directed Dijkstra*. TU Munich. 2010.

[Lufi16] Deutsches Zentrum für Luft- und Raumfahrt e.V. et al (2016). "GroFi: Large-scale fiber placement researchfacility". In: *Journal of large-scale research facilities, 2, A58 http://dx.doi.org/10.17815/jlsrf-2-93* (2016).

[Mni+13] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[RN10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010.

[RSG18] Fatemeh Salehi Rizi, Jörg Schlötterer, and M. Granitzer. "Shortest Path Distance Approximation Using Deep Learning Techniques". In: *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)* (2018), pp. 1007–1014.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[Sch+16] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: 1511.05952 [cs.LG].

[Sch15] Markus Schreiber. *Erweiterung des Quay Crane Scheduling Problems für eine Zeitablaufsteuerung mehrerer schienengebundener Roboter sowie Entwicklung eines heuristischen Lösungsverfahrens für Fibre-Placement-Produktionsprozesse*. Institut Angewandte Stochastik und Operations Research, Fakultät für Mathematik/Informatik und Maschinenbau, TU Clausthal. Apr. 2015.

[Sut88] Richard S Sutton. "Learning to predict by the methods of temporal differences". In: *Machine Learning* 3.1 (1988), pp. 9–44. DOI: 10.1007/BF00115009.

[Tow20] Matthew Towers. *Bidirectional Dijkstra*. University College London, https://www.homepages.ucl.ac.uk/~ucahmto/math/2020/05/30/bidirectional-dijkstra.html. 2020.

[Wan+16] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. 2016. arXiv: 1511.06581 [cs.LG].

[WD92] Christopher J. C. H. Watkins and Peter Dayan. "Q-Learning". In: *Machine Learning* 8.3 (1992), pp. 279–292. DOI: 10.1007/BF00992698.

[WS92] David A. White and D. Sofge. *Handbook of intelligent control: Neural, fuzzy, and adaptive approaches*. New York: Van Nostrand Reinhold, 1992.

[WW07] Dorothea Wagner and Thomas Willhalm. “Speed-Up Techniques for Shortest-Path Computations”. In: *STACS 2007*. Ed. by Wolfgang Thomas and Pascal Weil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 23–36. ISBN: 978-3-540-70918-3.