POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Decision making in HRC combining offline path planning and Machine Learning methods

Tesi di Laurea Magistrale in Ingegneria Meccanica

Author: **Bianca Grieco**

Student ID: 976416
Advisor: Prof. Andrea Maria Zanchettin
Co-advisors: Ing. Martina Pelosi, Prof. Paolo Rocco
Academic Year: 2023-24

Abstract

The advent of Industry 4.0 marks a significant transformation towards a digitalized and automated manufacturing environment. Collaborative robotics is a milestone of this development, promoting a synergistic relationship between humans and robots to enhance productivity. Ensuring safety is one of the key challenges in this field, as cobots are designed to work in close proximity to humans without causing any harm.

This thesis explores the technical challenges of designing a collaborative robotic system that can operate both safely and effectively. The methodological approach includes an offline generation of a dataset of feasible paths connecting the starting configuration to the goal using a Rapidly Exploring Random Tree algorithm. Subsequently, a Reinforcement Learning method is employed to enable the robot to dynamically select on which of the previously computed paths to travel and when to transition from a path to another, depending on the human worker presence. The goal is to empower the robot to autonomously identify the path that optimally balances maintaining a safe distance from the human worker with reducing the distance between its current configuration and the target one. The training and testing phases of the proposed algorithm are performed in a simulated environment. The established optimal policy is then validated on the GoFa™ robotic arm.

Keywords: Human-Robot Collaboration, safety, path planning, Reinforcement Learning, industrial automation.

Sommario

L'avvento dell'Industria 4.0 segna una significativa transizione verso un ambiente di produzione digitalizzato e automatizzato. La robotica collaborativa rappresenta una pietra miliare di questo sviluppo, promuovendo una relazione sinergica tra umani e robot per migliorare la produttività. Garantire la sicurezza è una delle principali sfide in questo campo poiché i cobot sono progettati per lavorare in stretta prossimità con gli umani senza causare danni. Questa tesi esplora le sfide tecniche nella progettazione di un sistema robotico collaborativo che possa operare in modo sicuro ed efficace.

L'approccio metodologico include la generazione offline di un dataset di percorsi permessi che collegano la configurazione di partenza a quella finale utilizzando un algoritmo di Rapidly Exploring Random Tree. Successivamente, un metodo di Reinforcement Learning è impiegato per consentire al robot di selezionare dinamicamente su quale dei percorsi precedentemente calcolati viaggiare e quando passare da un percorso all'altro, a seconda della presenza dell'operatore umano. L'obiettivo è consentire al robot di identificare autonomamente il percorso che bilancia il mantenimento di una distanza di sicurezza dall'umano con la riduzione della distanza tra la sua configurazione attuale e quella obiettivo. Le fasi di addestramento e test dell'algoritmo sono state eseguite in un ambiente simulato. Successivamente, la politica di apprendimento è stata validata sul braccio robotico GoFa™.

Parole chiave: Collaborazione Uomo-Robot, sicurezza, pianificazione dei percorsi, apprendimento per rinforzo, automazione industriale.

Contents

List of Figures

List of Tables

List of Algorithms

1 | Introduction

1.1. Field of application

The fourth industrial revolution, commonly referred to as Industry 4.0, has given rise to the concept of the smart factories characterized by seamless connectivity, sensors integration, and the operation of autonomous and self-organizing systems. This transformation is driven by digital technologies, like Internet of Things (IoT) and Artificial Intelligence (AI), which are revolutionizing traditional manufacturing processes. The fusion of these advanced technologies within smart factories serves to significantly enhance productivity, marking a huge transformation in how industrial operations are conducted.

A prominent aspect of this revolution is the advent of robots in production lines, which are becoming fundamental in various industries: robots are sophisticated, intelligent systems equipped with advanced sensors, which are designed to operate autonomously, taking real-time decisions based on environmental data. The use of robots in industries brings several advantages: they excel in repetitive tasks, hazardous, with a high degree of precision, thereby freeing human workers from monotonous and potentially risky activities. Moreover, robots in Industry 4.0 are often equipped with Machine Learning (ML) capabilities, enabling them to learn and adapt to new tasks and scenarios, providing industries with the flexibility needed to respond quickly to continuously evolving market dynamics.

With such opportunities, the adoption of robots in industries simultaneously presents questions about the potential impact on employment, economic implications, cybersecurity threats and ethical considerations [9]. Striking a balance between automation, human labor and surveillance is a critical consideration in managing the current challenges. Another issue is the upfront cost and investment required to integrate sophisticated robotic systems, which may be prohibitive for small and medium-sized enterprises. Furthermore, reliance on robots raises concerns about cybersecurity and the vulnerability of manufacturing systems to hacking and data breaches. Finally, there is the matter of ethical considerations and the need to ensure that the deployment of robots in industries does not compromise human safety or well-being.

1.2. Human-Robot Collaboration

A significant development in the context of the fourth industrial revolution has been the rise of collaborative robots, also known as cobots, able to work alongside human in a shared workspace, enhancing Human-Robot Collaboration (HRC) and exploiting their combined abilities to achieve optimal outcomes. Collaborative robotics represent not only a technological advancement but also a paradigm shift in how we envision the role of robots in the workplace. As industries continue to embrace these innovative solutions, collaborative robotics is expected to play a central role in determining the future of automation and human-machine collaboration [33].

With respect to conventional industrial robots, which usually replace human workers and operate in isolation or behind safety barriers, collaborative robotics aims at creating a synergistic cooperation between humans and robots in a shared workspace, enabling them to help humans to accomplish the sequence of actions required to perform a certain task. In the context of industrial applications, certain operations demand advanced capabilities and cognitive skills, making them better suited for human execution. On the other hand, repetitive activities, requiring precision and reliability beyond human capacity, are better handled by robots. This specific approach to automation has the potential to revolutionize fields such as manufacturing, logistics, and healthcare [31]. The most commonly used robots are shown in Figure 1.1.

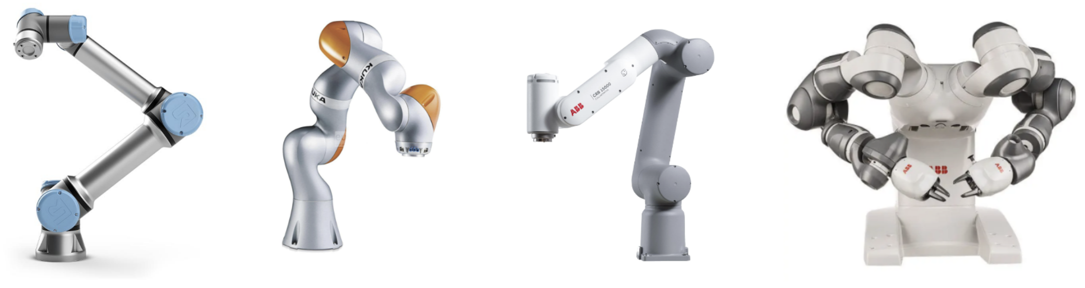


Figure 1.1: From the left to the right: UR5, KUKA LBR iiwa, ABB GoFa™ CRB 15000, ABB IRB14000 (YuMi)

In a collaborative context, ensuring a safe environment deserves thoughtful attention. Cobots are designed to respect safety specifications, featuring compact sizes, lighter structures and protective padding to minimize the risk of injury upon impact while handling light loads. They are equipped with advanced sensors, vision systems, and control algorithms which allow them to detect the presence and the movements of human workers, allowing for real-time adjustments to avoid collisions and ensure a secure environment.

The survey by Villani *et al.* [44] highlights the three main challenges in HRC, defined in the following:

- Safe Interaction refers to the set of principles and technologies that ensure physical safety of humans when they are working in close proximity to robots. This involves adhering to guidelines and specifications, which dictate how robots should behave around humans to prevent injuries, as well as implementing robot operational modes that are specifically designed for human interaction.
- Intuitive Interfaces refer to user-friendly technologies that allow humans to interact with robots in a natural and effortless way. These interfaces are designed to reduce the cognitive load on human operators, enabling them to communicate commands to robots and receive information back from them without complex training or technical knowledge. They include programming by direct manipulation or simple demonstrations, using straightforward input methods like gestures or voice commands, and employing Augmented Reality (AR) or Virtual Reality (VR).
- Design Methods cover the development of control laws, techniques, and strategies involved in creating robotic systems that can work effectively and safely alongside human workers. They include establishing clear task planning and allocation to harmonize robotic functions with human activities. Control laws must be designed to respond to the presence and actions of human collaborators dynamically, ensuring smooth and safe interactions. Furthermore, equipping robots with sensors enables them to accurately detect and react to their surroundings and human movements.

Cobots must adhere to the *ISO/TS 15066* safety specification [12]. This technical specification addresses the unique safety challenges that arise when humans and robots share a workspace. It outlines the critical parameters and methodologies to ensure a safe interaction between humans and cobots, providing guidelines for the design and implementation of this kind of systems. It identifies four types of collaborative operating modes: safety-rated monitored stop (SMS), hand guiding (HG), speed and separation monitoring (SSM) and power and force limiting (PFL).

These categories, illustrated in Figure 1.2 and further elaborated below, are designed to mitigate risks associated with direct human-robot interaction, ensuring that cobots can work alongside human operators without the need for physical barriers or safety cages:

- In SMS, the robot detects the presence of the human in the workspace and automatically pause its operations, maintaining its position until the person leaves the space or it is deemed safe to resume.
- In HG, operators can directly instruct the robot by manually guiding it to desired positions. The robot arm's weight is compensated for, allowing it to maintain its position while the operator physically interacts with the machine.
- SSM involves the continuous monitoring of the speed and position of both the robot and the human worker during operations. Based on the distance between them, the system dynamically scales of the velocity of the robot as it approaches the operator, while preserving the path consistency of the trajectory [26].
- Finally, PFL allows the robot to operate near the human by limiting the amount of force a robot can apply such that a potential impact remains below a predetermined threshold for pain-free contact.

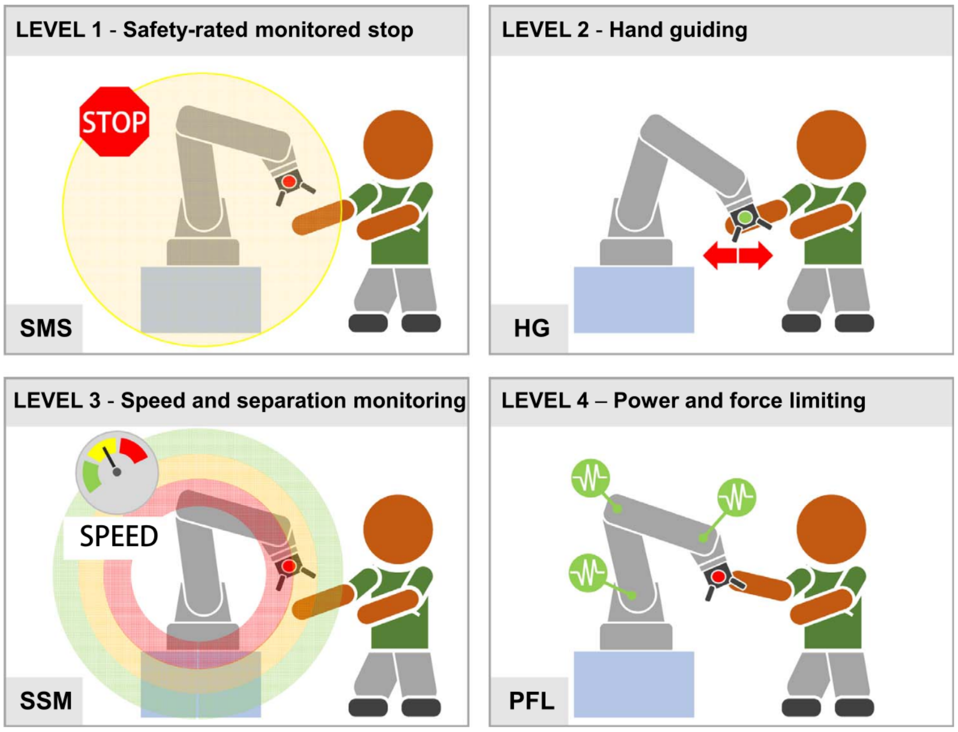

LEVEL 1 - Safety-rated monitored stop
STOP
SMS
LEVEL 2 - Hand guiding
HG
LEVEL 3 - Speed and separation monitoring
SPEED
SSM
LEVEL 4 – Power and force limiting
PFL

Figure 1.2: The four collaborative operative modes identified by robot safety specifications (from [44])

Alongside these stringent safety requirements and protocols, there also lies an inherent optimization problem related to an efficient execution of the tasks. This entails a careful balance between upholding safety specifications and achieving operational efficiency.

1.3. Thesis purpose

In a collaborative context, the recognition and prediction of human activities must be exploited by the robot for decision-making. This thesis falls within the Design Methods addressing the challenges associated with HRC as it focuses on the development of a control logic allowing the robot to choose actions to be performed depending on the actual human position, while minimizing the distance between the actual robot configuration and the goal configuration. This approach find possible solutions which guarantees, at the same time, the maximum safety for the operator and an efficient execution of the task in terms of covered distance.

This thesis consists in an offline approach followed by an online validation. Initially, a database of admissible paths is created using the Bidirectional Rapidly Exploring Random Trees (BiRRT) algorithm to enable the robotic arm to navigate from its starting position to the designated target within the workspace. The BiRRT algorithm is designed to allow the manipulator to explore the configuration space by growing two trees with root nodes at the specified start and goal configurations in a static environment. Its random sampling strategy allows to efficiently cover large areas of the space, focusing on unexplored regions. Subsequently, the Reinforcement Learning method, in particular the Q-learning (QL) technique, is employed to enable the robot to dynamically select on which of the previously computed paths to travel and when to transition from a path to another, depending on the human worker presence. This approach aims at empowering the robot with the capacity to autonomously identify the path that strikes an optimal balance between ensuring a safe distance from the human worker and reducing time for the task completion. Through this adaptive strategy, the robot can navigate the environment prioritizing both safety and efficiency, adjusting its trajectory online to adapt to the dynamics of the human movements. The training and testing phases of the Q-learning algorithm utilize different sets of data from the Motion Capture (MoCap) Database HDM05, simulating human movements across different tasks. Upon the completion of the training phase, an optimal policy, characterized by an optimal Q-table, is derived and subsequently utilized during the testing phase. This policy encapsulates the most effective actions the robot should take in various states to achieve its objectives, balancing safety and efficiency based on the learned experiences. The optimal Q-table serves as a decision-making guide, enabling the robot to select actions that lead to the desired outcomes. Finally, the optimal policy is validated on the GoFa™ robotic arm in the MeRLIn Lab at the Politecnico di Milano. In the validation phase, a real-time detection of the human position guides the manipulator in choosing optimal actions to navigate the trees using the established optimal policy.

Typically, online path planning methods for robotic applications demand substantial computational resources as decisions are made in real time as the robot moves. These methods continuously analyze the environment, calculate possible routes, and make decisions on the fly. This real-time computation can be intensive, especially in complex and dynamic environments where obstacles or humans are moving, but they are useful when the robot is asked to perform repetitive tasks, quite common in industrial assembly lines. The novelty of this thesis lies in computing before all possible admissible paths the robot might take. Alongside mapping out all viable routes, this thesis also involves defining an optimal policy for the robot's navigation. As a consequence, when the robot is active and functioning, the algorithm no longer needs to perform complex calculations to decide its path. Instead, it simply observes the current condition, which corresponds to the current robot position and the current human position, and refers to the dataset of paths and to the optimal policy to quickly determine the best route to take. This drastically reduces the computational load during real-time operations, as the robot is effectively matching the current situation to a solution it already knows, rather than computing a new solution from scratch.

1.4. Thesis structure

This thesis is structured as follows:

- Chapter 1 presents an overview on the current industrial landscape, marked by the advent of digital technologies. Special attention is given on how the integration of collaborative robots into the production lines influences industrial operations.
- Chapter 2 offers a State of the Art review, focusing on the literature surrounding the most common path planning algorithms. Special attention is given to the Rapidly Exploring Random Trees algorithm and to the Reinforcement Learning method.
- Chapter 3 outlines the theoretical concepts behind the thesis work, describing the classes of models adopted to solve the robotic navigation and path optimization problems. This chapter is dedicated to a detailed description of the creation, expansion, and connection processes inherent to RRT. Furthermore, the chapter introduces the framework of Reinforcement Learning, focusing on its capacity to address sequential decision-making problems.
- Chapter 4 describes the application of the BiRRT algorithm to efficiently explore the robot's workspace and generate multiple feasible paths connecting the starting configuration to the goal configuration. The parent-child relationships graph is then

extracted from the dense interconnected network of paths and the essential nodes and their connections are identified.

- Chapter 5 outlines the formulation of the key components of a Reinforcement Learning problem and details the implementation of the Q-learning algorithm, enabling the robot to navigate both safely and efficiently. Moreover, this chapter presents an analysis of the algorithm's performances evaluated through a testing phase.
- Chapter 6 illustrates the experimental setup used to validate the proposed algorithm and describes the results achieved through the application of this work.
- Chapter 7 summarizes the conclusions of this study.

2 | State of the Art

2.1. Review of path planning algorithms

In recent years, the development of path planning algorithms for collaborative robots has garnered significant attention due to the growing integration of these robots into shared workspace with humans. The core challenge in designing these algorithms lies in balancing safety and efficiency. Ensuring safety involves maintaining a safe distance between the human worker and the robot at all times, preventing contacts or injuries. It is crucial to optimize task execution time to adhere to industrial production standards, and the complexity of achieving these goals has led to the development of various strategies and classes of algorithms.

The methodology introduced by Balan *et al.* [5] addresses the challenges of navigating through the shared workspace thanks to a sophisticated human occupancy stochastic model. The proposed solution uses a sphere-based geometric model to represent both the human and the robot, making it easier to calculate distances between them. This model predicts the likelihood of the human presence in different parts of the workspace at various times. By incorporating probability and uncertainty into the planning process, the robot can make informed decisions that minimize the risk of collision. This method allows for dynamic adjustment of the robot's path in real-time, based on the predicted movements and positions of human workers. A crucial part of this method is predicting where the human and the robot will move next, reducing problems caused by the robot's slow reactions. The robot's future movements are estimated with a mathematical model that understands how it responds over time. At the same time, human movements are predicted by looking at the average of where they have moved before.

A different approach, proposed in [20], solves the path planning problem focusing on task allocation. Instead of directly modifying the robot's path to avoid human workers, this strategy assigns tasks to the robot in a manner that inherently avoids collisions. The robot is programmed to engage in activities that do not intersect with human actions and are optimized to reduce idle time. This approach relies on a strategic overview of the workspace dynamics, where tasks are allocated based on current and anticipated human actions. As a result, the robot contributes productively to the task without the need for continuous path adjustments. Instead, in the study referenced by Ragaglia *et al.* [34], a sensor fusion algorithm integrates data from multiple depth sensors to generate a precise representation of a human worker's movements within the robotic environment. This representation helps to predict the areas the worker might occupy, taking into account the robot's stopping time, and models these areas as convex swept volumes. Utilizing this predictive model, the controller strategically alters the predetermined robot's trajectory. These adjustments ensure that safety is maintained by steering clear of these volumes, thereby minimizing any disruption to the task at hand.

An innovative approach, which inspired the development of this thesis, is proposed by Pellegrinelli *et al.* [32]. The time needed by the robot to execute its trajectory is computed, incorporating a probabilistic model of the human presence. This research aims at establishing a confidence interval for the robot's trajectory execution time in environments where human-robot interaction is essential. The human arm movements in specific collaborative assembly tasks are analyzed to determine the volumes and probabilities of the worker occupancy. In this research, the optimal path is chosen from a dataset prepared offline, designed to avoid areas frequently occupied by humans and maintain a safe distance from the worker's space. This process enables the estimation of the likelihood of the robot reducing its speed, along with a confidence interval for its overall execution time, as it can be seen in Figure 2.1.

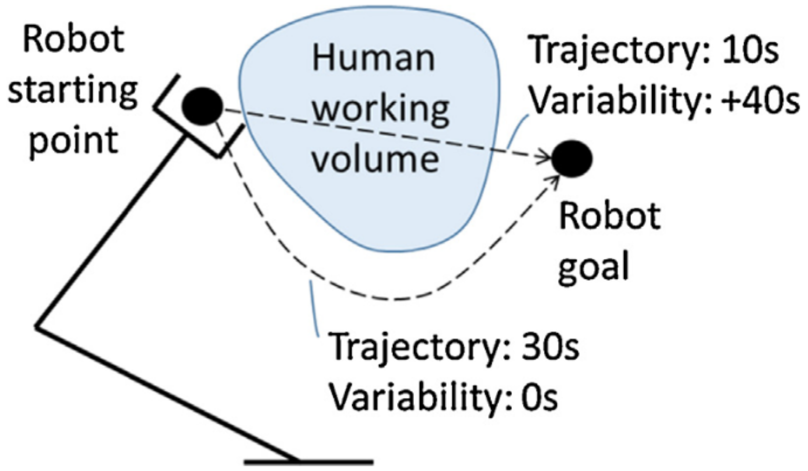


Figure 2.1: The shorter path crosses a zone where the human could be, and its execution time is subject to variations due to possible robot stops to guarantee safety (from [32])

A similar approach [3] introduces a virtual model of the collaborative workspace using a 3D data structure known as a voxel-grid. This technique, highlighted for its relevance to the thesis, focuses on robot trajectory planning with an emphasis on avoiding human-populated areas and maintaining safety margins. Each voxel represents a volumetric element within the workspace, allowing the robot to distinguish between free and occupied spaces. This voxel-grid approach serves for efficiently navigating the shared workspace, ensuring the robot avoids areas that are occupied by humans and maintains a safe distance from previously human-occupied zones.

An interesting class of methods widely used in robotic navigation and obstacle avoidance is represented by Artificial Potential Field (APF) algorithms. Drawing inspiration from the concept of potential fields in physics, APF algorithms are formulated to guide the robot by simulating a virtual force field, allowing for continuous adjustments of the robot movements and enabling real time responses to changes in the surrounding environment. The overall force acting on the robot is a combination of attractive forces towards the goal and repulsive forces to avoid obstacles. The study presented in [36] is dedicated to online collision avoidance for collaborative manipulators by adjusting offline generated paths. The approach presented involves representing both the human coworker and the robot as geometric shapes to facilitate the use of hypothetical repulsion and attraction vectors to manage the space between humans and robots effectively. By integrating these vectors with the robot's kinematics, the robot is able to dynamically alter its predetermined path to avoid collisions with humans, ensuring safety without compromising the completion of industrial tasks.

Other classes of path planning algorithms worth analyzing in the context of path planning with collision avoidance are grid-based search algorithms and visibility graph algorithms. Grid-based search algorithms discretize the robot's configuration space into a grid of cells, where each cell represents a possible configuration or pose of the robot. The purpose of grid-based search algorithms is to find a collision-free path for the robot to move from an initial configuration to a goal configuration within this grid map. The paper by Lau *et al.* [21] introduces novel algorithms for dynamic environment navigation in robotics, focusing on efficiently updating grid maps like Euclidean distance maps, generalized Voronoi diagrams, and configuration space maps. By incrementally updating only the affected cells, these algorithms enhance collision checking and path planning. The methods presented in the paper have been validated through real-world data experiments, showing significant improvements in computational efficiency and adaptability to complex structures. On the other hand, visibility graph algorithms aim at creating a graph by considering a direct line-of-sight connections between different points in the environment, allowing a robot to plan paths avoiding collisions with obstacles. While visibility graph algorithms are not exclusively designed for collaborative robotics, they are especially useful in scenarios where robots share a workspace with humans. The ability to plan paths based on visible connections aids in creating trajectories that are not only collision-free but also consider the natural movement patterns and limitations of both robots and humans. This enhances safety and efficiency in collaborative settings. The paper by Blasi [6] introduces an innovative approach for creating optimal flight paths for unmanned aircraft, navigating around obstacles and no-fly zones with a real-time collision avoidance algorithm. The proposed algorithm employs the Essential Visibility Graph for minimum cost piecewise linear path search and incorporates a re-planning procedure to update paths in dynamic environments. The use of Dubins curves ensures smooth, flight mechanics-compliant paths. Numerical simulations across diverse scenarios highlight the algorithm robustness, adherence to collision avoidance standards, and suitability for real-time deployment due to to its minimal computational requirements.

Two different classes of algorithms fundamental for the development of this thesis are sampling-based algorithms and reward-based algorithms. Sampling-based algorithms algorithms focus on generating a discrete set of samples in the robot's configuration space and building a feasible path through the connections between these samples. Probabilistic Roadmap (PRM) and Rapidly Exploring Random Tree (RRT) are two types of sampling-based algorithms. These algorithms provide a flexible and computationally efficient way to plan paths in spaces with complex geometry and obstacles. However, they are time consuming, since re-planning when the initial position changes is required and in some cases, a feasible solution cannot be found. Moreover, Karaman and Frazzoli's research [16] deals with the asymptotic behavior of PRM and RRT algorithms. Their analysis uncovers that, under commons conditions, these algorithms frequently yield to solutions that are not optimal. To address this issue, PRM* and RRT* (where the symbol "*" stands for the enhanced version) algorithms are introduced: being asymptotically optimal, they ensure that the solution converges to the optimum as the number of samples increases, without significantly increasing computational complexity.

On the other hand, reward-based algorithms algorithms, based on the mathematical theory known as Markov Decision Process (MDP), provide a powerful framework for enabling robots to learn and optimize their behavior in collaborative environments. MDP models the system as a set of states, actions, transitions, and rewards. States represent the different conditions of the system, actions are the choices available to the robot, transitions describe the probability of moving from one state to another based on a chosen action and rewards are numerical values assigned to every state-action pair in the environment. Reward-based algorithms provide a positive reward to associate desirable actions with positive outcomes and negative reinforcement to link undesired actions with negative outcomes. The ability of the robot to learn from negative outcomes enhances safety by discouraging actions that may lead to undesirable consequences. The main feature of reward-based algorithms is to determine the optimal sequence of choices that leads to the maximum cumulative reward. In the specific context of HRC, rewards can be designed to encourage safe, efficient, and collaborative behaviour.

In this thesis, the RRT algorithm, categorized under sampling-based algorithms, and the Reinforcement Learning method, which belongs to the family of reward-based algorithms, are coupled to solve the path planning problem for the GoFa™ robotic manipulator. With respect to already mentioned contributions, the objective of this work is to develop a dataset of known, precomputed and feasible paths. By discretizing the shared workspace into voxels, a precise identification of the human positions is ensured. Based on this spatial discretization, along with a comprehensive graph of potential paths, the robot is equipped to learn, through an offline training phase within a simulated environment, which route to select, balancing safety and efficiency.

2.2. Rapidly Exploring Random Tree algorithms for path planning

Sampling-based algorithms have proven to be effective in addressing numerous complex, high-dimensional motion planning problems for articulated manipulators. Rapidly Exploring Random Tree algorithms (RRT), through a process of random sampling of points in the configuration space, construct a graph connecting sampled points which represents potential collision-free paths. This ensures the algorithm's efficiency in navigating the space and discovering viable paths for the robot. Since its formulation, the RRT algorithm has undergone significant improvements, evolving into an highly efficient tool for specific path planning problems. The majority of the RRT algorithms found in the literature provide feasible paths guaranteeing only that there is no contact between the robot and the obstacles. More sophisticated RRT-based algorithms impose extra safety oriented conditions on the generated path, such as minimizing proximity to specific areas whenever possible, adding an extra layer of safety to the paths they generate.

In [17] an anytime motion planning algorithm using the RRT* is introduced, focusing on enhancing efficiency and safety of HRC in industrial settings. It emphasizes the RRT* algorithm's ability to quickly generate an initial feasible solution and iteratively refine it towards an optimal solution without significant computational overhead. Through various experiments, the algorithm demonstrates superior performance in dynamically adjusting paths in real-time to avoid collisions with humans while maintaining task efficiency.

Instead, the approach discussed in [30] introduces the real-time version of RRT*, an innovative method for path planning in dynamic environments, which exploits online tree rewiring. This technique enables real-time adaptation of paths to changing goals and emerging obstacles, ensuring continuous alignment with the agent's current position. The algorithm's ability to re-calibrate routes swiftly, without discarding previous computations, significantly enhances its efficiency. Through comparative studies, RT-RRT* is demonstrated to surpass conventional methods, providing faster, adaptable solutions for scenarios demanding immediate responsiveness in complex settings.

The paper by [28] introduces the Particle RRT algorithm, an enhancement of the RRT path planning algorithm, incorporating uncertainty handling similar to particle filters. This method treats each search tree extension as a stochastic process, allowing for simulation of multiple outcomes. It enables characterization of robot behavior under specified environmental uncertainties, offering performance guarantees. Paths are selected based on the likelihood of successful execution, with benefits demonstrated through a rover simulation navigating rough terrain with uncertain friction coefficients.

The study proposed by Lacevic *et al.* [19] utilizes the RRT framework to incorporate a kinetostatic danger field to guide the expansion of the trees towards safer regions, ensuring that the planner not only delivers collision-free paths but actively pursues safer alternatives. Two modifications of RRT-based planner are presented: a unidirectional RRT algorithm using the Jacobian transpose for targeted growth, and a bidirectional RRT-connect approach rooted in specific start and goal configurations. Simulations demonstrate the algorithms effectiveness in enhancing path safety over traditional methods.

2.3. Reinforcement Learning methods for path planning

Reinforcement Learning (RL) is one of the most effective approaches to solve complex goal-oriented problems from interaction and it has been rapidly evolving in the last decades, marked by significant progresses in algorithms development, practical applications, and theoretical understanding. Reinforcement Learning has been widely used in many fields, including robot control [18], autonomous driving [2], games theory [39], computer vision [23], and natural language processing [37].

Within the framework of control theory, the application of RL is integrated with a simulation phase. This step is prior to real-world deployment and it is useful to improve the effectiveness and safety of RL techniques. By harnessing controlled virtual environments for algorithm training and refinement, simulation enables robots to achieve remarkably precise and efficient task execution. The approach proposed in [25] explores the development and training of a digital twin for a robotic arm using RL within a virtual environment created in Unity. This approach seeks to enhance AI training efficiency by simulating real-world observations and applying the acquired knowledge to a twin. This research demonstrates the potential of digital twins in robotics, offering insights into training protocols and the mapping of virtual learning to physical applications. The paper by James *et al.* [13] explores the integration of deep Q-learning and 3D simulation to train a 7-DOF robotic arm to perform a control task without prior knowledge. The task involves locating, grasping, and lifting a cube using images of the environment as input. This innovative approach allows the robotic arm to learn and execute the task without any preprogrammed knowledge, showcasing the potential of combining advanced Machine Learning techniques with virtual simulations for robotic training. Furthermore, the research in [27] investigates the use of different RL algorithms for the precise control and positioning of robotic arms, focusing on how these algorithms can improve accuracy and efficiency in task execution. By examining four RL algorithms across six distinct setups, this work evaluates their performance in terms of positioning accuracy, motion trajectory, and the number of steps required to achieve the goal. Simulations and real-world tests with a robot were conducted to compare the effectiveness of these RL algorithms in practical scenarios, demonstrating that RL can be successfully applied to robotic arm control tasks. An interesting application is described in the paper by Shehawy *et al.* [38], where a single-arm robot employs Reinforcement Learning for the task of flattening and folding a piece of cloth. The Deep Deterministic Policy Gradient (DDPG) algorithm is employed to instruct the robot on the best actions to take based on the state of the cloth. As a result, the robot was able to exploit the collected experiences to learn how to flatten and fold a towel with a constrained edge. Concurrently, computer vision technology is applied to observe and determine the cloth condition. This work details the creation of a simulation environment and the development of a policy as well as methods for cloth corner recognition using computer vision, comparing traditional and deep learning approaches. The training phase showed successful results both in the simulations and in the real-world application, conducted using an ABB robot and a 2D camera.

In the context of Reinforcement Learning, Q-learning is used for mobile robot navigation due to its simple and well-developed theory. However, in the literature, it is not frequently applied for solving the robot arm path planning problem due to the necessity to account for the motion of each individual joint. In this thesis, this complexity is overcome by previously establishing a database of feasible paths for the robot arm using the RRT algorithm. By adopting this strategy, the robot joints configurations associated with each point along these paths are already known and they are defined as the states for the QL problem, while the actions are defined as the transitions between different configurations. In this thesis, the problem formalization was inspired by the approach proposed in [4], where the robot locations and orientations are defined as states for the Q-learning. The tests conducted reveal that, in nearly all scenarios, the QL generated path guarantees minimal orientation error when the robot reaches the target point. Nevertheless, the resultant path lacks in smoothness due to the discretization of states and actions. A similar approach for mobile navigation in the dynamic environment is discussed in [14]. The relative position and angle between the robot's current pose and the goal pose are defined as states for the QL method. This choice allows to operate effectively in an unknown dynamic environments while maintaining a compact Q-table. The experimental tests reveal that the paths generated by this method exhibit remarkable smoothness, with a success rate of *98%* for the robot reaching the target pose along a collision-free trajectory.

Also in [15], the C space of the robot undergoes a discretization. As the movements of the robot joints must be coordinated to navigate around obstacles, the joint state space is defined as a vector containing all joint information while the action space is composed by the angular resolutions for each region. The reward function is used to evaluate an action taken at a particular state and it is designed to impose a penalty when a collision occurs. To mitigate the possibility of collisions between the robot arm and the obstacles, the algorithm responds by assigning a negative infinite feedback value, effectively steering the robot away from potential obstacles. In addition, the reward function is designed to consider the distance from the current position to the target position within the configuration space. This ensures that the algorithm not only prioritizes obstacle avoidance but also takes into account in the spatial efficiency of the robot movements towards its goal. According to this criterion, two terminal states and one transient state are defined. A state is categorized as winning (WS) if the robot reaches the goal, as failure (FS) if the robot collides with any obstacle, and as safe (SS) if the robot is located in safe regions without obstacles. The reward function is set equal to *-inf* if the robot transitions from an SS to an FS, it equals *10* if the robot moves from an SS to a WS, and it is calculated as -$\mu \cdot D$ if the robot shifts from one SS to another SS, where D represents the Euclidean distance between the safe states and μ is the penalty coefficient.

3 | Theoretical background

3.1. Introduction

To gain a comprehensive understanding of the tools and methodologies employed in this thesis, a theoretical foundation is provided. This chapter provides a general overview of the Rapidly Exploring Random Tree family of algorithms, analyzing the main features and operational processes of standard RRT, RRT* and BiRRT. Additionally, it explores the Reinforcement Learning technique, with a specific focus on Q-learning method.

3.2. Rapidly Exploring Random Tree algorithm

A wide variety of path planning algorithms that address the navigation problem in a static environment is present in the literature. Compared to other path planning algorithms, the Rapidly Exploring Random Tree is advantageous for its flexibility, its simplicity of implementation and its speed in finding a solution in complex environments, both creating a graph and finding a feasible path. The RRT is a tree-based motion algorithm designed to efficiently explore non-convex, high-dimensional spaces, by building a search tree incrementally from samples randomly selected from a given state space. The RRT key feature lies in its randomized nature, which allows it to explore complex configuration spaces effectively by growing a tree from an initial state towards randomly sampled states. This exploration strategy is effective in handling environments with obstacles and unknown spaces, making RRT well-suited for real-world robotic scenarios.

3.2.1. RRT algorithm

The operational process of the Rapidly Exploring Random Tree involves a systematic exploration of the configuration space by dynamically growing a tree structure to discover feasible paths from an initial configuration to a desired goal configuration. The efficiency of the algorithm lies in its capacity to adapt and extend the tree based on the random sampling of successive configurations. Algorithms 3.1 and 3.2, illustrated in [7], present the pseudo-codes corresponding to the process of generation and expansion of the tree structure. The algorithm consists in sampling randomly a node x_{rand} in the configuration space C, which defines the set of all feasible positions and orientations the robot can assume during its movements, and identifying an existing node x_{near} in the search tree that is the closest to x_{rand}. If the distance between x_{near} and x_{rand} is higher than a threshold ϵ, the planner expands from x_{near} towards x_{rand}, until a state x_{new} is reached. Otherwise, the new state x_{new} is equal to x_{rand}. The new state x_{new} is added to the search tree and the expand operation, illustrated in Figure 3.1, continues until the final node x_{goal} is reached. The connection between x_{near} and x_{new} is set only after a check that the edge connecting the two nodes lies entirely within the collision-free space C_{free}. These steps are repeated until the final state is hit or the maximum number of iterations is reached. If the final state is reached, the series of connected nodes which form a path from the starting configuration to the goal configuration is extracted from the RRT data structure.

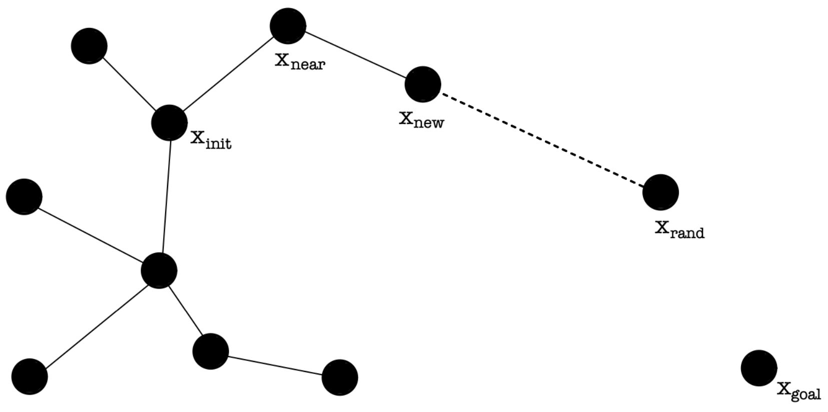


Figure 3.1: The expand operation

Algorithm 3.1 RRT Algorithm - Build

```
Require: $x_{start}$, $x_{goal}$
1: $T$ ← Initialize tree with $x_{start}$
2: for $k$=1 to $K$ do
3:     $x_{rand}$ ← Sample random node
4:     $E$ ← Expand
5:     if $E$ = Reached then
6:         return $T$
7:     end if
8: end for
9: return Failure
```

Algorithm 3.2 RRT Algorithm - Expand

```
Require: $T$, $x_{rand}$, $x_{goal}$
 1: $x_{near} \leftarrow$ Nearest neighbour node
 2: if $|x_{near}$ - $x_{rand}$ $| < \epsilon$ then
 3:     $x_{new} = x_{rand}$
 4: else
 5:     $x_{new} = x_{near} + \epsilon$
 6: end if
 7: $e \leftarrow$ Edge from $x_{near}$ to $x_{new}$
 8: if $e \in C_{free}$ then
 9:     $T \leftarrow$ Add $x_{new}$ to the tree
10:     $T \leftarrow$ Add $e$ to the tree
11:     if $x_{new} = x_{goal}$ then
12:         return  $Reached$
13:     else
14:         return  $Advanced$
15:     end if
16: end if
17: return  $Trapped$
```

The advantages of the RRT are illustrated in [22]:

- It is robust in scenarios where the environment is not entirely known, as it relies on random sampling and exploration to discover feasible paths.
- The expansion of the tree is strongly directed towards unexplored regions of the configuration space by biased random sampling.
- The distribution of the nodes tends to a uniform sampling distribution, meaning that the nodes are evenly spread throughout the configuration space, leading to a consistent behavior.
- The tree always remains connected, even if the number of nodes is low.
- It is probabilistically complete, meaning that as the number of iterations approaches infinity, the algorithm is guaranteed to find a solution if one exists.
- It is straightforward to implement and can be included into a wide variety of planning systems, including both single-query and multi-query scenarios, as well as problems with dynamic obstacles.

Finally, the use of Voronoi regions in conjunction with uniform sampling enhances the efficiency of the motion planning algorithm by guiding the exploration towards regions that are more likely to yield valuable paths. This bias promotes a rapid and uniform exploration of the configuration space, avoiding unnecessary exploration of less promising regions and leading to a faster and more efficient search for a solution. Figure 3.2 demonstrates how the algorithm, with the introduced bias, explores the configuration space more rapidly and uniformly compared to a scenario without such bias.

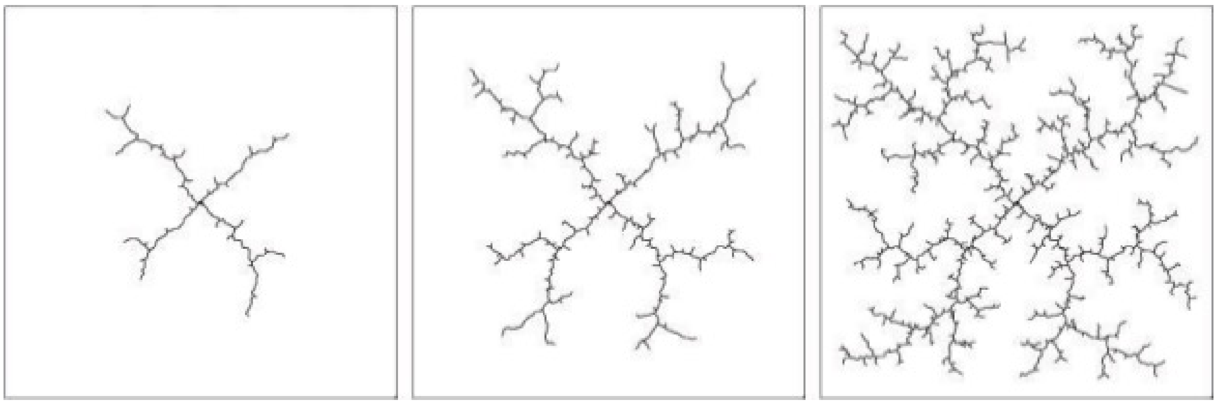
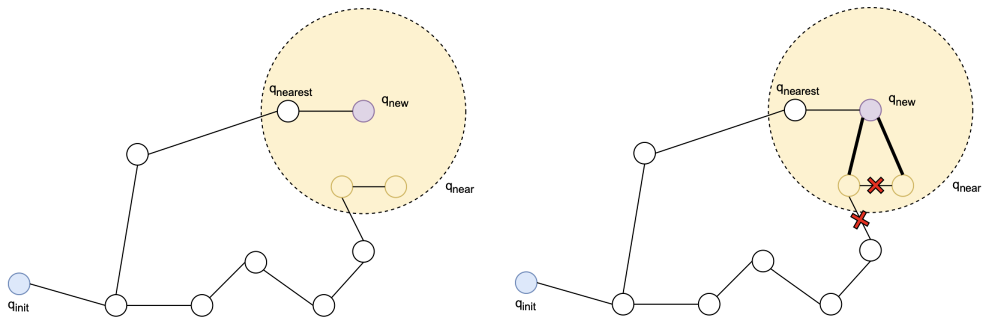


Figure 3.2: Voronoi Regions as the RRT explores the configuration space (from [22])

3.2.2. RRT* algorithm

The RRT* is the optimized version of the RRT algorithm, which produces smoother graphs and finds the shortest path connecting the starting point to the goal, whether considering the distance or other metrics. Unlike RRT, which tends to generate sub-optimal paths due to the cubic graph structure, RRT* aims at improving the optimality of the generated paths. The basic principle of RRT* is the same as RRT, but an improved search of the nearest neighbor node and the rewiring of the tree, produce in significantly better results. To find the optimal solution, the RRT* algorithm considers a cost function C_{min} associated to each node, defined as the cumulative cost of reaching that node from the starting point. This cost function is used to evaluate and compare different paths, facilitating the selection of more optimal routes. Instead of connecting the newly generated node only to its nearest neighbor, RRT* considers a set of nearby nodes within a certain radius. In the rewiring phase, if an alternative path offers a lower cumulative cost than the existing connection, the algorithm re-configures the tree by rewiring the nodes. This means establishing a new, more optimal connection between the nodes to improve the overall quality of the generated paths. Figure 3.3 shows the advantage of the rewiring of the tree and Algorithms 3.3 and 3.4 present the pseudo-codes of the illustrated procedure.

The rewiring step is performed iteratively as the algorithm progresses through multiple iterations. This continuous adjustment process allows RRT* to refine the tree structure over time, ultimately converging towards smooth paths that are optimal in terms of the defined cost function. RRT* is designed to be asymptotically optimal, meaning that as the number of iterations increases, the probability of finding the optimal solution approaches the unit. The drawbacks of RRT* are its time and computational expenses, consequences of the continuous re-connections of the tree.



(a) Before rewiring (b) After rewiring

Figure 3.3: Tree connections before and after rewiring

Algorithm 3.3 RRT* Algorithm

```
Require: $x_{start}$, $x_{goal}$
 1: $T$ ← Initialize tree with $x_{start}$
 2: for $k$=1 to $K$ do
 3:     $x_{rand}$ ← Sample random node
 4:     $x_{nearest}$ ← Nearest neighbour node
 5:     $x_{new}$ ← Steer between $x_{rand}$ and $x_{nearest}$
 6:     $e$ ← Edge from $x_{nearest}$ to $x_{new}$
 7:     if $e \in C_{free}$ then
 8:         $T$ ← Add $x_{new}$ to the tree
 9:         Rewiring
10:     end if
11: end for
12: return $T$
```

Algorithm 3.4 RRT* Algorithm - Rewiring

```
Require: $x_{new}$, $x_{near}$, $x_{nearest}$
 1: $x_{min} \leftarrow x_{nearest}$
 2: $C_{min} \leftarrow \text{cost}(x_{nearest}) + \text{edge}(x_{nearest}, x_{new})$
 3: for all $x_{near} \in X_{near}$ do
 4:     if $e \in C_{free} \wedge cost(x_{near}) + edge(x_{near}, x_{new}) < c_{min}$ then
 5:         $x_{min} \leftarrow x_{near}$
 6:         $C_{min} \leftarrow \text{cost}(x_{near}) + \text{edge}(x_{near}, x_{new})$
 7:     end if
 8: end for
 9: $E \leftarrow E \cup (x_{min}, x_{new})$
10: for all $x_{near} \in X_{near}$ do
11:     if $e \in C_{free} \wedge cost(x_{new}) + edge(x_{near}, x_{new}) < cost(x_{near})$ then
12:         $x_{parent} \leftarrow \text{Parent}(x_{near})$
13:         $E \leftarrow E \setminus (x_{parent}, x_{near}) \cup (x_{new}, x_{near})$
14:     end if
15: end for
16: return $E$
```

3.2.3. Bidirectional RRT algorithm

Bidirectional RRT, also known as BiRRT, is an extension of the standard RRT algorithm which explores the configuration space using two trees T_A and T_B starting from both the starting node and the goal node simultaneously, and if the trees meet, a solution is found. Algorithms 3.5 and 3.6 present the pseudo-codes corresponding to the process of generation, expansion and connection of the two trees structures. The difference with respect to the standard RRT algorithm is that, after the expand operation of each single tree, the best node within T_B is selected from the sorted list such that it provides collision-free low-cost connection between the two trees. If the operation is successful, the algorithm returns the end-to-end feasible path solution, connecting the starting and goal nodes. Otherwise, the roles of the trees are swapped and another connection is tried, until one is established. The main advantage of BiRRT lies in its ability to find a feasible path with significantly lower number of iterations with respect to the standard RRT, especially in scenarios where the start and goal configurations are far apart or the environment to be explored is more complex. Moreover, BiRRT can help in avoiding dead ends in the configuration space. If one tree encounters an obstacle or reaches a dead end, the other tree might continue exploring and finding a valid path to the goal.

Algorithm 3.5 Bidirectional RRT Algorithm

```
Require: $x_{start}$, $x_{goal}$
1:  $T_A \leftarrow$ Initialize tree start with $x_{start}$
2:  $T_B \leftarrow$ Initialize tree goal with $x_{goal}$
3:  for $k$=1 to $K$ do
4:      $x_{rand} \leftarrow$ Sample random node
5:      $E \leftarrow$ Expand $T_A$ towards $x_{rand}$
6:      if $E \neq$ Trapped then
7:          S $\leftarrow$ Connect
8:          if $S$ = Reached then
9:              return Path
10:         end if
11:     end if
12:     $(T_A, T_B) \leftarrow$ Swap
13: end for
14: return Failure
```

Algorithm 3.6 Bidirectional RRT - Connect

```
Require: $T_A$, $T_B$, $x_{new}$
1: $x_{near} \leftarrow$ Nearest Neighbour within $T_B$
2: $E \leftarrow$ Extend from $x_{near}$ to $x_{new}$
3: return $E$
```

3.3. Reinforcement Learning method

Everyday humans have to take decisions. Decisions are sometimes driven by the pursuit of an immediate gratification, while on other occasions, we carefully weigh options, sacrificing short-term pleasures for enduring satisfaction. Additionally, we are not immune to errors, often making choices that result in undesirable outcomes. Yet, our ability to learn from mistakes sets us apart as our brains, working as adaptive learning systems, assimilate the consequences of our actions, gradually refining our decision-making processes. Our brains learn by interacting with the environment in an endless sequence of trial and error. We are likely not to repeat actions causing a negative reinforcement. Conversely, behaviors associated with positive reinforcement tend to be revisited, showing our inclination to repeat actions that have brought us satisfaction in the past. Reinforcement Learning focuses on developing algorithms and models to enable agents to make intelligent decisions in an environment, being inspired by how humans and animals learn through interaction with their surroundings.

Reinforcement Learning (RL) is defined by Sutton and Barto [41] as *"learning what to do - how to map situations to actions - so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them"*. A Reinforcement Learning problem is generally formalized as a Markov Decision Process (MDP), which will be analyzed in detail in Section 3.3.1. The agent interacts with the environment and learns to make sequential decisions to achieve a specific goal by receiving feedback from the environment in the form of rewards or penalties based on the actions it takes. The agent's aim is to learn an optimal policy, a mapping between states and actions, that maximizes the expected cumulative reward over time. This learning process often involves exploration, where the agent tries different actions to discover their consequences, and exploitation, where the agent uses the learned policy to choose actions that are expected to yield high rewards.

Reinforcement Learning, supervised learning, and unsupervised learning are three fundamental approaches within the field of Machine Learning, each aiming at distinct purposes and addressing specific types of learning problems. Supervised learning [10] relies on a labelled datasets to train algorithms, enabling accurate classification of data and precise prediction of outcomes. The model refines its weights as input data are introduced, achieving an appropriate fit during the cross-validation process. An example of supervised learning are Neural Networks, which learn from data through a process called training. During training, the network is presented with labeled examples, and it adjusts its weights and biases to minimize the difference between its predictions and the actual outcomes. This is typically done using optimization algorithms like gradient descent. In unsupervised learning [11], instead, the algorithm explores the inherent structure or patterns within the data without specific guidance on what to find. The goal is to discover hidden patterns, group similar data points, or reduce the dimensionality of the data. An example of unsupervised learning algorithm is K-means clustering, which is used for partitioning a dataset into K distinct, non-overlapping subgroups or clusters. The goal of K-means is to group similar data points together and assign them to a cluster, such that the variance within each cluster is minimized. Algorithms used in RL include Q-learning, Policy Gradient methods, and Deep Reinforcement Learning, where Neural Networks are employed to handle complex state-action spaces.Reinforcement Learning has recently received considerable attention due to its successful application in several fields, including game theory, operations research, combinatorial optimization, information theory, simulation-based optimization, control theory, and statistics.

3.3.1. Markov Decision Process

RL algorithms operate on systems that are formalized as Markov Decision Processes (MDP), which are mathematical frameworks used for modeling sequential decision-making problems. In a MDP, the agent is the decision-maker or learner of the system and it takes actions in response to the current state of the environment, with the goal of maximizing the cumulative reward. The states represent the different configurations the environment can be in and the agent's perception of the environment at a given time is captured by the current state. The reward is a numerical value associated with the transition from one state to another based on the agent's action and it provides a feedback to the agent, indicating the immediate desirability or undesirability of the action taken. The agent's goal is to learn a policy that maximizes the expected cumulative reward over time. A policy is a mapping from states to actions, so it defines the agent's behavior, specifying the action to be taken in each possible state.

In this thesis, the focus will be on discrete-time MDP, so, at a generic time step t, the interaction dynamics between the agent and the environment, illustrated in Figure 3.4, can be described as:

- The agent observes the current state of the environment $S_t \in S$, where S is the set of all possible states.
- Based on the observed state, the agent selects an action $A_t \in A(S_t)$, where $A(S_t)$ is the set of all possible actions enabled at state S_t.
- The environment responds to the action by transitioning to a new state S_{t+1} and provides a numerical reward $R_{t+1} \in R \subset \mathbb{R}$ to the agent based on the transition.

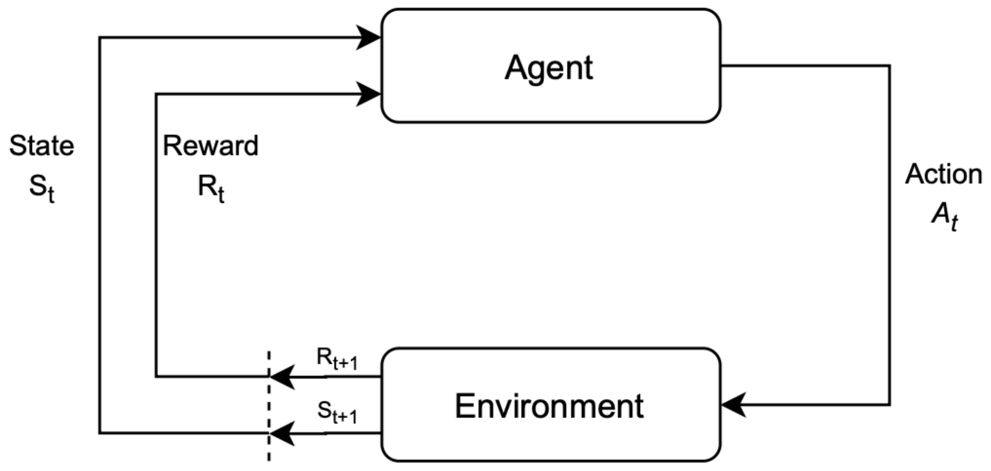


Figure 3.4: Agent–environment interaction in a Markov Decision Process (from [41])

At each step, the policy π_t guides the agent's decision-making process, influencing the selection of the action at that state. The agent's objective in a Reinforcement Learning setting is to maximize the return, a function representing the sum of future rewards that the agent seeks to optimize in terms of expected value. The definition of the return varies based on the task's nature and whether the problem involves discounting delayed rewards. The undiscounted formulation, presented in Equation (3.1), is suitable for episodic tasks, where the agent-environment interaction naturally breaks into episodes. In this case, the return is computed by performing the cumulative sum of the rewards over the time sequence until time T. Conversely, the discounted formulation, whose formal definition is given in Equation (3.2), is well-suited for continuing tasks, where the interaction persists without a natural episodic structure. The return is computed by performing the cumulative sum of the rewards over time, while incorporating a discount rate γ, which varies between *0* and *1*. This factor reflects the agent's preference for immediate rewards over future rewards. In particular, a higher γ tends to prioritize long-term rewards, encouraging the agent to consider future consequences more heavily, while a lower γ places greater emphasis on immediate rewards.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ...R_T \tag{3.1}$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.2}$$

A property of particular interest in a Markov Decision Process is the Markov property, which ensures that the future state depends only on the current state and action, not on the sequence of events that preceded it, simplifying the modeling and analysis of sequential decision-making problems. Moreover, in a finite MDP, the set of states, actions, and rewards are bounded by a finite number of elements. This allows the random variables of the current state S_t and reward R_t to be precisely described by a discrete probability distributions, which depend only on the immediately preceding state S_{t-1} and action A_{t-1}. Equation (3.3) defines the transition dynamics function p, which completely characterizes the dynamics of the environment of the finite MDP:

$$p(s', r \mid s, a) = Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{3.3}$$

3.3.2. Policies and value functions

The brain of the RL agent is the policy, which is defined as a mapping from states to probabilities of selecting each possible action. The policy can be deterministic, if, for each state, there is a single corresponding action the agent will always take: $a = \pi(s)$, or stochastic, if the policy defines a probability distribution over the set of actions given a state: $\pi(a \mid s) = Pr[A \mid s]$. This last expression can be interpreted as the likelihood to execute action a if the state is s. Alongside the concept of policy, Reinforcement Learning algorithms involve the estimation of value functions, which are used to evaluate and compare different policies, by defining how good it is for an agent to be in a particular state or to take a specific action in a given state. By estimating the expected cumulative rewards associated with states or state-action pairs, one can discern the quality of different strategies. Once value functions are computed, they can be used to iteratively improve the policy. Actions or policies that lead to higher values can be favored over those with lower values. The value function $v_\pi(s)$ of a state s under a policy π is the expected return when starting in s and following the same policy afterwards. For a MDP, the value function is defined by Equation (3.4), where $\mathbb{E}_\pi[\cdot]$ represents the expected value of a random variable given that the agent follows the policy π.

$$V_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \tag{3.4}$$

The function V_π is called state-value function for policy π.
Similarly, it is possible to define the action-value function for policy π, as the value associated with taking a specific action a being in the state s under the policy π, denoted $Q_\pi(s,a)$. Equation (3.5) represents the expected cumulative reward starting from state s, executing action a and subsequently following the policy π.

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s\,, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \tag{3.5}$$

The value functions $V_\pi(s)$ and $Q_\pi(s,a)$ can be estimated based on the agent's experiences in the environment. In the following, Bellman equation is introduced to simplify the state-value or state-action value calculations. Equation (3.6), named after the mathematician Richard Bellman, expresses a recursive relationship between the value of a state or state-action pair and the values of its successor states [40]. The idea of the Bellman equation is that instead of calculating each value as the sum of the expected return, which is a long process, each value is given by the sum of the immediate reward plus the discounted value of the state that follows.

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma\ v_\pi(S_{t+1}) \mid S_t = s] \tag{3.6}$$

Solving a Reinforcement Learning problem means, roughly speaking, finding a policy that achieves a the maximum reward over an extended period [8]. For a finite MDP, a policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all the states. However, multiple policies that are considered optimal may exist, called $\pi*$. They share the same optimal state-value function, denoted $V*$ and defined in Equation (3.7).

$$V^*(s) = \max_\pi V_\pi(s) \tag{3.7}$$

Similarly, optimal policies share the same optimal action-value function, denoted $Q*$, and defined in Equation (3.8).

$$Q^*(s, a) = \max_\pi Q_\pi(s, a) \tag{3.8}$$

The importance of the Q-value function with respect to the V-value function is that the optimal policy can be obtained directly from the same optimal action-value function, as stated in Equation (3.9).

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s, a) \tag{3.9}$$

3.3.3. Monte Carlo and Temporal Difference methods

Monte Carlo (MC) and Temporal Difference (TD) are two different strategies to estimate value functions, which quantify the expected cumulative rewards associated with being in a certain state or taking a specific action. Both methods are used to learn from experiences and improve the decision-making of an agent in an environment.

Monte Carlo methods are well-suited for episodic tasks, where the agent-environment interaction naturally breaks into episodes, and each episode terminates in a terminal state. It requires a complete entire episode of interaction before updating our value function as stated in Equation (3.10), where α is the learning rate. The value of a state or state-action pair is estimated by averaging the returns observed in multiple episodes, providing a statistical estimate of the expected reward for a state or state-action pair.

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right] \tag{3.10}$$

On the other hand, Temporal Difference methods estimate value functions by updating their estimates after each time step based on the observed experiences rather than waiting for an entire episode to be completed. This accelerates learning in online applications. TD methods combine the principle of Monte Carlo methods (which estimate values from a complete episode) and dynamic programming (which involves bootstrapping to update estimates based on other estimates). The key concept in TD methods is the use of temporal differences, representing the difference between the estimated value of a state or state-action pair and the immediate reward plus the estimated value of the next state or state-action pair, as defined in Equation (3.11).

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma \; V_\pi(S_{t+1}) - V(S_t) \right] \tag{3.11}$$

TD methods, such as Q-learning, which will be extensively described in the following Section 3.3.5, are widely used in Reinforcement Learning for tasks that involve continuous interactions with the environment, where learning occurs incrementally over time.

3.3.4. Features of Reinforcement Learning algorithms

In the literature, a wide variety of RL methods are presented. A review is provided, highlighting the most important features used to distinguish among several classes of algorithms:

- Value-based and policy-based algorithms: value-based algorithms aim at estimating and optimizing the value functions associated with different states or state-action pairs. The goal is to learn a value function that guides the decision-making by selecting actions that lead to states with higher expected returns. Contrarily, policy-based algorithms directly parameterize and optimize the policy that maps states to actions, without explicitly estimating value functions.

- Model-free and model-based algorithms: model-free algorithms, like Q-learning, directly learn optimal strategies by interacting with the environment, through trial-and-error and without explicitly modeling the environment. While simple to implement and versatile, they may require substantial sampling for effective learning.

In contrast, model-based algorithms, such as Monte Carlo Tree Search, create and use a model of the environment's dynamics, in order to simulate and analyze possible future scenarios before they occur. Although potentially more sample-efficient, model-based algorithms heavily depend on the accuracy of the learned model.

- Online and offline learning algorithms: in online learning, the RL agent learns and updates its policy or value function while actively interacting with the environment in real-time. The learning process occurs incrementally, with the agent adapting its strategy based on immediate feedback and experiences. On the other hand, in the offline learning, also known as batch learning, the RL agent learns from a fixed dataset of precollected experiences without further interaction with the environment. The agent processes historical data to update its policy or value function, making it suitable for scenarios where the agent has a predefined set of experiences and wants to improve its learning efficiency.
- On-policy and off-policy algorithms: on-policy algorithms, such as SARSA (State Action Reward State Action), optimize the policy that is currently being used to interact with the environment. The agent learns from its own experiences and directly updates the policy while currently executing it. On-policy methods often involve a trade-off between exploration and exploitation, as the agent refines its strategy while actively interacting with the environment. Off-policy algorithms, on the other hand, optimize a policy different from the one used to generate the data. These methods can learn from experiences generated by another policy, allowing for more flexibility. The policy being optimized is called the target policy, while the policy generating the data is the behavioral policy.

Moreover, hybrid approaches, combining elements of both methods for each category, are available to exploit the advantages of each paradigm for more efficient and robust Reinforcement Learning.

3.3.5. Q-learning method

Q-learning is a model-free Reinforcement Learning algorithm designed to enable the agent to learn optimal actions in a given environment through direct interaction. It is particularly effective in scenarios where the agent has limited prior knowledge about the environment. Q-learning is a value-based method, as it finds its optimal policy indirectly by training a value-function or action-value function, and an off-policy method, as it learns the optimal policy by updating the Q-values based on the maximum expected future reward, regardless of the policy that generated the data. Q-learning adopts the Temporal

Difference approach to iteratively train its action-value function at each step. Q-learning is the algorithm used to train a Q-Function, an action-value function that determines the value of being in a particular state and executing a specific action at that state. Internally, the Q-learning operates by employing a Q-table to store quality estimates (Q-values), for specific actions in different states, thus guiding the agent's decision-making process. The idea of the Q-learning algorithm is that the agent acquires knowledge about the environment by executing actions and learning the values associated with each state-action pair based on the received rewards.

In a finite MDP, characterized by a finite number of states and actions, a Q-table is a valuable tool to map these pairs to corresponding values. The table is initialized to zeros, then the agent performs a random action, transitioning to a new state and receiving a reward. The agent uses this reward as a new information to update the value of the previous state and the action that it just took using the Bellman Equation (3.12).

$$Q'(s, a) \leftarrow Q(s, a) + \alpha \left[R(s, a) + \gamma \; \max Q'(s', a') - Q(s, a) \right] \tag{3.12}$$

The Bellman equation allows the agent to iteratively refine the Q-table over time by breaking down the whole problem into more manageable steps. Instead of attempting to determine the true value of a state-action pair in a single step, the agent updates the value each time it revisits a state-action pair through dynamic programming [35]. The Q-table is constructed during the training phase, where the agent alternates exploration and exploitation to populate the table with learned values. Once the training phase is completed, the information contained within the Q-table dictates the actions constituting the policy. Thus, during the testing phase, the agent will make optimal choices by selecting the best action from this policy, relying on the Q-values.

The Q-learning algorithm, whose pseudo-code is provided in the following Algorithm 3.7, uses the epsilon-greedy policy to handle the exploration/exploitation trade-off, by making the agent choose the action that currently has the highest estimated Q-value with probability (*1*- ϵ), while it selects a random action to explore the environment with probability ϵ. The tuning of the parameter ϵ, which is called exploration rate, is crucial in Q-learning, as it determines the balance between exploration and exploitation within the learning process. When the exploration rate is set to a high value, the agent prioritizes the exploration, meaning that it is more inclined to try out various actions to discover potentially beneficial outcomes. As time progresses and ϵ decreases, the agent shifts its focus towards exploiting the knowledge it has gained from Q-values. The epsilon-greedy policy allows the agent to explore various actions, even those with lower Q-values, which is essential for discovering potentially better strategies. As the agent gains more experience and refines its Q-values, the exploitation becomes more dominant, and the policy tends to select actions with higher Q-values. This dynamic adjustment of ϵ allows the learning agent to strike a balance between discovering new possibilities and leveraging existing knowledge for more refined decision-making.

Algorithm 3.7 Q-learning algorithm

```
Require: learning rate $\alpha$, discount factor $\gamma$, number of episodes $n$, exploration strategy
    (e.g., GLIE with $\epsilon$ decay)
 1: $Q(s,a) = 0 \leftarrow$ Initialize Q-table to zeros for all $s \in S$ and $a \in A(s)$ and
    $Q(terminalstate, \cdot) = 0$
 2: for $i$=1 to $n$ do
 3:     Update $\epsilon$ according to the exploration strategy
 4:     Observe initial state $S_0$
 5:     repeat
 6:         Choose action $A_t$ using the $\epsilon$-greedy policy
 7:         Take action $A_t$ and observe $R_{t+1}$ and $S_{t+1}$
 8:         $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\ [R_{t+1} + \gamma\ \max_a\ Q(S_{t+1},a) - Q(S_t,A_t)]$
 9:         Update state $S_t \leftarrow S_{t+1}$
10:     until $S_t$ is terminal
11: end for
12: return $Q$
```

3.3.6. Exploitation/Exploration trade-off

A critical challenge within Reinforcement Learning technique is commonly referred to as the exploration/exploitation dilemma [42]. It is worth noticing that this dilemma extends beyond RL, impacting the general ability to learn. This concept revolves around the crucial decision of balancing the exploration of potentially new and improved solutions versus exploiting the best currently known policy. When an agent engages exploration, its average performance is usually lower than if it were to exploit the current best policy consistently. A strategy exclusively centered on exploration is impractical when training on physical hardware, as it may lead the agent to perform actions causing damage. Even in simulated environments without the concern of damage, pure exploration is inefficient as the agent tends to cover a larger portion of the state space, potentially slowing down the learning process. On the other hand, using the same policy without exploration hinders the discovery of potentially better alternatives. A learning approach focused only on pure exploitation may increase the amount of time to find the optimal policy or may cause the learning algorithm to converge on a sub-optimal policy, as certain areas of the state space remain unexplored. Therefore, optimal learning algorithms strike a balance between exploring and exploiting the environment.

The ϵ-greedy strategy is commonly used in Q-learning to address the trade-off between exploration and exploitation. This approach ensures that the agent opts for the action associated with the highest estimated Q-value with a probability of $(1 - \epsilon)$. Conversely, to promote exploration, it introduces the possibility of selecting a random action with a probability of ϵ. This method allows the agent to both exploit its current knowledge for immediate rewards and explore the environment for potentially better future rewards.

The dilemma between exploration and exploitation is particularly pronounced in online methods, where the average learning performance serves as a metric for the quality of learning. In this context, the goal is not only to find an optimal policy but to do so while maintaining a trajectory of continuously improving performances. In contrast, offline methods mitigate this concern, as the learning process begins only after a set of trajectories has been collected. This approach ensures that the agent gains the necessary experience to make informed decisions and optimize the policy without the immediate pressure of real-time performance evaluations.

4 | Offline generation of a dataset of feasible paths

4.1. Introduction

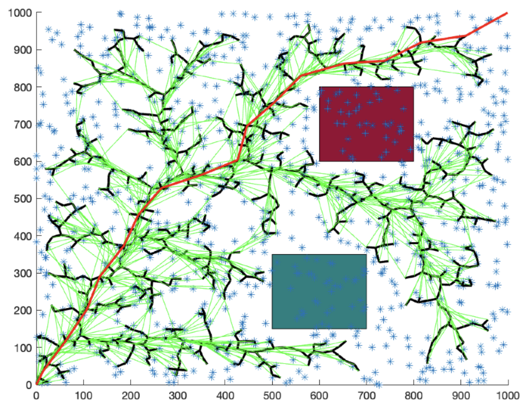
In the first part of this chapter, the focus lies in the application of the BiRRT algorithm to efficiently explore the robot's workspace and generate multiple feasible paths from an initial position to a desired endpoint. The use of this algorithm holds strategic significance for the subsequent phase of Reinforcement Learning. It enhances the robot's adaptability and decision-making capabilities, allowing it to select the most appropriate path from a range of options, especially when navigating in a shared workspace alongside humans. As the trees expand within the workspace, it is necessary to identify only the feasible paths which effectively lead to the goal configuration. Additionally, the parent-child relationships graph is extracted from the trees structure, which provides a clear hierarchy of the nodes within each path and helps in streamlining the decision-making process.

In the second part of this chapter, a model designed to represent the presence of the human operator within the workspace is introduced. This is achieved by discretizing the workspace into voxels, which are uniformly sized cubic volumes.

4.2. BiRRT algorithm

For the purpose of this thesis, multiple feasible paths reaching the final state must be available for the robotic manipulator. The key point of this work is to enable the robotic arm to switch between paths at any moment to avoid excessive proximity or, in the worst case, impact with the human operator. Calculating these paths using the RRT algorithm is extremely advantageous because, by imposing a high maximum number of iterations, the tree extends in all directions simultaneously, creating a dense net of connections throughout the workspace. Figure 4.1 shows the countless connections a RRT algorithm is able to create throughout its branches while avoiding static obstacles. The black edges represent the pathways generated by the standard RRT algorithm implementation, while the green edges are the result of the application of the optimized algorithm. The green connections extend not only to connect nodes within the same branch of the tree but also nodes that belong to different branches.

In particular, after a new node is added to the tree, the RRT* algorithm examines nearby existing nodes within a certain radius, as the new node is considered a potential connection point for both nodes belonging to the same branch and nodes belonging to other branches. These two procedures are called within-branch rewiring and between-branch rewiring, respectively. If connecting the new node to a node in a different branch results in a lower cost path, a connection is established. This mechanism aims at minimizing the cost of reaching a node by considering alternative paths, allowing for the refinement of paths and the discovery of more efficient connections. The combination of within-branch and between-branch connections allows the algorithm to build a dense net of possible paths in the entire configuration space. Moreover, once the execution of the algorithm is completed, i.e. the final state is reached or the maximum number of iterations is hit, the series of connected nodes which form a path from the starting configuration to the goal configuration and characterized by the lower cost, is extracted from the RRT data structure. Being based on a random process of selection of the nodes, the RRT algorithm generates different tree branches and provides a different optimal path every time the procedure is repeated.


(a) Execution of the RRT algorithm

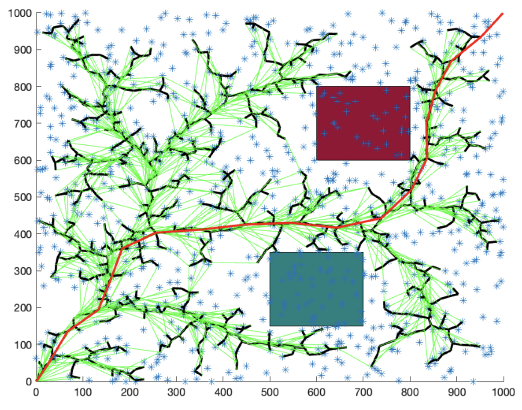
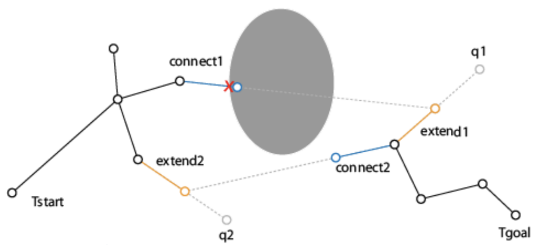
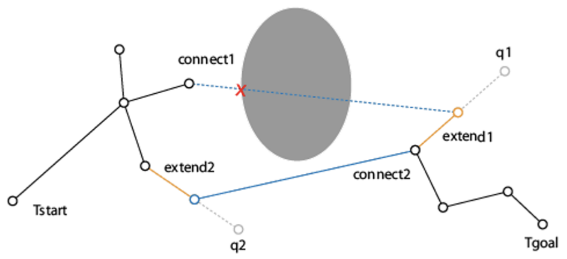

(b) Another execution of the RRT algorithm

Figure 4.1: The combination of within-branch rewiring and between-branch connections allows RRT* to iteratively adapt and refine its solution

In this thesis, a BiRRT planner is employed to explore the configuration space, creating two distinct trees with root nodes at predefined start and goal configurations. Two static obstacles, around which the tree must extend, are present in the workspace. The planner explores the joint configuration space and searches for collision-free paths among different robot configurations. The extension of each tree involves the generation of a random configuration and the definition of a connection with the nearest node, based on the *Max-ConnectionDistance* property. The algorithm employs a connect heuristic to increase the speed of the process, promoting a more efficient connection between the start and goal trees. The choice of enabling or disabling the connect heuristic, via the *EnableConnec-tHeuristic* property, becomes a trade-off between planning times and success rates. The *EnableConnectHeuristic* property is set to false to limit the extension distance for connecting the two trees to the value specified by the *MaxConnectionDistance* property, leading to longer planning times and paths but increasing the likelihood of successfully finding a path within the given number of iterations [43]. Fine-tuning of the planner parameters is a crucial step in ensuring the effectiveness of the robot manipulator's path planning. The effects of these parameters on the tree extension are represented in Figure 4.2. Invalid configurations or connections in collision with the environment are not added to the tree.



(a) *EnableConnectHeuristic* set to false



(b) *EnableConnectHeuristic* set to true

Figure 4.2: Effect of the *EnableConnectHeuristic* property (from [43])

In the Matlab implementation of the bidirectional RRT planner, the *EnableConnec-tHeuristic* property is set to false, the *MaxConnectionDistance* is set equal to 0,005 m and the maximum number of iterations *MaxIterations* is set equal to *10.000*. By limiting the extension distance for connecting the two trees and imposing a maximum high number of iterations, but accepting a long planning time, the algorithm enables the trees to extend widely in the workspace, resulting in the generation of a very high number of paths connecting the start configuration to the goal configuration. The primary focus of the planner is to enable the two trees to fully explore the workspace and establish connections at various points. However, some drawbacks have to be taken into account and ultimately be solved. The drawbacks include long planning times, which do not substantially affect the result since this procedure is performed offline, and the presence of irregularities in the branches, which appear edgy and tortuous.

Subsequent sections 4.2.1 and 4.2.2 will address these challenges by introducing methods to eliminate dead branches and to achieve smoother paths. The key point in the implementation of the planner is that the generated paths are feasible for the robot as they exist within the configuration space. The planner ensures that the paths it constructs satisfy the constraints and the limitations of the robot's joints, making them immediately suitable and practical for the robot to follow. As a result, the paths generated are specifically tailored for the robot selected for this thesis work, which is the GoFa™ robotic arm. The result of the application of the bidirectional RRT planner in a simulated environment created in Matlab is shown in Figure 4.3. The tree in blue colour is generated at the starting configuration, which corresponds to a set of angles of the robot joints equal to *[-51,6° 62,39° -8,24° 2,87° 34,49° 1,14°]*, while the tree in green colour starts from the goal configuration, which corresponds to *[56,17° 48,49° 20,9° -3,79° 19,82° 1,14°]*. The two trees extend in every direction towards each other, and when the maximum number of iterations is reached or the distance between two branches of the trees becomes lower that the one imposed by *MaxConnectionDistance*, a connection is established.

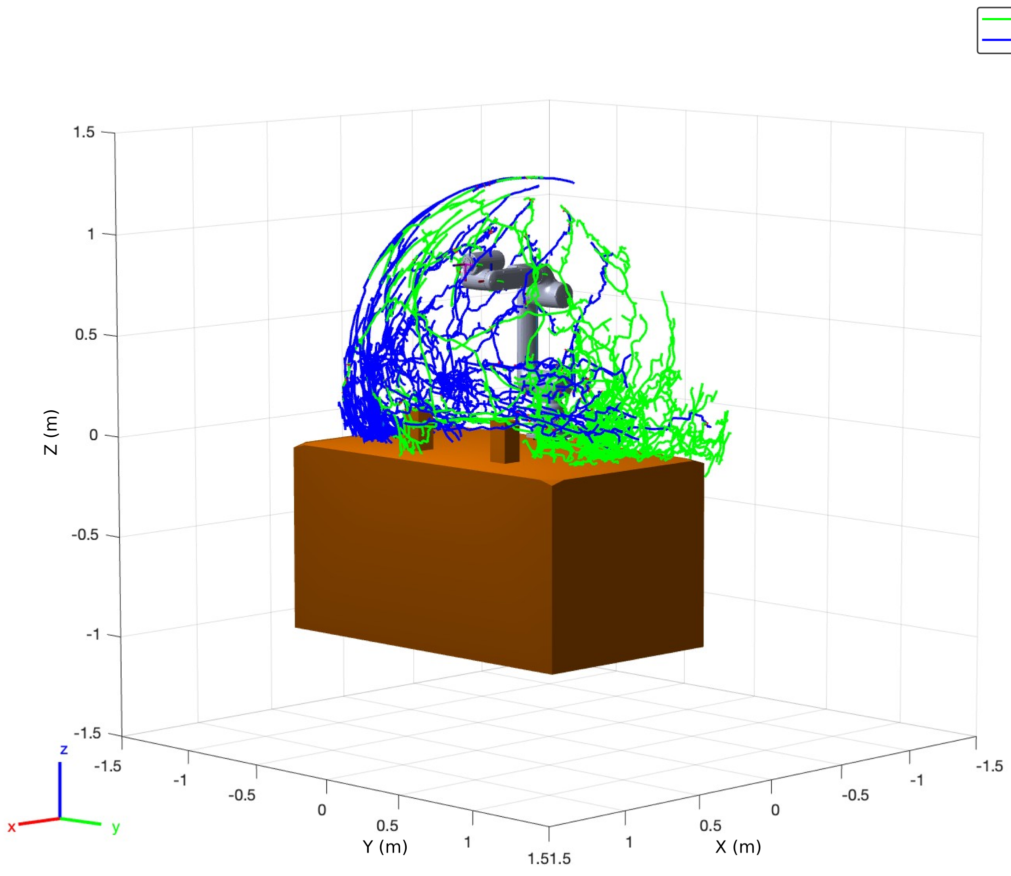


Figure 4.3: Result of the BiRRT algorithm application in the simulated environment

4.2.1. Paths identification

The methodology outlined in Section 4.2 is designed to construct a dense network of interconnected paths throughout the workspace. Thus, the subsequent step involves identifying and extracting valuable paths that connect the starting point to the end goal, removing dead ends, which lead to no viable solution. To achieve this, the Matlab function *allpaths* is employed, which returns all the paths between two graph nodes. This function aids in navigating through the interconnected network and identifying feasible paths for the robot which effectively lead to the goal configuration. For this application, the *allpaths* function has successfully identified *165* feasible paths within the graph, which are shown in Figure 4.4.

The identified paths are described in terms of sequences of robot configurations, where each configuration denotes a specific robot pose that connects the initial configuration to the goal. Additionally, these paths are described using sequences of identification numbers for the nodes within the graph. This dual representation is not only useful for referencing and navigating through the graph efficiently, as it simplifies the path description to a series of node identifiers, but it also allows to easily derive the parent-child relationships among the nodes. Defining the parent-child relationships in the graph is fundamental because they represent the connectivity and the hierarchy between different configurations or states, providing a clear framework for the robot's movements within the workspace.

The network of paths in the configuration space, mapping the transitions from the initial to the goal configuration, is quite intricate. Paths intersect each other at various points, and there are many branching and junction nodes where different paths come together or diverge. This complex network allows the robot to have multiple paths to choose from, especially when it reaches significant nodes in the graph. This multiplicity of routes is strategic for the subsequent phase of Reinforcement Learning, designed to offer the robot an array of navigational choices. In this phase, the robot undergoes a training phase to learn how to navigate this complex graph, aiming to optimize its route selection. By learning the layout of the paths network and how humans move within it, the robot uses the acquired knowledge optimally, becoming able to find the shortest path to reach the goal but also to choose a path that ensures safety, maintaining at every time instant a minimum distance from the current position of the human operator. This emphasizes the robot's ability to adapt and make informed decisions based on the intricate network structure it has learned.

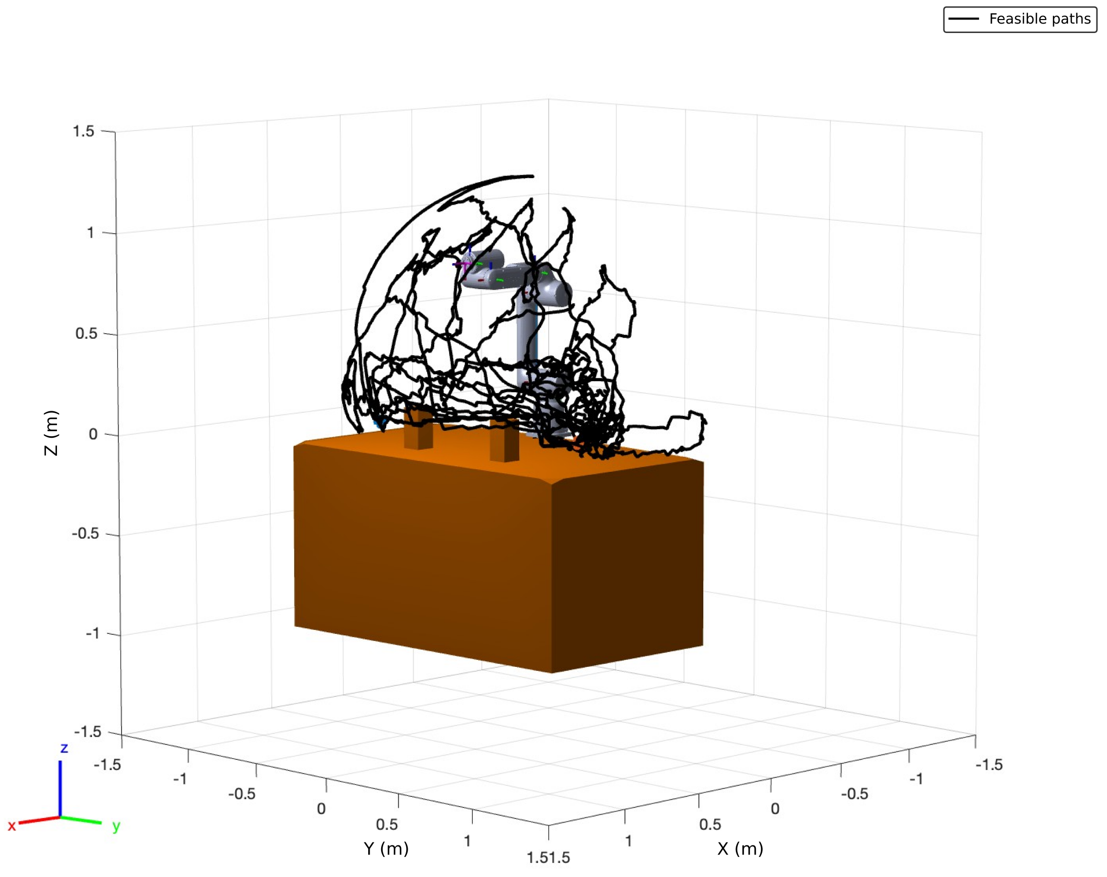


Figure 4.4: Identification of the *165* feasible paths within the network connecting the initial configuration to the goal configuration

4.2.2. Ramer-Douglas-Peucker algorithm

Once the feasible paths are identified as described in Section 4.2.1, the goal is to make them smoother and more linear in order to facilitate the robot's motion, improving the fluidity of movements along the identified paths. This process of path smoothing can be applied only on specific segments of the paths, specifically those without branching or junction nodes. Considering that a branching node is a parent node with two or more children nodes and a junction node is a child node with two or more parent nodes, the sub-paths which can be simplified without altering the tree structure are exclusively those one made by parent nodes which have only one child node and viceversa. The underlying purpose is to guarantee that the robot reaches the key nodes, where a decision must be taken in the subsequent phase of Reinforcement Learning. The remaining nodes, which constitute unique paths with no possibility of shifting to other paths, can undergo a process of simplification.

Once these sub-paths are identified, the Ramer-Douglas-Peucker (RDP) algorithm is applied, which is a technique used for simplifying a curve represented by a series of points. It is commonly employed in computer graphics, geographic information systems, and various applications where the reduction of data points in a curve is desired without significantly altering its shape. The RDP algorithm, whose pseudo-code is shown in Algorithm 4.1, operates by iteratively identifying and removing points that contribute minimally to the overall shape of the curve. First of all, the point in the curve that has the maximum perpendicular distance from the line segment connecting the first and last points of the curve is identified. If the maximum distance is greater than a predefined threshold ϵ, the algorithm identifies the portion of the curve between the first and last points which contains complex details that need further refinement. The algorithm is recursively applied to simplify this specific portion, determining whether a point is essential for representing the curve or if it can be approximated by a straight line segment connecting adjacent points. Finally, the algorithm builds the final result list, which contains all the key points that represent the essential shape of the curve [24].

Algorithm 4.1 Ramer-Douglas-Peucker Algorithm

```
Require: $d_{max}$, $\epsilon$, $PointList[]$, $index$
 1: $d_{max} = 0$
 2: $index = 0$
 3: for $i$=2 to $length(PointList)$ -1 do
 4:     $d \leftarrow$ perpendicularDistance($PointList[i]$, Line($PointList[1]$, $PointList[end]$))
 5:     if $d > d_{max}$ then
 6:         $index = i$
 7:         $d_{max} = d$
 8:     end if
 9: end for
10: return $ResultList[] = empty$
11: if $d_{max} > \epsilon$ then
12:     $Result1[]$ = DouglasPeucker($PointList[1,...,index]$, $\epsilon$)
13:     $Result2[]$ = DouglasPeucker($PointList[index,...,end]$, $\epsilon$)
14:     $ResultList[] = \{Result1[1,...,length(Result1) - 1], Result2[1,...,length(Result2)]\}$
15: else
16:     $ResultList[] = \{PointList[1], PointList[end]\}$
17: end if
18: return $ResultList[]$
```

The following Figures demonstrate the influence of the factor ϵ on the approximation of the original curve (depicted in yellow) with the approximated curves (depicted in blue). Specifically, Figure 4.5 illustrates the simplification of the original curve, achieved by setting a value of ϵ equal to *19,6*. This process of simplification reduces the number of points describing the original curve from *56* to *13*. Similarly, Figure 4.6 results from setting a value of ϵ equal to *7,9*, which brings to *22* points of the approximated curve, while Figure 4.7 is derived by setting a value of ϵ equal to *2,3*, which leads to *37* points of the approximated curve. Thus, smaller thresholds preserve more detail, while larger thresholds result in greater simplification. The simplified curves present roughly the same shape of the original curve but consist only of a subset of the points that defined the original curve. In the practical application focus of this thesis, the parameter ϵ, used for the robot's paths simplification, has been set to *10 cm*. A higher value would be risky as the robot's paths are designed to navigate around static obstacles present in the workspace, so taking a higher value could result in a non collision-free path.

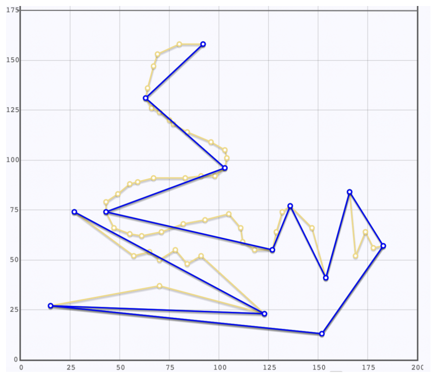


Figure 4.5: Simplification of the curve in yellow using the RDP algorithm with ϵ=*19,6*

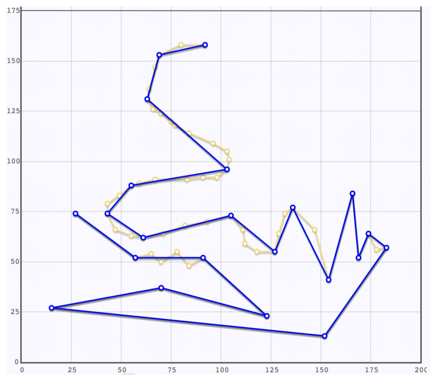


Figure 4.6: Simplification of the curve in yellow using the RDP algorithm with ϵ=*7,9*

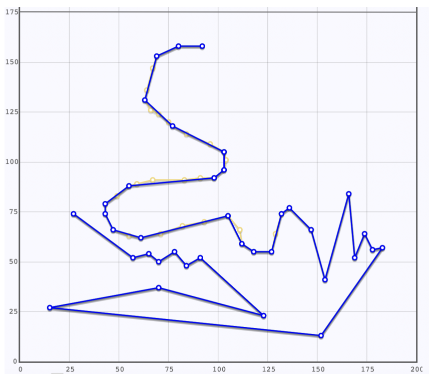


Figure 4.7: Simplification of the curve in yellow using the RDP algorithm with ϵ=*2,3*

Figure 4.8 shows the *165* paths after the simplification with the RDP algorithm by removing the nodes that are non-essential and contribute minimally to the overall shape of the curve. It is worth noticing that all the branching and junction nodes are still present in the tree structure and the nodes which fall outside the RDP threshold are preserved. This ensures that the robot still has some reference nodes between significant points, which contribute to effective navigation and guidance for the robot in following a specific path.

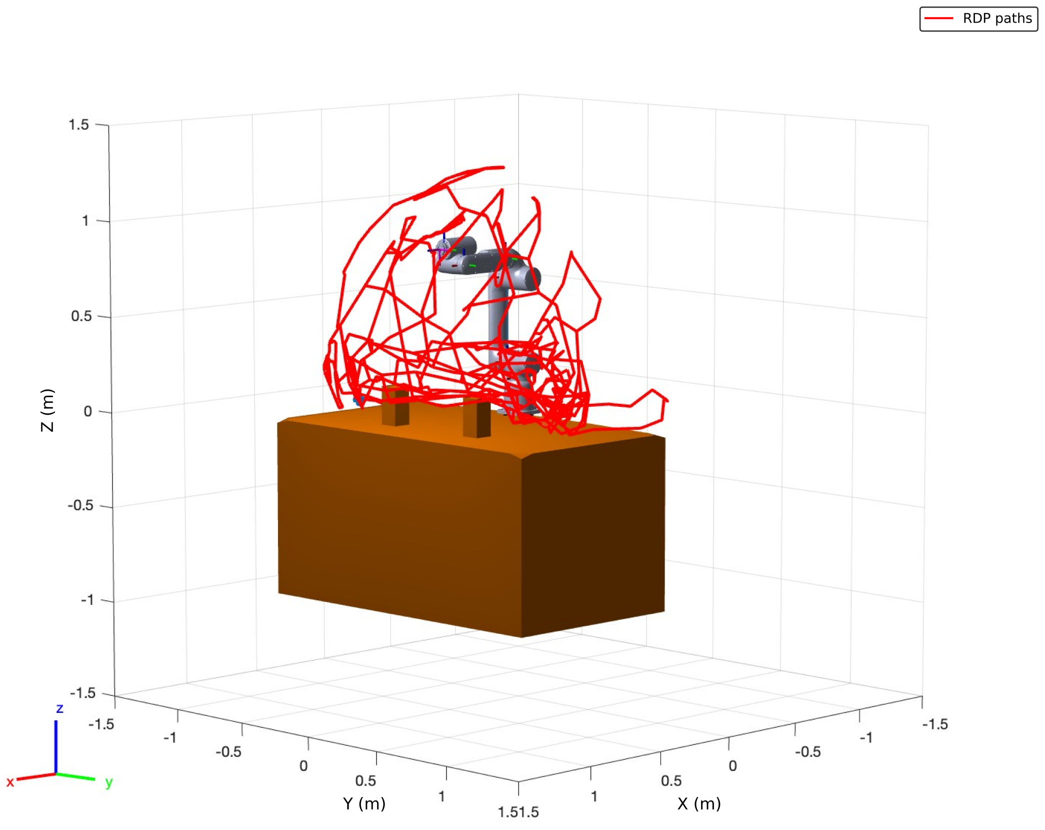


Figure 4.8: Result of the application of the RDP algorithm to the *165* paths

4.2.3. Parent-child relationships graph reduction

The dense network, which results from the trees expansion and connection processes, provides a foundation for the subsequent application of the Reinforcement Learning method. The aim is to teach the robot how to navigate efficiently within the workspace, relying on the knowledge derived from this network and on the parent-child relationships. As a result, the graph representing the parent-child relationships undergoes a process of reduction as only the significant nodes are essential to define the states and actions required in Section 5.3.1 to construct the Q-table. The significant nodes are defined as the junction nodes and the branching nodes, as well as those nodes that are indispensable to uniquely identify a path. The significance of these nodes lies in the fact that, in certain situations, there may be numerous alternative routes connecting a branching node to a junction node, as demonstrated in Figure 4.9. Therefore, it is necessary to keep some of those nodes to preserve the ability to distinguish between paths that might otherwise become indistinguishable. All the nodes that are not filtered out by the RDP algorithm are removed from the graph, as they do not contribute to the robot's movement, which involves transitions between notable nodes. This process effectively minimizes the number of nodes to only those essential for the navigation. Moreover, for the future development of Reinforcement Learning, it is crucial to focus exclusively on identifying the significant nodes and their connections, providing a parent-child relationships graph containing only these relations.

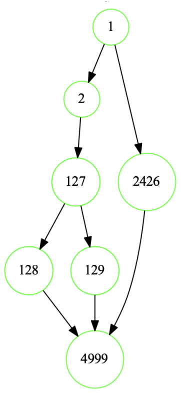


Figure 4.9: Example of different pathways connecting node *1* to node *4999*

The process of the parent-child relationships graph reduction can be summarized through the following steps:

1. Determine the list N of the nodes that currently constitute the total parent-child relationship graph G.
2. Define c as the counter of the nodes within the list N and initialize it to zero.

3. If c is different from the number of nodes constituting G, proceed to step *4*; otherwise, the algorithm terminates.
4. Select the node in position c from N, named current node n_{corr}.
5. If n_{corr} has a single parent p and a single child f, proceed to step *6*; otherwise, increment c by one unit (c=c+*1*), and return to step *3*.
6. If f has p as its parent, increment c by one unit (c=c+*1*), and return to step *3*. Otherwise, remove n_{corr} from G and N, along with the parent/child relationships between p and n_{corr} and between n_{corr} and f. Generate a new parent/child relationship between p and f; then return to step *2*.

This process allows to simplify the full parent-child relationships graph from a total of *2.815* nodes to *131* nodes. The reduced graph containing the essential parent-child relationships is shown in Figure 4.10. The nodes are linked with *201* connections, which represent the actions of the subsequent training phase.

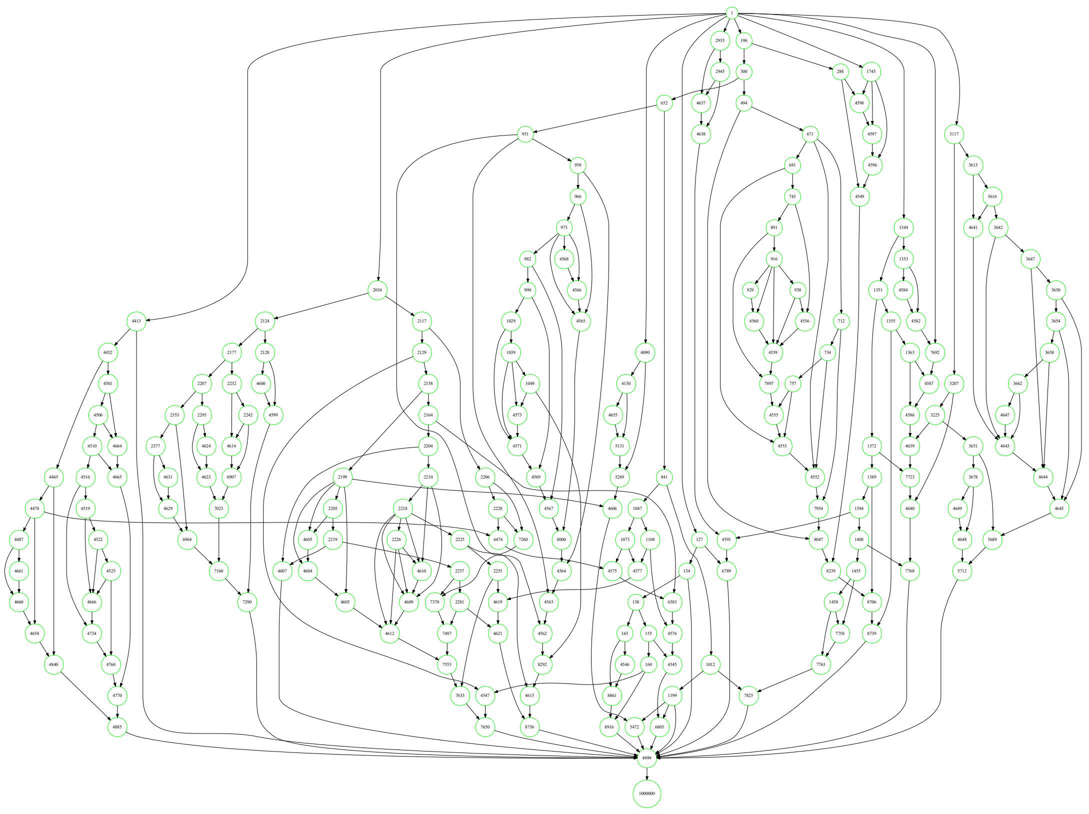


Figure 4.10: Parent-child relationships graph with *131* nodes and *201* connections

4.3. Workspace discretization

Workspace discretization into voxels involves dividing the entire workspace into uniform cubic volumes, known as voxels. Each voxel is precisely defined by its cubic shape and is characterized by Cartesian coordinates relative to a global reference frame, which coincides with the robot reference system, as represented in Figure 4.11. This process aims at representing the workspace in a structured and computationally manageable manner. The number of voxels in the three-dimensional space is determined based on the dimensions of the real workspace and the desired resolution. The higher the desired resolution, the more voxels are required, subsequently increasing the computational cost.

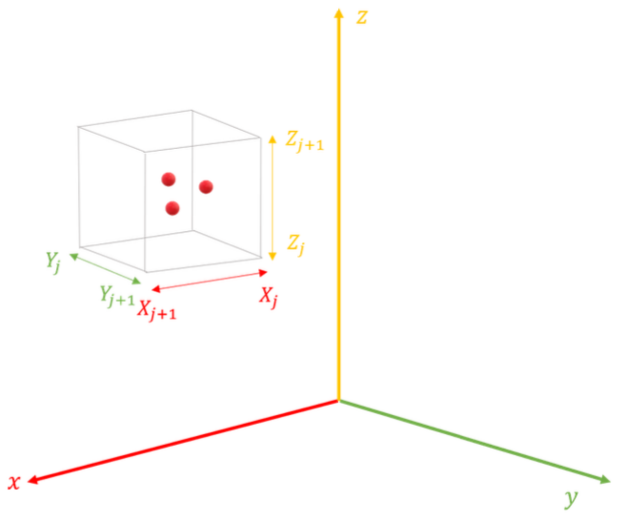


Figure 4.11: Voxel coordinates are defined with respect to a global reference system

The motion of the human worker in the workspace is depicted through a series of occupied voxels. Utilizing voxels to model the human presence offers a significant advantage, particularly because human movements are inherently non-repetitive. The dimension of a voxel is sufficiently large to encapsulate not only the average position of the movements but also their standard deviation. This ensures that the model accounts for the range of motion within the workspace, providing a more realistic representation of human activity.

For the application under study in this thesis, the shared workspace has been divided into a total of *30* voxels, with each voxel being a cube with a side length of *30 cm*. Additionally, the introduction of an extra voxel, assigned the identification number *31*, is necessary to take into account the condition where specific skeletal positions of the human body are located outside the designated workspace area. Figure 4.12 visually illustrates how the three-dimensional space has been segmented into voxels. The spatial coordinates delineating each voxel are indicated by 'x' marks, whereas the centers of the voxels are highlighted with red dots. A unique identification number is assigned to each voxel within the workspace, ensuring a systematic way to reference and track each cubic volume in the space. These identification numbers are used to pinpoint which voxels are occupied by each of the considered skeletal joints of the human at each time instance. This process aims at mapping the skeletal points while executing the task to specific voxels. In this way, the movements of the human necessary to execute the task are translated into an evolution of the voxels occupied over time.

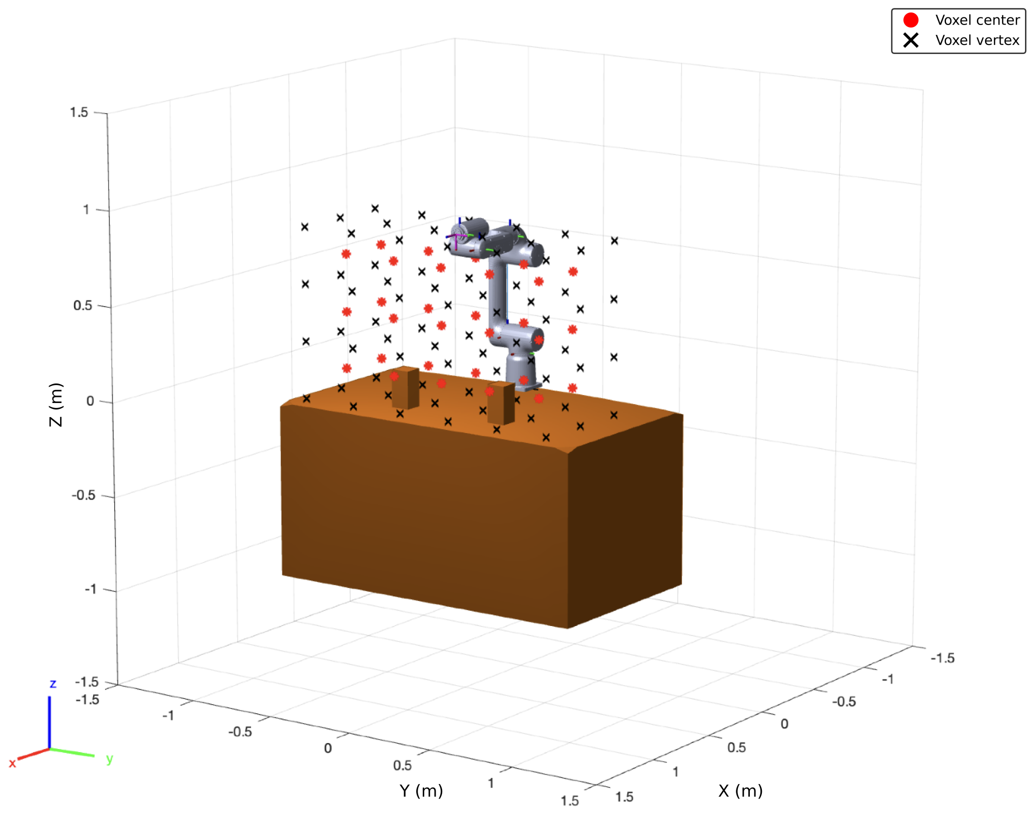


Figure 4.12: Workspace discretization in voxels

5 | Reinforcement Learning application

5.1. Introduction

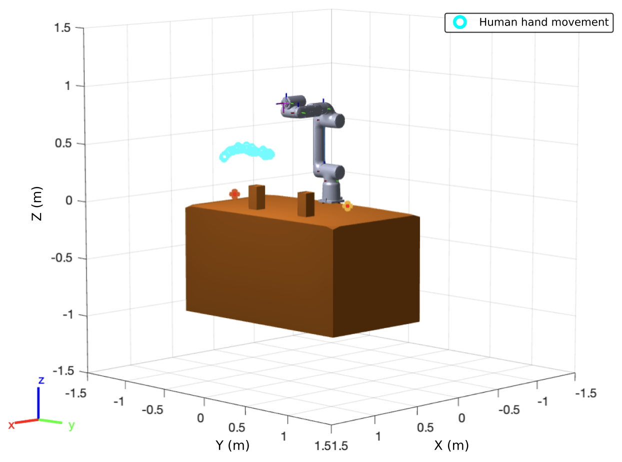
In this chapter, the Q-learning technique is employed to train the robotic manipulator. The key point is to guarantee that the manipulator motion is not limited to a single path, but instead has access to various alternatives to achieve its final state. The aim of this research is to equip the robotic arm with the capability to dynamically switch between these paths once a branching node is reached. This flexibility is essential for maintaining a safe distance from the human operator and preventing any potential collisions. By empowering the robotic arm with a deeper understanding of its operational environment and with the ability to adapt its movements in response to human movements, the interaction between robots and humans in a shared space is significantly enhanced, leading to safer, more efficient, and harmonious collaborations.

This chapter is structured into two main sections: initially, the focus lies on defining the elements required for the MDP problem, including a human occupancy model. The model leverages data pertaining to human movements during various tasks, sourced from the Motion Capture (MoCap) database. For each of the selected tasks and at every moment in time, the model determines the occupancy within each voxel, thereby providing a detailed spatial and temporal analysis of human presence in the workspace. The second part of the chapter is devoted to the training phase of the Q-learning algorithm and to evaluate its performances through a testing phase. This approach allows for a comprehensive evaluation of the algorithm's capabilities, ensuring it learns the desired behaviors before assessing its into the validation phase on the GoFa™ robotic arm.

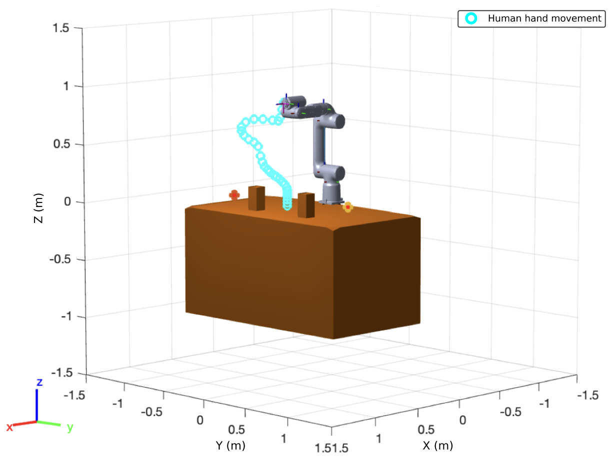
5.2. Human occupancy model

Human occupancy data within the workspace during real-time operations are collected using the Kinect camera. Instead, during the training phase of the Q-learning algorithm, input data regarding human movements within the workspace are sourced from the MoCap database. The choice of this dataset proves to be particularly beneficial in this context as the aim is to generalize human movements as much as possible. This ensures that the final real-time model is capable of accommodating any human motion and task.

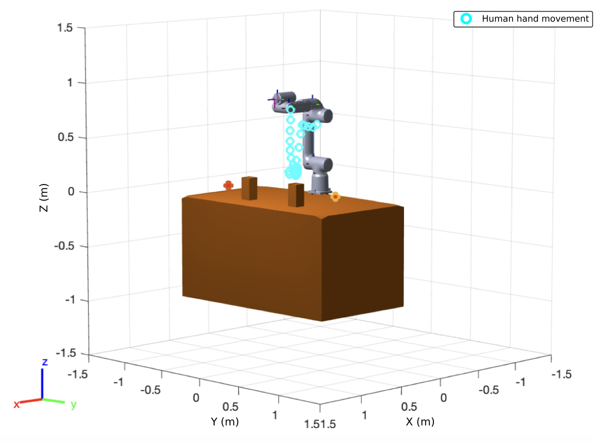
Referenced in [29], the MoCap database contains over three hours of systematically recorded and well-documented motion capture data. This extensive collection features more than *70* categories of motions, including walking, dancing, sports activities, and gestures, realized in *10* to *50* variations by different actors and performed by over *140* subjects. For the purposes of the training, a selection of tasks was made from the MoCap database, specifically focusing on those involving arm movements within the workspace and excluding tasks related to activities such as walking. During the training phase, the robotic manipulator has to learn how to navigate within the workspace to reach its final configuration with the shortest path while also maintaining a safe distance from the human. The manipulator has to determine the best path to take in response to different human movements, and, once the optimal policy is derived, in the subsequent testing phase, the robot will be presented with new, previously non analyzed MoCap tasks. It will then need to decide on a course of action to evaluate the effectiveness of the established policy. Human movements from the MoCap dataset are represented as a series of points for each joint of the human body. This detailed representation facilitates the robot's understanding and anticipation of human movements, allowing for more informed decision-making in maintaining safe and efficient interactions within the shared workspace. For the training phase, the tasks selected from the MoCap database are equal to *40*, while the optimal Q-table is then tested on *10* new tasks. Examples of the chosen tasks are reported in Figure 5.1 only considering the movements relative to one hand.



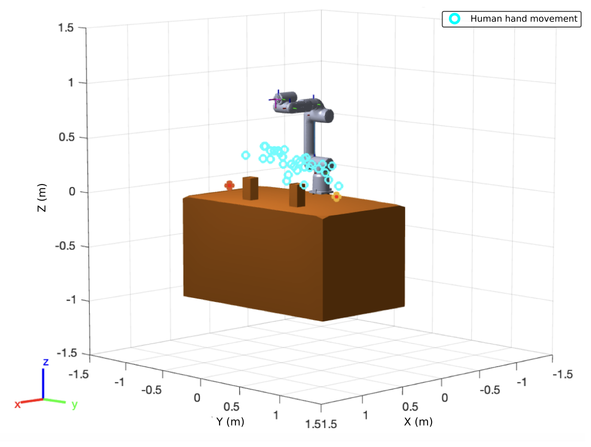
(a) Task: "drink soda, screw on bottlecap"



(b) Task: "high five"



(c) Task: "shake hands"



(d) Task: "conversation with hand gestures"

Figure 5.1: Human hand motion in the workspace for different tasks

The occupancy determination within each voxel is achieved for each of the selected tasks from the MoCap database and for each time instant. Algorithm 5.1 is employed to execute this identification procedure in a systematic and computational manner, ensuring consistency and accuracy in determining the occupied voxels for the human's hand across different tasks and temporal moments. In particular, for a fixed task, the algorithm evaluates whether the *i-th* point of the hand of the human body, with coordinates x_i, y_i and z_i, belongs to the volume of the *j-th* voxel, which is defined by coordinates X_j, X_{j+1}, Y_j, Y_{j+1} Z_j and Z_{j+1}. The algorithm provides as output the identification numbers of the occupied voxels at each time instant.

Algorithm 5.1 Voxel occupancy map Algorithm

```
Require: $[x_i\ y_i\ z_i]$ of each skeletal point, $[X_j\ X_{j+1}\ Y_j\ Y_{j+1}\ Z_j\ Z_{j+1}]$ of each voxel,
    $N_{skeletal\ points}$, $N_{voxels}$
1: for $i=1$ to $N_{skeletal\ points}$ do
2:     for $j=1$ to $N_{voxels}$ do
3:         if $X_j \leq x_i \leq X_{j+1}$ & $Y_j \leq y_i \leq Y_{j+1}$ & $Z_j \leq z_i \leq Z_{j+1}$ then
4:             $Voxel_{occupied} = Voxel_{id}(j)$
5:         else
6:             $Voxel_{occupied} = 31$
7:         end if
8:     end for
9: end for
10: return $Voxel_{occupied}$
```

5.3. Algorithm training phase

Q-learning is a model-free Reinforcement Learning algorithm that seeks to find the best action to take given the current state. It works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following a fixed policy thereafter. Training a Q-learning model requires several key components, which are illustrated in the following and then defined specifically for this problem:

- The state space contains the set of all possible states in which the agent can find itself. It captures all relevant information about the environment that allows the agent to make decisions.
- The action space contains the set of all possible actions the agent can take in each state. The action space defines the choices available to the agent at each step.
- The reward signal is the feedback that the agent receives from the environment after taking an action. The reward signal indicates the immediate value of the action taken and it allows the agent to learn which actions lead to better outcomes.
- The policy is the strategy that the agent follows to decide the actions to be taken.
- The Q-table is the table that contains the Q-values. The rows of the Q-table corresponds to the states and the columns to the actions. Each entry in the table represents the expected cumulative reward of taking an action in a given state, following the optimal policy thereafter.
- The learning rate α determines the extent to which the new information affects the existing Q-values. A higher learning rate means that more weight is given to newer information.
- The discount factor γ determines the importance of future rewards. A discount factor close to *1* places nearly equal importance on future rewards as on immediate rewards, encouraging the agent to consider long-term consequences of its actions.
- The exploration strategy is the method for balancing exploration, which consists in trying new actions and exploitation, which consists in taking actions known to yield high rewards. The ϵ-greedy strategy is commonly used in Q-learning, where ϵ represents the probability of taking a random action.
- The Q-value update rule is the formula used to update the Q-values in the Q-table based on the reward received and the estimated optimal future rewards. The update rule enables the agent to improve its policy over time.

5.3.1. MDP environment definition

The parent-child relationships graph, introduced in Chapter 4, serves as the fundamental component for defining discrete states and actions essential for the construction of the Q-table. This framework supports the Reinforcement Learning model, facilitating the systematic exploration of feasible paths and enabling the algorithm to learn optimal decision-making strategies through interaction with the environment.

The actual 3D problem counts for *201* possible actions, *131* nodes, and *31* voxels. To facilitate a clearer comprehension of how the Q-learning problem is formulated, a simplified 2D version of the problem, which includes *26* possible actions, *19* nodes, and *24* voxels, is analyzed. Subsequently, the same method is applied to analyze the real, more complex 3D problem. Figure 5.2 illustrates the structure of the 2D problem. The nodes, which represent feasible positions for the manipulator, are depicted within circles, while the actions the robot can take are illustrated by arrows linking two nodes. The numbers displayed in pink color correspond to the states associated with each node, providing a clear mapping between the robot's location and its current state within the environment. The manipulator motion starts in correspondence of node *1* and ends when the final node *19* is reached. Throughout the grid, multiple paths are available for the manipulator to navigate and ultimately arrive at its designated endpoint. The presence of humans within the grid is represented by a series of occupied voxels. To simplify the model, it is assumed that at any given moment, only one voxel is occupied, specifically the voxel that is occupied by the hand of the human worker. The grid is divided into *24* voxels, with an additional voxel included to account for situations where the considered skeletal position falls outside the designated grid area. Consequently, each node is characterized by *25* states to accommodate this spatial modeling. The total number of states is computed using Equation (5.1), with the assumption that the final node is represented by a single state. This is due to the fact that, once the robot reaches the final node, no further decisions are required and the human presence is irrelevant for the task's completion.

$$NumStati_{tot} = (NumNodi_{tot} - 1) \cdot NumVoxel_{tot} + 1 \tag{5.1}$$

In a Markov Decision Process problem, states and actions are designed to comprehensively describe the environment to the extent necessary for decision-making purposes. This means that the states should capture all relevant information that could influence future decisions and the outcomes of those decisions. Similarly, the actions should contain all possible choices available to the agent that can affect the state of the environment. Consequently, each state is defined by a specific pairing: a voxel that is occupied by the human operator and a node that is occupied by the manipulator. In other words, the combination of a node and a voxel occupancy defines a particular state. For instance, state *1* indicates that the robot is located at node *1* with voxel *1* being occupied, while state *2* signifies that the robot is still at node *1*, but now voxel *2* is occupied. Similarly, state *28* shows the robot has moved to node *2* and voxel *3* is occupied, an so on.

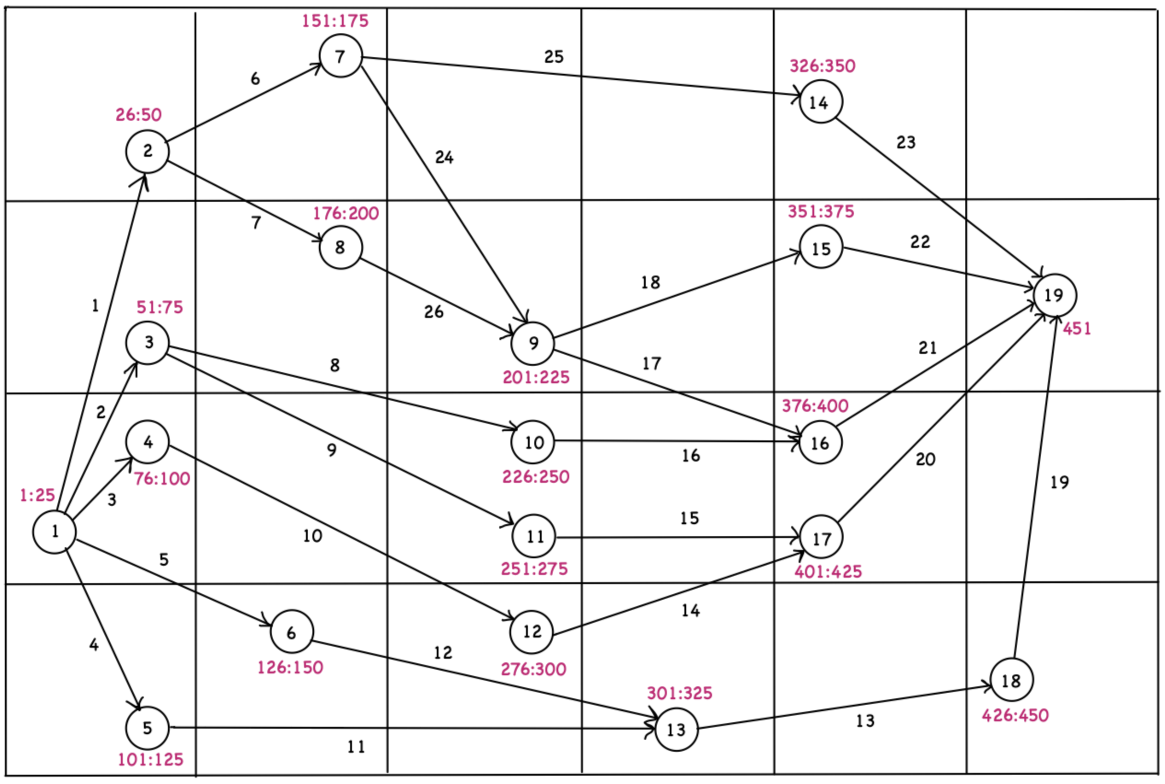


Figure 5.2: 2D grid structure with *26* possible actions, *19* nodes and *24* voxels

5.3.2. Transitions and rewards definition

Transition and reward matrices are square matrices used to model the dynamics of the environment and the outcomes of actions taken by an agent within that environment. In the context of MDP, there is typically a separate transition matrix and reward matrix for each action that can be taken. Given that there are *26* distinct actions for the 2D simplified case, this results in the creation of *26* separate transition matrices and *26* corresponding reward matrices. On the other hand, for the three-dimensional real case, the number of transition and reward matrices is equal to *201*, as the number of actions is equal to *201*. The transition matrix describes the probabilities of moving from one state to another in a given environment. Each element of the matrix represents the probability of transitioning from one state to another as a result of a particular action.

To facilitate the definition of the transition matrix and of the reward matrix, which are fundamental for the development of the Q-table, the initial step involves creating a matrix which contains all possible and impossible transactions based on the current state and the action taken. In this matrix, the transitions the robot can execute in a particular state are marked with a *1* in the last column, signifying their availability. Conversely, actions that are not viable in the current state are denoted by a *0*, indicating their unfeasibility. This preparatory step ensures a structured framework, allowing for a systematic representation of how actions taken in specific states lead to subsequent states and the associated rewards or penalties. A portion of this matrix involving nodes *1* and *2* is illustrated in Table 5.1, providing an example of how the information contained in Figure 5.2 is applied in practice.

In the MDP implementation, it is necessary to define transitions for all state-action pairs, including situations where certain actions may not be feasible within specific states. This approach is adopted to prevent undefined behavior and to meet the specifications of the particular MDP framework in use. To manage infeasible actions within the code, a self-transition, which means transitioning back to the same state, with a probability of *1* is assigned for actions that are not applicable in a given state. For example, node *1*, whose associated states are from *1* to *25*, permits actions from *1* to *5*. By taking action *1*, the manipulator transitions to node *2*, which is associated with states from *26* to *51*. Similarly, action *2* leads to node *3*, corresponding to states from *51* to *75*, and so forth. Conversely, actions from *6* to *26* are not viable at node *1*. Therefore, executing one of these prohibited actions while at node *1* results in the robot remaining at node *1*. Implementing this strategy ensures that taking an infeasible action in any state results in remaining in that state with absolute certainty. This approach not only simplifies the handling of actions that cannot be executed in certain states but also maintains the integrity of the MDP model by avoiding undefined behavior. By doing so, the model accounts for the consequences of all actions, including those that do not lead to a change in state, thereby maintaining a complete and consistent representation of the environment essential for effective decision-making and policy evaluation within the MDP framework.

Moreover, given that each node is linked to *25* distinct states, the feasible transitions must consider that there is an equal probability of moving from any of the *25* states of the starting node to any of the *25* states of the target node. For instance, when transitioning from state *1*, which is associated with node *1* and occupied voxel *1*, there exists a probability of *1/25* of moving to each state within the range of *26* to *50*, corresponding to node *2*, upon executing action *1*. This equal distribution of probabilities ensures a uniform approach to state transitions, reflecting the inherent uncertainty about the specific end state within the target node's state set, given a particular action.

Node$_{start}$	**State$_1$**	**State$_2$**	**Node$_{end}$**	**State$_3$**	**State$_4$**	**Action**	**Feasible**
1	1	25	2	26	50	1	1
1	1	25	3	51	75	2	1
1	1	25	4	76	100	3	1
1	1	25	5	101	125	4	1
1	1	25	6	126	150	5	1
1	1	25	1	1	25	6	0
1	1	25	1	1	25	7	0
...	...	...	...	...	...	...	...
2	26	50	7	151	175	6	1
2	26	50	8	176	200	7	1
2	26	50	2	26	50	1	0
2	26	50	2	26	50	2	0
...	...	...	...	...	...	...	...

Table 5.1: Columns from left to right: ID of the starting node, range of states associated to the starting node, ID of the end node, range of states associated to the end node, actions, feasibility of the action

The reward function quantifies the benefit or consequence of performing a specific action in a given state, serving as a critical feedback mechanism that guides the learning algorithm. By assigning a numerical value to each action's outcome, the reward function helps the agent discern which actions are beneficial for achieving its goal. This relationship between actions and their consequent rewards is fundamental for the agent to learn optimal behaviors through trial and error, optimizing its strategy to maximize cumulative rewards over time.

Once infeasible actions are identified for a given node, the reward function assigned to these actions is set to *-1.000*. This ensures a substantial penalty is associated with attempting any impossible action within the current context, effectively discouraging the selection of such actions. Instead, for feasible actions, the reward function is designed to achieve a balance between ensuring that the robotic manipulator consistently maintains a certain minimum distance from the voxel occupied by the human and minimizing the distance from the current configuration of the manipulator to the goal configuration. Thus, the computation of the reward function takes into account two quantities: the distance between each node and every voxel within the grid, which evaluates the proximity to the human occupied voxel, and the distance between each node and the goal node, which assesses the efficiency of the path towards the objective. This approach enables the robot to navigate safely and efficiently, avoiding too close an approach to the human while optimizing its path to the goal. Algorithm 5.2 presents the mathematical formulation of this theoretical concept. The reward associated to a specific action is composed by two terms, which are called R_1 and R_2. R_1 is derived from the proximity between the node that could be occupied in the next time step by the robot and the voxel that is occupied by the human at the current moment. This part of the reward incentives the agent to maintain an appropriate distance from the human, promoting safety and efficient navigation within the shared environment. In fact, the calculation of R_1 incorporates a specific formula that is selected according to a threshold parameter, denoted as a, which defines some zones of close proximity within the voxel's area. This parameter, which is computed in Equation (5.2) for the 3D real case, quantifies the distance from the vertex of a voxel to its center, serving as a key factor in determining the proximity-based reward.

$$a = \sqrt{3 \cdot \left(\frac{\text{DimensioneVoxel}}{2}\right)^2} \tag{5.2}$$

Furthermore, the use of the logarithmic function results particularly advantageous because by dividing the node-voxel distance by $2 \cdot a$, when the logarithm's argument falls below *1*, it takes on a negative value. This mechanism effectively imposes a substantial penalty on very small node-voxel distances. Conversely, when the logarithm's argument exceeds *1*, indicating that the node is sufficiently distant from the occupied voxel, the logarithm results in a positive value. This value increases as the node-voxel distance grows, thus encouraging the agent to maintain a safe distance from the human-occupied voxel by rewarding greater separation with higher rewards. On the other hand, R_2 pertains to the distance between the node that might be occupied in the subsequent time step and the designated goal node. This segment of the reward focuses on guiding the agent towards its goal configuration. R_2 is computed by considering a factor μ equal to *25* and considering the inverse of the node-goal distance, as the aim is to penalize actions that bring to nodes that are far from the goal node. In the end, the reward function for states associated with voxel number *25* for the 2D case or *31* for the 3D real case, when the human worker's hand is positioned outside the workspace, is calculated focusing only on the shortest path from the current to the goal configuration. In these cases, the absence of a human within the workspace eliminates any collision risk, making the term R_1 equal to zero.

Algorithm 5.2 Rewards definition

```
Require: $Dist_{Node\text{-}Voxel}$, $Dist_{Node\text{-}GoalNode}$, $a$, $VoxelID$
 1: if $Dist_{Node\text{-}Voxel} = 0$ then
 2:     $R_1 = -1.000$
 3: else if $Dist_{Node\text{-}Voxel} \leq a$ then
 4:     $R_1 = \ln\left(\frac{\text{Dist}_{\text{Node-Voxel}}}{2a}\right) \cdot \text{Dist}_{\text{Node-Voxel}} \cdot 50$
 5: else
 6:     $R_1 = \ln\left(\frac{\text{Dist}_{\text{Node-Voxel}}}{2a}\right) \cdot \text{Dist}_{\text{Node-Voxel}}$
 7: end if
 8: if $VoxelID == 31$ then
 9:     $R_1 = 0$
10: end if
11: $R_2 = \mu \cdot \frac{1}{\text{Dist}_{\text{Node-GoalNode}}}$
12: $Reward = R_1 + R_2$
13: return $Reward$
```

5.3.3. Policy and training procedure

The training phase involves iterating through episodes of interactions within the environment. The goal is to train the agent to make decisions based on the state of the environment, which includes the robot's position and the human trajectory. Each episode of the training uses a trajectory selected randomly from the MoCap database to simulate the human's movement through the environment. At the beginning of each episode, the initial state is determined by the robot's starting node and the initial position of the human from the trajectory. The current state combines the robot's and the human's positions to represent the situation accurately. For each step within an episode, the model selects an action based on the current state, using a policy derived from the Q-table.

The action selection process implemented in the training procedure draws inspiration from the ϵ-greedy algorithm but it is adapted to this specific problem. The decision-making environment is defined by an intricate graph of parent-child relationships, as depicted in Figure 4.10, which showcases an extensive array of possible paths starting from each branching node. Given the width of alternatives in terms of possible paths connecting the starting configuration to the goal configuration and the complexity of navigating through such a diversified parent-child relationships graph, this approach allows to prioritize exploration over exploitation, especially in the initial episodes of the training process. The action selection process implemented for this problem is detailed in Algorithm 5.3. The total number of episodes, denoted as $NumEpisodes_{tot}$, is set equal to *10.000*, while the exploration rate, ϵ, is set equal to *0,9*. The process for deciding the next action adopts an epsilon-decreasing approach to balance exploration and exploitation. Exploration involves selecting a random action within the range of *1* to *26* in the 2D case scenario, or within *1* to *201* for the 3D real-world setting. Upon choosing such an action, if it is viable at the current node, the reward calculation takes into account two specific contributions, R_1 and R_2, as previously described. Conversely, if the selected action is not feasible, it incurs a penalty with a reward set to *-1.000*, reflecting the significant negative impact of infeasible actions within the exploration process. Exploitation, in contrast, involves choosing the action that is associated with the highest value in the state-action pair according to the Q-table. This approach prioritizes the selection of actions that the agent has learned to yield the most favorable outcomes, based on its accumulated experience. For the first half of the episodes, exploration is strongly encouraged, while during the second half of the episodes, the value of epsilon is dynamically adjusted based on the episode count. This helps to focus more on exploration in the initial stages of learning and more on exploitation in later stages, when the Q-table has accumulated more knowledge of the environment. With this strategy, ϵ_1 gradually decreases with each episode, allowing the agent to transition from an initial phase of intensive exploration to a more advanced phase of exploitation, based on the acquired knowledge, in a smoother and controlled manner.

Algorithm 5.3 Action selection process

```
Require: $Qtable$, $episode$, $NumEpisodes_{tot}$, $\epsilon$, $CurrentState$
 1: $Qtable \leftarrow$ Inizialize $Qtable$ with zeros
 2: if $episode > \frac{NumEpisodes_{\text{tot}}}{2}$ then
 3:     $DecayRate = -\log(0,1) / \left(\frac{NumEpisodes_{tot}}{2}\right)$;
 4:     $\epsilon_1 = \epsilon \cdot \exp\left[-DecayRate \cdot \left(episode - \frac{NumEpisodes_{\text{tot}}}{2}\right)\right]$;
 5: else
 6:     $\epsilon_1 = \epsilon$;
 7: end if
 8: if $rand(1) > \epsilon_1$ then
 9:     $NextAction \leftarrow$ Take the maximum value of $Qtable(CurrentState, :)$
10: else
11:     $NextAction \leftarrow$ Choose random between 1 and 201
12: end if
13: return $NextAction$
```

Once the agent selects its next action based on the policy and on the current state, it interacts with the environment by executing that action, which results in a transition from the current state to the next state according to the dynamics of the environment. Concurrently, this transition provides a reward, representing the immediate consequence of the action taken. The reward is used as a new information to update the values associated with the action taken and the previous state. The mechanism of updating of the Q-values in the Q-table is done through the Bellman equation, which represents the utility of taking a given action in a given state and following the optimal policy thereafter. This equation represents a foundational principle in dynamic programming and Reinforcement Learning, enabling the agent to iteratively adjust values to reflect the expected long-term benefit of its actions. Q-learning uses the Temporal Difference (TD) error, which quantifies the discrepancy between the predicted value of a state or state-action pair and the immediate reward plus the estimated value of the next state or state-action pair. Such update rule allows the algorithm to recursively improve its estimates of the Q-values towards the optimal action-value function, which dictates the best action to take in every state to maximize future rewards. These concepts are formalized using the equations in the following. The TD error is defined in Equation (5.3). The *Reward* used in this formula is the feedback associated to the transition from the *OldState* to the *CurrentState* taking the *ChosenAction*. Instead, Equation (5.4) illustrates the process of updating the *NewQValue*, starting from the *OldQValue* and using the *TD* error adjusted by the learning rate factor.

$$TD = Reward + \gamma \cdot \max(Qtable(CurrentState, :)) - OldQValue \tag{5.3}$$

$$NewQValue = OldQValue + \alpha \cdot TD \tag{5.4}$$

5.3.4. Q-table analysis and convergence strategies

The outcome of the training phase is the Q-table, which is a matrix with number of columns equal to the number of available actions and number of rows equal to the number of possible states. For the 2D case, the Q-table matrix has *451* rows and *26* columns. A number of episodes $NumEpisodes_{tot}$ equal to *5.000* is more than enough to obtain a complete matrix at the end of the training phase for the two-dimensional scenario. This implies that throughout these episodes, the state space is exhaustively explored, ensuring that every possible state-action combination is evaluated at least once. On the other hand, in the real 3D scenario, the Q-table is a significantly larger matrix, counting specifically *4.000* rows and *201* columns. For such a complex environment, a number of episodes equal to *5.000* proves to be insufficient for achieving a fully populated matrix. The number of rows in the Q-table whose maximum state-action pair value is null is equal to *1.677*, which corresponds to the *41,9%* of the total number of rows. Thus, the complexity of the environment results in a high percentage of unexplored or under-explored state-action pairs. This gap in exploration can impact the algorithm's ability to fully understand and optimize its decision-making process across the entirety of the state-action space, potentially leaving certain strategies less refined or certain outcomes less predictable. An obvious solution to this problem is increasing the number of episodes, giving the algorithm more opportunities to interact with the environment, thereby increasing the likelihood of encountering and evaluating more state-action combinations. Thus, the number of episodes is increased first up to *10.000* and then up to *50.000.* This solution is not definitive on its own because the number of rows in the Q-table, where the maximum value of the state-action pair is null, equals *1.187* in case of *10.000* episodes and equals *673* in case of *50.000* episodes. This represents respectively the *29,7%* and *17%* of the total number of rows.

Two different strategies are applied. The primary strategy makes the most of a fundamental characteristic of MDP: the outcome of future states is determined only by the current state and the action taken, rather than by the history of events that preceded it. This intrinsic property of MDP allows for a significant optimization in the algorithm's functionality. The algorithm begins its process at a fixed starting point, which is the initial node labeled as node *1.* However, instead of commencing every episode from this predetermined starting node, the algorithm is modified to introduce an element of randomness in selecting its starting position. At the beginning of each episode, it selects a starting node at random from within the range of *1* to *130.* Regardless of the starting node selected at random within the specified range, the algorithm is capable of navigating the robot through the graph to its intended destination. Another advantage of this approach is that it ensures equal probability for all states to be analyzed. Typically, if the algorithm always initiated from node *1*, the nodes at the beginning of the graph, including node *1* itself and those in higher positions, would be examined more frequently. Conversely, states corresponding to nodes in lower positions might not receive adequate analysis due to the graph's complexity and breadth. By randomizing the starting node, the algorithm mitigates this imbalance, allowing for a more uniform exploration of the graph. The training procedure is repeated with this improvement and the number of episodes is set equal to *5.000*, *10.000* and *50.000.*

The enhancement of the algorithm's efficacy incorporates a second strategy, which involves using the knowledge of the parent-child relationship graph and, in particular, the knowledge of the actions that are feasible and not feasible in each node, which are reported in Table 5.2. This approach simplifies the decision-making process and optimizes the algorithm's performance by clearly delineating the actionable paths and constraints at each juncture. As a consequence, the approach of selecting actions undergoes a significant modification. During the exploration phase, the next action is no longer chosen randomly from the entire range from *1* to *201*, which counts for both feasible and not feasible actions, for each node independently. Instead, for every node, the subsequent action is selected exclusively within the range of feasible actions at that specific node. The major advantage of this strategy is that the time needed to complete the training procedure is significantly reduced, as a consequence a number of episodes is set equal to *50.000* and *500.000*.

To summarize, the various training sessions, along with their characteristics and outcomes, are presented in the following:

- Results of the training session with node *1* as initial node and random action chosen between feasible and not feasible actions for every node in the exploration phase:
  - over the course of *5.000* episodes, the Q-table exhibits the *41,9%* of the rows with maximum value of zero.
  - over the course of *10.000* episodes, the Q-table exhibits the *29,7%* of the rows with maximum value of zero.
  - over the course of *50.000* episodes, the Q-table exhibits the *17%* of the rows with maximum value of zero.
- Results of the training session with initial node selected randomly between *1* and *130* and random action chosen between feasible and not feasible actions for every node in the exploration phase:
  - over the course of *5.000* episodes, the Q-table exhibits the *32%* of the rows with maximum value of zero.
  - over the course of *10.000* episodes, the Q-table exhibits the *20%* of the rows with maximum value of zero.
  - over the course of *50.000* episodes, the Q-table exhibits the *9%* of the rows with maximum value of zero.
- Results of the training session with initial node selected randomly between *1* and *130* and random action chosen only between feasible actions for every node in the exploration phase:

  - over the course of *50.000* episodes, the Q-table exhibits the *7%* of the rows with maximum value of zero.
  - over the course of *500.000* episodes, the Q-table exhibits the *7%* of the rows with maximum value of zero.

The analysis of the training sessions conclusively identifies the most effective Q-tables as the ones derived from a setup where the initial node is randomly selected from a range between *1* and *130*, random action chosen between feasible and not feasible actions for every node in the exploration phase. The Q-tables derived from this setup and training sessions with *50.000* and *500.000* episodes both exhibit a commonality: *7%* of the rows have a maximum value of zero. This observation suggests that the outcomes have reached a state of convergence, indicating that further improvements do not produce positive effects even with an increase in the number of episodes.

Despite the fact that *7%* of the rows in the Q-table display a maximum value of zero, the table itself is fully populated. This means that every state-action pair has been evaluated and assigned a specific value by the algorithm. Rows exhibiting a maximum value of zero are characterized by negative values across all actions. In fact, actions that are not feasible at a given node are assigned a negative value but also certain states, which represent particular robot node and occupied voxel combinations, present negative values in correspondence of the feasible actions. In such cases, the negative values indicate that, although the actions are permitted, they lead to undesirable outcomes because the robot node is either located within the same voxel occupied by the human or is extremely close to it. Given that each node within the workspace is mapped to a specific voxel, it is reasonable and expected that at least one state associated with each node would have a negative reward value, a reflection accurately captured within the Q-table.

5.4. Algorithm testing phase

Once the policy is sufficiently optimal, the learning process can be stopped and the optimal Q-table is extracted. The testing phase of the learning algorithm is useful to evaluate the performances of the algorithm, ensuring it has learned the intended behaviors. This phase is essential before advancing to the validation stage on the GoFa™ robotic arm. During the testing phase, the algorithm is exposed to new sets of data of the of human hand movement, sourced from the MoCap database, which are different from the data it was trained on. By utilizing previously unseen data, the testing phase aims at assessing the algorithm's generalization capabilities and its ability to apply learned patterns and behaviors to new, unexplored scenarios.

An example of the results of the testing phase for the 2D scenario is presented in the following. The algorithm is tested on the following sequence of occupied voxel: *1*, *2*, *3*, *4* and *10*. In Figure 5.3, the scenario begins with voxel *1* occupied by the human hand, and the robot is initially positioned at node *1*. The first step is to identify the state corresponding to the combination of the robot position and the voxel occupied by the human, which in this case is state *1*. Consequently, the algorithm consults the optimal Q-table previously obtained, specifically targeting the first row, to determine the optimal action based on the highest value present in that row. Therefore, the algorithm selects action *1*, which corresponds to a transition from node *1* to node *2*. The voxel occupied at the following time instant is number *2*, as depicted in Figure 5.4, while the current position of the robot is node *2*. This specific scenario is represented by state *27*, and the analysis of the *27-th* row in the Q-table reveals that the index corresponding to the maximum value is *7*. Consequently, the robot executes action *7*, which results in its transition to node *8*. Subsequently, state *178* identifies the scenario where voxel *3* is occupied and the robot is located at node *8*. In this context, the action chosen is number *26*, as illustrated in Figure 5.5. The robot transitions to node *9* and since the following voxel occupied is *4*, the corresponding state is *204*. The algorithm selects action *18*, transitioning to node *15*, as shown in Figure 5.6. Finally, the state corresponding to node *15* and voxel *10* is *360*. The robot correctly selects action *22* and arrives to its final node *19*, which is represented only by one state, as illustrated in Figure 5.7. The final state is *451* and no further actions are needed. Table 5.2 summarizes the chosen actions based on the current state and on the information contained in the Q-table.

Voxel	**Node**	**State**	**Action**
1	1	1	1
2	2	27	7
3	8	178	26
4	9	204	18
10	15	360	22
10	19	451	-

Table 5.2: Actions are selected by the algorithm based on the current state of the agent and the information contained in the Q-table, which provides a mapping of state-action pairs to Q-values

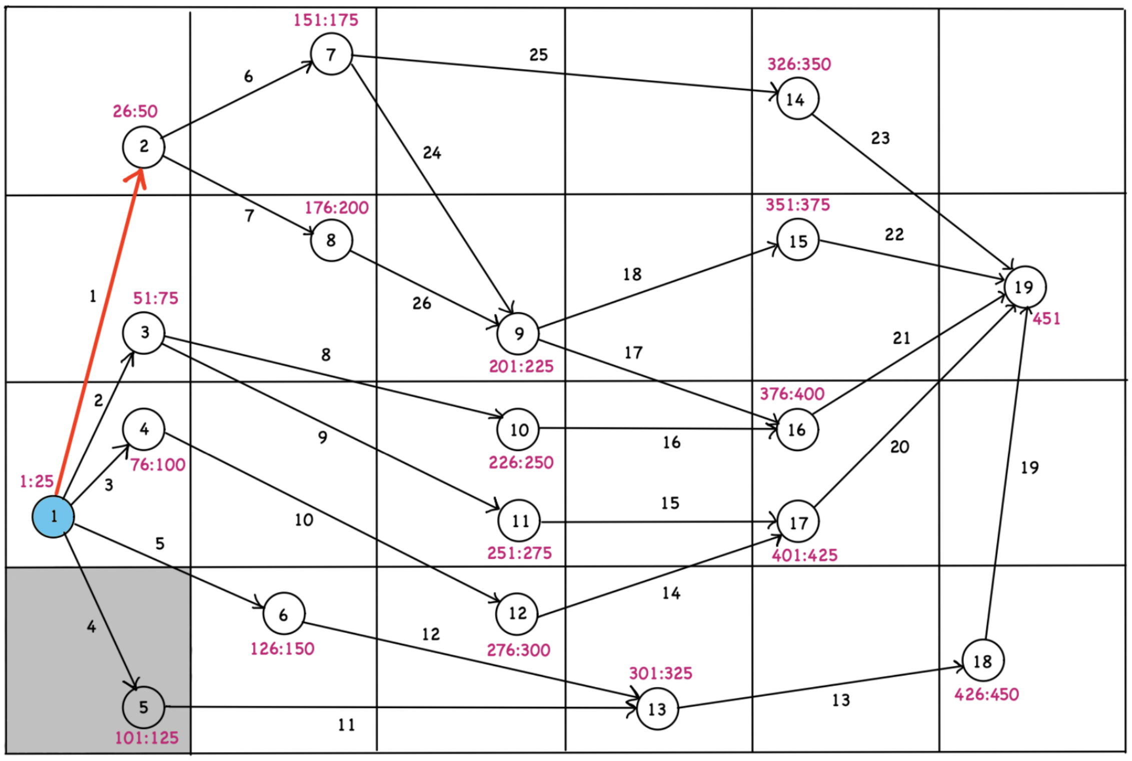


Figure 5.3: Robot node = *1*, Next voxel = *1*, Current state = *1*, Chosen action = *1*.

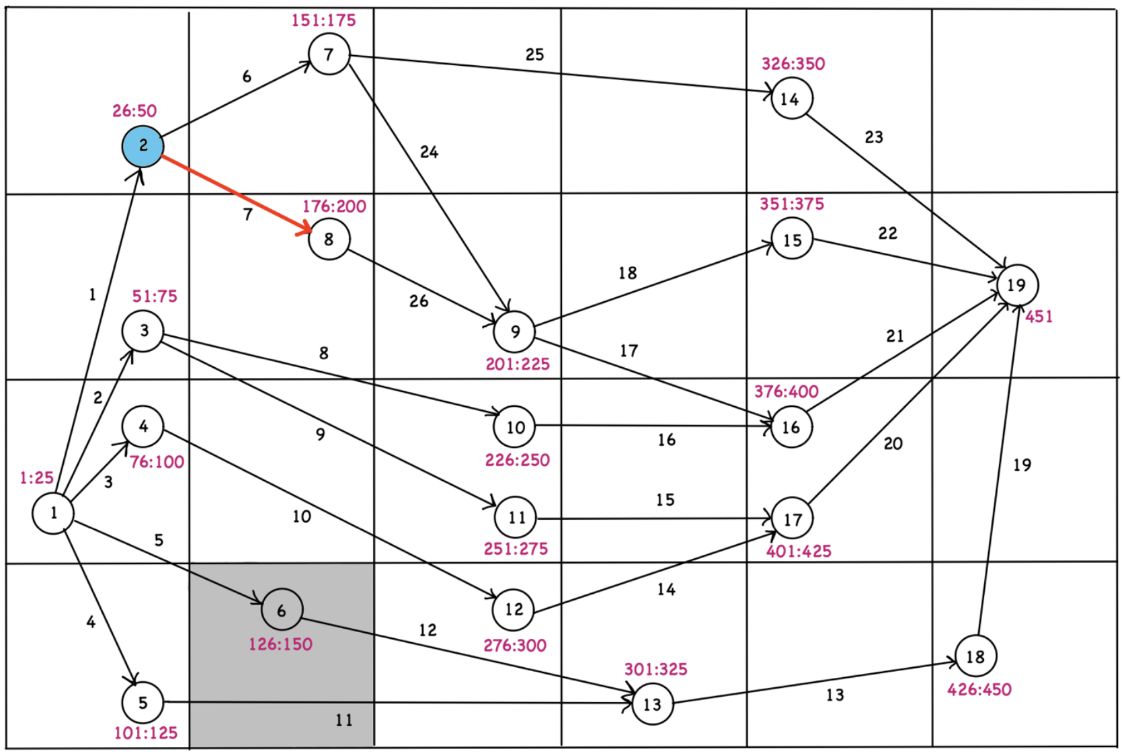


Figure 5.4: Robot node = *2*, Next voxel = *2*, Current state = *27*, Chosen action = *7*

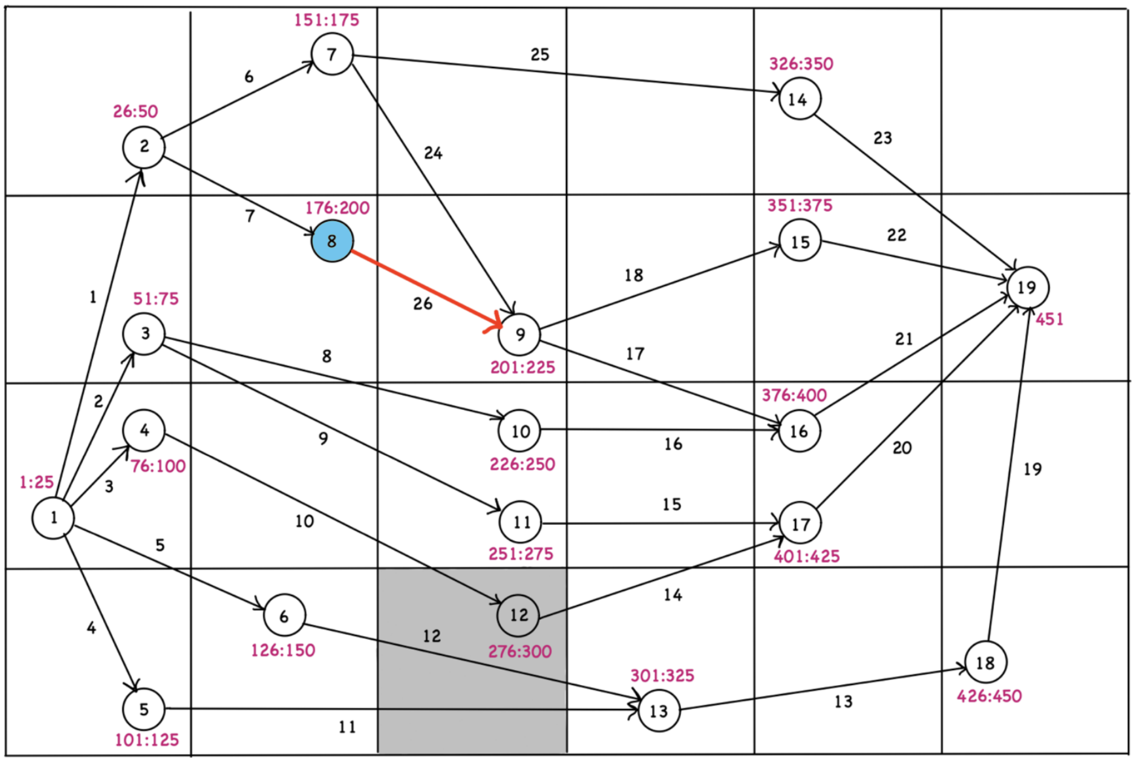


Figure 5.5: Robot node = *8*, Next voxel = *3*, Current state = *178*, Chosen action = *26*

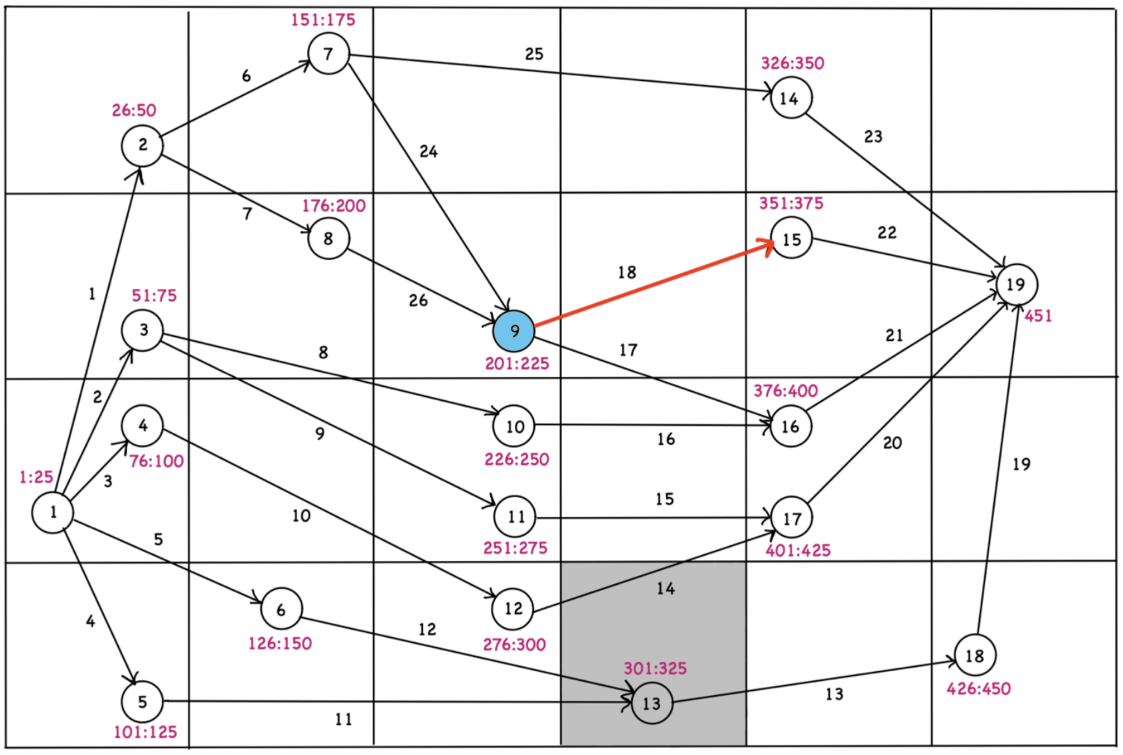


Figure 5.6: Robot node = *9*, Next voxel = *4*, Current state = *204*, Chosen action = *18*

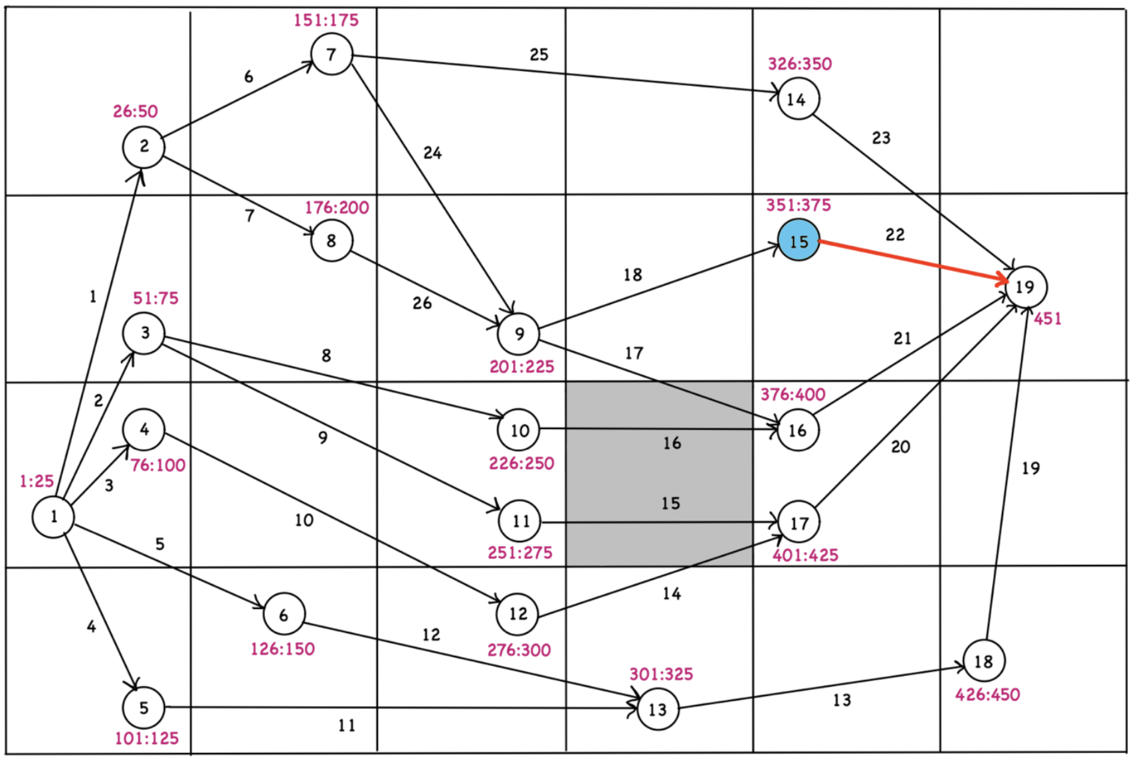


Figure 5.7: Robot node = *15*, Next voxel = *10*, Current state = *360*, Chosen action = *22*

The methodology implemented in the testing phase for the 3D case study is coherent to the conceptual framework just described for the two-dimensional scenario. The Q-table is subjected to evaluation using *10* new MoCap datasets, each representing distinct patterns of human hand movement. These datasets are chosen to differ from the data utilized during the training phase. The outcomes of this testing phase are detailed subsequently.

In *9* cases out of *10*, the testing algorithm successfully identifies a solution, effectively determining a sequence of nodes that avoids and circumvents the series of occupied voxels for that specific task. However, in one case, the algorithm fails to discover a valid path. This failure occurs because the maximum value in the row corresponding to the specific state under examination equals zero. As described in Section 5.3.3, the *7%* of the rows in the Q-table display a maximum value of zero, even though the table is fully populated. When the maximum value of the row equals zero, no actions are deemed feasible at that state. This scenario occurs when specific actions are disallowed due to the limitations set by the parent-child relationship graph, and the remaining actions carry a negative state-action value. This is because their associated transitions would result in a node that is dangerously close to an occupied voxel, making these actions unsuitable for navigating the workspace without breaching the minimum safety distance required from the operator.

Figure 5.8 illustrates the outcome of the testing algorithm given a sequence of occupied voxels equal to: *5*, *5*, *5*, *5*, *15*, *15*, *15*, *30*, *30* and *29*. In this representation, the green dots represent the centers of the occupied voxels, while the asterisks mark the vertices of these voxels. The algorithm first determines the current state of the robot based on its current position and a predictive model estimating the next voxel to be occupied by the human hand. Actions to be performed by the manipulator are then selected according to this current state and the knowledge derived from the Q-table. The testing algorithm calculates that the path to be followed in this case, adhering to the optimal policy derived from the optimal Q-table, consists of the sequence: *1*, *8*, *9*, *86*, *76*, *123*, *125*, *126* and *130*. This sequence of nodes and their connections are depicted by the red path in the Figure. Similarly, Figure 5.9 represents the optimal path to be followed by the robotic manipulator when navigating through the following sequence of voxels: *4*, *19*, *18*, *18*, *31*, *19*, *31* and *20*. For this task, the optimal path according to the optimal policy derived from the optimal Q-table, consists of the sequence: *1*, *59*, *62*, *63*, *95*, *101*, *102* and *130*. Table 5.3 and Table 5.4 provide a summary of the decision-making process, containing in each column the sequences of occupied voxels, of nodes connecting the starting point to the goal, of corresponding states and of chosen actions for the selected tasks.

Figure 5.10 and Table 5.5 depict an unsuccessful outcome of the training phase. The algorithm is not able to find a valid path connecting the starting point to the target destination. The initial state is equal to *22* and action *6* leads to a transition towards node *66*. Subsequently, the algorithm selects action *39*, determining a transition to node *67*. At this point, state *2.068*, linked to node *67* and voxel *22*, is analyzed. Unfortunately, the maximum value in the Q-table for this state is equal zero, indicating that all actions within the Q-table are unfeasible, effectively bringing the algorithm's progress to a standstill. It is simple to verify that node *67* belongs to voxel *22*, resulting in a negative reward associated to the only permitted transaction. To address this issue, the proposed solution involves implementing a self-transition mechanism for the robot. The policy is improved by imposing that the robot remains at the current node whenever the algorithm is unable to select an optimal action for a given state. As a result, when the algorithm analyzes state *2.068* without identifying a viable solution, the robot stays at node *67* until the voxel in input changes, thereby altering the state and enabling the algorithm to discover a feasible action based on a new combination of previous node and new voxel occupied. This approach ensures that the algorithm stops only when the final state is reached by awaiting new conditions that facilitate the progress. Moreover, the introduction of this safety mechanism guarantees that if the human and the robot occupy the same voxel, the robot automatically stops, effectively preventing any potential collisions.

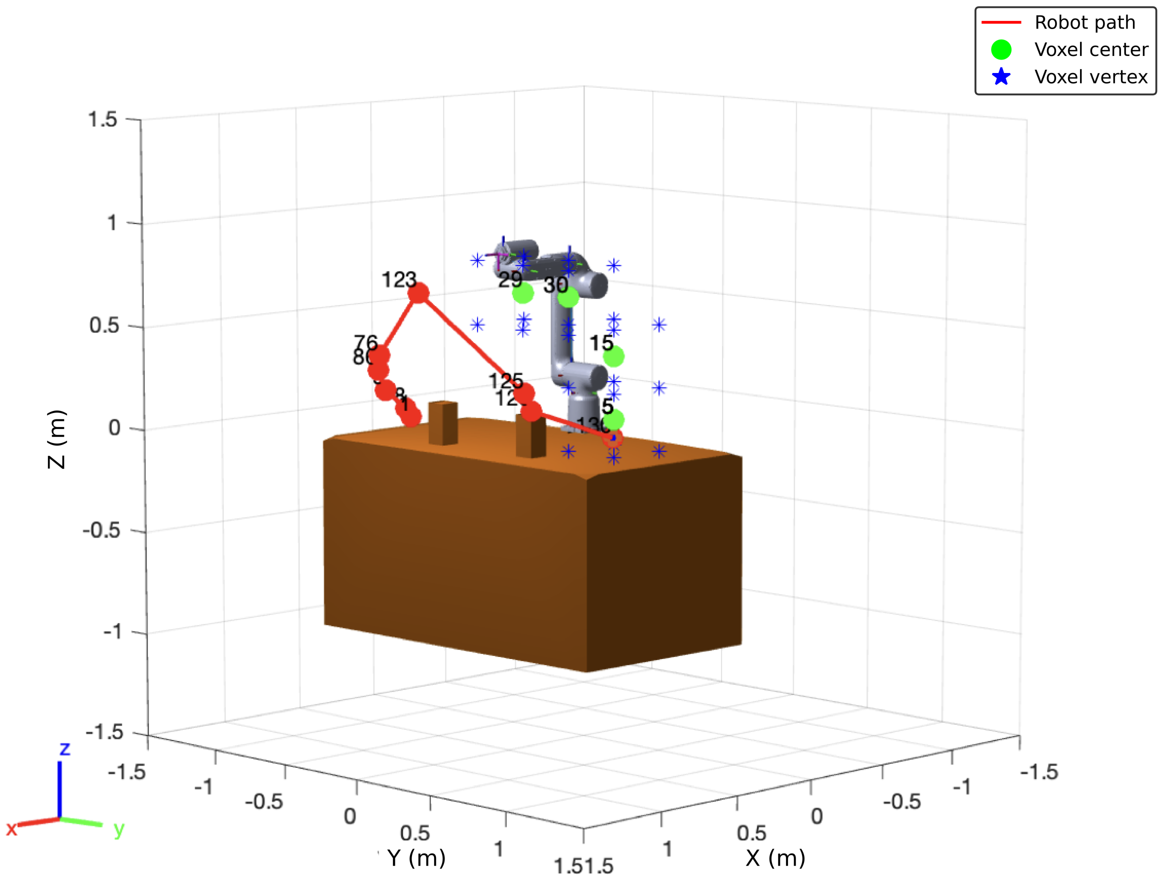


Figure 5.8: The optimal path to be followed by the manipulator to maintain in every moment a safe distance from the occupied voxel of the selected task is depicted in red

Voxel	Node	State	Action
5	1	5	6
5	8	222	69
5	9	253	91
5	86	2.640	92
15	76	2.340	93
15	123	3.797	75
30	125	3.874	76
30	126	3.905	77
29	130	4.000	-

Table 5.3: Sequences of voxels, nodes, states and actions for the selected task

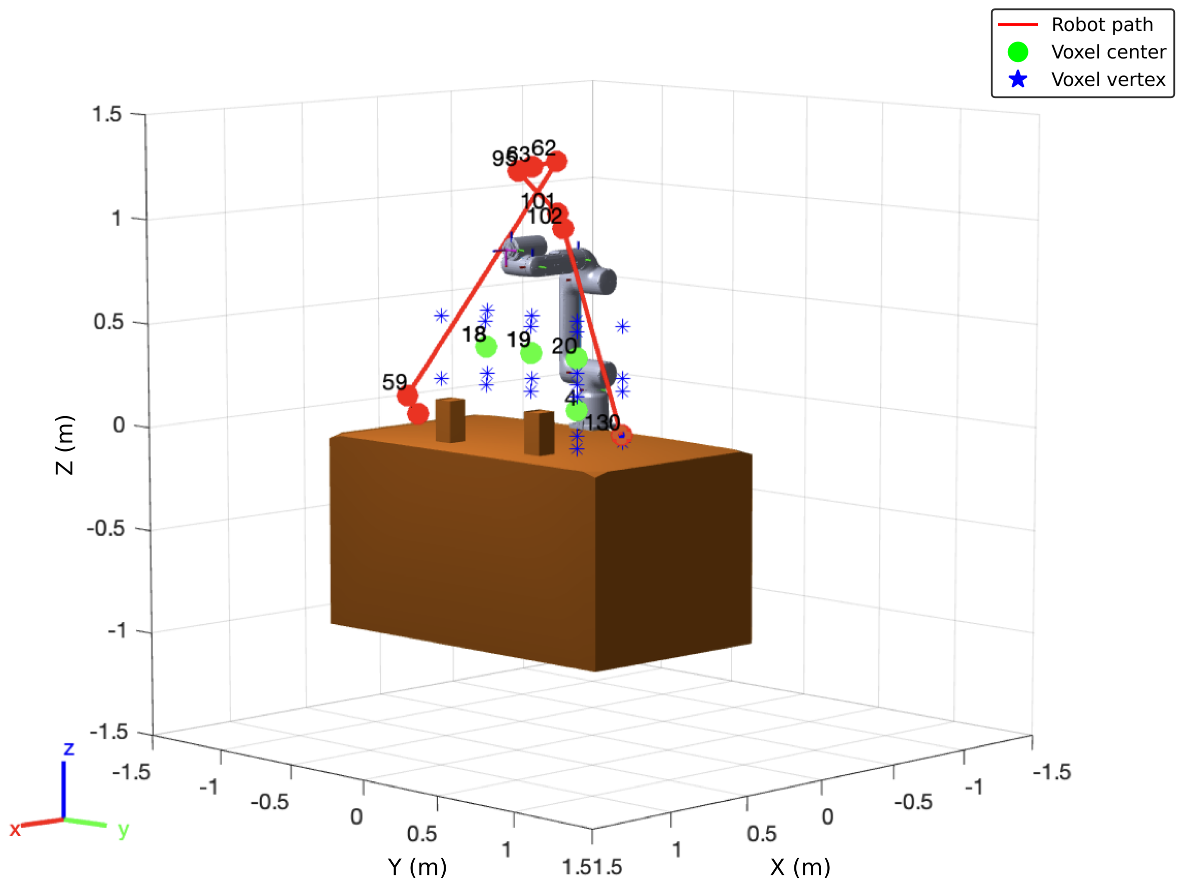


Figure 5.9: The optimal path to be followed by the manipulator to maintain in every moment a safe distance from the occupied voxel of the selected task is depicted in red

Voxel	Node	State	Action
4	1	4	10
19	59	1.817	96
18	62	1.909	98
18	63	1.940	118
31	95	2.945	99
19	101	3.119	100
31	102	3.162	101
20	130	4.000	-

Table 5.4: Sequences of voxels, nodes, states and actions for the selected task

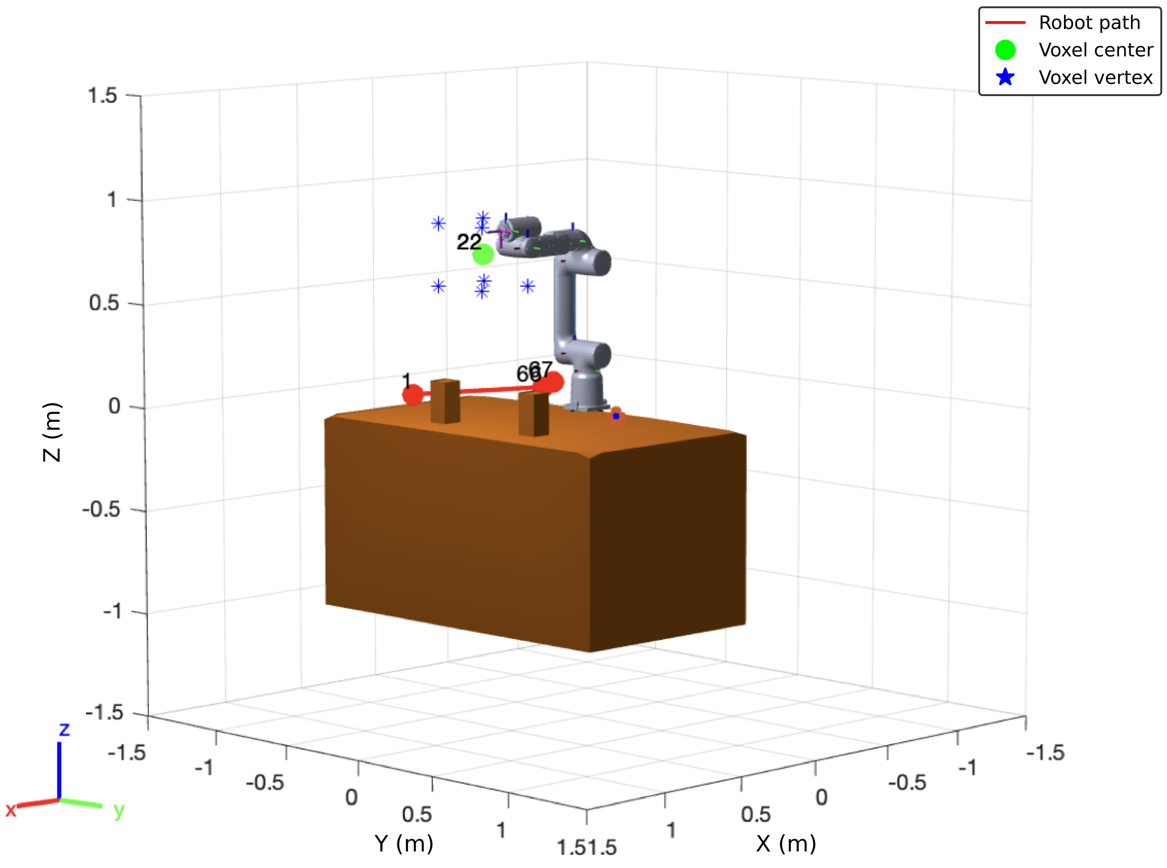


Figure 5.10: The optimal path to be followed by the manipulator is incomplete for the selected task

Voxel	Node	State	Action
22	1	22	6
22	66	2.037	39
22	67	2.068	?
?	?	?	?

Table 5.5: Sequences of voxels, nodes, states and actions for the selected task until the algorithm reaches a state where all the actions listed in the Q-table are unfeasible

6 | Experimental validation

6.1. Introduction

This chapter is dedicated to the validation of the proposed algorithm through its application on the GoFa™ CRB 15000 ABB robotic arm. The use case implementation took place in the MeRLIn Lab at Politecnico di Milano. In particular, this chapter details the experimental setup, including the robot and its controller, a workspace with static obstacles, and a Kinect for human detection. The chapter concludes by showcasing the validation results, demonstrating the algorithm effectiveness in the real world scenario.

6.2. Experimental setup

The experimental setup employed for the validation process is depicted in Figure 6.1 and Figure 6.2. The robot is fixed towards the front end of the table, leaving a wide space in front of the robot for operations and human interactions. The control pad is used for manually control the GoFa™ robotic arm. The screen displays a graphical interface that can show the status of the robot or provide different options for operations. The red button serves as an emergency stop mechanism and, when pressed, it allows operators to immediately stops all robot's movements. A Kinect sensor, used for detecting human presence and movement within the workspace, is positioned to monitor the area around the robot to ensure safe human-robot interaction, dynamically adjusting the robot's actions based on the sensor's input. In front of the GoFa™ robot, an ArUco marker is placed on the table. This black and white marker is used for pose estimation and it is often utilized in robotic applications for precise detection and spatial orientation, enabling the robot to interact with objects at specific coordinates within its operational space.

The simulation environment used during the algorithm training and testing phases mirrors the actual setup, including identical dimensions of the table, same positions of the static obstacles, and equal starting and goal configurations. Ensuring consistency between the simulated and real environments is critical for validating the algorithm's effectiveness.

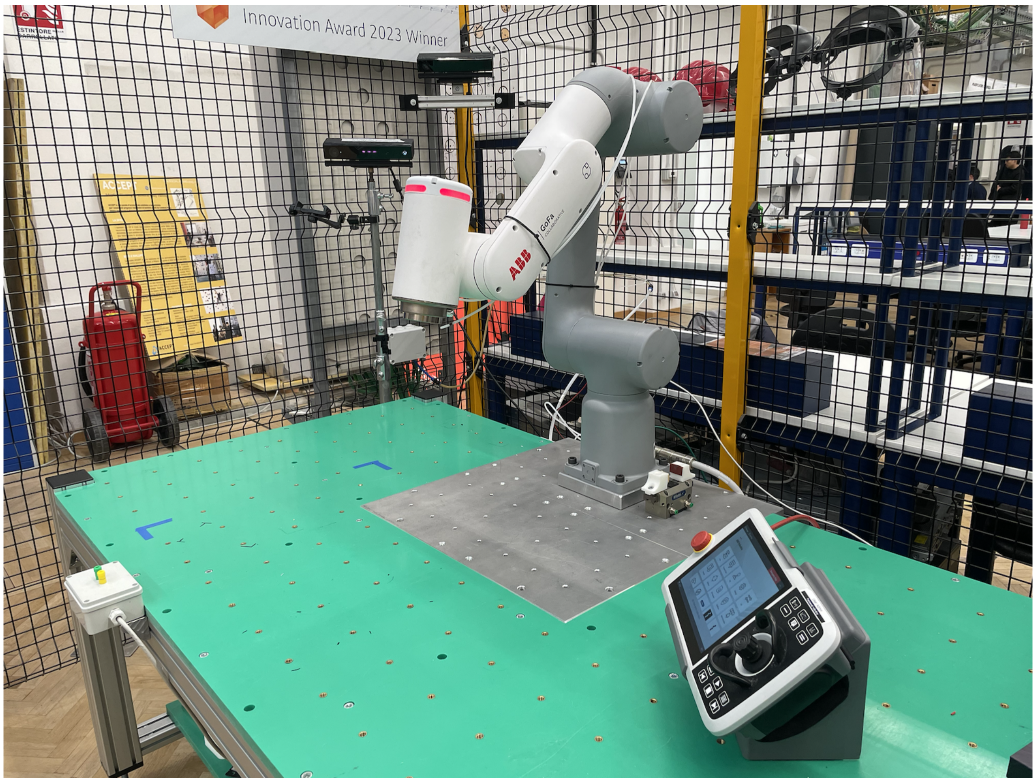


Figure 6.1: GoFa™ robotic arm in the workspace and Kinect sensing camera on the left

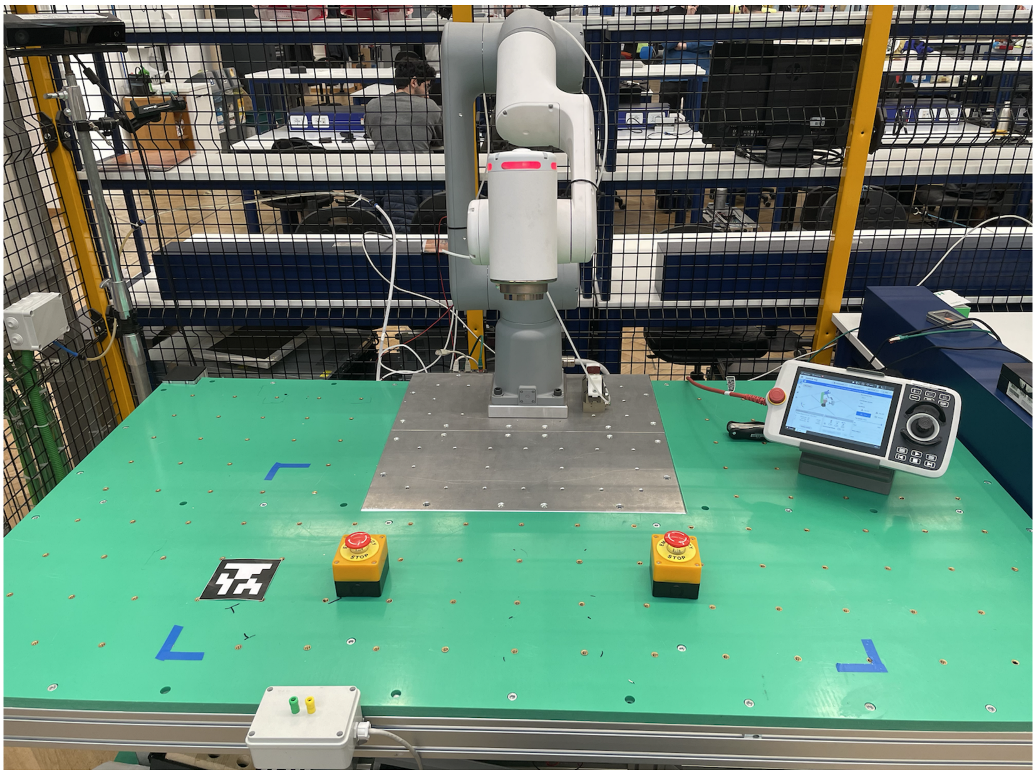


Figure 6.2: GoFa™ robotic arm in the workspace with static obstacles and ArUco detector

6.2.1. GoFa™ robotic arm

The proposed algorithm is validated on the GoFa™ CRB 15000 robotic arm, illustrated in Figure 6.3 and produced by ABB [1]. The 6-axis GoFa™ manipulator is tailored for safe, continuous operation alongside workers, simplifying installation and use in various industrial settings. Like all collaborative robots, it complies with *ISO/TS 15066* safety specification, incorporating sophisticated sensors for instantaneous contact response, soft pads, and a design free from trap points that could otherwise catch body parts or clothing. It is equipped with advanced torque and position sensors across its six joints, which ensure safety through superior power and force limitation capabilities, immediately halting in case of unexpected contact to prevent operator injuries. The GoFa™ robotic arm presents a payload capacity up to *5 kg*, speed up to *2,2 m/s*, and a working radius of *950 mm*. Designed for a wide array of tasks including material handling, assembly, packaging, machine maintenance, and more, its compact design does not sacrifice power for size. Further details can be found on the technical datasheet in Figure A.1 in the Appendix.

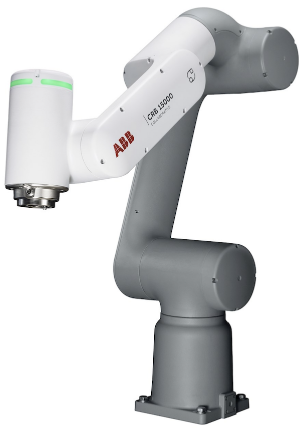


Figure 6.3: ABB GoFa™ CRB 15000

6.2.2. Kinect camera and ArUco markers

The human occupancy data in the workspace are collected by the Kinect Version 2 camera. Originally developed by Microsoft for gaming console, the Kinect V2 has found a wide range of applications, including human detection. Shown in Figure 6.4, the Kinect V2 is a sensing camera designed for motion tracking and spatial analysis. This technology integrates two cameras with differing resolutions: a RGB camera at *1920*x*1080* pixels for visual capture data and a depth camera at *512*x*424* pixels for detailed spatial analysis.

The RGB camera provides color coordinates in the two-dimensional space, whereas the depth camera offers depth coordinates, allowing for the calculation of distances from the camera to objects in the scene.

Figure 6.4: Kinect V2 camera

ArUco marker detectors, shown in Figure 6.5, are employed in the fields of computer vision and robotics to facilitate the translation of camera coordinates into world coordinates. ArUco markers are simple, black and white square markers that can be easily detected and decoded in images. The utility of ArUco markers lies in their ability to allow accurate pose estimation. Upon identifying an ArUco marker within an image, the system is capable of calculating the camera's orientation and position relative to the marker's real-world location. This calculation is based on the marker's predefined dimensions and its unique identifier, which determines its orientation.

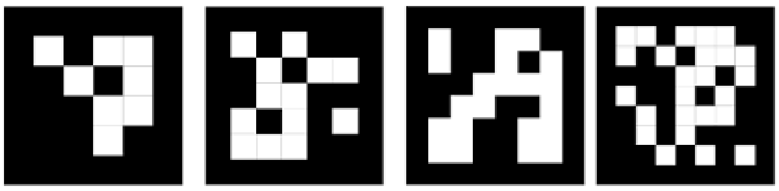


Figure 6.5: ArUco marker detectors with different IDs

6.3. Validation process

In the validation phase, the Kinect camera, in conjunction with an ArUco marker attached to the operator's wrist, is utilized for real-time detection of the human hand's position. This setup enables precise identification of the ArUco marker's center point by the camera, effectively pinpointing the exact location of the operator's right wrist. Then, the voxel occupancy map Algorithm, illustrated in Section 5.2, is employed to determine to which voxel, within the subdivided workspace, the given detected point belongs to.

A schematic representation of the validation process is detailed in Figure 6.6. The robot and the computer establish a communication via a socket connection, which facilitates real-time data exchange essential for the robot's operation and coordination. The validation algorithm, which has been developed in Python and operates on the computer side, requires the occupied voxel as input. The algorithm processes the current robot pose and the occupied voxel to compute the current state. It then employs the optimal policy to calculate the action to which the maximum cumulative reward is associated and generates the identification number, coordinates, and joint configurations for the subsequent node. On the other hand, the GoFa™ robot is operated using RAPID, an high-level programming language, which is specifically designed for articulating and controlling the movements of ABB robots. The RAPID program interprets the positional data provided by the Python validation algorithm. Upon receiving the coordinates for the next target point from the computer, the robot navigates towards that point. Every time the robotic arm reaches a new node, it sends a confirmation signal back to the computer and, at the same time, the Kinect system captures a new frame of the human hand, feeding this data into the algorithm. This triggers the Python algorithm to process and send the next set of movement instructions, ensuring a continuous operation. Throughout every test, this loop is continuously running to ensure the robotic arm's movements are synchronized with the human operator's hand movements, ensuring for real-time interaction and allowing the system to adapt dynamically to the actions of the human operator.

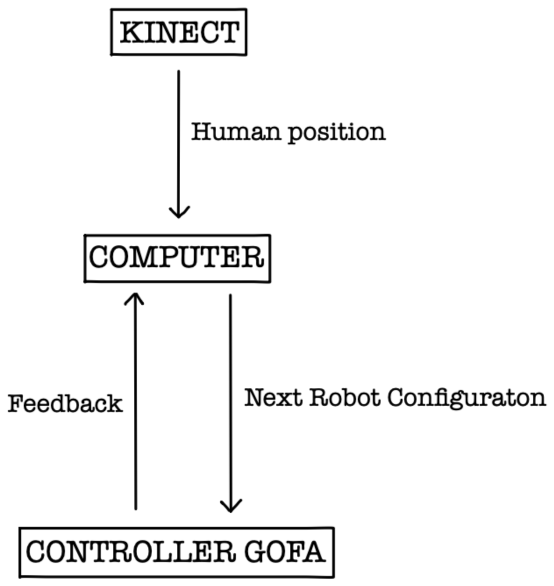


Figure 6.6: Schematic representation of the validation process

6.4. Results of the validation process

This section is dedicated to demonstrate the operational integrity of the deployed system. Here, the theoretical and computational efforts spent on training the Reinforcement Learning agent are translated into a practical application. Various tests have been conducted to validate the effectiveness of the algorithm, ensuring that the deployment not only works as intended but also delivers on the anticipated performance outcomes:

- "No human" category includes one test carried out without human presence in the workspace.
- "Static pose" category accounts for six tests carried out with the human hand maintaining a static position in the workspace, meaning that a single voxel is occupied throughout the task's duration.
- "One axis" category presents sixteen tests conducted with the human hand moving linearly along a single axis, resulting in a sequential occupation of voxels along either the x, y, or z-axis.
- "Random case" category comprises six tests in which the human is performing diverse tasks that require hand movement in all three axes concurrently.

All the tests confirmed the absence of collisions between the operator's hand and the robot's end-effector, affirming the algorithm's reliability to address safety issues in a real-world application. However, for a comprehensive validation of the algorithm, two distinct criteria have been delineated. Firstly, the time needed for the task completion serves as a critical benchmark. This parameter measures the algorithm's operational efficiency and its impact on productivity. Secondly, the maintenance of a minimum safety distance between the human's hand and the robot's end-effector throughout task execution is imperative. Their relative distance must exceed a predetermined threshold at all times during the task execution. This safety threshold has been set to be half the dimension of a voxel, which is equal to *15 cm*. This measure ensures a buffer zone around the operator, providing a standard to evaluate the algorithm's performance in maintaining workplace safety.

Thus, the robot's poses, captured during its movement from the starting point to the goal, are directly obtained from the GoFa™ controller at intervals of *0,1* seconds. Instead, the updates on the human hand's position are provided by the Kinect camera only when the robot reaches a node defined in the parent-child relationship graph. Consequently, the synchronization of human positional data with the robot's poses occurs exclusively at these significant nodes. Human positions are then subjected to quadratic interpolation from one frame to the next, ensuring a continuous representation of human movement in relation to the robot's trajectory.

Utilizing the recorded robot poses and human positions over time, the data points can be visualized on a graph. In the subsequent figures, the path followed by the robot is illustrated in blue, whereas the path traced by the human hand is marked in red. Robot's starting and goal positions are highlighted using a green and a red point, respectively. Figure 6.7 corresponds to test *1* from "One axis" category, specifically showcasing a human movement exclusively along the x-axis. Meanwhile, Figure 6.8 is associated with test *3* from "Random case" category, while Figure 6.9 refers to test *6* of the same category.

The movement patterns are crucial to understand the interaction dynamics between the robot and the human operator. These plots represent an accurate reproduction of what happened during the tests performed in the laboratory. The robot is able to respond to the real-time detection of the human trajectory and is also able to adjusts and deviates its trajectory adapting to the human movement, trying to strike a balance between always ensuring a maintaining safety distances and avoiding potential collisions while reaching the goal in the minimum possible time. These plots demonstrate the robot's agility and the effectiveness of the collision avoidance strategy, confirming that the policy developed during the training phase provides good results.

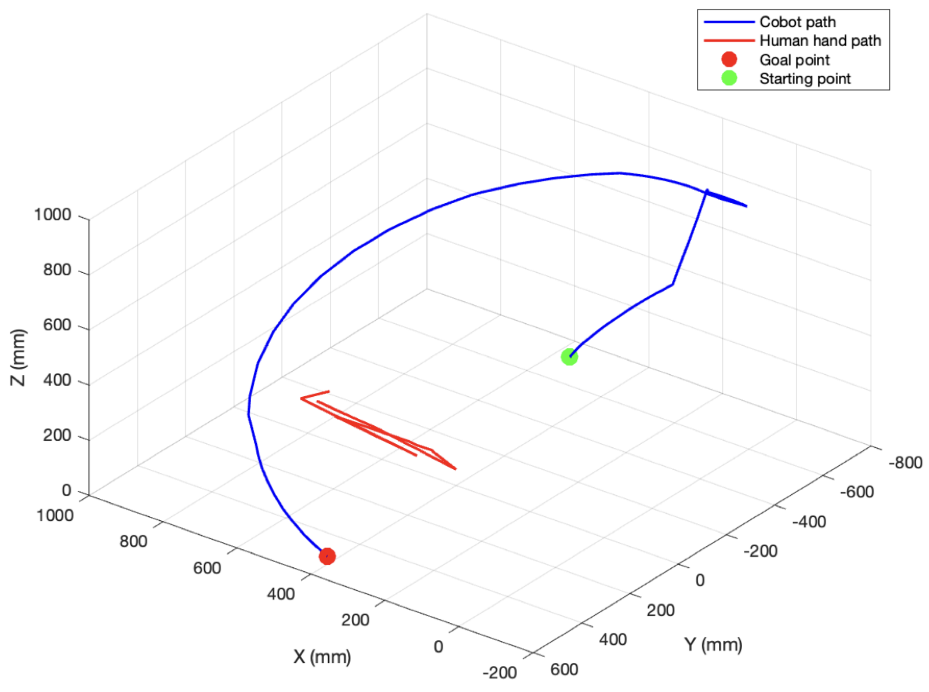


Figure 6.7: Robot and human hand paths for test *1* from "One axis" category

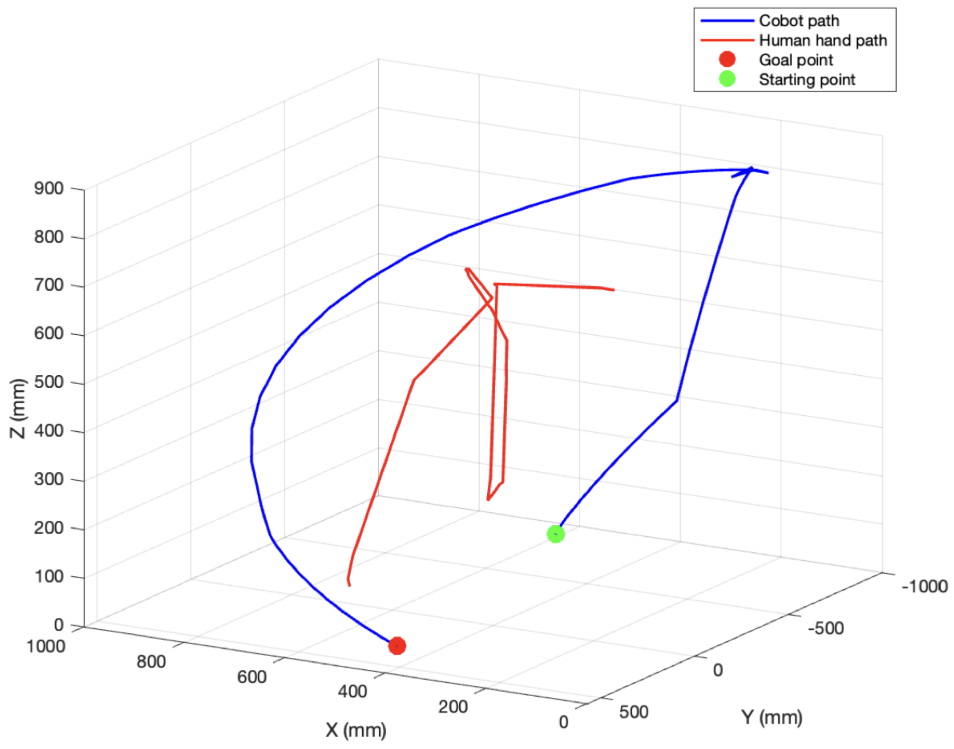


Figure 6.8: Robot and human hand paths for test *3* from "Random case" category

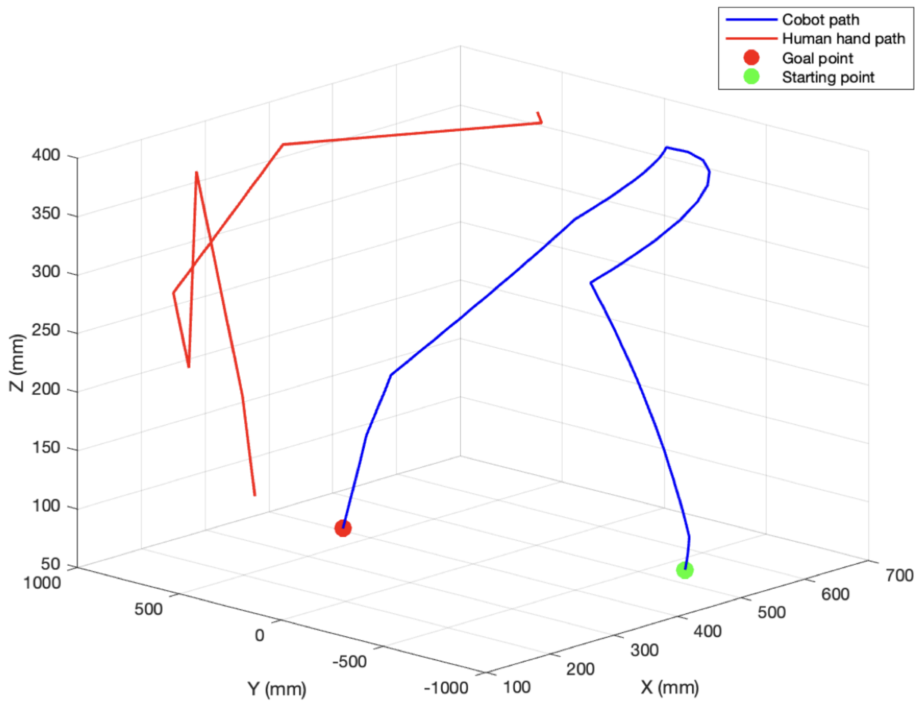


Figure 6.9: Robot and human hand paths for test *6* from "Random case" category

The tests results in terms of time required to complete the task, human occupancy volume, minimum distance between the human and the robot, maximum distance, average distance and standard deviation are reported in Table 6.1.

Test type	**ID**	**T [s]**	**V [cm^3]**	**D$_m$ [cm]**	**D$_M$ [cm]**	**D$_{avg}$ [cm]**	**σ [cm]**
No human	-	18,48	0,00	0,00	0,00	0,00	0,00
Static pose	1	28,79	6,23	44,58	97,00	78,18	15,09
	2	18,48	7,93	52,88	123,01	105,08	17,48
	3	30,58	10,23	21,78	89,68	69,12	19,33
	4	39,48	17,28	36,17	113,53	89,30	26,42
	5	29,71	8,22	32,80	129,66	108,07	25,56
	6	35,95	5,89	34,36	105,64	79,64	22,18
One axis	1	50,52	145,60	30,84	114,78	87,07	24,87
	2	29,70	252,31	54,76	135,78	119,21	23,04
	3	29,14	191,24	37,81	104,59	89,64	15,55
	4	39,28	269,56	34,98	114,47	87,62	24,67
	5	35,81	160,09	16,30	148,23	117,58	43,08
	6	28,07	9,03	35,85	149,62	76,43	36,11
	7	29,00	196,64	4,34	75,80	50,52	19,40
	8	43,05	430,37	29,34	141,31	91,63	29,39
	9	42,58	686,46	44,57	108,53	70,45	20,24
	10	39,17	960,02	19,24	137,15	81,38	34,93
	11	25,13	23,61	41,39	136,96	110,16	25,57
	12	35,11	67,91	49,44	111,44	92,37	21,19
	13	35,83	118,81	47,45	94,86	72,39	14,53
	14	86,30	68,90	55,69	91,41	83,78	9,47
	15	35,81	134,47	56,83	146,83	122,27	32,36
	16	38,04	79,94	36,89	143,32	118,64	35,29
Random case	1	35,82	1147,55	31,08	125,46	86,82	21,98
	2	36,11	621,71	17,52	104,23	85,42	16,73
	3	35,85	416,96	18,49	119,34	81,56	22,10
	4	39,59	967,10	25,71	113,53	79,86	21,01
	5	22,06	49,54	36,29	143,44	88,68	39,84
	6	27,60	347,98	22,56	98,70	72,12	33,56

Table 6.1: Results of the validation process

The shortest time needed to perform the task is recorded at *18,48 s*, which is associated to the test where no human is detected in the workspace and to the test with ID equal to *2* from the "Static pose" category. Conversely, the longest task completion time, clocking in at *86,30 s*, is observed for the task labeled as ID *14* within the "One axis" test category. The volume is computed by considering the sequence of point coordinates occupied by the human's hand over time as a point cloud volume, which encapsulates the three-dimensional space covered by a point cloud. This volume represents an indirect metric of the distinct voxels occupied by the human's hand during its movement. The occupancy volume varies significantly, ranging from *0* cm^3 to *1147,55* cm^3. As expected, this parameter is lower in the "Static pose" tests, where the human is confined to a single voxel, reaching its peak in the "Random case" test category as it reflects the dynamic nature of human movement within the workspace.

The minimum distance maintained between the human hand and the robot's end-effector consistently exceeds the established safety threshold of *15 cm*, except for one case. In the test *7* of the "One axis" category the minimum distance is reported to be equal to *4,34 cm*. Upon detailed examination of the case at hand, it was clear that this deviation occurred because the reward for moving to the following next node was calculated as positive, given the node's sufficient distance from the currently occupied voxel. Nevertheless, the actual trajectory required for the robot to transition to this next node passed dangerously close to the occupied voxel, despite both the starting and subsequent nodes meeting the distance criteria for reward assignment. To avoid such situations and enhance safety, introducing additional nodes along this segment would be necessary to guide the robot along a safer path. Moreover, a notable constraint is that the positions of the hand across successive frames are derived through interpolation, which means they don't accurately mirror the hand's real-time location. Consequently, while the actual real-time distance may have been greater, relying on interpolated data introduces a degree of approximation, potentially distorting the true spatial relationship between the hand and the robot.

Figure 6.10 showcases the data collected from all the tests using a scatter plot. The x-axis reports the volume occupied by the human within the robot's operational space while the y-axis displays the task execution time. The plot illustrates the observations based on the typology of the tests conducted. The singular test executed without any human presence in the workspace is depicted by a light blue circle. Data from the "Static pose" experiment are marked in red, while the blue marks represent the "One axis" tests and the green marks highlight the "Random case" experiments. The correlation coefficient, calculated across all the experiments, between the variables task execution time and volume occupied by the human in the workspace, is equal to *0,16*. This low value suggests a weak relationship between the two variables. As a result, it is possible to conclude that the method's efficiency, identified by the time taken to complete the task, is not influenced by whether the human's movement is static or dynamic within the workspace. The task execution time is not influenced by the nature of the test, i.e. whether the hand is stationary or in motion, but rather by the specific voxel occupied at the moment the robot is asked to make a decision. The identification number of the occupied voxel, which identifies its position in the workspace, affects the decision-making process. The extensive range of the path network, with its numerous alternative routes, enables the robot to adjust its trajectory responsively to the human's changing location and the associated spatial occupation within the workspace.

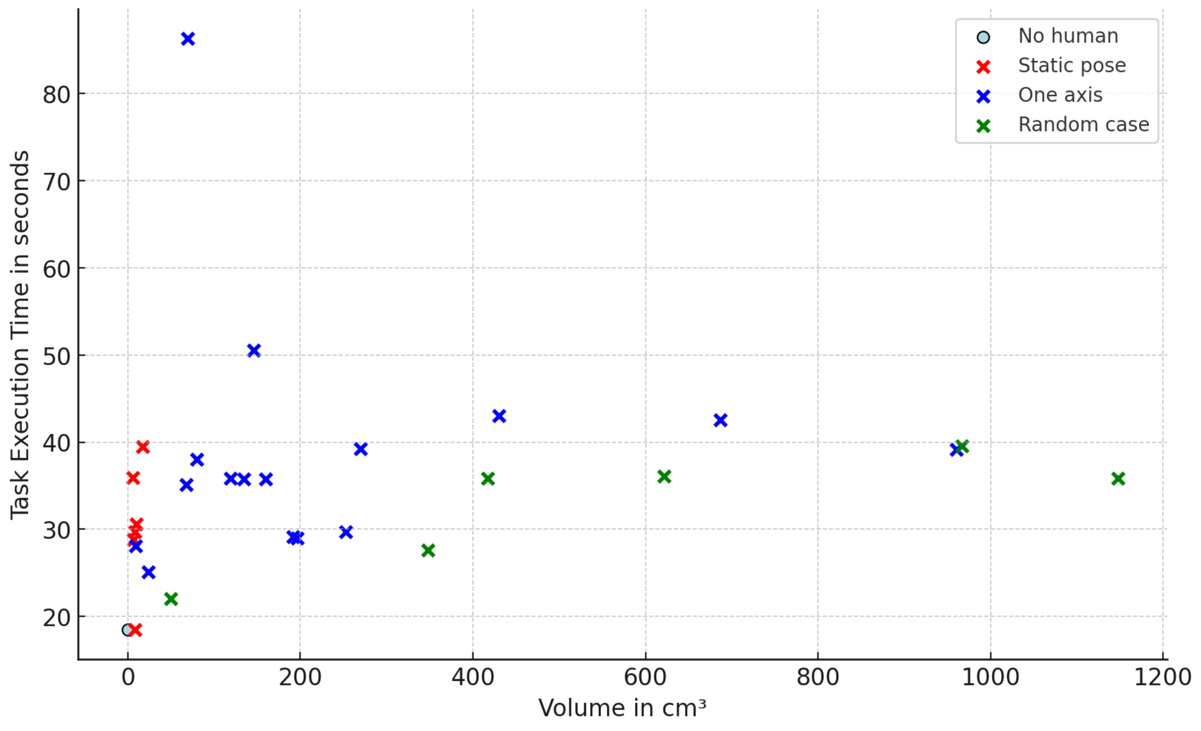


Figure 6.10: Data collected from each category of the performed tests in terms of time to task execution vs. human occupancy volume

It can be noticed that an outlier, which is a data point that significantly deviates from the other points, is presented in the plot. The outlier point (x = *68,9 cm*^{3} and y = *86,3 s*), is characterized by a high value of time to task execution compared to the others. This outlier is distinct from the rest of the data, potentially indicating a test where the task execution time was unusually long, likely due to unforeseen circumstances or complexities during the test. The presence of the outlier point can be justified by analyzing more in details the sequences of nodes and actions performed during test *14* of the "One axis" category. In the initial phase of the test, the robot efficiently adjusts its trajectory, selecting a certain path in response to the presence of the human. The robot's route progression is mapped out through the following series of nodes: *1*, *8*, *10*, *11*, *13*, *16*, *121*, *78*, *77* and *122*. Upon reaching node *122*, the execution of the task is momentarily suspended as the algorithm determines that the following node *123* is within an unsafe proximity to the voxel currently occupied by the human. After consulting the Q-table for this particular scenario, the most appropriate action to be taken results in a self-transition to the same node, until it is safe to proceed. The same situation occurs during the transition between nodes *123* and *125* and between nodes *125* and *126*. A detailed description of the time intervals for each segment of the trajectory is outlined in Table 6.2. The task execution is paused at node *122*, requiring *8,05 s* to progress the next node. Similarly, navigating to node *125* requires *10,19 s*, while advancing to node *126* is accomplished in *7,99 s*.

Start	**Goal**	**Time [s]**
1	8	0,79
8	10	3,40
10	11	5,20
11	13	1,80
13	16	1,48
16	121	1,65
121	78	1,80
78	77	0,80
77	122	2,94
122	**123**	**8,05**
123	**125**	**10,19**
125	**126**	**7,99**
126	130	4,39

Table 6.2: Time intervals required to move between two subsequent nodes for the outlier

In conclusion, the task execution time registers a minimum of *18,48 s* seconds across two tests. The inclusion of the human within the workspace leads to an increase in this value, ranging from no change *0%* to as much as *173%*, with this calculation omitting the outlier. Including the outlier in our analysis, the percentage increase in execution time rises up to *300%*, highlighting the significant impact human presence can have on operational efficiency in this context.

7 | Conclusions and future developments

7.1. Final considerations

In collaborative robotic environments, safety is a critical concern. It is essential for robots to employ techniques for human activities recognition and prediction as a fundamental part of their decision-making processes. This thesis focused on Human-Robot Collaboration, in particular on the development of a Machine Learning algorithm that allows robots to detect and respond to the human presence and movements. This capability is crucial for preventing collisions and injuries, thereby enhancing safety in environments where humans and robots work closely together. Additionally, the proposed algorithm ensures an efficient execution of the task, demonstrating that safety measures can coexist with productivity in a shared workspace.

An intriguing aspect of this research is that motion planning and decision-making algorithms are entirely developed either offline or within a simulated environment to bypass complex real-time calculations. The robot undergoes a training phase to learn how to navigate the complex network of previously created paths, focusing on enhancing its path selection strategies. By familiarizing with the network configuration and human movement patterns within the shared workspace, the robot exploits the acquired knowledge optimally, becoming able to find the shortest path to reach the goal but also to choose a path that ensures safety, maintaining at every time instant a minimum distance from the current position of the human operator. Following this, the optimal policy is deployed to be evaluated and proven in a real-world application. The experimental validation of the algorithm on the GoFa™ robotic arm confirmed its effectiveness, initially demonstrated during the offline testing phases, in navigating the workspace without coming into contact with the operator's hand. The algorithm enables real-time detection of human presence within the workspace, allowing the robot to alter its trajectory timely to maintain a safe distance from humans or to halt completely in the event of an imminent collision.

A significant advantage of the algorithm is that if the hand and the robot are in close proximity, occupying the same voxel, the robot automatically stops, effectively preventing any potential collisions. This safety mechanism is added to enhance the policy developed during the training phase. Specifically, when the robot encounters a scenario where all possible actions from a given node are associated with negative values in the Q-table, it remains stationary until the human moves to a different voxel, thereby updating the current state. This criterion was designed to elevate safety levels during real-time operations. However, it is important to note that this safety mechanism, while elevating safety levels, also increases the task completion time, as the robot must pause its operation, potentially extending the duration from the start to the final configuration.

7.2. Future developments

Although the proposed algorithm successfully meets the task requirements and fulfills the objectives of this thesis, it presents some limitations. This sections outlines potential areas for improvement and future developments of the algorithm.

The BiRRT algorithm, employed for creating the dataset of paths, is precisely customized for the specific environment and the robot being used. In fact, the algorithm generates paths connecting a starting point to a final point, thereby fixing the robot starting and goal configurations. Furthermore, the robot possesses distinct features, including six degrees of freedom and predefined joint lengths. Consequently, the paths identified are expressed in configuration terms, making them uniquely applicable to the robot in question. The environment also contains two static obstacles, with the algorithm's generated paths maneuvering around them. Should there be alterations to the workspace, robot specifications, starting and goal locations, or the positions and sizes of obstacles, a reconfiguration is needed to recreate the navigational trees and extrapolate paths. This requirement underscores the algorithm's adaptability in accommodating changes within its operational environment.

Another limitation of the proposed algorithm is that it focuses only on the interaction between a single human hand and the robot's end-effector, ensuring collision-free paths exclusively concerning these two elements. Essentially, while it effectively prevents collisions between the human hand and the robot's end-effector, it does not account for potential collisions involving other parts of the human body or other components of the robot. To enhance the algorithm's capability in addressing this limitation, future improvements could involve incorporating the detection of multiple voxels simultaneously occupied by the operator to better represent different parts of the human body. This enhancement necessitates enlarging the data structures, such as matrices and the Q-table, as the system states would now encompass a robot's pose in conjunction with various combinations of occupied voxels (for instance, a robot's pose with voxels *1* and *2* occupied, followed by voxels *1* and *3*, followed by voxels *2* and *3*, and so on). Consequently, this adjustment would lead to an expansion in the size of matrices and an increase in computational costs. While moving towards a more comprehensive approach for collision avoidance significantly improves safety measures in human-robot interactions, it simultaneously demands more computational power and extends the training time for the algorithm.

A similar potential refinement of the current algorithm that involves increasing the dimension of the problem, could incorporate the examination of a temporal sequence of voxels occupied by the hand. Through the application of predictive human behavior models, it is possible to anticipate the future positions of humans and the specific voxels they are likely to occupy. This strategy necessitates a redefinition of states within the Reinforcement Learning framework to account for multiple robot nodes and voxels simultaneously occupied by the human. Additionally, this enhancement necessitates a re-calibration of the reward system to prioritize positions in the near future with greater precision, while attributing lesser significance to more distant, uncertain positions. The feasibility of this approach is supported by the knowledge of the parent-child relationship network, which delineates the sequence of nodes the robot must navigate to achieve its final configuration, thereby facilitating strategic long-term decision-making informed by a predictive understanding of human movement patterns.

Moreover, the implemented algorithm halts its operation when the voxel associated with the next node is occupied by the operator's hand and resumes movement only after the operator has moved away. While this measure significantly increases safety during the task execution, it also extends the duration required for the robot to move from the starting point to the goal. A possible improvement for the algorithm could involve allowing the robot not to stop but to backtrack along the previously followed nodes until it reaches the nearest prior junction node. At this point, it could select an alternate route. This modification would be advantageous in situations where a particular voxel remains occupied by a human for an extended period. It wouldn't necessitate additional calculations since the Q-table remains unchanged. The modification lies in how the algorithm interprets it, as the logic and data driving the robot's decision-making process stay constant. This strategy could optimize the robot's path-finding efficiency, particularly in dynamic environments, by reducing idle times and adapting to changes in the workspace.

Bibliography

[1] ABB. Gofa™ CRB 15000. URL `https://tinyurl.com/ABBGoFA`.

[2] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

[3] L. Antão, J. Reis, and G. Gonçalves. Voxel-based space monitoring in human-robot collaboration environments. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 552–559. IEEE, 2019.

[4] D. B. Aranibar and P. J. Alsina. Reinforcement learning-based path planning for autonomous robots. In *EnRI-XXIV Congresso da Sociedade Brasileira de Computaç ao*, volume 10, 2004.

[5] L. Balan and G. M. Bone. Real-time 3d collision avoidance method for safe human and robot coexistence. In *2006 IEEE/RSJ international conference on intelligent robots and systems*, pages 276–282. IEEE, 2006.

[6] L. Blasi, E. D'Amato, M. Mattei, and I. Notaro. Path planning and real-time collision avoidance based on the essential visibility graph. *Applied Sciences*, 10(16):5613, 2020.

[7] B. Boyacioglu and S. Ertugrul. Time-optimal smoothing of rrt-given path for manipulators. In *ICINCO (2)*, pages 406–411, 2016.

[8] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, J. Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

[9] E. Gambao. Analysis exploring risks and opportunities linked to the use of collaborative industrial robots in Europe. *Panel for the Future of Science and Technology*, 2023.

[10] IBM. What is supervised learning?, . URL `https://tinyurl.com/IBMulearning`.

[11] IBM. What is unsupervised learning?, . URL `https://tinyurl.com/IBMslearning`.

[12] *Robots and robotic devices – Collaborative robots, ISO/TS 15066:2016*. International Organization for Standardization.

[13] S. James and E. Johns. 3d simulation for robot arm control with deep q-learning. *arXiv preprint arXiv:1609.03759*, 2016.

[14] M. A. K. Jaradat, M. Al-Rousan, and L. Quadan. Reinforcement based mobile robot navigation in dynamic environment. *Robotics and Computer-Integrated Manufacturing*, 27(1):135–149, 2011.

[15] M. Ji, L. Zhang, and S. Wang. A path planning approach based on q-learning for robot arm. In *2019 3rd International Conference on Robotics and Automation Sciences (ICRAS)*, pages 15–19. IEEE, 2019.

[16] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[17] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller. Anytime motion planning using the rrt. In *2011 IEEE international conference on robotics and automation*, pages 1478–1483. IEEE, 2011.

[18] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[19] B. Lacevic, P. Rocco, and M. Strandberg. Safe motion planning for articulated robots using rrts. In *2011 XXIII International Symposium on Information, Communication and Automation Technologies*, pages 1–7. IEEE, 2011.

[20] P. A. Lasota and J. A. Shah. A multiple-predictor approach to human motion prediction. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2300–2307. IEEE, 2017.

[21] B. Lau, C. Sprunk, and W. Burgard. Efficient grid-based spatial representations for robot navigation in dynamic environments. *Robotics and Autonomous Systems*, 61 (10):1116–1130, 2013.

[22] S. LaValle. Rapidly-exploring random trees: A new tool for path planning. *Research Report 9811*, 1998.

[23] N. Le, V. S. Rathour, K. Yamazaki, K. Luu, and M. Savvides. Deep reinforcement learning in computer vision: a comprehensive survey. *Artificial Intelligence Review*, pages 1–87, 2022.

[24] E. Lee. Javascript implementation of the Ramer Douglas Peucker algorithm. URL `https://karthaus.nl/rdp/`.

[25] T. Lindner, A. Milecki, and D. Wyrwał. Positioning of the robotic arm using different reinforcement learning algorithms. *International Journal of Control, Automation and Systems*, 19:1661–1676, 2021.

[26] N. Lucci, B. Lacevic, A. M. Zanchettin, and P. Rocco. Combining speed and separation monitoring with power and force limiting for safe collaborative robotics applications. *IEEE Robotics and Automation Letters*, 5(4):6121–6128, 2020.

[27] M. Matulis and C. Harvey. A robot arm digital twin utilising reinforcement learning. *Computers & Graphics*, 95:106–114, 2021.

[28] N. A. Melchior and R. Simmons. Particle rrt for path planning with uncertainty. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 1617–1624. IEEE, 2007.

[29] M. Müller, T. Röder, M. Clausen, B. Eberhardt, B. Krüger, and A. Weber. Documentation mocap database hdm05. Technical Report CG-2007-2, Universität Bonn, June 2007.

[30] K. Naderi, J. Rajamäki, and P. Hämäläinen. Rt-rrt* a real-time path planning algorithm based on rrt. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, pages 113–118, 2015.

[31] S. Patil, V. Vasu, and K. Srinadh. Advances and perspectives in collaborative robotics: a review of key technologies and emerging trends. *Discover Mechanical Engineering*, 2(1):13, 2023.

[32] S. Pellegrinelli, F. L. Moro, N. Pedrocchi, L. M. Tosatti, and T. Tolio. A probabilistic approach to workspace sharing for human–robot cooperation in assembly tasks. *CIRP Annals*, 65(1):57–60, 2016.

[33] D. Preuveneers and E. Ilie-Zudor. The intelligent industry of the future: A survey on emerging trends, research challenges and opportunities in industry 4.0. *Journal of Ambient Intelligence and Smart Environments*, 9(3):287–298, 2017.

[34] M. Ragaglia, A. M. Zanchettin, and P. Rocco. Trajectory generation algorithm for safe human-robot collaboration based on multiple depth sensor measurements. *Mechatronics*, 55:267–281, 2018.

[35] R. Ris-Ala. *Fundamentals of Reinforcement Learning*. Springer Nature, 2023.

[36] M. Safeea, P. Neto, and R. Bearee. misc collision avoidance for collaborative robot manipulators by adjusting offline generated paths: An industrial use case. *Robotics and Autonomous Systems*, 119:278–288, 2019.

[37] A. R. Sharma and P. Kaushik. Literature survey of statistical, deep and reinforcement learning in natural language processing. In *2017 International conference on computing, communication and automation (ICCCA)*, pages 350–354. IEEE, 2017.

[38] H. Shehawy, D. Pareyson, V. Caruso, S. De Bernardi, A. M. Zanchettin, and P. Rocco. Flattening and folding towels with a single-arm robot based on reinforcement learning. *Robotics and Autonomous Systems*, 169:104506, 2023.

[39] D. Silver, R. S. Sutton, and M. Müller. Reinforcement learning of local shape in the game of go. In *IJCAI*, volume 7, pages 1053–1058. Citeseer, 2007.

[40] T. Simonini. The Bellman Equation: simplify our value estimation. URL `https://huggingface.co/learn/deep-rl-course/unit2/bellman-equation`.

[41] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[42] The MathWorks, Inc. Reinforcement Learning with MATLAB, . URL `https://tinyurl.com/RLwMatlab`.

[43] The MathWorks, Inc. manipulatorRRT, . URL `https://it.mathworks.com/help/robotics/ref/manipulatorrrt.html`.

[44] V. Villani, F. Pini, F. Leali, and C. Secchi. Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications. *Mechatronics*, 55: 248–266, 2018.

A | Appendix

Specification	
	GoFa 5
Reach (mm)	950 (wrist) 1050 (flange)
Payload (kg)	5
Arm load (kg)	1 (mounted on axis 4)
Number of axes	6
Protection	IP54
Mounting	Any angle, including table mounting, wall mounting, and ceiling mounting
Controller	OmniCore C30
Customer power supply	24V/2A supply
Customer signals	4 signals (for IO, Fieldbus, or Ethernet)
Tool flange	Standard ISO 9409-1-50
Functional safety	SafeMove Collaborative included all safety functions certified to Category 3, PL d

Performance	
	GoFa 5
Max TCP Velocity	2,2 m/s
Max TCP acceleration (Controlled motion for nominal load)	36,9 m/s²
Max TCP acceleration (e-stop for nominal load)	61,6 m/s²
Pose repeatability	0,02 mm

Physical	
	GoFa 5
Dimensions robot base	165 x 165 mm
Weight	28 kg

Movement		
	GoFa 5	
Axis movement	Working range	Axis max. speed
Axis 1 rotation	-180° to 180°	125 °/s
Axis 2 arm	-180° to 180°	125 °/s
Axis 3 arm	-225° to 85°	140 °/s
Axis 4 wrist	-180° to 180°	200 °/s
Axis 5 bend	-180° to 180°	200 °/s
Axis 6 turn	-270° to 270°	200 °/s

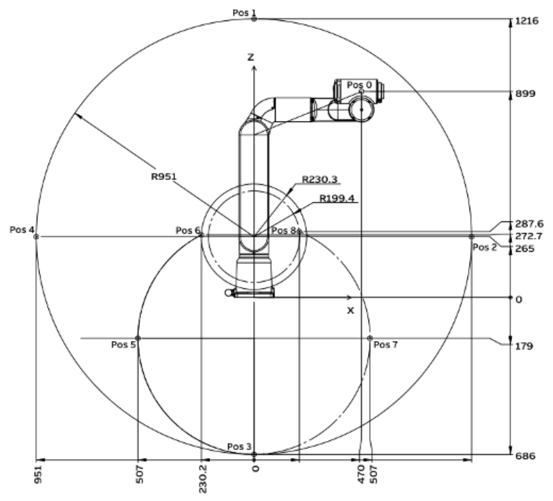


Figure A.1: Technical datasheet of the ABB GoFa™ CRB 15000 cobot (from [1])