POLITECNICO DI MILANO

School of Industrial and Information Engineering
Master of Science in Automation and Control Engineering

POLITECNICO
MILANO 1863

Collaborative robot scheduling based on reinforcement learning in industrial assembly tasks

Supervisor:
Prof. Paolo ROCCO
Co-supervisor:
Prof. Andrea Maria ZANCHETTIN
Ing. Riccardo MADERNA

Master of science dissertation of:
Giovanni FIORAVANTI, student ID 899805
Daniele SARTORI, student ID 899680

Academic Year 2018-2019

Ringraziamenti

Milano, 6 Giugno 2020

Prima di presentare il lavoro sviluppato in questa tesi, desideriamo ringraziare le persone che ci hanno supportato durante questo progetto e per tutto il nostro percorso universitario. In particolare ringraziamo il professor Paolo Rocco e il professor Andrea Maria Zanchettin per il contributo ed il supporto nella stesura di questa tesi. Con gratitudine, desideriamo ringraziare l'ingegner Andrea Casalino e l'ingegner Riccardo Maderna, che ci hanno seguito durante la nostra attività di ricerca senza mai farci mancare i loro consigli.
Desideriamo inoltre ringraziare tutti i nostri amici, che ci hanno accompagnato durante questi anni di studio. Infine un grazie particolare alle nostre famiglie, che ci hanno permesso di intraprendere questo percorso universitario e che ci hanno sostenuto ed accompagnato in ogni momento. Con grande riconoscenza,

Giovanni e Daniele

Contents

Abstract **VIII**

Sommario **IX**

1 Introduction **1**

1.1 Field of application 1

1.2 Research question and contribution 2

1.3 Thesis structure 4

2 State of the Art **6**

3 Theoretical Background **9**

3.1 Markov Decision Process 10

3.2 Stochastic Shortest Path problem 12

3.3 Reinforcement Learning: an overview 13

3.3.1 Classification 14

3.3.2 Exploitation and Exploration trade-off 15

4 Use case: Industrial Assembly **16**

4.1 YuMi 16

4.2 Description of the product 17

4.3 Setup Design 18

4.4 Modeling a collaborative task as an MDP 20

5 Reinforcement Learning solution **25**

5.1 Implemented Algorithms 25

5.1.1 Q-Learning 26

5.1.2 Delayed Q-Learning 28

5.2 Analysis of the results 31

5.2.1 Sensitivity analysis of the parameters 33

5.2.1.1 Test: Q-Learning 33

5.2.1.2 Test: Delayed Q-Learning 38

5.2.2 Performance comparison 43
5.2.2.1 Scenario: n=6 free actions 43
5.2.2.2 Scenario: n=55 free actions 45
5.2.2.3 Scenario: n=326 free actions - scheduling fully undefined . 48
5.2.3 Optimal scheduling 52

6 GUI for Digital Twin generation 57
6.1 Motivations 57
6.2 From the workflow to the digital twin 58

7 Simulation-based RL solution 73

8 Conclusions and Future Developments 78

Appendix 80
.1 IRB 14000 YuMi Datasheet 80

List of Figures

1.1 History of the Industry 1
1.2 Human-robot collaboration 2
1.3 Transition A and B simultaneously enabled 3

2.1 Allocation method proposed in [6] 6
2.2 Hierarchical framework of the allocation technique proposed in [9] 7

3.1 Sequential decision making loop 9
3.2 Example of a MDP 10
3.3 Example of a SSP 13

4.1 IRB 14000 YuMi 17
4.2 The shaker assembled in our use case 17
4.3 Pieces of the upper part 18
4.4 Pieces of the lower part 18
4.5 Setup of our use case 19
4.6 Caps support 20
4.7 The SSP of the use case 24

5.1 Plot of the parameter strategies 34
5.2 Case: normal reward. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best strategy highlighted 35
5.3 Case: averaged reward. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best strategy highlighted 35
5.4 Dispersion of the minimum number of episodes to achieve the optimal policy with normal reward (left) and averaged reward (right) 37
5.5 Case: zero human variance. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best $\epsilon 1$ setting highlighted 38

5.6 Case: low human variance. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best $\epsilon 1$ setting highlighted 38
5.7 Case: high human variance. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best $\epsilon 1$ setting highlighted 39
5.8 Dispersion of the minimum number of episodes to achieve the optimal policy with zero human variance (left), low human variance (center) and high human variance (right) 40
5.9 Case: low human variance. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best m setting highlighted 41
5.10 Case: high human variance. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best m setting highlighted 41
5.11 Dispersion of the minimum number of episodes to achieve the optimal policy with low human variance (left) and high human variance (right) 42
5.12 MDP with n=6 free actions 44
5.13 Scenario: n=6 free actions. Plot of the total reward per episode (top left) and plot of the 10-episodes averaged total reward (bottom left) with a low human variance. Plot of the total reward per episode (top right) and plot of the 10-episodes averaged total reward (bottom right) with a high human variance 44
5.14 Dispersion of the minimum number of episodes to achieve the optimal policy with low human variance (left) and high human variance (right) 45
5.15 MDP with n=55 free actions 46
5.16 Scenario: n=55 free actions. Plot of the total reward per episode (top left) and plot of the 10-episodes averaged total reward (bottom left) with a low human variance. Plot of the total reward per episode (top right) and plot of the 10-episodes averaged total reward (bottom right) with a high human variance 46
5.17 Dispersion of the minimum number of episodes to achieve the optimal policy with low human variance (left) and high human variance (right) 47
5.18 MDP with n=326 free actions 48
5.19 Scenario: n=326 free actions. Plot of the total reward per episode (top left) and plot of the 10-episodes averaged total reward (bottom left) with a low human variance. Plot of the total reward per episode (top right) and plot of the 10-episodes averaged total reward (bottom right) with a high human variance 49

5.20 Dispersion of the minimum number of episodes to achieve the optimal policy with low human variance (left) and high human variance (right) 50
5.21 Regret of a policy 51
5.22 Plot of the regret per episode with a low human variance (left) and with a high human variance (right) 51
5.23 Optimal path of the industrial assembly use case 52
5.24 Utilization and efficiency of a manufacturing task 54
5.25 Plot of the wait time per episode for the human (left) and for the robot (right) 54
5.26 Plot of the sum of the human wait time and robot wait time per episode . . 55

6.1 Pipeline of the application 58
6.2 Graphical elements 59
6.3 Use case workflow 60
6.4 Focus on a piece of the use case workflow and associated piece of graph . . . 61
6.5 Focus on sections of the digital twin representation 66

7.1 Policy iteration alternation between evaluation and improvement phases . . 75
7.2 Plot of the performances with Q-Learning (top) and Delayed Q-Leaning (bottom). Each circle represents a optimal scheduling returned by a technique. The circle size measures the closeness of the optimal scheduling cycle time to the reference one (larger implies closer). The circle colour measures the probability that the optimal scheduling is the reference one (green implies high probability) 76

List of Tables

4.1 Sub-action descriptions 21
4.2 Sub-actions assignment to a sub-agent 22

5.1 Sub-action expected durations 32
5.2 Case: normal reward. Median values of the minimum number of episodes to achieve the optimal convergence. The best result is highlighted 36
5.3 Case: averaged reward. Median values of the minimum number of episodes to achieve the optimal convergence. The best result is highlighted 36
40table.caption.36
42table.caption.40
5.6 Scenario: n=6 free actions. Minimum number of episodes to achieve the optimal policy 45
5.7 Scenario: n=55 free actions. Minimum number of episodes to achieve the optimal policy 47
5.8 Scenario: n=326 free actions. Minimum number of episodes to achieve the optimal policy 49
5.9 Optimal scheduling of the industrial assembly use case 53

6.1 Description of the classes and their attributes 72

List of Algorithms

1 Q-Learning . . . 27
2 Delayed Q-Learning . . . 29
3 Graph . . . 62
4 Graph_Generator . . . 63
5 Define_inp_transitions . . . 64
6 Assign_ID . . . 65
7 MDP_States_Generator . . . 67
8 Move_among_branch . . . 68
9 Move_along_branch . . . 69
10 MDP_Actions_Generator . . . 70
11 Generate_Actions . . . 70
12 Find_State . . . 71
13 Policy Iteration . . . 74

Abstract

In classical automation, control systems are frequently guided by PLC logics. In this sequential process, the problem of how to choose when two or more actions are simultaneously available may arise. To overcome this problem, precedence rules are normally used. Sometimes, the definition of these rules is based on a priori knowledge of the system. Most often, they rely on intuition or implement simple tie-breaking rules with no clear foundation. In the thesis we solve this problem exploiting the tools that the fourth-generation industrial revolution, called Industry 4.0, offer. We face it in a human-robot collaboration (HRC) domain, in which humans and robots work together to achieve a common goal, and we compute a solution using reinforcement learning (RL) techniques. From a trial-and-error interaction with the environment, these techniques learn the "best" action to execute among the simultaneously available ones. To validate these techniques we have designed a use case that consists in an industrial assembly task. The manufacturing task is modeled as a Markov Decision Process to which two RL algorithms are applied with the aim of learning the optimal scheduling. Specifically, we have analysed the behavior and the performance of the Q-Learning with averaged reward and the Delayed Q-Learning in three scenarios of increasing complexity. Then, the manufacturing task performed following the optimal scheduling is evaluated through standard industrial metrics. In the third scenario, in which the optimal scheduling is learnt ex-novo with the maximum amount of flexibility, the learning phase duration proves to be not suitable for an effective utilization in industry. Hence, to speed up the learning, we have developed an application that converts a drawing of a manufacturing task workflow into its digital twin, which simulates the interaction between the agent and the environment. Finally, in this simulation-based RL framework, we have used the two RL algorithms to compute a static and a dynamic operation assignment, whose adaptability is tested in the face of a non-stationary human behavior.

Key words: Human-robot collaboration, Reinforcement learning, Industrial assembly, Digital Twin

Sommario

Nell'automazione classica i sistemi di controllo sono spesso gestiti con una logica PLC. In questo processo sequenziale può sorgere il problema di come scegliere tra due o più azioni contemporaneamente abilitate. Per risolverlo, normalmente, vengono utilizzate delle regole di precedenza. Qualche volta la definizione di queste regole è basata su una conoscenza a priori del sistema, più spesso si fa affidamento sull'intuizione o su semplici meccanismi "tie-breaking" senza nessun chiaro fondamento. Nella tesi utilizziamo i nuovi strumenti offerti dalla quarta rivoluzione industriale, chiamata Industria 4.0, per risolvere tale problema. Esso viene affrontato nell'ambito della robotica collaborativa, dove gli umani e i robot lavorano assieme per raggiungere uno scopo comune. La soluzione viene trovata utilizzando tecniche di Reinforcement Learning, che, tramite un approccio a tentativi e analisi dei successivi feedback, apprendono qual è la miglior azione da eseguire tra quelle contemporaneamente abilitate. In sintesi, tale apprendimento segue una logica del "sbagliando si impara". Per validare queste tecniche abbiamo simulato una tipica lavorazione industriale, ovvero l'assemblaggio di un prodotto. L' assemblaggio viene modellato come un Markov Decision Process, sul quale applichiamo due algoritmi di reinforcement learning allo scopo di imparare lo scheduling ottimo delle azioni. Nello specifico abbiamo analizzato il comportamento e le prestazioni del Q-Learning con una funzione ricompensa mediata e del Delayed Q-Learning in tre scenari caratterizzati da una complessità crescente. Dunque, l'assemblaggio effettuato con lo scheduling ottimo è valutato tramite delle metriche industriali standard. Nel terzo scenario, dove lo scheduling è appreso ex-novo e con il massimo livello di flessibilità, il tempo di apprendimento ha dimostrato di non essere adatto per un'effettivo utilizzo industriale. Quindi, per velocizzarne l'apprendimento, abbiamo sviluppato un'applicazione che converte il workflow di una lavorazione manifatturiera nel suo digital twin, dove viene simulata l'interazione tra agente ed ambiente. In conclusione, data questa struttura del tipo Simulation-based Reinforcement Learning, abbiamo utilizzato i due algoritmi per definire lo scheduling delle azioni in maniera statica e dinamica, testando l'adattabilità a fronte di un comportamento umano non stazionario.

Parole chiave: Robotica collaborativa, Reinforcement Learning, Assemblaggio Industriale, Digital Twin

Chapter 1

Introduction

This chapter aims to introduce the reader to the topic of the thesis. In section 1.1 the so called Industry 4.0, the technological context in which our project lays, is outlined. In section 1.2 the research question, the beginning point from which we have developed the thesis, is introduced. We also briefly describe how this question has been answered. Finally, in section 1.3 the organization of the thesis dissertation is explained.

1.1 Field of application

The fourth generation industrial revolution, Industry 4.0, has created the notion of smart factory where everything is interconnected, equipped with sensors and works as an autonomous and self-organising system. This revolution follows three main directions: further increasing automation, digitalization and miniaturization. The final purpose is the creation of smart products and services through smart processes [1] [2].

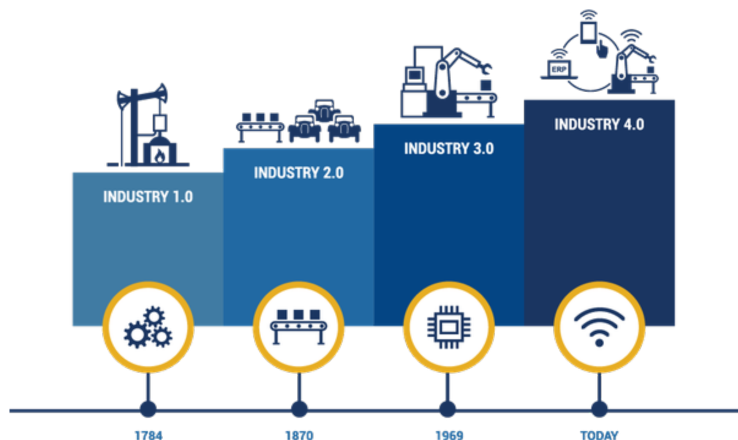


Figure 1.1: History of the Industry

The technological pillars of Industry 4.0 are advanced robotics, internet of things, 3D printing, machine learning, cloud computing, etc. Among them, we deal with an advanced

robotic concept, specifically the human-robot collaboration (HRC). It is defined as humans and cobots that work together, sharing the same workspace, in order to reach a common goal. The word “cobot” stands for “collaborative robots” and denotes a robot optimized for the collaboration with humans, which means that the robot is provided with high safety systems, rounded edges and limitation on speed and force. Industrial automation guarantees high efficiency and repeatability for mass production but it lacks flexibility to deal with the fast changes in the consumers’ demand. Humans, on the other hand, can face such uncertainties and variability but they are limited by their physical capabilities, in terms of repeatability, physical strength, endurance, speed etc. The human-robot collaboration is a productive balance that catches the benefits from both industrial automation and human work [3]. Beyond that, another technological pillar is treated in the thesis: machine learning. It is defined as an application of artificial intelligence that provides systems with the ability to automatically learn and improve from experience without being explicitly programmed. Specifically, we use a set of machine learning techniques called reinforcement learning, that allows the system to learn from a trial-and-error interaction with the environment in order to, in our case, efficiently manage the collaboration between humans and cobots.

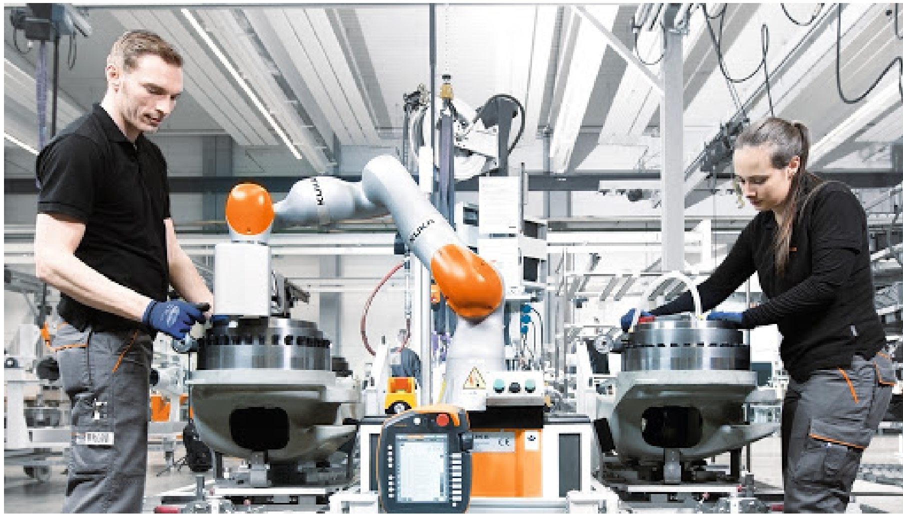


Figure 1.2: Human-robot collaboration

1.2 Research question and contribution

In classical automation, control systems are frequently guided by PLC logics: depending on the inputs and the state of the operating system, the controller decides which action to perform next. In this sequential process, the problem of how to make a decision when two or more actions are simultaneously available may arise. An example of this in the domain of HRC assembly can be found in [4]. The action choice of the cobot (specifically the right and left arms of the ABB YuMi® robot) is guided by the logical SFC program shown in

figure 1.3, in which it can bed noticed that two transitions can be simultaneously enabled, representing a case in which two actions are available at the same time.

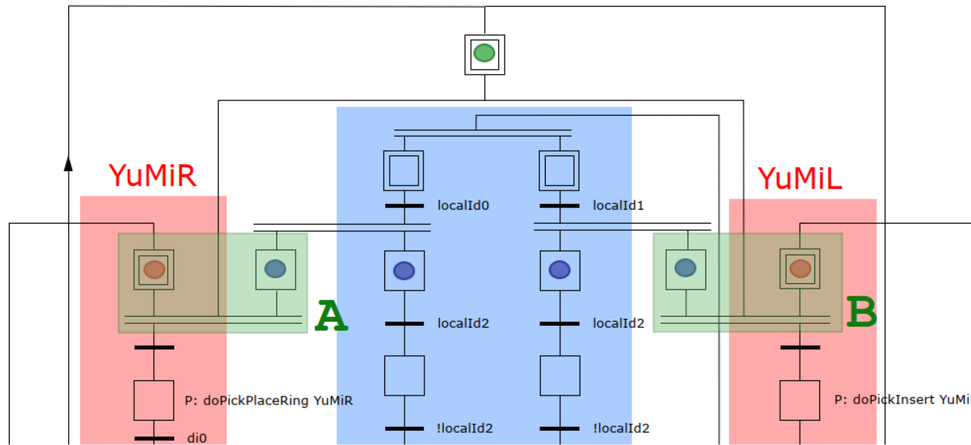


Figure 1.3: Transition A and B simultaneously enabled

To overcome this problem precedence rules are normally used, which assign different priorities to different actions. Sometimes, the definition of these rules is based on a priori-knowledge of the system. Most often, they rely on intuition or implement simple tie-breaking rules with no clear foundation.
The research question consists in using reinforcement learning (RL) techniques to process experienced samples and learn the "best" actions to perform with respect to a metric. This metric, which defines what "best" means, is the time needed to execute an entire sequence of actions that brings the system from the initial state to the achievement of the goal. So, given more alternative actions available, the point is not only about choosing the fastest one but instead choosing the one that enables the fastest sequence. In the field of industrial HRC this problem is declined in the issue of determining the scheduling of human and cobot actions when it is not fully defined, with the purpose of minimizing the time required to manufacture a product.

In the thesis we adapt two RL algorithms to fit a manufacturing task and we test them in an industrial assembly use case. The robot involved is YuMi® by ABB and the assembled product is the Domyos shaker produced by Decathlon. The results are collected for three scenarios that differ for the amount of freedom that is left in the scheduling. In the first one, the scheduling is completely defined apart from a few free actions among which the learner must choose. In the second one, the number of free actions is increased. In the third one, we fully generalize the problem: the scheduling is not defined at all and so all the actions are free to be chosen.
Focusing on the case of computing the entire scheduling, RL techniques prove to be not suitable for an industrial field. They carry out the optimal scheduling but require a long

learning phase that implies a waste of time. So, to speed up the process, we have developed an application that converts an easily drawable workflow of a manufacturing task into its digital twin in which it's possible to learn without physically interacting with the environment. In this simulation-based RL framework, we test the two RL algorithms to analyze their adaptability in the face of a non-stationary human behavior.

1.3 Thesis structure

The rest of the thesis is articulated as follows:

- **Chapter 2 - State of the Art**
  This chapter contains a review of the literature regarding the existing techniques to determine an optimal operation scheduling in a HRC domain.

- **Chapter 3 - Theoretical Background**
  This chapter outlines the theoretical concepts that lie behind the thesis work. Specifically, we describe the classes of models adopted for the use case and we introduce the reinforcement learning framework specifying how it can solve a sequential decision-making problem.

- **Chapter 4 - Use case: Industrial Assembly**
  In this chapter the use case is described: we introduce the robot, the product to assemble and the customized workspace setup. Then, we illustrate how the use case is modeled. Particular attention is placed to the actions necessary to assemble the product, which are listed and described.

- **Chapter 5 - Reinforcement Learning solution**
  In this chapter we firstly define the reinforcement learning algorithms we have applied to the use case, which are Q-Learning and Delayed Q-Learning. Then, we show the results of the implemented reinforcement learning techniques. Specifically, the results of a sensitivity analysis of the parameters and a performance comparison. Finally, we evaluate the resulting optimal scheduling through some indexes that are normally used in the industrial field.

- **Chapter 6 - GUI for Digital Twin Generation**
  In this chapter we initially present the technical motivations that have led to develop an application that transforms a workflow of a manufacturing task into its digital twin. Then, the application is explained. It is composed of four steps. The first step involves the drawing of the workflow on a GUI. The second step consists in the translation of the workflow to a graph, which is a manageable representation in the form of data structure. The third step manages the conversion of the graph into the model of the use case. The fourth step simulates the functioning of the use case.

- **Chapter 7 - Simulation-based RL solution**
  This chapter outlines a simulation-based RL technique. This technique involves the two RL algorithms and the digital twin with the aim of determining a static and a dynamic action assignment. Then, we show a qualitative analysis of the adaptability of this technique.

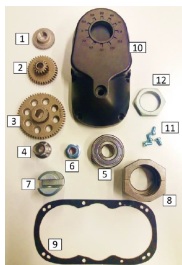
- **Chapter 8 - Conclusions and Future Developments**
  The final chapter completes this document with a resume of the main contributions and obtained results. Moreover, some suggestions of possible future developments are proposed.

Chapter 2

State of the Art

In this chapter we review the state of the art of human-robot collaboration, focusing on the studies that deal with the operation assignment problem. In the literature, several approaches are introduced for scheduling the activities that human and robots have to perform in order to complete a determinate task. The majority of the proposed techniques share the purpose of setting an optimal scheduling, which consists in minimizing the working time in order to increase the productivity. On the other hand, they can be divided into static or dynamic approaches.

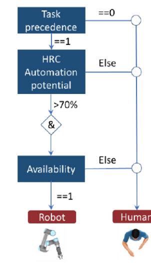
A static assignment determines which operations have to be executed by either the human or the robot in an a-priori way. This methodology can be useful when changes in the workplace are not observable, agent performance is not measurable and/or the system is observable and measurable but the agents are not controllable anymore once the task has begun. An example of static allocation is reported in [5] and in [6]. In this last study the authors propose a method based on giving a score to each operation. This score models the potential of each operation to be automated and it depends on the operation cycle time and adaptability, the properties of the components involved and the collaborative workspace. Figure 2.1 shows the product they have assembled in the case study (figure 2.1a), the scores giving to each component (figure 2.1b) and the algorithm used to allocate the operations (figure 2.1c).



(a) Linear actuator

Part	Cp	Mt	Fd	Sf	Jn	Misc	Potential
Part 1	100	75	50	100	100	83	85%
Part 2	100	75	37,5	100	100	25	73%
Part 3	100	75	37,5	100	100	25	73%
Part 4	100	75	37,5	100	100	50	77%
Part 5	100	62,5	37,5	100	100	25	71%
Part 6	87,5	75	37,5	100	75	0	63%
Part 7	100	100	37,5	100	100	50	81%
Part 8	100	87,5	37,5	100	100	0	68%
Part 9	43,7	87,5	37,5	100	100	25	66%
Part 10	93,7	87,5	37,5	100	100	50	75%
Part 11	100	87,5	37,5	100	75	50	75%
Part 12	100	87,5	37,5	100	75	100	75%

(b) Table of the scores



(c) Allocation algorithm

Figure 2.1: Allocation method proposed in [6]

Other techniques, conversely, involve dynamic allocation. They have the advantage of reacting and modifying the schedule in response to changes in the state of the process that occur while the robot and the human are already performing the task. In [7] the authors propose a method similar to [6] (the scores are replaced by a decision tree in the allocating process), but they also design a procedure of operation dynamic reassignment in order to counteract delays and disturbances that can happen during the process. In [8] the authors follow a different approach. They introduce the Adaptive Preferences Algorithm that computes an optimal flexible scheduling for the robot. The scheduling is flexible in the sense that it accommodates the human preferences in terms of changes in the workflow, but preserving strong guarantees for synchronization and timing of the activities. The Adaptive Preferences Algorithm has been applied to aerospace manufacturing and it has proved to be fast, robust and adaptable.
In [9] the authors present a hierarchical framework to solve the operation allocation problem. The structure is shown in figure 2.2. The first two layers, called assembly-level and team-level, work offline and treat the planning of the assignments considering a multiagents human-robot team interaction. In the assembly-level, the collaborative task is modeled using an AND/OR graph. In the team-level, this graph is used as input of a planner that, exploiting the so-called A* algorithm (with suitable cost function assigned to each member of the team), outputs the optimal scheduling for the robots. The third layer, called agent-level, works online and use complex hierarchical and concurrent hybrid state machines to handle the operation execution and conduct a real-time control. It is designed to cope with unpredictable events that, given the human presence, are likely to happen.

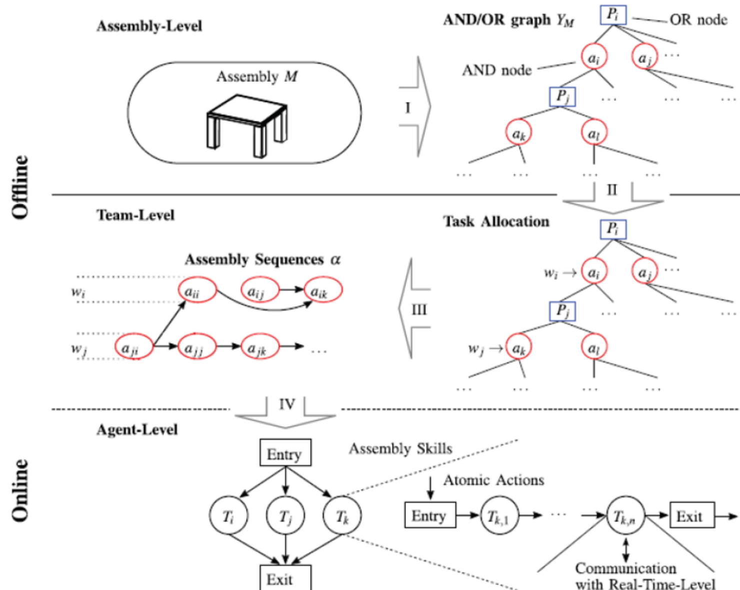


Figure 2.2: Hierarchical framework of the allocation technique proposed in [9]

In these last two studies, it's clear how the robots have a follower role with respect to human intentions. This trend is further pronounced in [10] and [11] in which the robot scheduling depends directly on a model of the human behavior. For example in [10], the author describes an algorithm that, given a model of the human behavior and the available resources, returns the operations the robot has to perform. The future evolution of the system is simulated with a timed Petri net in order to predict how many autonomous actions of the robot can be done before a collaborative action with the human must start. The main purpose is to reduce the waiting time for the operator by avoiding the possibility of the robot being late in case of collaborative action, which has the maximum priority. In the already cited [11], the authors explain an adaptive scheduling based on a model of the human behavior (a Gaussian mixture model for the position and a Gaussian mixture regression for the motion trajectory) that is continuously updated during work. In this study, the human performs the task while the robot delivers to him/her the necessary parts and tools. So, the robot actions are scheduled using the predictive results of the model. Specifically, the starting time of the robot actions depend on the estimates of the human position and arrival time of his/her current operation.
Finally, in [12] the allocation of the operations is determined with the aim of improving ergonomics. In this case, a capability score method to compute the assignment is subordinated to an ergonomics evaluation. For each operation, the ergonomics evaluation is carried out utilizing an automatic postural method based on the REBA score and a workload defined as the REBA scores averaged over the operation duration. The REBA score assesses the risk of suffering from musculoskeletal disorders. So, given the allocation resulting from the capability method, an action assigned to the human remains to the human only if both the REBA score and the workload of that action are lower than established bounds.

The literature lacks studies in which reinforcement learning techniques are used to determine the optimal scheduling, although there are many examples of learning by interaction in the HRC domain (an example is proposed in [13] where the authors also show an overview of the available techniques). Instead, in multi-robots tasks, the RL techniques have been widely used to compute the operation allocation. The thesis dissertation [14] is one of the most thorough work in the field. The author deals with the dynamic organization of a team of robots that share a common goal(s). He provides a market-based reinforcement learning algorithm (it is based on the economic principles behind the market) to determine the operation allocation. The robots interact with their environment and communicate with each other to allocate tasks that maximize their utilities based on their previous experience and cost function. Then, the algorithm has been validated in four scenarios: a centralized task allocation (a single robot assigns the task for the whole team) with homogeneous robots and with heterogeneous robots, a distributed task allocation (each robot decides by itself) with homogeneous robots and with heterogeneous robots.

Chapter 3

Theoretical Background

In the thesis we deal with a sequential decision making problem, whose underlying theoretical concepts are mainly four: agent, environment, reward and policy. The agent is responsible for interacting with the world and making decisions. The environment is everything external to the agent and, according to some rules, it changes from state to state after an agent action. The reward describes how good the action has been with respect to the purpose of the agent. The policy represents the agent behavior.
Figure 3.1 show how these elements interact. Based on a policy, the agent chooses an action, it acts on the environment and then observes how the action has changed the environment configuration and receives a reward.

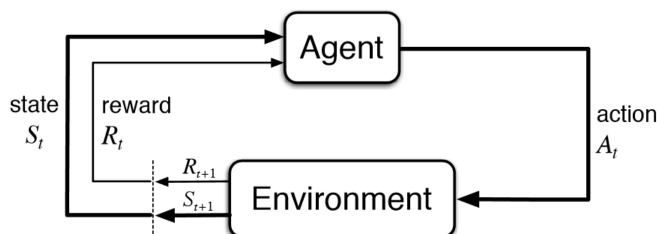


Figure 3.1: Sequential decision making loop

The agent's objective is to shape its policy in order to select actions that maximize the reward over time (sometimes it may be better to sacrifice immediate reward to gain more long–term reward).

In the next sections we analyze in detail the sequential decision making problem. In section 3.1 we introduce a class of models called Markov Decision Processes. In section 3.2 we illustrate the class of Stochastic Shortest Path problems, a subset of MDPs. In section 3.3 we outline the fundamentals of Reinforcement Learning, a framework in which the agent has to maximize the reward without knowing the model. This preamble and the rest of the chapter is based on [15], [16] and [17].

3.1 Markov Decision Process

Markov Decision Process (MDP) is a class of models involving an agent that interacts with its environment in a sequential decision making problem. Specifically, we treat discrete-time finite MDP in which the states and action spaces are finite and the time is considered discrete.
The main property of a MDP declares that all the states of the environment are Markovian. To be Markovian a state s has to satisfy the Markov assumption:

$$\Pr(s_{t+1} = s'|s_t = s, s_{t-1}, ..., s_1, s_0) = \Pr(s_{t+1} = s'|s_t = s) \tag{3.1}$$

The state resumes all the necessary information from the history, hence it will be possible to take decisions based only on the current state.

Formally a MDP is defined as a 6-tuple $\langle S, A, T, R, \gamma, \mu \rangle$ where:

- S is a (finite) set of states;
- A is a (finite) set of actions;
- $T : S \times A \times S \rightarrow$ [0,1] is the transition function that specifies $\Pr(s'|s, a)$, the probability to reach states s' from state s taking action a;
- $R : S \times A \rightarrow \mathbb{R}$ is the reward function that specifies $\mathbb{E}[r|s, a]$, the expected gained reward taking action a from state s;
- $\gamma \in$ [0,1] is the discount factor, a parameter that denotes if the problem is myopic (immediate rewards favourited) given by γ close to 0 or far-sighted (long-term rewards favourited) given by γ close to 1;
- μ is a set of initial probabilities $\Pr(S_0 = s)\ \forall s$.

Normally, an MDP is subject to a graphical representation. An example, specifically of a recycling robot [16], is the following:

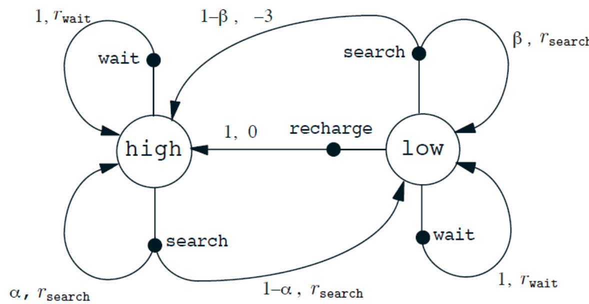


Figure 3.2: Example of a MDP

The states are drawn as circles and the actions as black dots. On each arrow it is instead possible to notice the probability and the reward associated with reaching the pointed state.

The agent should maximize some measure of the long-run reward received. The Sutton hypothesis states that "*All what we mean by goals can be well thought of as the maximization of the cumulative sum of a received scalar reward*" [16]. We focus on the return v_t defined as the total discounted reward from time-step t.

$$v_t = r_{t+1} + \gamma \cdot r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} \tag{3.2}$$

where r_i are the scalar reward received at time i=t+1,..,+∞ (infinite time horizon).
A stationary policy π: $S \times A \rightarrow$ [0,1] specifies, independently of the time step, the probability of an action a to be chosen by the agent given the current state s.

$$\pi(a|s) = Pr(a|s) \tag{3.3}$$

In order to evaluate a policy with respect to the agent's goal we adopt the so called state-value function $V^\pi : S \rightarrow \mathbb{R}$ that is equal to the expected return starting from state s and then following policy π.

$$V^\pi(s) = \mathbb{E}_\pi[v_t|s_t = s] \tag{3.4}$$

In order to carry out the optimal policy and not only evaluating one, it can be simpler to consider the action-value function $Q^\pi : S \times A \rightarrow \mathbb{R}$ that is equal to the expected return starting from state s, taking action a and then following policy π.

$$Q^\pi(s,a) = \mathbb{E}_\pi[v_t|s_t = s, a_t = a] \tag{3.5}$$

The Bellman expectation equations provide a recursive decomposition of state-value and action-value function into immediate reward plus discounted value of successor state.

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\left[r_{t+1} + \gamma V^\pi\left(s_{t+1}\right)|s_t = s\right] \\ &= \sum_{a \in A} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in S} Pr\left(s'|s,a\right) V^\pi\left(s'\right) \right) \end{aligned} \tag{3.6}$$

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_\pi\left[r_{t+1} + \gamma Q^\pi\left(s_{t+1}, a_{t+1}\right)|s_t = s, a_t = a\right] \\ &= R(s,a) + \gamma \sum_{s' \in S} Pr\left(s'|s,a\right) V^\pi\left(s'\right) \\ &= R(s,a) + \gamma \sum_{s' \in S} Pr\left(s'|s,a\right) \sum_{a' \in A} \pi\left(a'|s'\right) Q^\pi\left(s',a'\right) \end{aligned} \tag{3.7}$$

The Bellman expectation equation of the state-value function can be expressed concisely using a matrix form.

$$V^\pi = R^\pi + \gamma \cdot T^\pi \cdot V^\pi = (I - \gamma \cdot T^\pi)^{-1} \cdot R^\pi \tag{3.8}$$

Where $T^{\pi} = \sum_{a \in A} \pi(a|s) \cdot T(s'|s,a)$ and $R^{\pi} = \sum_{a \in A} \pi(a|s) \cdot R(s,a)$.
Value functions define a partial ordering over policies:

$$\pi \geq \pi' \;\; if \;\; V^{\pi}(s) \geq V^{\pi'}(s) \tag{3.9}$$

For any MDP there always exists a deterministic optimal policy π^* that is better or equal to all other policies $\pi^* \geq \pi$, $\forall \pi$. This optimal policy represents the best possible performance in the MDP.
The Bellman optimality equations provide a recursive decomposition for the optimal state-value function $V^{\pi^*}(s) = V^*(s)$ and action-value function $Q^{\pi^*}(s,a) = Q^*(s,a)$.

$$\begin{aligned} V^*(s) &= \max_a Q^*(s,a) \\ &= \max_a \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} Pr\left(s'|s,a\right) V^*\left(s'\right) \right\} \end{aligned} \tag{3.10}$$

$$\begin{aligned} Q^*(s,a) &= R(s,a) + \gamma \sum_{s' \in \mathcal{S}} Pr\left(s'|s,a\right) V^*\left(s'\right) \\ &= R(s,a) + \gamma \sum_{s' \in \mathcal{S}} Pr\left(s'|s,a\right) \max_{a'} Q^*\left(s',a'\right) \end{aligned} \tag{3.11}$$

From which it's possible to compute the deterministic optimal policy:

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} Q^*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

3.2 Stochastic Shortest Path problem

The class of Stochastic Shortest Path (SSP) problems, belonging to the more general MDPs, is of central importance to AI: they are the representation of the classic minimum path search problem in case of stochastic transitions and general reward functions. In a SSP the optimal policy determines that from any initial state (often a fixed one), after n periods of time, a target state is achieved maximizing the expected return.
A SSP is formally described as a discrete-time finite MDP, a 6-tuple $\langle S, A, T, R, \gamma, \mu \rangle$, with a state space $S = \{1, ..., n, t\}$ such that t is an absorbing target state. An absorbing state satisfies the following conditions: $T(t,u,t) = Pr(t|t,u) = 1 \; \forall \; u \in U(t)$ and $R(t,u)$ is the maximum admissible value $\forall \; u \in U(t)$ where $U(t)$ is a subset of the action space A containing all the actions that can be taken from state t. In other words, all the actions performed when the current state is absorbing leads with probability 1 to the same absorbing state and the reward to stay in the absorbing state is maximum.
The existence of an optimal policy is guaranteed under the following conditions:

- There exists a policy, called proper policy, that from any initial state, after n periods of time, achieves the target state with a probability greater than 0.

- Except for $R(t, a)$ with t absorbing state, the rewards are all negative (a cost perspective is often used in SSP, maximizing a negative expected return is equivalent to minimizing the sum of the costs) or all positive.

These two assumptions preclude the case in which a policy remains "stuck" without reaching the target state. For example a problem having zero-cost cycle (in the state space) is excluded by the second assumption [18].

In figure 3.3 a SSP is shown:

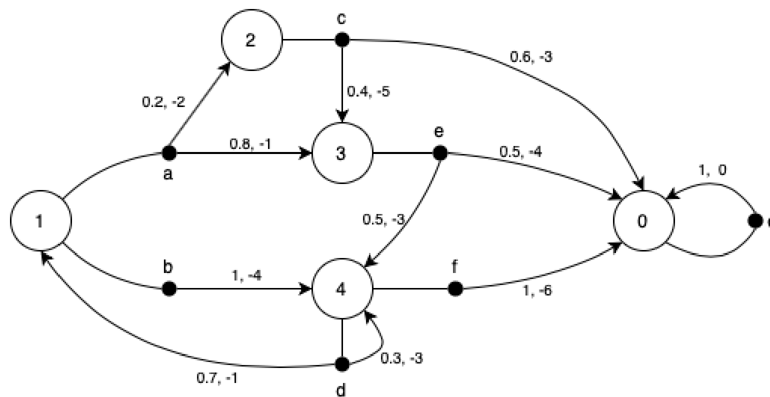


Figure 3.3: Example of a SSP

The target state is labeled by "0" and it is absorbing since action "o", the only one that can be executed from the target state, leads to the target state with the maximum reward. It's possible to notice that the two assumptions are satisfied: from any state it's possible to reach the target state in n periods of time (also from the target state itself) and all the rewards are negative (the reward of "o" action is allowed to be equal to 0 since it has to be maximum and so cost-free).

3.3 Reinforcement Learning: an overview

"Reinforcement learning is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment". [19]

This quote introduces to Reinforcement Learning (RL): a branch of machine learning focused on solving sequential decision-making problems when the MDP model is unknown (or formally, the transition and reward functions are unknown). In order to learn about the sequential decision-making problem the RL framework is fed by a sequence of experience tuples $\langle s, a, r, s' \rangle$. An experience tuple is a sample of a single interaction: the environment is in state s, the agent takes action a, the environment sends a reward r to the agent

and changes its configuration to state s'. These samples are finally processed by an RL algorithm to determine the optimal policy.
In the next sub-sections we provide a classification of the RL algorithms and we outline the exploitation and exploration trade-off, which is an essential topic in RL.

3.3.1 Classification

RL algorithms can be divided into:

- Model-free vs Model-based
  Model-free RL aims to carry out the optimal policy without reconstructing the whole model whereas model-based RL does (despite the misleading name the model keeps not to be available). Model-based algorithms require more samples and, if the purpose is only finding the optimal policy, are less efficient since normally a lot of samples are wasted to estimate part of the model that are useless with respect to the goal.

- On-policy vs Off-policy
  On-policy algorithms estimate the policy that generates the collected samples while off-policy ones estimate another policy, called target policy, from experience sampled by the acting policy, called behavior policy. An example may be an AI that tries to learn how to play chess optimally (the target policy) processing video records of old matches (the behavior policies).

- Online vs Offline
  Online learning is characterized by a continuous interaction between the agent and the environment: a sample is collected and the policy is immediately updated. Instead, offline learning is composed of two phases: in the first one the samples are collected and in the second one the policy is learned. An offline learning implies that the use of off-policy algorithms is mandatory.

- Tabular vs Function Approximation
  A tabular representation consists of the storage of precise values of V(s), one for each state, indeed it can be seen as a table (specifically a column vector). In a function approximation representation V(s) is modeled as a function: a linear combination of features with coefficients computed by minimizing a suitable cost function. The outcome will not be anymore precise but an approximation, anyway, beyond this disadvantage, we will generalize and achieve a faster computation. The same distinction can be done for the action value function Q(s,a).

- Value-based vs Policy-based vs Actor-Critic
  A value-based approach implies to store value functions in order to estimate improved value functions from which an action is selected, instead in a policy-based approach we directly store the policy to estimate new policies. The actor-critic approach implies to store both the value function and the policy: the policy architecture is called

the Actor, because it chooses the action, while the value function architecture is called the Critic, because it criticizes the actions chosen by the actor.

3.3.2 Exploitation and Exploration trade-off

A fundamental topic in machine learning, and in general in the organizational learning, is the trade-off between exploitation and exploration. Every data scientist involving in RL problems has to face the dilemma of setting the right dynamics of exploitation, i.e. how to make the best decision given the current information, and exploration, i.e. gather more information. Since the purpose of RL is finding the best long-term strategy, exploration is a necessary immediate sacrifice to make the best overall decision. In other terms, in case we find a path bringing to a success it may be a local optimum. In fact, if previously we have not tried enough alternative options, there may exist paths that yield better results and displace the previous misinterpreted optimal path [20].

Chapter 4

Use case: Industrial Assembly

The purpose of this chapter is to show how HRC in manufacturing tasks can be modeled as a MDP and so manageable by RL techniques. There are many manufacturing tasks that can be carried out in a collaborative way, such as assembly, pick and place, painting and welding [21]. Among them, we have developed an industrial assembly use case. In the following sections, the use case is further described: in section 4.1 we introduce the adopted cobot, in 4.2 we present the product subjected to the assembly task and in 4.3 we illustrate how the experimental setup of the working area has been organized. Finally, in section 4.4, we describe how the use case has been modeled as a MDP: the states, actions and rewards are defined.

4.1 YuMi

The use case has been implemented in the MERLIN Lab at Politecnico di Milano. The robot that has been used is a IRB 14000 YuMi® produced by ABB. It is a 7 axes dual-arm robot optimized for the collaboration with humans. Therefore, it must adhere to stringent safety requirements, such as power and speed limiting, soft padding, and the absence of trap points (i.e. points that can trap body parts or clothing) (ISO-TS 15066 [22]). Each arm of YuMi can reach points within a hemisphere of radius 0.56 m and lift up to 0.5 kg (decreasing with the distance from the mounting). Its TCP (tool center point, the center of the hand section) has a maximum velocity of 1.5 $\frac{m}{s}$ and an acceleration of 11 $\frac{m}{s^2}$. The controller is integrated and the mounting is designed for tables. For each arm, the integrated hand has a gripper and a vacuum. Further information can be found on the technical datasheet reported in the appendix.

Figure 4.1: IRB 14000 YuMi

The YuMi program is written in RAPID, a high-level programming language used to control ABB robots, and it runs in the controller. It describes the assembling actions as a sequence of robot movements needed to achieve the action goal, for example using the vacuum to pick a piece and insert it in the semi-finished good or assembling two pieces using a screwing movement of the gripper (all the use case actions will be explained in section 4.4). The program also specifies trajectories (normally from a home position to a working area passing through a warehouse) and sets the robot speed and acceleration.

4.2 Description of the product

The product to be assembled is the Domyos Shaker 500ml produced by Decathlon that is shown in figure 4.2. The selected object represents a meaningful example since its assembly is enough complexx, as it is composed of 8 pieces that can be combined following several assembly sequences. Moreover, this product is made of polypropylene, that is a quite robust and light material, and its size is suitable to be managed by YuMi.

(a) Domyos shaker 500 ml

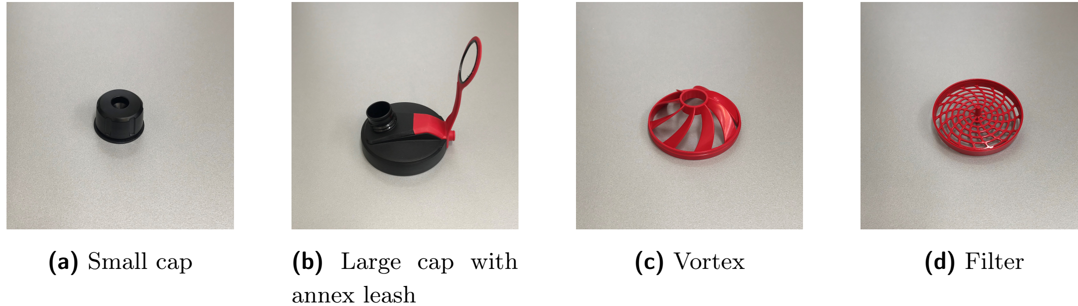
(b) Exploded view of the shaker

(c) Shaker with detached cap

Figure 4.2: The shaker assembled in our use case

Focusing on the single component of the shaker we have ideally divided the product into an upper and a lower part for the sake of clarity.

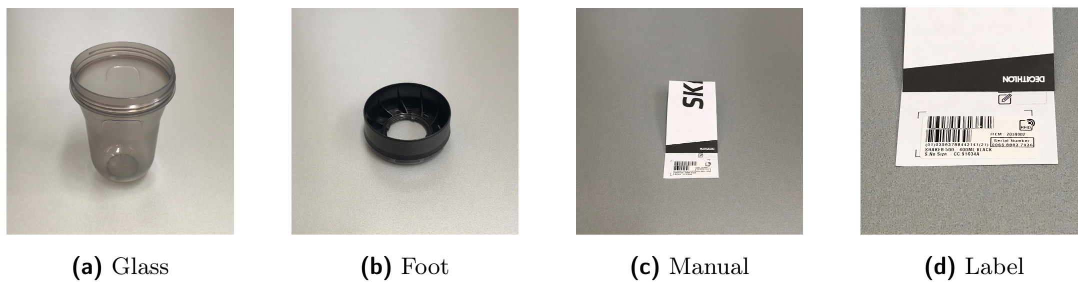
The pieces of the upper part are the following:



(a) Small cap **(b)** Large cap with annex leash **(c)** Vortex **(d)** Filter

Figure 4.3: Pieces of the upper part

The pieces of the lower part are the following:



(a) Glass **(b)** Foot **(c)** Manual **(d)** Label

Figure 4.4: Pieces of the lower part

4.3 Setup Design

The setup of the workspace consists of the positioning of the warehouses, the necessary tools and eventual supports. The workspace is the portion of a table (YuMi has to be mounted on it) between YuMi and the human, which works in front of the robot.
To design our setup we have followed two guidelines:

- All the setup elements have to be reachable from both human and YuMi. If this is not possible the element has to be doubled. This layout is due to the fact that the RL algorithm decides which agent has to execute the action and so the elements have to be available for both.
- Being the arm span of YuMi limited, all the setup elements used, for example, by the right hand of YuMi has to be placed on its right side and viceversa.

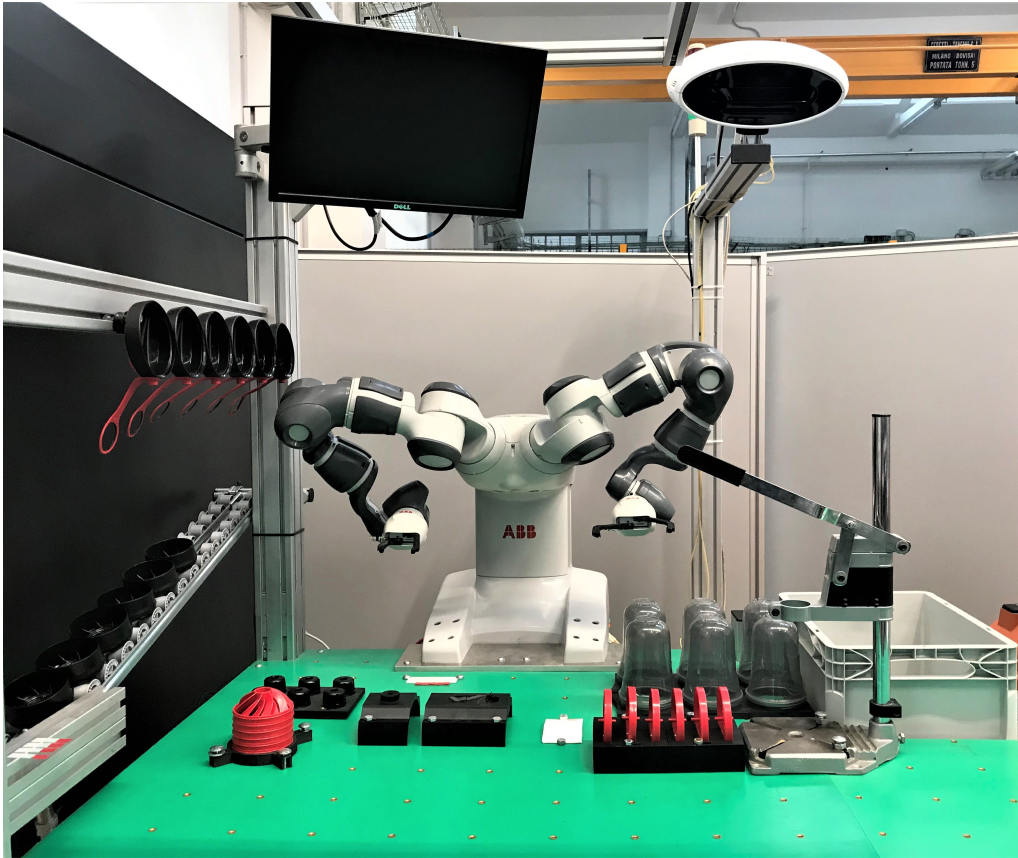


Figure 4.5: Setup of our use case

Figure 4.5 shows the implemented working cell, where it's possible to notice how the two guidelines have been satisfied. All the setup elements are reachable, there are not elements in inaccessible zones as, for example, behind the human. The only element that has been doubled is the label warehouse: one is placed in front of YuMi, the other on the left side of human. Moreover, we have placed the setup elements on the right or left side of YuMi depending on if they are involved in an action done by the right arm of YuMi o the left one respectively. For this reason, the small cap, large cap, vortex, foot and label warehouses have been placed on the Yumi's right side while the glass and the filter warehouses and the hand press have been positioned on the YuMi's left side.
Most of the warehouses have been customly designed and made with a 3D-printer. Each warehouse contains at maximum 6 pieces and, in order to pick the piece from its right "cell", the functions in the YuMi program are modified based on a counter that tracks the number of finished products. Moreover, an assembly support has been made. It is necessary to hold up the two caps and have a precise location where the robot can press the leash, the vortex and the filter. In this way, a high repeatability of these actions is guaranteed.

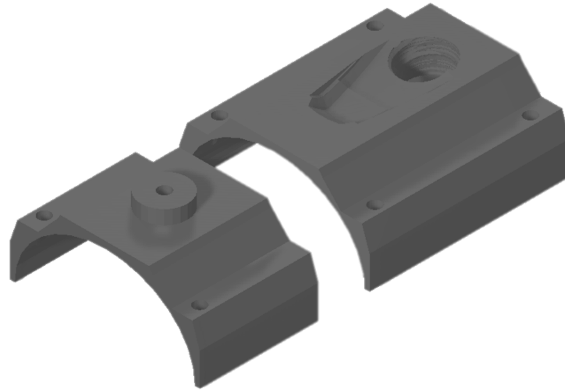


Figure 4.6: Caps support

Please notice that after the setup of the experimental facility and the writing of the RAPID program, it was not actually possible to perform experiments due to the unavailability of the lab for the COVID-19 pandemic. The thesis will then report simulation results referred to the same use case.

4.4 Modeling a collaborative task as an MDP

The use case is an example of sequential decision-making problem that can be modeled as an MDP. The human and the robot act, in a collaborative way, on a semi-finished product to assemble it. As a consequence the semi-finished product become closer to be completed. Then, the human and the robot have to decide how to act next.
The agent comprises both the human and the robot, more specifically, the human, the right arm of YuMi and the left arm of YuMi (individually they will be called sub-agents). In principle, they may be considered as three separated agents but we have adopted a single-agent framework since they collaborate to the same purpose and can be controlled by centralized algorithms, so that a multi-agents framework doesn't provide any meaningful advantage. The actions of each sub-agent, called sub-actions, are the actions required to assemble the shaker. In table 4.1 we present the sub-actions. The first seven sub-actions contribute to the lower part assembly, while, from the eighth to the eleventh sub-action, the upper part assembly is involved. The sub-actions are described following the point of view o the robot, i.e. how the sub-actions have been programmed, since the human is allowed to be more flexible in the realization of the sub-actions. It is assumed that the human knows how to execute each sub-action without errors. Besides, the human is modelled as a controllable sub-agent, i.e. he/she always performs the requested sub-action.

Sub-Action	Description
Place_C1	The small cap is taken from its warehouse and placed on a specific support, called caps support (see section 4.3), on the setup.
Place_C2	The large cap is taken from its warehouse and placed upside-down on the caps support. The trajectory is such that the leash lays down on the small cap.
Press_Leash	The leash is pressed on the small cap.
Press_Filter	The filter is taken from its warehouse and pressed on the large cap.
Press_Vortex	The vortex is taken from its warehouse and pressed on the filter fixed to the large cap.
Press_VortexFilter	The Press_Vortex and Press_Filter actions are realized together coordinating the movements of both arms in order to shorten the working time.
Screw_C1	The large cap is overturned and, thanks to a specific turn trajectory, the small cap lays down on the threaded spout of the large cap. Then the large cap is placed again on the support and the small one is screwed.
Place_Glass	The glass and the foot are picked from their warehouse and the glass is placed over the foot.
Press_Glass	The glass is pressed into the foot with the aid of a hand press (operable also by YuMi) and it is placed again in its warehouse.
Manual_ready	The label is taken from their warehouse and attached to the manual (that is positioned on its warehouse).
Place_Manual	The ready manual is picked and inserted into the glass (that is positioned on its warehouse).
Screw_C2	The large cap, that has been fully assembled (the upper part is finished), is taken from the support. The glass, that has been fully assembled (the lower part is finished), is taken from its warehouse. Then the upper part is screwed on the lower one.
Place_inBox	The completed product is picked and placed in a box.
Wait	The agent stays at its home position.

Table 4.1: Sub-action descriptions

In the table 4.2, instead, it is possible to find, for each sub-action, which sub-agent can execute it (YuMiR and YuMiL stand, respectively, for the right and left arm of YuMi).

Sub-Action	YuMiR	YuMiL	Human
Place_C1	✓		✓
Place_C2	✓		✓
Press_Leash	✓		✓
Press_Filter		✓	✓
Press_Vortex	✓		✓
Press_VortexFilter	✓	✓	✓
Screw_C1	✓	✓	✓
Place_Glass			✓
Press_Glass		✓	✓
Manual_ready	✓		✓
Place_Manual		✓	✓
Screw_C2			✓
Place_inBox		✓	✓

Table 4.2: Sub-actions assignment to a sub-agent

The sub-action “Wait” is not in the table because it is not properly a sub-action but it represents a break from action of a sub-agent. A sub-action that can be executed by all the sub-agents means that the sub-action can be carried out by the human or by the two arms of YuMi working together. It does not exist a sub-action that can be done by either YuMiR or YuMiL singularly.
An action is defined as the vector of the three sub-actions performed simultaneously by the three sub-agents.

$$\text{Action}= \begin{bmatrix}\text{sub-action(YuMiR)} & \text{sub-action(YuMiL)} & \text{sub-action(Human)}\end{bmatrix}$$

The environment is defined as the level of completion of the assembly task. After each action is executed the environment configuration changes toward an increased level of completion. From this point of view it’s natural to define the states as the level of the Work in Progress (WIP), as shown in [28] and [29]. This definition satisfies the Markov assumption, see section 3.1, since each WIP already stores all the necessary information about the history (the previous WIPs).
The reward must be a measure of what we want to maximize in the long term. Since the purpose is minimizing the time required to assemble the product, we set the reward as the negative duration of an action. The duration of an action is defined as the maximum value among the duration of the three sub-actions.
We consider a discrete-time MDP. The measure of time is called timestep and it increases

of a unit each time a new interaction starts. For example, at timestep t there is the current WIP, which is represented by s, on the workspace and the agent performs an action a chosen according to the current policy. When the execution of each sub-action is terminated, we have a new WIP, which is represented by state s', generated by the application of each sub-action on the previous WIP and a reward r is sent. Now, a new interaction starts: the timestep increases of a unit, the new WIP becomes the current WIP and the agent chooses and performs the next action. This definition can lead to a situation in which the sub-agents wait for each other. The efficiency of this choice is analysed in section 5.2.3. Finally, it's possible to notice that an assembly is an episodic task, which means a task in which the agent-environment interaction naturally breaks down into a sequence of separate episode [16]. In our use case, for example, when a shaker is assembled the interaction starts again to assemble a new one. An episodic task can be modeled as a SSP (Stochastic Shortest Path), a sub-class of the MDPs described in section 3.2, with an absorbing state that is the one representing the finished WIP, i.e. the assembled product.

Formalizing the SSP:

- S, the set of states, is the set of all possible WIPs that may appear during the composition of the final product.
- A, the set of actions, is the set of all the combinations, in the form of vectors, of the sub-actions.
- The transition function T is a 3-dimension matrix where the value of the cell (i, j, k) can only be 0 or 1 depending on the probability $\Pr(s' = k|s = i, a = j)$

  $$T(i,j,k) = \begin{cases} 0, & \text{if } Pr(s' = k|s = i, a = j)\text{=}0 \\ 1, & \text{if } Pr(s' = k|s = i, a = j)\text{=}1 \end{cases}$$

  The reason is that, given a WIP (s) and an admissible action (a), there is not doubts that the action achieves the expected result and that the new WIP (s') is unique.
- The reward function R is the negative duration of an action.
- The discount factor γ=0.99 because we adopt a far-sighted strategy.
- The initial probability μ is zero for all the states except for the initial one corresponding to the "null WIP". The initial probability of this state will be equal to 1 since it is unique.

The SSP is shown in figure 4.7. It has 58 states, 82 actions and 330 state-action pairs. The initial state is marked by “S”, while the absorbing state is marked by “E”.

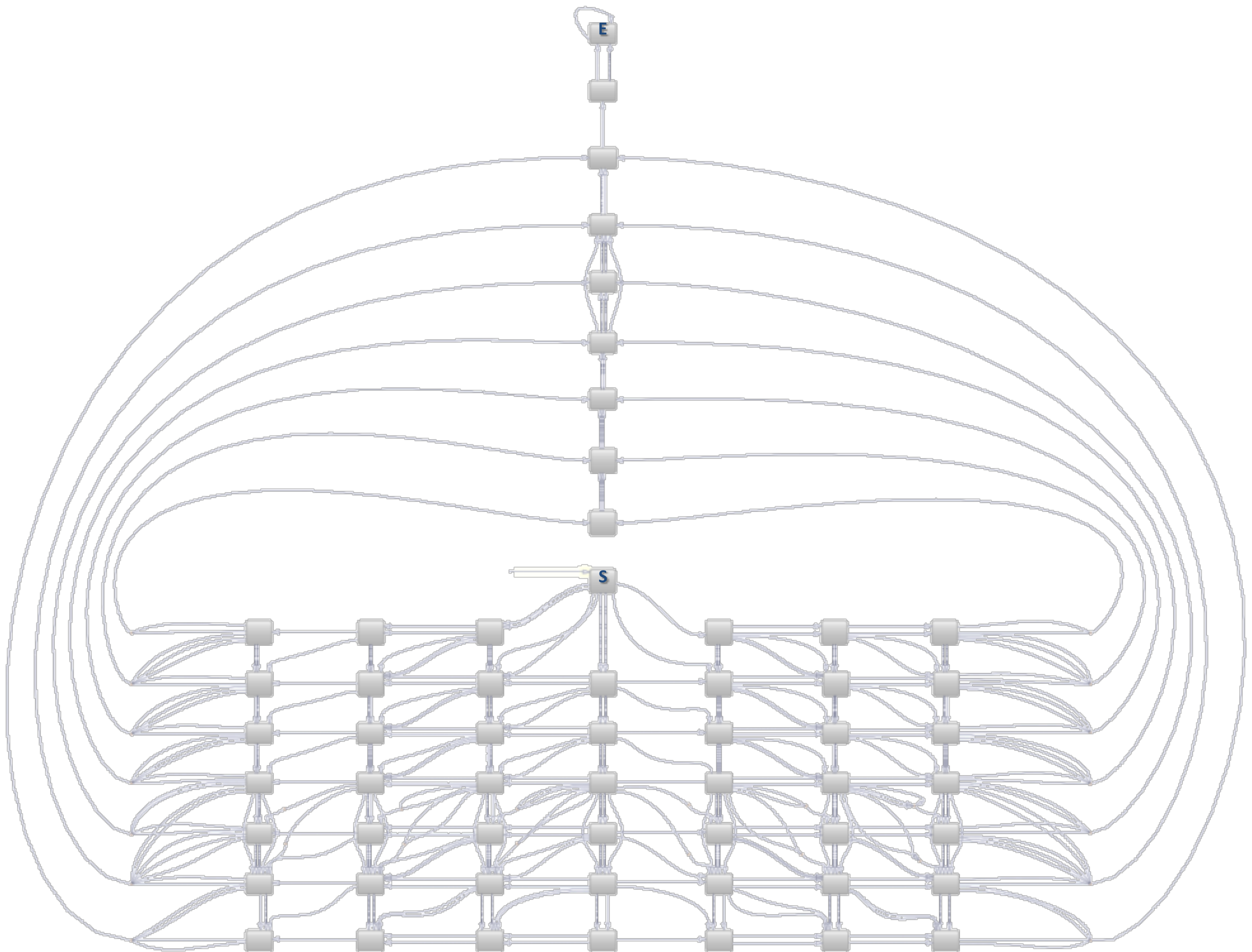


Figure 4.7: The SSP of the use case

Chapter 5

Reinforcement Learning solution

In this chapter the RL techniques applied to the use case are described and their results are shown and commented. Firstly, in section 5.1, the implemented algorithms, Q-Learning and Delayed Q-Learning, are introduced. Then, in section 5.2, we discuss the performances of the RL algorithms in a comparative way, we analyse the sensitivity of the tunable parameters and we evaluate the resulting optimal scheduling.

5.1 Implemented Algorithms

The use case is modeled as a SSP where the state and action spaces are finite. The choice of tabular and value-based RL algorithms is therefore appropriate since it is not required to store a remarkable amount of value functions. Moreover, SSP has an absorbing state so the RL framework is fed with terminating episodes that are defined as finite sequences of samples $\langle s, a, r, s' \rangle$ (the single episode terminates when s' is equal to the absorbing state). For what concerns value-based algorithms applied to episodic tasks, there are mainly two methods to learn the value functions: the Monte-Carlo (MC) method and the Temporal Difference (TD) one. Given the kth episode $\langle s_i, a_i, r_{i+1}, s_{i+1}, a_{i+1}, r_{i+2}, ..., a_{n-1}, r_n, s_n \rangle$ that lasts from timestep i to timestep n (when the absorbing state is met), the MC method updates the action-value functions at the end of it. The following update rule holds:

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha \cdot (v_t - Q_k(s_t, a_t)) \quad \forall t = i, ..., n-1 \tag{5.1}$$

where $v_t = r_{t+1} + \gamma \cdot r_{t+2} + ... + \gamma^{n-t-1} \cdot r_n$ is the return from timestep t.
The parameter α is called forgetting factor ($\alpha \in$ [0,1]) and it sets how much of what has already been learned from old samples affects the next update. Considering the boundary values: if α=1 all the past is forgotten, if α=0 no weight is given to the current sample. The forgetting factor can be either a time-dependent or time-independent parameter.
Instead, the TD method updates the action-value functions after each timestep t, without waiting the end of the episode. So, giving a sample $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$, the following update

rule holds:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \cdot (v_t - Q_t(s_t, a_t)) \tag{5.2}$$

where $v_t = r_{t+1} + \gamma \cdot Q_t(s_{t+1}, a_{t+1})$ is the expected return at timestep t.
The MC and TD update rules for the state-value function are analogous.

The TD method is subjected to a lower variance with respect to the MC one since each TD return depends on one random reward (r_{t+1}), while each MC return depends on many random rewards (from r_{t+1} to r_n). For this reason and because the TD method is generally more efficient than the MC one, we adopt RL algorithms that use a TD method.
So, the TD-based RL algorithms called Q-Learning and Delayed Q-Learning are respectively introduced in sub-section 5.1.1 and 5.1.2. In these algorithms action-value functions are used and, for the sake of brevity, they will be called Q-value for the rest of the chapter. Moreover, for the same reason, a (state, action) pair will be abbreviated to (s, a) pair.

5.1.1 Q-Learning

According to the RL algorithms classification of section 3.3.1 Q-Learning is a model-free, off-policy, online, tabular and value-based algorithm. Moreover, it is based on a TD method for updating the value functions. The Q-value is updated at each sample $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$ and its update rule is the following:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t \cdot (r_{t+1} + \gamma \cdot \max_{a \in A} Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \tag{5.3}$$

Under the condition of Robbins–Monro sequence of step–sizes, which bounds the value of the forgetting factor α_t: $\sum_{t=1}^{\infty} \alpha_t = \infty \wedge \sum_{t=1}^{\infty} \alpha_t^2 < \infty$, Q-Learning converges to the optimal Q-value, that means $Q_t(s, a) \to Q^*(s, a)$ for $t \to +\infty$ [23].
For what concerns the policy, it is important to highlight that Q-Learning is an off-policy algorithm. The target policy is obviously the optimal one while the behavior policy can be whatever. The agent may continuously use a random policy along the entire learning phase and the algorithm still learns the optimal policy (it is due to the max operator in the update formula). However, normally, the behavior policy is ϵ-greedy in order to speed up the convergence. A ϵ-greedy strategy takes into account the exploitation/exploration trade-off (see section 3.3.2). Specifically, with probability 1-ϵ a greedy action is chosen (exploitation), while with probability ϵ, a random action is chosen (exploration).

$$\pi(s, a) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & \text{if } a = \underset{a \in A}{argmax}\, Q(s, a) \\ \frac{\epsilon}{m} & \text{otherwise} \end{cases} \tag{5.4}$$

The parameter m corresponds to the number of actions and it makes, for all the actions, the probability to be chosen greater than 0.

Q-Learning is a very general RL algorithm for tasks that can be modeled as MDPs. It also efficiently fits SSP problems as shown in [24], where the authors apply a Q-Learning, called Q-SSP, to a wireless sensor network. Moreover, they adopt a reward shaping approach, which means to manipulate the reward in order to stimulate the RL technique to learn the correct policy. Their reward shaping technique, called reward-averaging, is designed to mitigate the stochasticity of the rewards. The reward of our use case, as mentioned in 4.4, consists in the negative duration of an action that, mainly in the case of human actions, suffers from a considerable variance. Hence, the reward-averaging is suitable for our use case, too. The shaped reward is called averaged reward $R_{t+1}^{avg}(s_t, a_t)$ and it is equal to the weighted average between the received reward r_{t+1} and the old value of the averaged reward $R_t^{avg}(s_t, a_t)$. The weight is $Num_t(s_t, a_t)$ that is the number of time that the pair (s_t, a_t) has been visited up to instant t.

$$R_{t+1}^{avg}(s_t, a_t) = \frac{Num_t(s_t, a_t) \cdot R_t^{avg}(s_t, a_t) + r_{t+1}}{Num_t(s_t, a_t) + 1} \tag{5.5}$$

Of course, for each pair (s_t, a_t) it is required to store a value of R^{avg} and Num.
Therefore, the Q-value update rule can be rewritten as:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t \cdot (R_t^{avg}(s_t, a_t) + \gamma \cdot \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \tag{5.6}$$

The complete algorithm is shown in 1.

Algorithm 1 Q-Learning

1: **Inputs** : S,A,γ, s_{init}, s_{abs}, α, ϵ
2: **for all** (s, a) **do**
3: $\quad Q(s, a) \leftarrow 0$
4: $\quad R^{avg} \leftarrow 0$
5: $\quad Num(s, a) \leftarrow 0$
6: **end for**
7: episode $\leftarrow 0$
8: **while** episode < episode$_{\max}$ **do**
9: $\quad s \leftarrow s_{init}$
10: $\quad$ **while** $s \neq s_{abs}$ **do**
11: $\quad\quad$ Let A$_s$ denotes the set of admissible actions from s
12: $\quad\quad a \leftarrow \epsilon$-greedy$(s)$ in A$_s$
13: $\quad\quad (r, s') \leftarrow$ Env.Step(s, a)
14: $\quad\quad R^{avg}(s, a) \leftarrow \frac{Num(s,a) \cdot R^{avg}(s,a) + r}{Num(s,a)+1}$
15: $\quad\quad Num(s, a) \leftarrow Num(s, a) + 1$
16: $\quad\quad Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (R^{avg}(s, a) + \gamma \cdot \max_a Q(s', a) - Q(s, a))$
17: $\quad\quad s \leftarrow s'$
18: $\quad$ **end while**
19: $\quad$ episode $\leftarrow$ episode + 1

20: **end while**
21: FinalPath ← greedy path w.r.t (Q, s_{init}, s_{abs})
22: **Return**(FinalPath)

From row 1 to row 7, three tables (the first to store the Q-values, the others two to store $R_{avg}(s_t, a_t)$ and $\text{Num}(s_t, a_t)$) and an episode counter are initialized. Then there are two nested loops. The way out condition of the inner one (from row 10 to row 18) is that the current state s has to be equal to the absorbing state s_{abs}, i.e. the episode is concluded. Each iteration of the loop represents an interaction between the agent and the environment. In row 11, the set A_s is filled by all the admissible actions from the current state s. In row 12, among all the actions in A_s, the action a to take is determined by following an ϵ-greedy policy. In row 13, the reward r and the observation about the next state s' are received after the execution of action a. From row 14 to row 16, the reward shaping and the Q-value updates are performed. At the end of the loop, in rows 17, the next state becomes the current one.
The outer loop (from row 8 to row 20) manages the convergence. If the counter of episodes reaches a bound called $episode_{max}$ (that has to be set high enough) the convergence is considered achieved. In the loop the first current state is set equal to the initial state in order to restart the episode (row 9). In row 19, the episode counter is increased. Finally, in row 21, the convergence is achieved and the greedy path (the sequence of state-action pairs that maximizes the returns) from the initial state to the absorbing one is computed and returned (row 22).

5.1.2 Delayed Q-Learning

Delayed Q-Learning is an RL algorithms introduced by Strehl et al. in [25] and [26]. According to the RL algorithms classification of section 3.3.1 Delayed Q-Learning is a model-free, off-policy, online, tabular and value-based algorithm. It is the first model-free RL algorithm proved to be PAC-MDP (PAC stands for Probably Approximately Correct). The definition of PAC-MDP involves the sample complexity that is, given an algorithm, the number of timesteps t for which the non-stationary policy at time t, π_t, is not ϵ-optimal from the current state at time t, s_t, which means $V^{\pi_t}(s_t) < V^*(s_t) - \epsilon$. Informally, it is the amount of experience needed to learn how to behave well. Thus, an algorithm is PAC-MDP if, for any ϵ and δ, the sample complexity can be bounded by a polynomial in the relevant quantities (S, A, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, $\frac{1}{1-\gamma}$), with probability at least 1-δ.
Delayed Q-learning is similar in many aspects to traditional Q-learning, the main differences are:

- A (s, a) pair has to be experienced m times before updating the associated Q-value (that is why it is called "delayed"). The update will be the average of the m missed update opportunities. In this way the noisy effect of randomness has been miti-

gated (therefore the reward shaping applied to the Q-Learning case is not necessary anymore).

- To encourage exploration an "optimism in the face of uncertainty" approach is used. It consists in a more efficient method to guide the exploration: the initial Q-value of each (s, a) pair is set to some overwhelmingly high number, in our case equal to $\frac{1}{1-\gamma}$. In this way if a (s, a) pair is often visited its estimated Q-value will become more exact, and therefore, lower. Thus, the algorithm will lead to the more rarely visited options, where the Q-values are still high [27]. This approach is made possible by the combined effects of the already described parameter m and $\epsilon 1$ that is a constant "exploration bonus" added to each action value function when it is updated.

The complete algorithm is shown in 2.

Algorithm 2 Delayed Q-Learning

1: **Inputs** : S,A,$\gamma, s_{init}, s_{abs}, m, \epsilon 1$
2: **for all** (s, a) **do**
3: $\quad Q(s, a) \leftarrow \frac{1}{1-\gamma}$
4: $\quad U(s, a) \leftarrow 0$
5: $\quad l(s, a) \leftarrow 0$
6: $\quad t(s, a) \leftarrow 0$
7: $\quad LEARN(s, a) \leftarrow$ true
8: **end for**
9: $t^* \leftarrow 0$
10: $t \leftarrow 0$
11: episode $\leftarrow 0$
12: **while** episode $<$ episode$_{\text{max}}$ **do**
13: $\quad s \leftarrow s_{init}$
14: $\quad$ **while** $s \neq s_{abs}$ **do**
15: $\quad\quad$ Let A_s denotes the set of admissible actions from s
16: $\quad\quad a \leftarrow \underset{a \in A_s}{argmax}\, Q(s, a)$
17: $\quad\quad (r, s') \leftarrow$ Env.Step(s, a)
18: $\quad\quad$ **if** $LEARN(s, a) =$ true **then**
19: $\quad\quad\quad U(s, a) \leftarrow U(s, a) + r + \gamma \cdot \max_a Q(s', a)$
20: $\quad\quad\quad l(s, a) \leftarrow l(s, a) + 1$
21: $\quad\quad\quad$ **if** $l(s, a) = m$ **then**
22: $\quad\quad\quad\quad$ **if** $Q(s, a) - \frac{U(s,a)}{m} \geq 2 \cdot \epsilon 1$ **then**
23: $\quad\quad\quad\quad\quad Q(s, a) \leftarrow \frac{U(s,a)}{m} + \epsilon 1$
24: $\quad\quad\quad\quad\quad t^* \leftarrow t$
25: $\quad\quad\quad\quad$ **else if** $t(s, a) \geq t^*$ **then**
26: $\quad\quad\quad\quad\quad LEARN(s, a) \leftarrow$ false

27: **end if**
28: $t(s,a) \leftarrow t$
29: $U(s,a) \leftarrow 0$
30: $l(s,a) \leftarrow 0$
31: **end if**
32: **else if** $t(s,a) < t^*$ **then**
33: $LEARN(s,a) \leftarrow$ true
34: **end if**
35: $s \leftarrow s'$
36: $t \leftarrow t+1$
37: **end while**
38: episode $\leftarrow$ episode + 1
39: **end while**
40: FinalPath $\leftarrow$ greedy path w.r.t (Q, s_{init}, s_{abs})
41: **Return**(FinalPath)

Practically the algorithm works in the following way. Some locals variables are introduced, for each (s,a): a flag $LEARN(s,a) \in \{\text{true, false}\}$ that indicates whether the learner is considering a modification to its Q-value estimate; a counter $l(s,a)$ whose value is the number of samples acquired for use in an upcoming update of Q(s,a); $U(s,a)$ that stores the running sum used to update Q(s,a) once enough samples have been gathered; $t(s,a)$ that is the instant of the last attempted update and t^* that is the instant of the most recent Q-value change. The initialization of these variable is between row 3 and row 9.
The loop from row 12 to row 39 cycles up to the achievement of the convergence ($episode_{max}$ has to be set high enough). The loop from row 14 to row 37 cycles until the current state reaches the absorbing one in order to allow the restart of the episode (row 13). In row 16 the action a to perform is determined following a greedy policy i.e. picking the action that, given the current state, leads to the highest Q-value. No ϵ-greedy actions are chosen since it is the "optimism in the face of uncertainty" approach that provides the necessary exploration. In row 17 the reward r and the observation about the next state s' are received after having executed action a.
From row 18 to row 34 there are subsequent conditions that, if satisfied, lead to an update of the Q-value. The update of Q(s,a) is firstly driven by the flag $LEARN$ (row 18) that has to be true and, when the counter l has reached m samples (row 21), an update is tried (the so called attempted update). Then there is a last condition $Q(s,a) - \frac{U(s,a)}{m} \geq 2 \cdot \epsilon 1$ (row 22) and, if it is verified, the update is executed (row 23) according to the following update rule:

$$Q(s,a) \leftarrow \frac{U(s,a)}{m} + \epsilon 1 \tag{5.7}$$

$U(s,a)$ is a temporary memory that has stored the sum of the m previous updates of the

action value function, ready to become the new Q-value (row 19).

$$U(s, a) \leftarrow U(s, a) + r + \gamma \cdot \max_{a'} Q(s', a') \tag{5.8}$$

The main rule of the algorithm is that every time a (s, a) pair is experienced m times, an update of Q(s, a) is attempted. In order not to allow an infinite number of attempted updates the computation has to be stopped: this is the role of $LEARN(s, a)$. Hence, attempted updates are only allowed for (s, a) when $LEARN(s, a)$ is true. Besides being set to true initially, $LEARN(s, a)$ is also set to true when any (s, a) pair is updated (rows 32, 33, 34) because our estimate Q(s, a) may need to reflect this change. $LEARN(s, a)$ can switch from true to false only when no updates are made during a period of time in which (s, a) is experienced m times and the next attempted update of Q(s, a) is rejected (rows 25, 26, 27). In this case, no more attempted updates of Q(s, a) are permitted until another Q-value is updated. Finally, in row 40, the greedy path is computed by acting greedy on Q from the initial state to the absorbing one and then returned (row 41).

5.2 Analysis of the results

In this chapter, we simulate the industrial assembly use case, on which we apply the two RL algorithms. The results of the simulations are shown and commented.

The simulations are fed with the durations of the sub-actions; each duration is not deterministic but draws a Gaussian distribution. Then, at each timestep and for each sub-action, the simulations are performed picking a random value of duration from the associated Gaussian distribution.
The expected durations of YuMiR and YuMiL sub-actions have been computed by running their RAPID programs on Robot Studio, which is an ABB's simulation and offline programming software. The resulting expected durations are summarised in table 5.1. Also the YuMi sub-actions should not be considered fixed, in fact we take into account a variance equal to var(YuMi sub-actions)=0.01.
The expected durations of human sub-actions have been found by experimentally trying the sub-actions an adequate number of times and measuring their durations. They are summarised in table 5.1. On the other hand, in the industrial field, factories usually collect a lot of data about the manufacturing task they perform, which can be exploited to estimate the duration of human operations. For example, data from the same manufacturing task if it was previously performed manually (often a manufacturing task is born manual and only secondly got automated) or from comparable manufacturing tasks. For what concerns the duration variance, we apply to all the sub-actions a variance equal to the largest among all the experimental variances. So, the resulting variance is var(Human sub-actions)=0.25 that leads to an interval equal, for each sub-action i, to $mean_i \pm 3 \cdot \sqrt{var} = mean_i \pm 1.5$s (with confidence of 99.5%). On average it corresponds to an interval $mean_i \pm 25\%$. In the

rest of the thesis dissertation this variance will be called "low variance". Since the human behavior may be very stochastic, we have decided to evaluate the RL algorithms also considering a higher duration variance. This variance is equal to var(Human sub-actions)=1 that leads to an interval equal, for each sub-action i, to $mean_i \pm 3 \cdot \sqrt{var} = mean_i \pm 3$s (with confidence of 99.5%). On average it corresponds to an interval $mean_i \pm 50\%$. In the rest of the thesis dissertation, this variance will be called "high variance".
The "Wait" sub-action has both the duration mean and variance equal to 0.

Sub-Action	**YuMiR** [s]	**YuMiL** [s]	**Human** [s]
Place_C1	3.200	/	3.9392
Place_C2	9.810	/	7.1450
Press_Leash	4.525	/	7.6700
Press_Filter	/	5.215	5.3500
Press_Vortex	2.656	/	5.4925
Press_VortexFilter	7.744	5.239	4.9358
Screw_C1	6.100	7.149	8.8750
Place_Glass	/	/	5.5342
Press_Glass	/	6.100	7.5975
Manual_ready	9.570	/	8.1175
Place_Manual	/	8.785	10.2475
Screw_C2	/	/	5.1233
Place_inBox	/	4.250	8.0375

Table 5.1: Sub-action expected durations

A metric we use to evaluate the two RL algorithms is the total reward, which is the sum of the rewards in an episode. An episode represents the assembly of an entire product. The trend of the total reward is inverse with respect to the cycle time i.e the time needed to manufacture a product (we remind that the total reward is proportional to the opposite of the cycle time and maximizing the rewards means minimizing the cycle time), but the total reward plots offer a better insight about the type of reward the algorithm adopts.
Another metric we use is the minimum number of episodes necessary to achieve the optimal policy (or, equivalently, optimal convergence), which is the episode from which the used RL algorithm always returns the optimal scheduling. Of course, the concept of optimality already implies that the total reward is maximized.

In sub-section 5.2.1 we present the results of a sensitivity analysis of the tunable parameters, which are α and ϵ for the Q-Learning (see sub-section 5.2.1.1) and $\epsilon 1$ and m for the Delayed Q-Learning (see sub-section 5.2.1.2). Then, in sub-section 5.2.2, we adopt the parameters that lead to the best performances and we compare the performances of the two

RL algorithms in three scenarios. These scenarios differ for the amount of freedom that is left in the scheduling. In sub-section 5.2.2.1 the scheduling is completely defined apart from a few free actions among which the learner must choose. In sub-section 5.2.2.2 the number of free actions is increased. In sub-section 5.2.2.3 we fully generalize the problem: the scheduling is not defined at all and so all the actions are free to be chosen. Finally, in sub-section 5.2.3, we focus on the optimal scheduling and we present some indexes, normally used in the industrial field, in order to evaluate it.

5.2.1 Sensitivity analysis of the parameters

The sensitivity analysis of the parameters consists in tuning the parameters to examine how they affect the performances of the RL algorithms.
They are evaluated on the entire MDP.

5.2.1.1 Test: Q-Learning

The parameters we analyze are α, which sets how much of what has already been learned from old samples affects the next update, and ϵ, which drives the exploration. The tests are organized in the following way: we have selected eight meaningful strategies from the literature to vary the parameters and we evaluate them both in the case of a reward equal to the opposite of the sub-action duration and the averaged reward. In figure 5.1, we plot the strategies. They consist in the combination of specific trends of α and ϵ.
The trends of α are the following:

A. It decreases linearly starting from a value equal to 1.

B. It is constant and equal to 0.95 up to the 85% of the learning phase, then it decreases linearly.

The trends of ϵ are the following:

C. It decreases exponentially starting from a value equal to 1.

D1. It is constant and equal to 0.6 up to the 40% of the learning phase, then it decreases exponentially.

D2. It is constant and equal to 0.5 up to the 50% of the learning phase, then it decreases exponentially.

D3. It is constant and equal to 0.4 up to the 60% of the learning phase, then it decreases exponentially.

For all the trends, the slope depends on the learning phase length we set, since the parameter has to reach a value equal to 0 in the last learning episode. For all the α trends, the Robbins-Monro conditions (see sub-section 5.1.1) are satisfied.

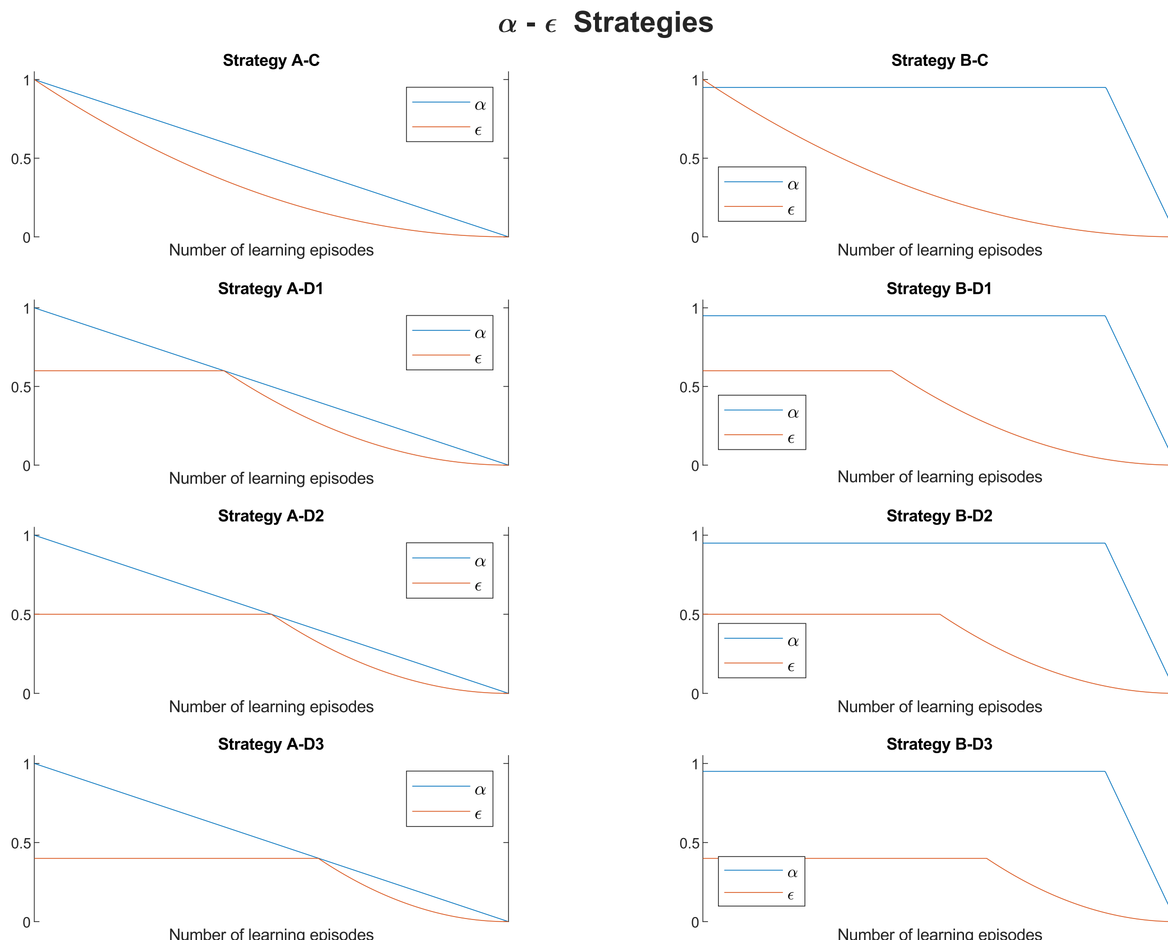


Figure 5.1: Plot of the parameter strategies

It's important to highlight that, in all the strategies, ϵ converges faster than α. This dynamics needs to speed up the learning, otherwise, if ϵ becomes higher than α, the exploration runs but its returns are disregarded by the fact that the weight assigned to the past is so much greater than the weight assigned to the current return.

Figure 5.2 shows the plots of the test in which we use the normal reward (the opposite of the sub-action duration), while figure 5.3 shows the plots of the test in which we use the averaged reward. In these tests, we consider a low human variance. The length of the learning phase changes according to the strategy and the considered reward and, for each case, it is equal to the minimum number of episodes to converge (see tables 5.2 and 5.3). For graphical reasons the simulations last more episodes than the learning phases.

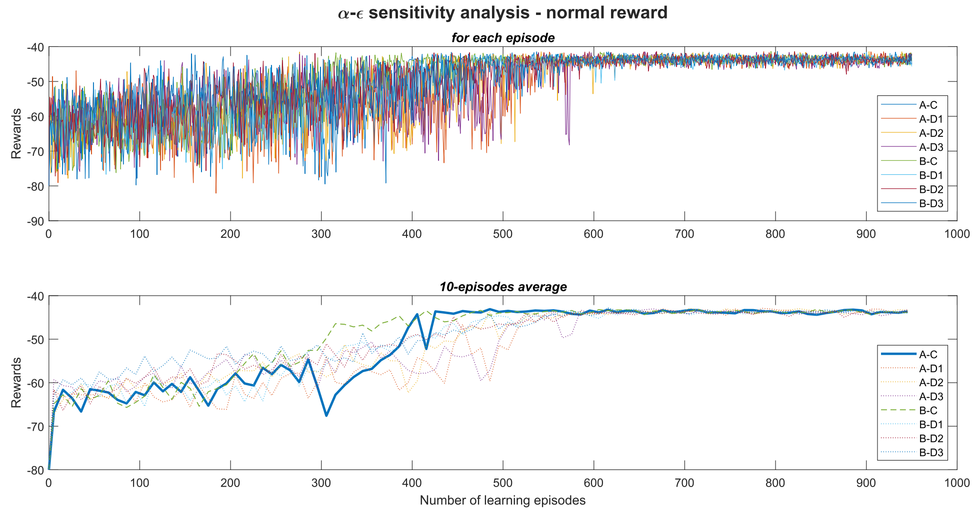


Figure 5.2: Case: normal reward. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best strategy highlighted

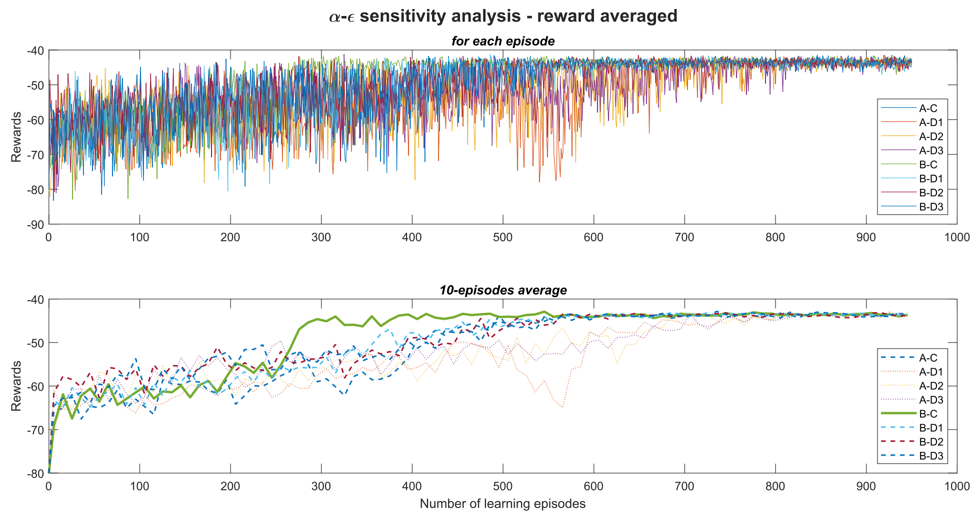


Figure 5.3: Case: averaged reward. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best strategy highlighted

The best strategy is the one that leads to achieve the maximum total reward with the lowest number of episodes. For the normal reward the best strategy is A-C, while for the averaged reward the best strategy is B-C. To explain this difference it's important to underline two concepts. The first concept involves the averaged reward, which behaves well in the face of stochasticity, it becomes more precise, but less reactive, as the number of episodes increases (it is due to an increasing denominator in its update formula 5.5). The second concept concerns α, which allows learning when it is close to 1 and encourages

the consolidation of what has been learnt when it is close to 0. The strategy A-C is not suitable for the averaged reward since it leads to a high learning intensity at the beginning, which decreases as the number of episodes increases. The averaged reward, on the contrary, has a low precision at the beginning, which increases as the number of episodes increases. So, when the learning intensity is high the averaged reward is imprecise, while when the averaged reward is very precise the learning intensity is low. The normal reward doesn't suffer from this problem since it doesn't behave differently with respect to the number of episodes. Indeed, the averaged reward works well with the strategy B-C in which α is kept close to 1 for more episodes. Moreover, the test in which we have an averaged reward and a strategy B-C shows a faster convergence with respect to the test in which we have a normal reward and a strategy A-C. The reason is that the averaged reward, if the parameters are set in a suitable way, is more "powerful" than the normal reward thanks to its ability to face stochasticity.

Finally, in tables 5.2 and 5.3, we summarise the results. For each strategy, we show the minimum number of episodes necessary to achieve the optimal policy, which is obtained by computing the median in 10 simulations. Furthermore, we highlight the values that lead to the best performances (the ones that find the optimal policy faster). In figure 5.4, we provide information about the dispersion of the results. For each strategy, we show the boxplot of the minimum number of episodes to converge.

ϵ \ α	**A**	**B**
C	447	514
D1	597	594
D2	604	597
D3	607	599

Table 5.2: Case: normal reward. Median values of the minimum number of episodes to achieve the optimal convergence. The best result is highlighted

ϵ \ α	**A**	**B**
C	554	398
D1	840	589
D2	845	592
D3	847	596

Table 5.3: Case: averaged reward. Median values of the minimum number of episodes to achieve the optimal convergence. The best result is highlighted

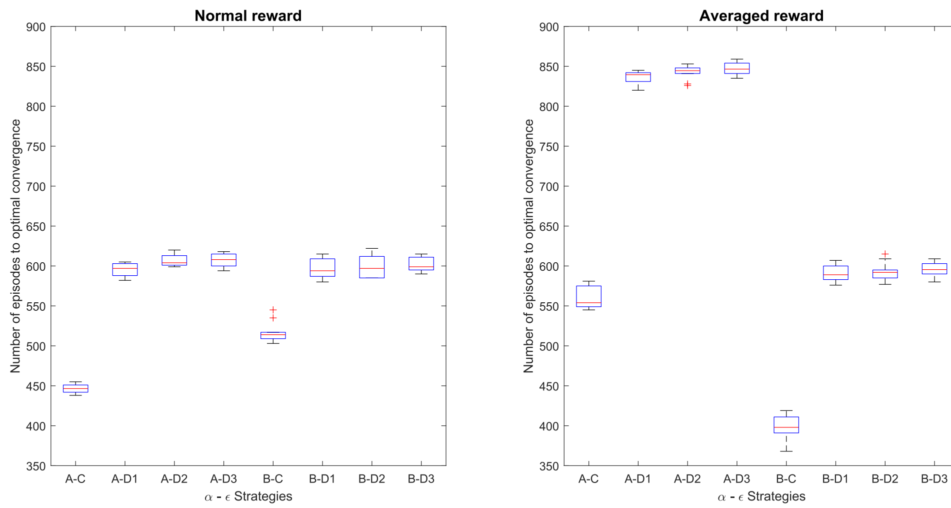


Figure 5.4: Dispersion of the minimum number of episodes to achieve the optimal policy with normal reward (left) and averaged reward (right)

The tests considering a human variance equal to the high variance lead to analogous results. The best strategy still is the B-C with averaged reward.

5.2.1.2 Test: Delayed Q-Learning

The first parameter we analyze is $\epsilon 1$. We carry out three tests that differ from the value of the human variance. Figure 5.5 shows the plots of the test in which the human variance is set equal to 0 (it is an unrealistic case, but it is interesting for the $\epsilon 1$ analysis), figure 5.6 shows the plots of the test in which the human variance is the low variance and figure 5.7 shows the plots of the test in which the human variance is the high variance. The other tunable parameter of Delayed Q-Learning, i.e. m, is set equal to 1 in the low variance test and equal to 5 in the high variance test (see the sensitivity analysis of m).

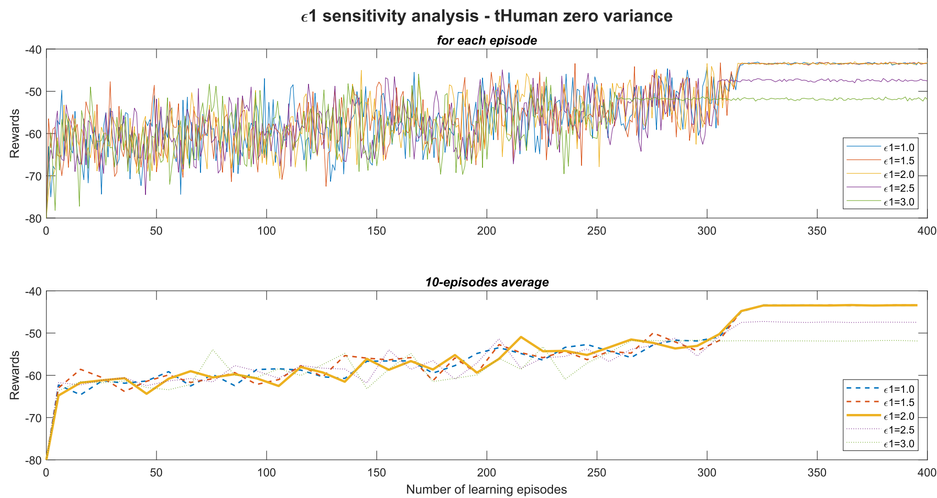


Figure 5.5: Case: zero human variance. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best $\epsilon 1$ setting highlighted

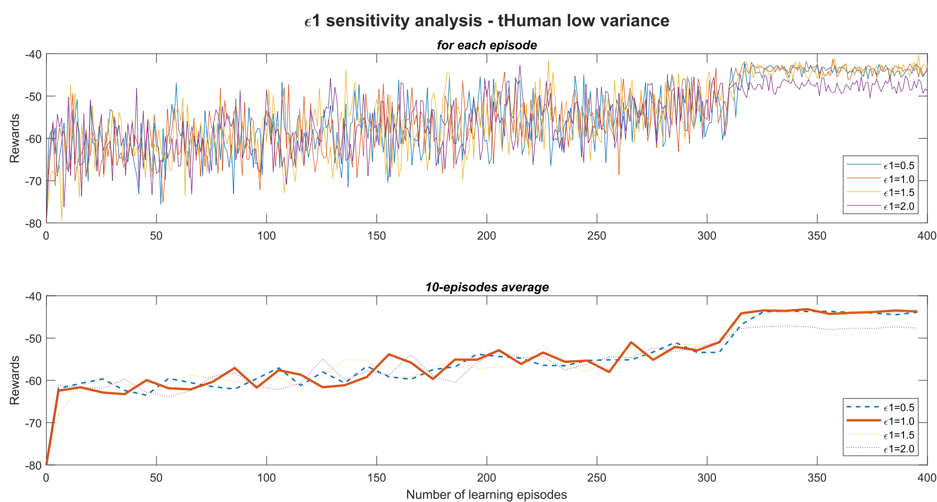


Figure 5.6: Case: low human variance. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best $\epsilon 1$ setting highlighted

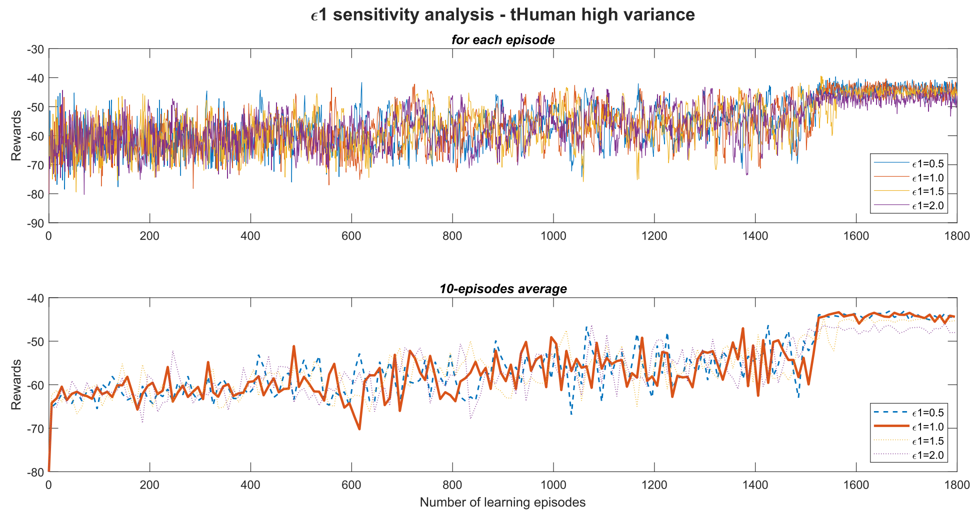


Figure 5.7: Case: high human variance. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best $\epsilon1$ setting highlighted

In these plots, it's possible to notice the role that $\epsilon1$ plays in the learning. It represents the sensitivity of the learning. The smaller is $\epsilon1$, the greater is the probability of converging to the optimal policy. On the other side, the higher is $\epsilon1$, the faster is the convergence. This is due to the update condition of the Q-Value (algorithm 2, row 22). If $\epsilon1$ is small, the condition is satisfied often, Q-values are updated many times and so they lead to a more precise result. Instead, if $\epsilon1$ is high, the condition is verified only at the beginning of the learning and we achieve soon the convergence (the Q-values don't change anymore).
For example, we can focus on $\epsilon1$=2. With this setting, the maximum reward is achieved in plot 5.5, in which the learning is "easy" since the human is not modeled as stochastic. Otherwise, in plots 5.6 and 5.7, we converge to a non-maximum reward that means a sub-optimal policy. The reason is that, in the two last cases, we use a stochastic human behavior and so the learning has to be more precise. Hence, we need a smaller $\epsilon1$. Indeed, with $\epsilon1$=1 and $\epsilon1$=0.5, the optimal policy is found. Between these two values, we prefer $\epsilon1$=1 since it implies a smaller number of episodes to converge.
Finally, it's important to underline that in Q-Learning the amount of exploration, which determines a correct learning, depends on the learning phase length we set (ϵ depends on it), while in the Delayed Q-Learning the exploration is only driven by $\epsilon1$ and m. So, in Delayed Q-Learning, it is useless to set a longer learning phase to reach a more precise learning.

We present, for each human variance, the minimum number of episodes necessary to achieve the optimal policy. We summarise the results obtained in 10 simulations. In the table 5.4, we show the median values, we highlight the best performances (the settings that allow to find the optimal policy faster) and we mark with "/" the settings that don't converge to

the optimal policy. In the boxplots 5.8, we provide information about the dispersion.

var / $\epsilon 1$	Zero variance	Low variance	High variance
0.5	314	324	1512
1	311	313	1508
1.5	310	/	/
2	309	/	/
2.5	/	/	/
3	/	/	/

Table 5.4: Median values of the minimum number of episodes to achieve the optimal policy. The symbol "/" indicates a sub-optimal convergence. For each variance, the best result is highlighted

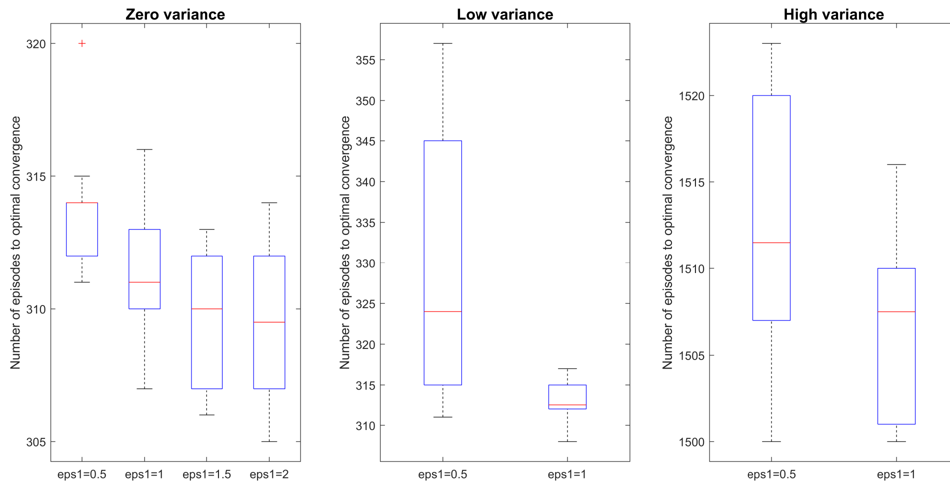


Figure 5.8: Dispersion of the minimum number of episodes to achieve the optimal policy with zero human variance (left), low human variance (center) and high human variance (right)

The second parameter we analyze is m. We carry out two tests that differ from the value of the human variance. Figure 5.9 shows the plots of the test in which the human variance is the low variance and figure 5.10 shows the plots of the test in which the human variance is the high variance. In all the tests $\epsilon 1$ is kept fixed and equal to 1 (the value that leads to the best performances).

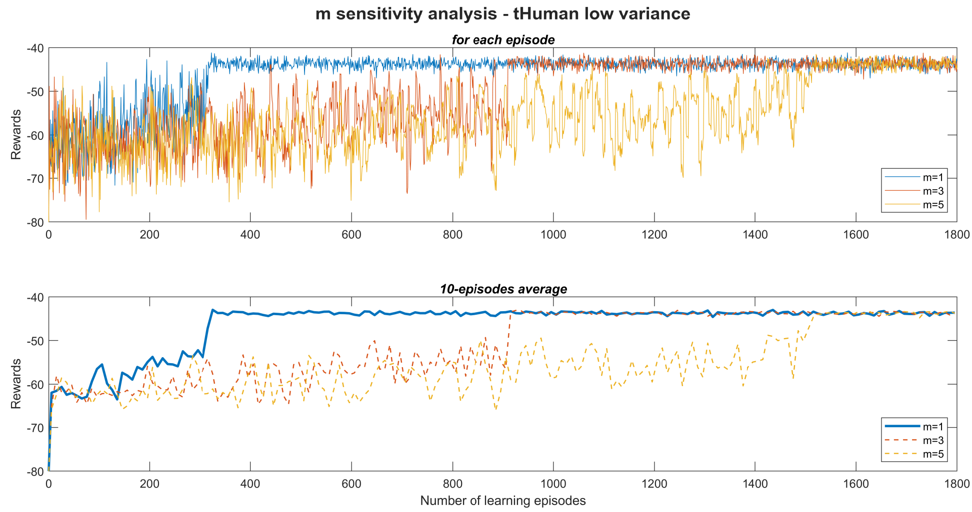


Figure 5.9: Case: low human variance. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best m setting highlighted

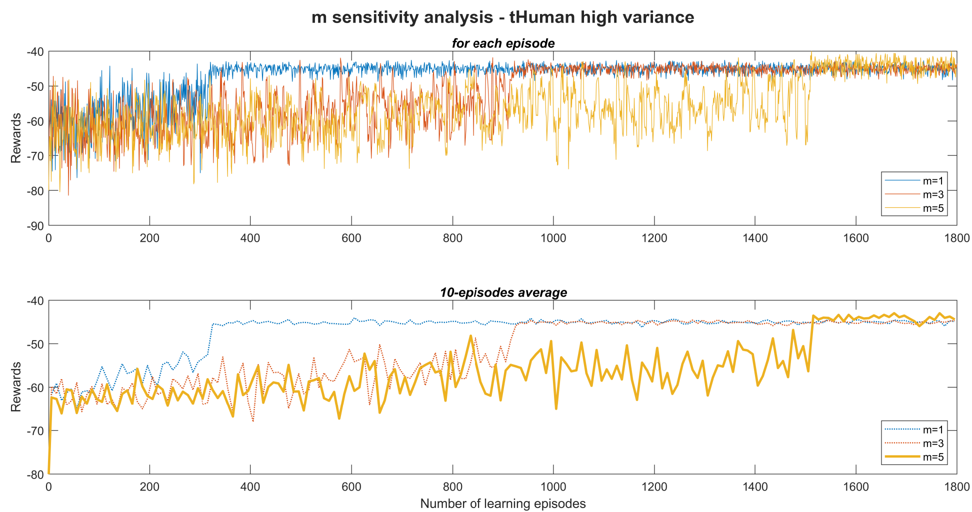


Figure 5.10: Case: high human variance. Plot of the total reward per episode (top) and plot of the 10-episodes averaged total reward (bottom) with the best m setting highlighted

In Delayed Q-Learning, the Q-value update consists in the average of the m missed update opportunities (algorithm 2, row 23). It's clear the role of m to mitigate the noisy effect of randomness. So, in order to achieve the optimal policy, we can't set m too small. Anyway, if we set m too high, we need a lot of episodes to converge since the attempted updates occur too rarely (algorithm 2, row 21).
In plot 5.9, the right trade-off is represented by m=1 since it is the smallest value that guarantees the achievement of the optimal policy. In plot 5.10, we need an higher m to

model the larger variance. The first setting that achieves the optimal policy is m=5 (the settings m=1 and m=3 output a policy barely sub-optimal, their total rewards are very close to the maximum value).

We present, for each human variance, the minimum number of episodes necessary to achieve the optimal policy. We summarise the results obtained in 10 simulations. In the table 5.5, we show the median values, we highlight the best performances (the settings that allow to find the optimal policy faster) and we mark with "/" the settings that don't converge to the optimal policy. In the boxplots 5.11, we provide information about the dispersion.

m \ var	Low variance	High variance
1	313	/
2	610	/
3	912	/
4	1218	/
5	1506	1508

Table 5.5: Median values of the minimum number of episodes to achieve the optimal convergence. The symbol "/" indicates a sub-optimal convergence. For each variance, the best result is highlighted

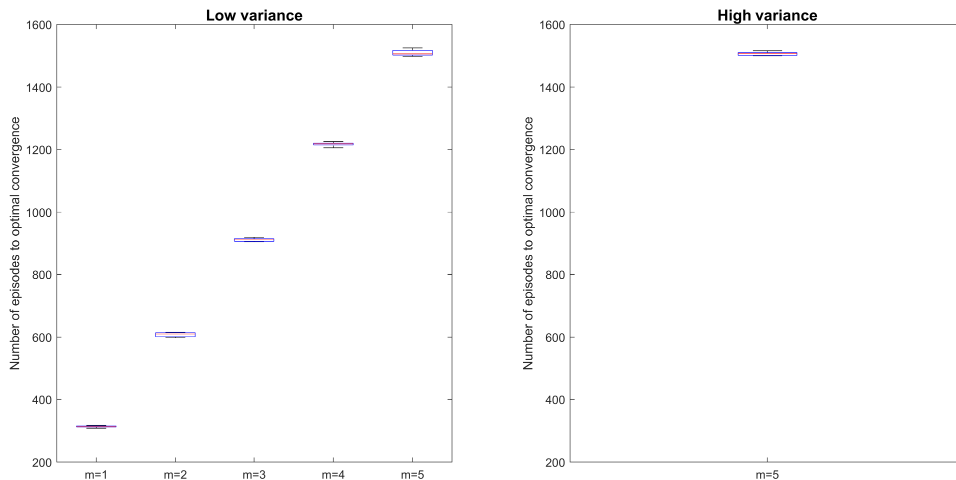


Figure 5.11: Dispersion of the minimum number of episodes to achieve the optimal policy with low human variance (left) and high human variance (right)

5.2.2 Performance comparison

The parameters of the RL algorithms are set in order to have the best performances, as described in sub-section 5.2.1. So, for the Q-Learning we use the strategy B-C with an averaged reward, while for the Delayed Q-Learning we set $\epsilon 1$=1 and m=1 or m=5 according to the human variance.

In the next sub-sections we show the results from three scenarios and, for each of them, we consider both a low human variance and a high human variance. These scenarios differ from the number of free actions among which the learner chooses in order to fully define the scheduling.
The three scenarios are characterized as outlined below.

- In sub-section 5.2.2.1 the scenario has a number of free actions equal to n=6.
- In sub-section 5.2.2.2 the scenario has a number of free actions equal to n=55.
- In sub-section 5.2.2.3 the scenario has a number of free actions equal to n=326. The scheduling is fully undefined, except for the choices that involve sub-actions executable only by the human (that is why n is different than the total number of state-action pairs of the MDP).

In the scenarios with n=6 and n=55, it's like to have reduced-size MDPs since the state and action space become small. A lower number of free actions implies a lower number of possible scheduling, which are translated into the MDP as a lower number of possible path (i.e. sequence of state-action pairs) that brings from the initial state to the final one. In these scenarios the overall optimal path, shown in section 5.2.3, is still available, hence the RL algorithms will surely converge to that.

5.2.2.1 Scenario: n=6 free actions

In figure 5.12, we introduce the resulting MDP with n=6 free actions. In figure 5.13, we show the total reward plots of a simulation. Then, we present, for each RL algorithm and human variance, the minimum number of episodes necessary to achieve the optimal policy. We summarise the results obtained in 10 simulations. In the table 5.6 we show the median values, while in the boxplots 5.14 we provide information about the dispersion.

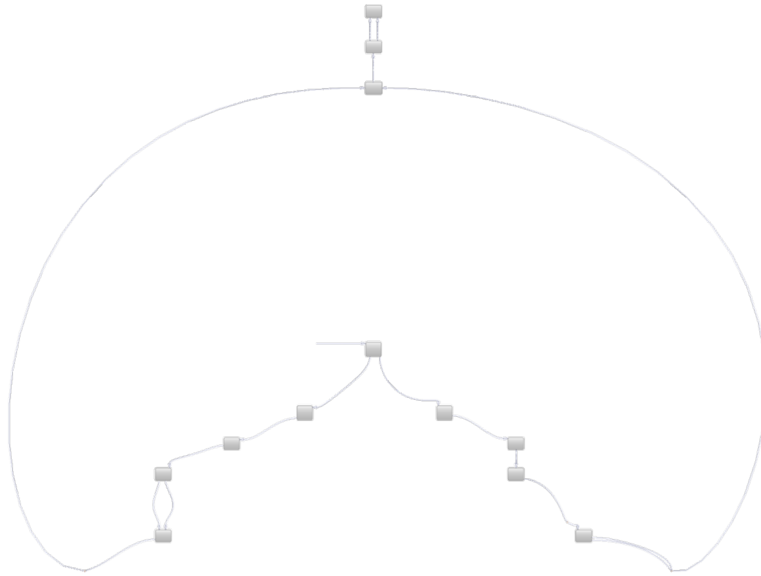


Figure 5.12: MDP with n=6 free actions

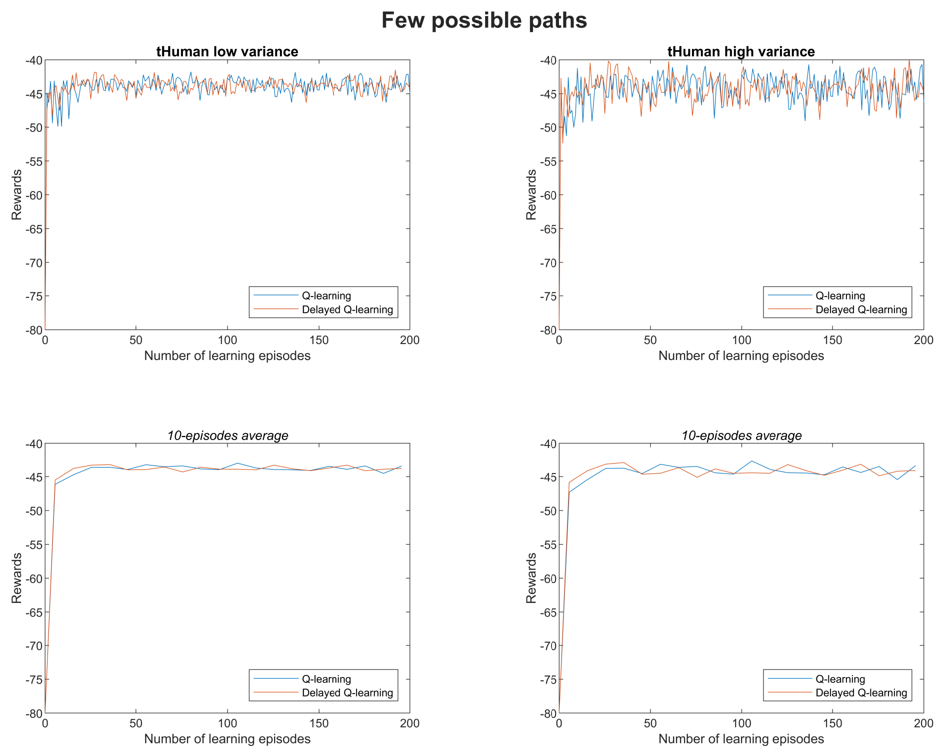


Figure 5.13: Scenario: n=6 free actions. Plot of the total reward per episode (top left) and plot of the 10-episodes averaged total reward (bottom left) with a low human variance. Plot of the total reward per episode (top right) and plot of the 10-episodes averaged total reward (bottom right) with a high human variance

Algorithm \ var	Low variance	High variance
Q-Learning	29	37
Delayed Q-Learning	7	40

Table 5.6: Scenario: n=6 free actions. Minimum number of episodes to achieve the optimal policy

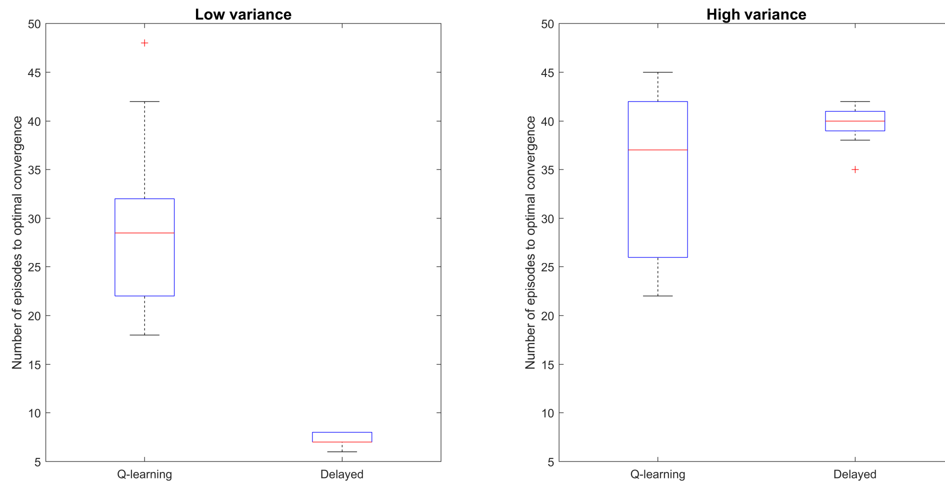


Figure 5.14: Dispersion of the minimum number of episodes to achieve the optimal policy with low human variance (left) and high human variance (right)

Delayed Q-Learning is faster than Q-Learning in the low variance case, while the performances of the two RL algorithms are roughly equivalent in the high variance case. The median values indicate that Q-Learning is faster than Delayed Q-Learning, but the boxplots show a dispersion in the results that doesn't allow to clearly detect the best.
Moreover, it's important to notice the different dispersions that the two RL algorithms show in the minimum number of episodes to achieve the optimal policy. Q-Learning has a random exploration that leads to boxplots showing a remarkable dispersion. Delayed Q-Learning has a guided exploration that leads to boxplots showing a low dispersion.

5.2.2.2 Scenario: n=55 free actions

In figure 5.15, we introduce the resulting MDP with n=55 free actions. In figure 5.16, we show the total reward plots of a simulation. Then, we present, for each RL algorithm and human variance, the minimum number of episodes necessary to achieve the optimal policy. We summarise the results obtained in 10 simulations. In the table 5.7 we show the median values, while in the boxplots 5.17 we provide information about the dispersion.

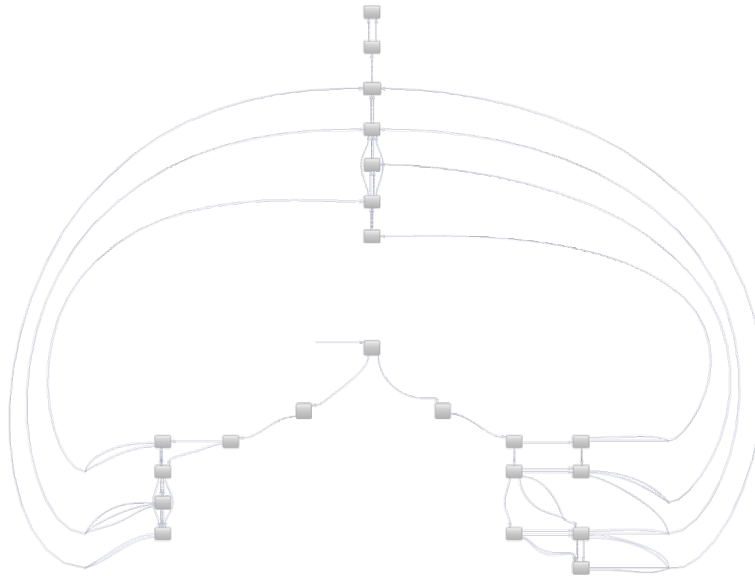


Figure 5.15: MDP with n=55 free actions

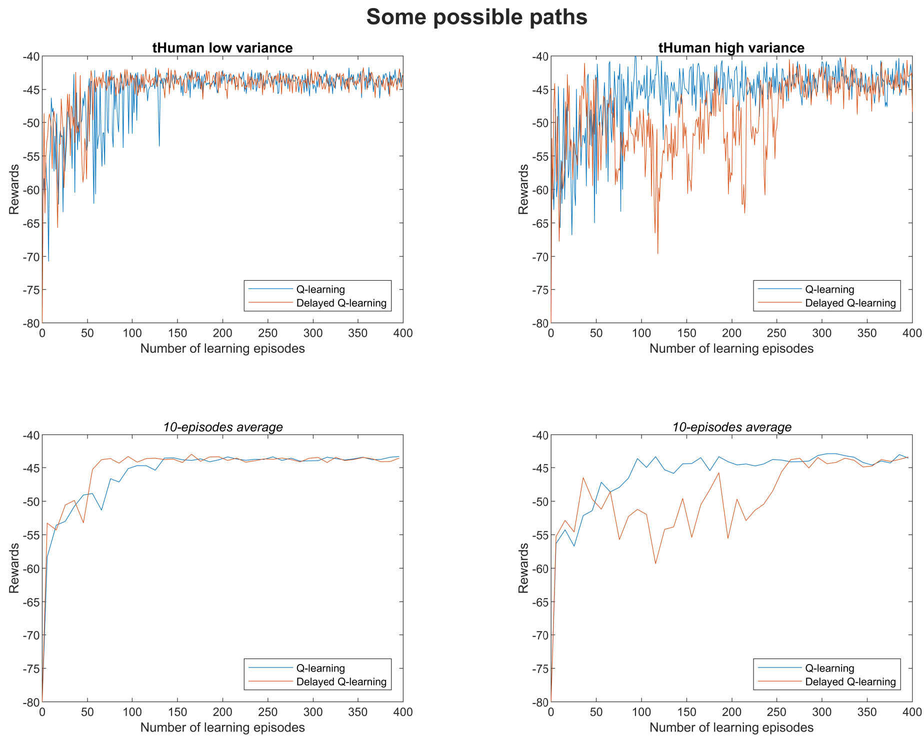


Figure 5.16: Scenario: n=55 free actions. Plot of the total reward per episode (top left) and plot of the 10-episodes averaged total reward (bottom left) with a low human variance. Plot of the total reward per episode (top right) and plot of the 10-episodes averaged total reward (bottom right) with a high human variance

Algorithm \ var	Low variance	High variance
Q-Learning	133	145
Delayed Q-Learning	54	257

Table 5.7: Scenario: n=55 free actions. Minimum number of episodes to achieve the optimal policy

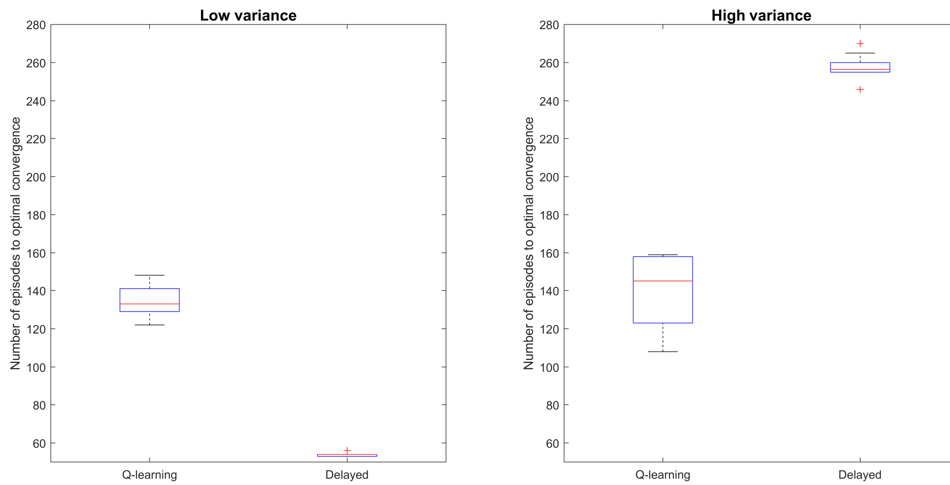


Figure 5.17: Dispersion of the minimum number of episodes to achieve the optimal policy with low human variance (left) and high human variance (right)

With respect to the previous scenario, in the low variance case we confirm that Delayed Q-Learning is faster than Q-Learning, in the high variance case we haven't anymore equivalent performances. In this scenario Q-Learning is faster than Delayed Q-Learning. Of course, the minimum number of episodes necessary to achieve the optimal policy is increased for both the RL algorithms since the complexity of the MDP is grown.

The different performances of the two RL algorithms are due to how they face stochasticity. Q-Learning uses an averaged reward that estimates the value of the reward without being affected by the variance. So, it leads to similar results both in the case of low and high human variance. The averaged reward allows a learning that becomes more precise and less reactive as the number of episodes increases. Instead, in Delayed Q-Learning the stochasticity is managed by the algorithm itself, specifically the parameters m has a central role, which changes according to the human variance. This method allows a learning that doesn't depend on the number of episodes and has a fixed reactivity linked to the value of m (the smaller is m, the more reactive is the learning). In the low variance case, Delayed Q-Learning is faster than Q-Learning because it shows a high reactivity (m=1). In the

high variance case, Delayed Q-Learning is slower than Q-Learning, because it shows a low reactivity (m=5).
To sum up, Q-Learning with averaged reward always finds the optimal policy in an acceptable number of episodes, showing a robust behavior towards different values of human variance. Delayed Q-Learning has a learning method that minimizes the number of episodes to converge when the human variance is low, but, when the human variance is high, it toils to find the optimal policy and so it becomes slower.

5.2.2.3 Scenario: n=326 free actions - scheduling fully undefined

In figure 5.18, we introduce the resulting MDP with n=326 free actions. In figure 5.19, we show the total rewards plot of a simulation. Then, we present, for each RL algorithm and human variance, the minimum number of episodes necessary to achieve the optimal policy. We summarise the results obtained in 10 simulations. In the table 5.8 we show the median values, while in the boxplots 5.20 we provide information about the dispersion.

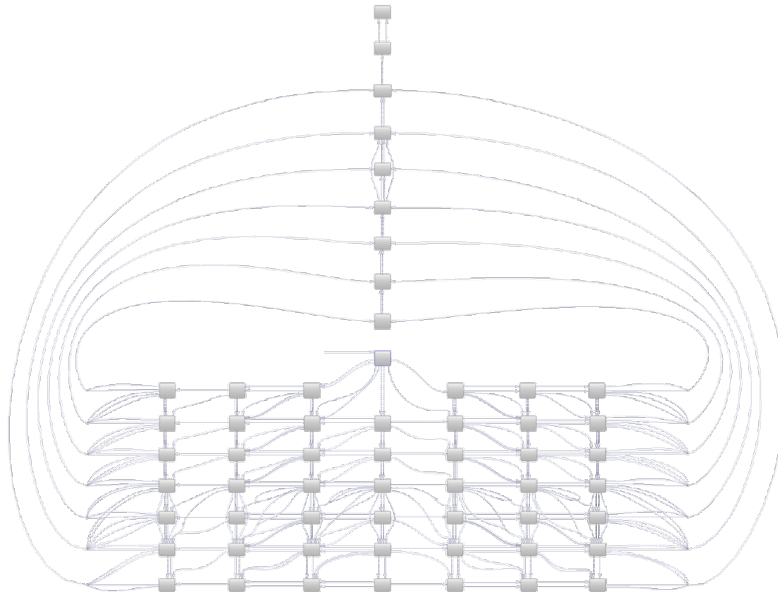


Figure 5.18: MDP with n=326 free actions

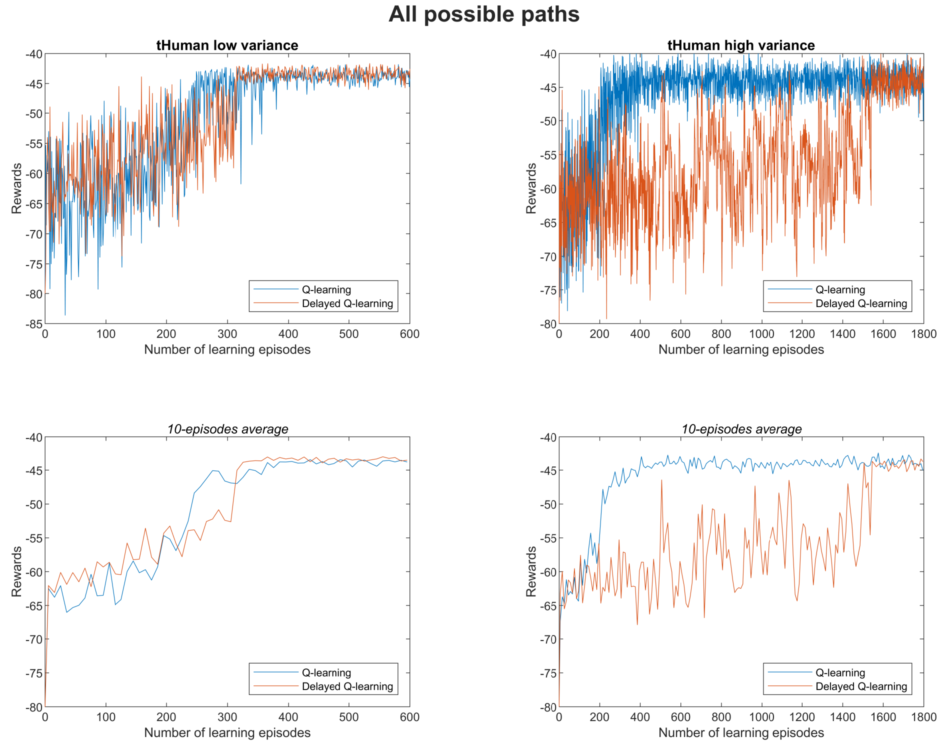


Figure 5.19: Scenario: n=326 free actions. Plot of the total reward per episode (top left) and plot of the 10-episodes averaged total reward (bottom left) with a low human variance. Plot of the total reward per episode (top right) and plot of the 10-episodes averaged total reward (bottom right) with a high human variance

Algorithm \ var	Low variance	High variance
Q-Learning	398	415
Delayed Q-Learning	313	1508

Table 5.8: Scenario: n=326 free actions. Minimum number of episodes to achieve the optimal policy

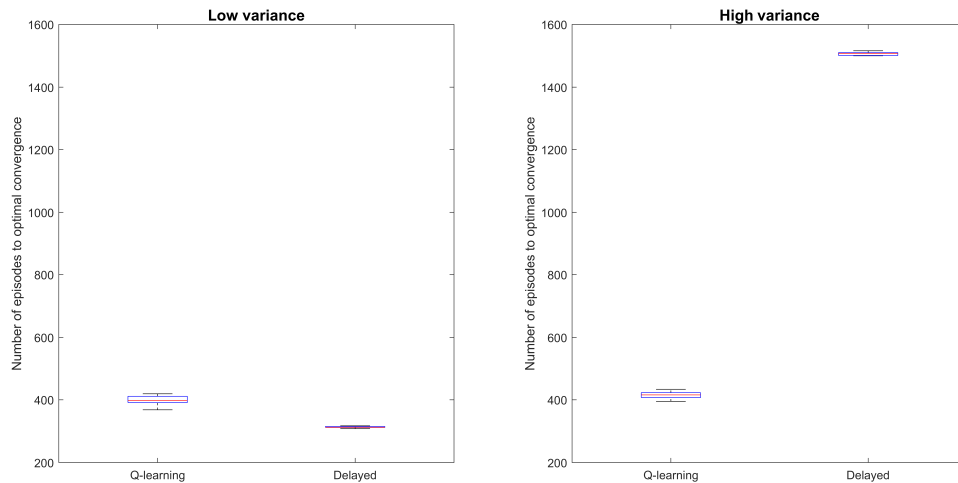


Figure 5.20: Dispersion of the minimum number of episodes to achieve the optimal policy with low human variance (left) and high human variance (right)

The remarks of the previous scenarios are confirmed. Delayed Q-Learning is faster than Q-Learning in the low variance case, while Q-Learning is faster than Delayed Q-Learning in the high variance case. Moreover, Q-Learning has a similar number of episodes to converge for both the low and the high variance cases.
Of course, the minimum number of episodes necessary to achieve the optimal policy is increased for both the RL algorithms since the complexity of the MDP is grown.

The amount of necessary episodes to converge starts to be remarkable. Considering the Delayed Q-Learning for the low variance case and the Q-Learning for the high variance case, the learning phase lasts 313 and 415 episodes. However, the working time "wasted" in the learning phase is not uniform between the two RL algorithms. Using Q-Learning the durations of sub-optimal schedulings become closer to the optimal one in a gradual way. Using Delayed Q-Learning the durations of sub-optimal schedulings are really high up to a trigger episode from which the algorithm starts to output the optimal policy. It is particularly clear in the high variance case in figure 5.19, in which it's possible to notice a "jump" around episode 1500. These behaviors are due to how the exploration is managed. In Q-Learning the exploration decreases exponentially as the episodes increase, while in Delayed Q-Learning it follows a "face to optimism" approach that leads to the "jump" just described (the conditions to update the Q-Value suddenly stop to be enabled).
A metric that allows to better analyse this aspect is the regret. It is defined as the cumulative difference between the total reward of the optimal policy π^* and that gathered by the current policy π and, as well shown in figure 5.21, the regret represents what we miss by adopting sub-optimal policies.

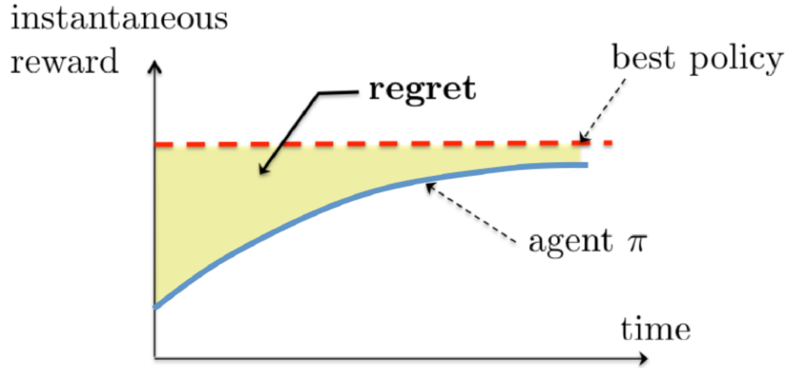


Figure 5.21: Regret of a policy

The formula of the regret at episode K is the following:

$$L_K = K \cdot V_h^{\pi^*} - \sum_{k=1}^{K} \mathbb{E}[V_h^{\pi_k}] \tag{5.9}$$

Where π_k is the policy that the algorithm returns at the end of the (k-1)th episode and h indicates the final timestep of an episode.
In figure 5.22 we show a comparison between the regrets of the two RL algorithms.

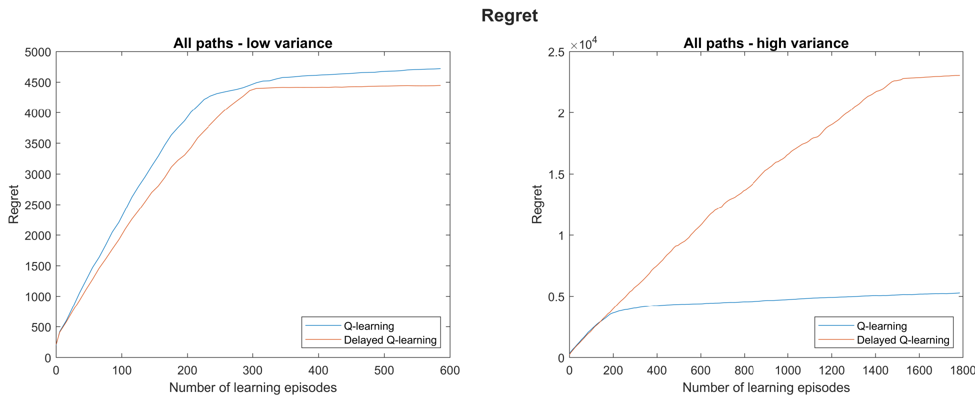


Figure 5.22: Plot of the regret per episode with a low human variance (left) and with a high human variance (right)

To sum up, the regret of the Delayed Q-Learning shows the "impact" of several highly sub-optimal policies, which leads to a long learning phase in terms of working time. Hence, if the comparison was computed in terms of working time, it will be less favorable for the Delayed Q-Learning. For instance, considering the low variance case, the Delayed Q-Learning learning phase lasts 18798 s (about 5 h and 10 min), while the Q-Learning learning phase lasts 21532 s (about 6h). These learning phase durations have been calculated as the average, over the 10 simulations, of the sums of the cycle times of the episodes that

occur before achieving the optimal convergence. The ratio between them is equal to:

$$\frac{\text{learning_phase}_{DelQ}(\text{s})}{\text{learning_phase}_{Q}(\text{s})} = \frac{18798}{21532} = 0.87 \tag{5.10}$$

While, in terms of episodes to converge, the ratio is equal to:

$$\frac{\text{learning_phase}_{DelQ}(\text{episodes})}{\text{learning_phase}_{Q}(\text{episodes})} = \frac{313}{398} = 0.79 \tag{5.11}$$

The ratios show that, if we adopt a working time perspective, the performances of Delayed Q-Learning get worse with respect to an episodes perspective. Anyway, it remains the more efficient algorithm to use with low human variance. As already cited, in the high variance case the worsening is even more clear. Indeed, the ratio in terms of working time is equal to $\frac{89231\ s}{22475\ s}$=3.97, while the ratio in terms of episodes to converge is equal to $\frac{1508\text{ episodes}}{415\text{ episodes}}$=3.63. This analysis holds also for the previous scenarios, but for them, being their learning phases so much shorter, the problem is less meaningful.

5.2.3 Optimal scheduling

As analysed in section 5.2.2, the optimal scheduling can be found by using preferably the Delayed Q-Learning in case of low human variance and preferably the Q-Learning in case of high human variance. In figure 5.23, we show the optimal scheduling in the form of a path, while, in table 5.9, we list the sub-actions that form the optimal scheduling.

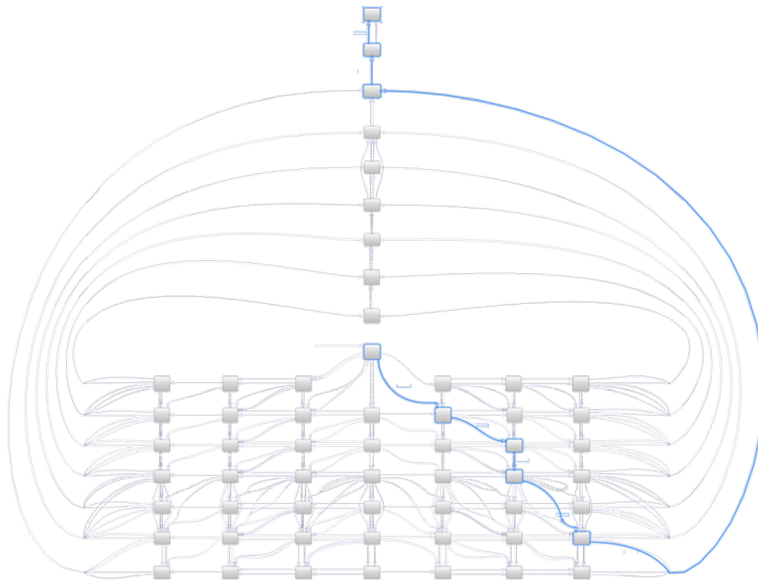


Figure 5.23: Optimal path of the industrial assembly use case

Order	YuMiR	YuMiL	Human
1°	Place_C1	Wait	Place_Glass
2°	Wait	Press_Glass	Place_C2
3°	Press_Leash	Wait	Wait
4°	Press_VortexFilter	Press_VortexFilter	Manual_ready
5°	Wait	Place_Manual	Screw_C1
6°	Wait	Wait	Screw_C2
7°	Wait	Place_inBox	Wait

Table 5.9: Optimal scheduling of the industrial assembly use case

To evaluate the manufacturing task, performed following the optimal scheduling, we use two standard industrial metrics: the Throughput (TH) and the Overall Equipment Effectiveness (OEE).
The TH, or productivity, is the frequency with which the manufacturing task generates the final product. It is calculated using the Little's formula, which is:

$$\text{TH} = \frac{\text{WIP}}{\text{cycle time}} \quad [\frac{pz}{h}] \tag{5.12}$$

The cycle time is the duration of an execution of the optimal scheduling. It can be calculated as the sum of the duration of the action that belongs to the optimal scheduling or as the difference between the starting time of the current task and the starting time of the previous task. The cycle time is a stochastic variable, we can use the mean without loss of correctness. In our use case the cycle time is equal to 43.25 s. The variable WIP indicates the average number of products in process. In our use case it is equal to 1. So, the productivity is equal to:

$$\text{TH} = \frac{\text{WIP}}{\text{cycle time}} = \frac{1\;pz}{43.25\;s} = 0.0231\;\frac{pz}{s} = 83.23\;\frac{pz}{h} \tag{5.13}$$

The OEE index measures, in terms of time, performances and quality of the final product, the overall efficiency of a production process compared to its full potential. The OEE is calculated as:

$$\text{OEE} = \text{utilization} \cdot \text{efficiency} \cdot \text{quality} \tag{5.14}$$

The utilization, or availability, indicates how the production process exploits the available time. It is defined as the period in which the production plants operates over the whole period in which they may be used.

$$\text{utilization} = \frac{\text{utilized time}}{\text{total time}} \tag{5.15}$$

Normally, a production process is organized to minimize all the events that can stop the production long enough, which means a high utilization. For our use case, we assume a

utilization equal to 0.95.
The efficiency, or performance, points out how much the utilized time is exploited. In HRC it is declined as how much the human and the robot works together rather than waiting for each other. It is calculated as:

$$\text{efficiency} = 1 - \frac{\text{wait time}}{\text{utilized time}} \tag{5.16}$$

Figure 5.24 summarises the link between the utilization and the efficiency.

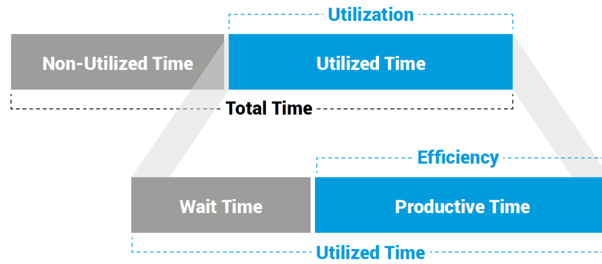


Figure 5.24: Utilization and efficiency of a manufacturing task

We focus on the efficiency of our use case. The two RL algorithms return, with the intent to minimize the cycle time, an optimal scheduling that makes the human and the robot collaborate as much as possible i.e. minimizing the amount of "Wait" sub-actions (unless an agent is deeply faster than the other). In this way we also tend to minimize the wait times, which are defined as follows. For each action, if the duration of the sub-action of the human is higher than the maximum duration between the sub-action of YuMiR and the one of YuMiL, the wait time of the human is 0 and the wait time of the robot is the difference between these two durations. Otherwise, it is viceversa. In figure 5.25 we show the trend of the wait times of the schedulings that have been attempted during the learning phase, while in figure 5.26 we show the trend of the sum of these wait times. The two figures refer to an entire MDP with low human variance (with high human variance is analogous).

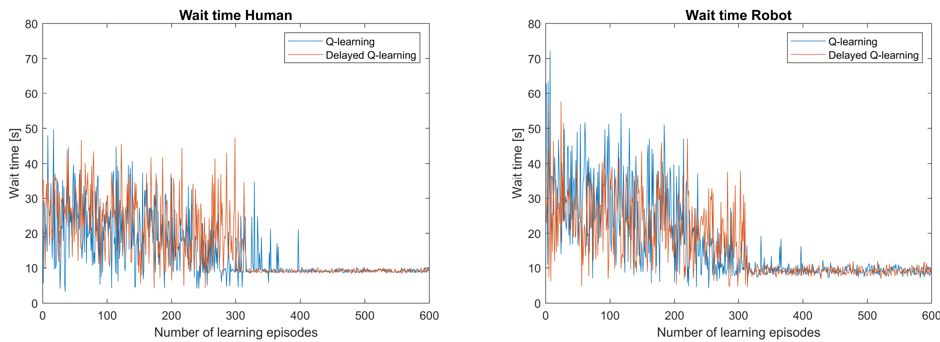


Figure 5.25: Plot of the wait time per episode for the human (left) and for the robot (right)

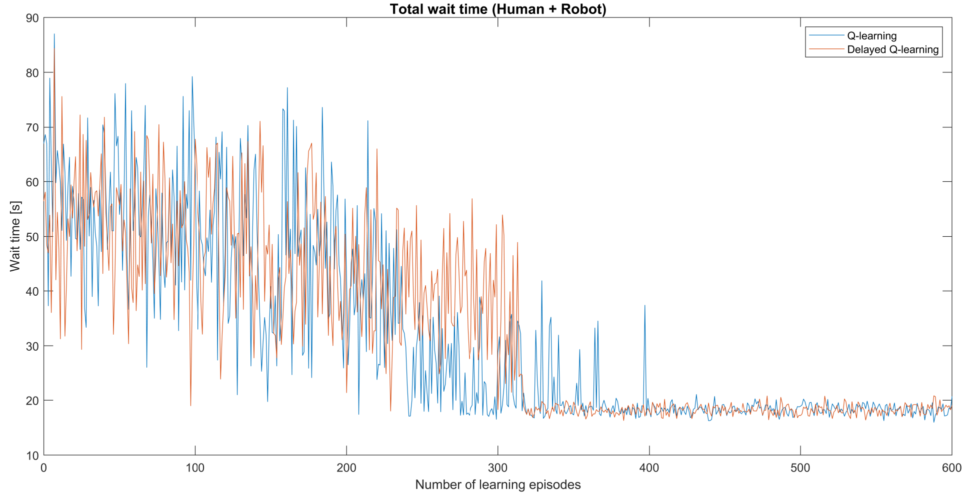


Figure 5.26: Plot of the sum of the human wait time and robot wait time per episode

The two RL algorithms don't minimize the wait times of both the human and the robot since there are spikes that reach lower values than the convergence one. Anyway, they minimize the sum of these wait times. The averaged efficiency for both the human and the robot will be equal since the two RL algorithms converge to the same wait time. It is normal that the optimal scheduling balances the wait times of the agents, but it's a coincidence that they are exactly equal. The wait time is stochastic but, without loss of correctness, we consider the mean that is equal to 9.2 s. Moreover, we normalize the utilized time for a single product, hence it is equal to the cycle time. So, the efficiency is:

$$\text{efficiency} = 1 - \frac{\text{wait time}}{\text{utilized time}} = 1 - \frac{9.2\ s}{43.25\ s} = 0.79 \tag{5.17}$$

Regarding the human efficiency, it is important to highlight that it doesn't overcome the value of 0.88 and so an excessive human effort is avoided. This upper bound is determined in [30] by exploiting the Methods-Time Measurement, a system widely used in the industrial field to decompose manual operations into the basic motions required to perform it and give to each motion a predetermined time standard. It takes into account the daily fatigue curve of an operator that works standing up and the classification of the assembly, which is a repetitive task.
Going back to the OEE formula, the quality is a measure of how the products are manufactured. It is defined as the number of well-made products over the overall number of manufactured products.

$$\text{quality} = \frac{\text{number good products}}{\text{total products}} \tag{5.18}$$

In the literature the quality of HRC tasks have been analysed. For instance, in [31], the authors present a fully manual assembly task and design a new cell production with HRC

in order to improve the efficiency. In terms of quality they show that the ratio between the number of badly assembled products and the overall number of assembled products decreases from a 15% down to be almost prevented. For our use case, we assume a quality equal to 0.98.
To conclude, the OEE is equal to:

$$\text{OEE} = \text{utilization} \cdot \text{efficiency} \cdot \text{quality} = 0.95 \cdot 0.79 \cdot 0.98 = 0.74 \tag{5.19}$$

The value of OEE that represents an efficient production process is still an open debate. From the literature we can cite some best practice benchmarks, in [32] Ericsson considers as acceptable values the indexes between 0.30 and 0.80, while Ljungberg, in [33], proposes a range between 0.60 and 0.75.

Chapter 6

GUI for Digital Twin generation

In this chapter, we describe an application that we have developed in order to convert the workflow of a manufacturing task, drawn on a Graphical User Interface (GUI), into its digital twin. In section 6.1, we illustrate the reason behind the choice of implementing such application and in section 6.2 we explain how it works.

6.1 Motivations

In chapter 5 we have answered the research question showing how the RL techniques can determine a scheduling when it is not fully defined. These techniques have been designed for working in an industrial field, hence, in the case of determining the entire scheduling, they can be considered not suitable. Although the RL techniques carry out the optimal scheduling also in that case, they need a lot of interactions to learn. Therefore, many products are manufactured with sub-optimal policies that lead to a remarkable waste of working time.
To avoid this disadvantage we have implemented an application that converts the workflow of a manufacturing task into its digital twin, which simply is an MDP that simulates the interaction between the agent and the environment. So, at each timestep t, we have a "digital" sample $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$. In this framework, called simulation-based RL, we use the Q-Learning and Delayed Q-Learning to carry out the optimal scheduling in a very fast way. In chapter 7, we analyse the performances of a simulation-based RL technique that involves the two RL algorithms and the digital twin to compute a static and a dynamic actions assignment in the face of a non-stationary human behavior.
Furthermore, it's important to underline, in a perspective of attractiveness for an industrial field, that the application has been designed in order not to depend on computer science specialists. It only needs a workflow, i.e. a simple drawing that involves only four graphical elements and that can be done by any operator aware of how the manufacturing task works.

6.2 From the workflow to the digital twin

The application converts the workflow of a manufacturing task into its digital twin. It is articulated in 4 steps. The first involves the drawing of the workflow, the second performs the codification of the workflow in order to have a more manageable "digital" graph that replies its structure. The third consists in the conversion from the graph to an MDP structure (i.e. states linked by actions associated with rewards). The fourth, given the MDP structure, manages the interaction between the agent and the environment. The following description of the application is based on our use case but the functioning is general, as it can be used to describe all the manufacturing tasks that can be performed in a collaborative way and can be modeled as an MDP. The pipeline of the application is sketched in figure 6.1.

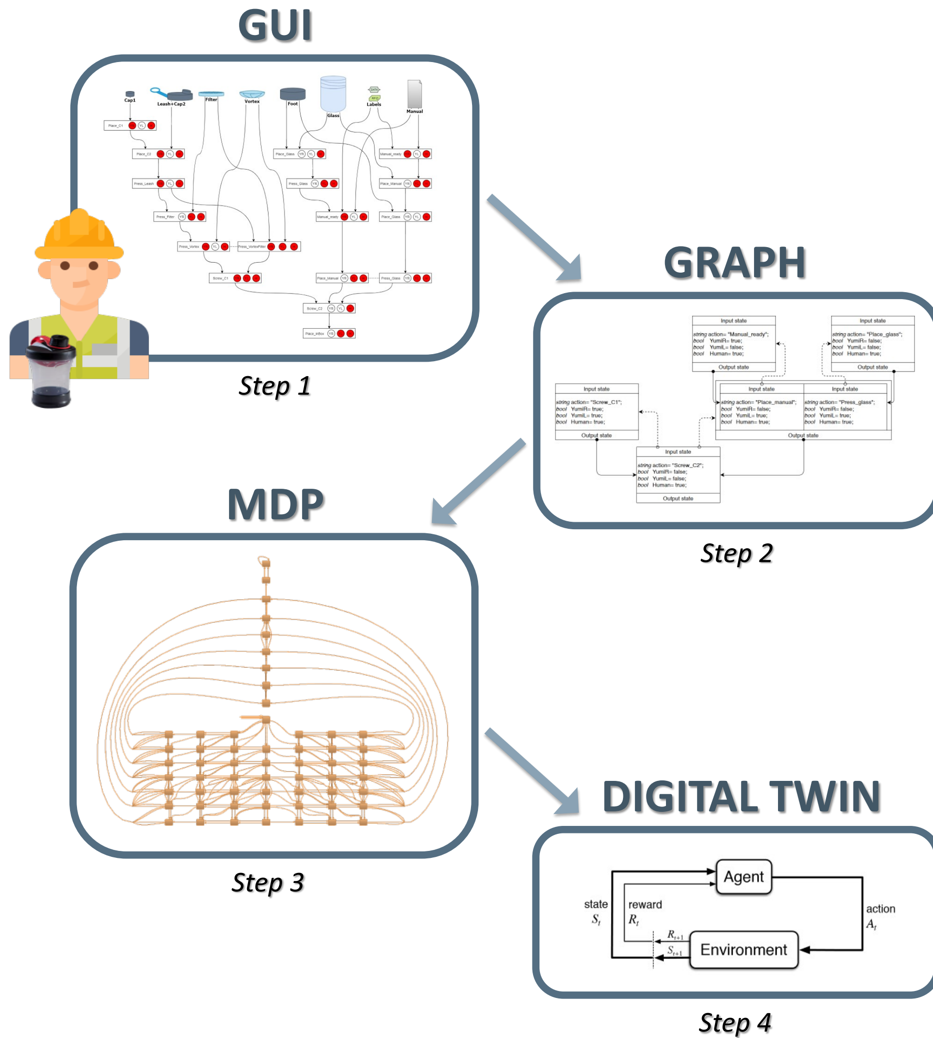


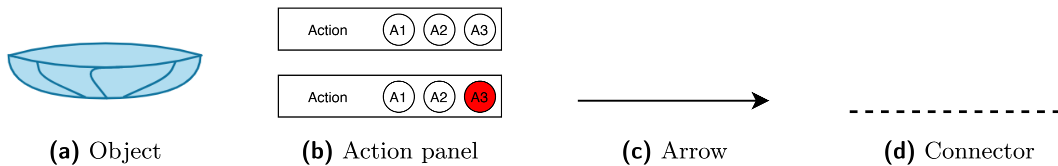
Figure 6.1: Pipeline of the application

The application is written in *C++*, one of the most spread object-oriented programming language. The explanation of the main function of the application is supported by pseudo-codes. In order to avoid misinterpretations in the reading of the pseudo-codes it's necessary to state some initial remarks:

- In *C++* language the variables are featured by a class. Each class can involve attributes, which are additional information for characterizing the variable. In the pseudo-codes the notation to indicate the attribute of a variable is the following: variable.attribute.
- In the pseudo-codes the notation used for the function is **function**(input) while the one used to define a value in a vector is vector(index).
- In the *C++* program we have mainly used pointers and lists in order to save memory and speed up the computational time. In the pseudo-codes, for the sake of simplicity, we replace them with variables and vectors without loss of correctness.

For the graphical part, instead, we rely on the website *Draw.io*, an open source technology stack for building diagramming applications, because it has a user-friendly interface and it allows exporting a XML file that can be easily processed by *C++* applications.

The application "step 1" is about the workflow drawn by the user. This workflow has to describe the precedence constraints among the various phases of the assembly. Specifically, he/she defines the workflow simply using the four graphical elements shown in figure 6.2.



(a) Object **(b)** Action panel **(c)** Arrow **(d)** Connector

Figure 6.2: Graphical elements

The element of figure 6.2a is called object and it represents a single piece of the product. The element of figure 6.2b is called action panel: a rectangle containing as many circles as the number of sub-agents. Inside the rectangle the operator inserts the name of a sub-action and selects, coloring the respective circles, which sub-agents can execute such sub-action. The element of figure 6.2c is an arrow, which links action panels and represents a precedence constraint: the sub-action of the panel from which the arrow starts has to be carried out before that one of the panel where the arrow ends. The element of figure 6.2d is a connector (a dashed line) which links two or more panels whose sub-actions led to the same WIP.

The final workflow of the use case assembly is shown in figure 6.3.

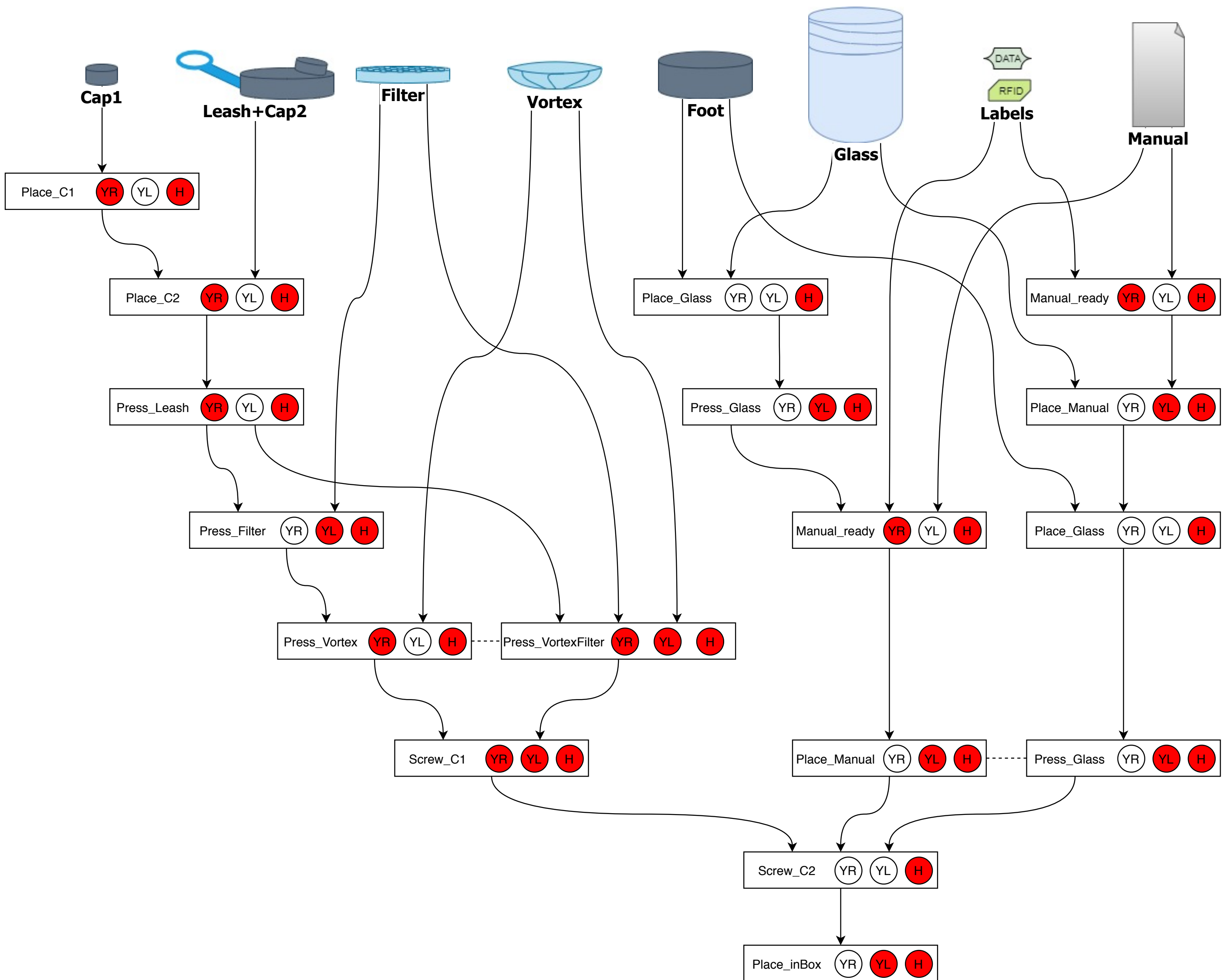
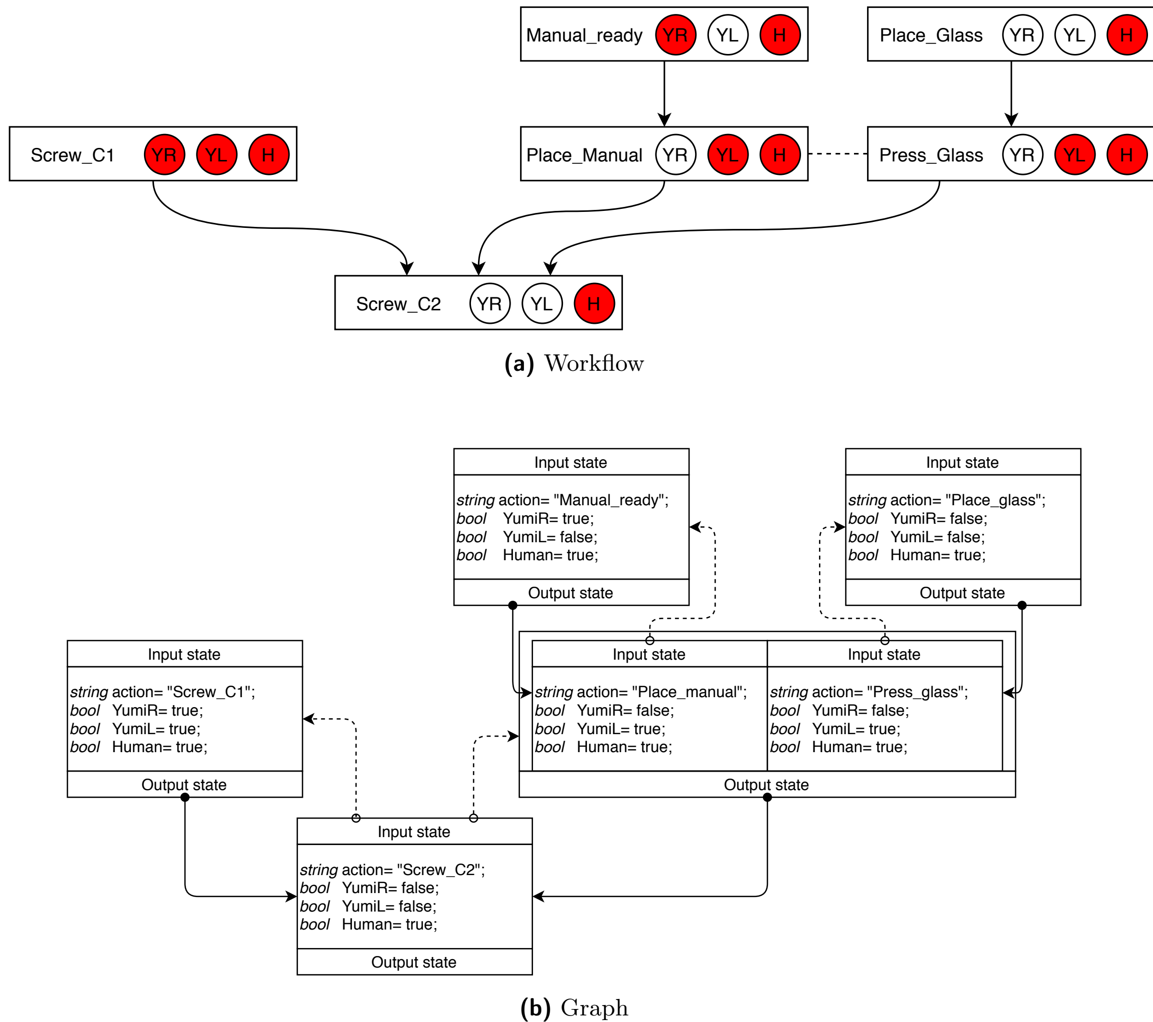


Figure 6.3: Use case workflow

The application "step 2" has as input the workflow XML file generated by the interface. It converts the XML file into a graph that replies the workflow structure, indeed a single node of the graph, called graph state, corresponds to an action panel in the workflow. A graphical representation of this conversion is presented in figure 6.4. Figure 6.4a shows a piece of workflow and figure 6.4b the corresponding piece of graph (it is a simplified snapshot, some attributes are omitted).



(a) Workflow

(b) Graph

Figure 6.4: Focus on a piece of the use case workflow and associated piece of graph

To make the subsequent "step 3" of the application simpler, the graph allows bi-directional exploration from leaves to root and viceversa.

The process of building the graph from the XML starts with algorithm 3. All the classes and attributes are summarised in table 6.1 at the end of the chapter. This function takes the XML as input (row 1). XML is a text file that codes images following specific rules. In row 2, it is decoded and four vectors are created in order to store objects, action panels,

arrows and connectors. From row 3 to row 5, we loop the *objects_vector* and each object ID is used as input of a function called Graph_Generator.

Algorithm 3 Graph

```
1: Inputs : XML_file
2: Decode_XML(XML_file)
3: for i=1 to length(objects_vector) do
4:     Graph_Generator(objects_vector(i).ID)
5: end for
6: Join_gstates(find_connected_gstates())
7: Object_gstates_generator()
8: Return()
```

This function is explained in algorithm 4 and is used to create graph states (abbreviated as gstates) and add them to a vector of graph states called *gstates_vector* (it is used as global variable). All the graph states are associated univocally to an action panel, which corresponds, for each gstate, to the attribute gstate.generator. So, the function works recursively to generate all the graph states associated to the action panels that are placed, on the workflow, among the input object and the root. It's important to highlight that the function is independent on which kind of ID it has as input. It is designed in such a way because in the first call it has an object ID as input while, in the subsequent recursive calls, the input is a panel ID.
In row 2, we use the find_arrow function that outputs a vector of arrows having, as source or target, the panel or the object marked with the ID passed as second input. So, *output_arrows_first* is a vector of arrows having as source the ID used as Graph_Generator input. In row 3, we loop *output_arrows_first*. In row 4, we define *curr_panel* as the panel having an ID equal to the target of the arrow (the objects can't have incoming arrows) using the find_panel function that cycles *panels_vector*. In rows 5 and 6, we create two vectors of arrows: the *input_arrows* vector collects the incoming arrows of the *curr_panel* while the *output_arrows* vector collects its outgoing arrows. In row 7, we verify if the *curr_panel* has already generated a gstate; this check is done by the find_gstate function that cycles *gstates_vector* and output the gstate having gstate.generator=*curr_panel*. If it doesn't exist, a gstate called *curr_gstate* is created (from row 8 to row 14) that means determining all its attributes and then adding it to *gstates_vector*. The *curr_gstate* ID is defined in row 8, the *curr_gstate* generator in row 9, the input and output transition (if the panel is not the root) in, respectively, row 10 and 12. Then, in row 14, the well-defined *curr_gstate* is added to *gstates_vector* and, if the *curr_panel* is not the root, the Graph_Generator function is launched again (row 16).

Algorithm 4 Graph_Generator

```
1:  Inputs : ID
2:  output_arrows_first ← find_arrow("source", ID)
3:  for i=1 to length(output_arrows_first) do
4:      curr_panel ← find_panel(output_arrows_first(i).target)
5:      input_arrows ← find_arrow("target", curr_panel.ID)
6:      output_arrows ← find_arrow("source", curr_panel.ID)
7:      if find_gstate(curr_panel) = NULL then
8:          curr_gstate.ID ← Assign_ID(curr_panel)
9:          curr_gstate.generator ← curr_panel
10:         Define_inp_transitions(input_arrows)
11:         if length(output_arrows) > 0 then
12:             Define_out_transitions(output_arrows)
13:         end if
14:         Add(curr_gstate)
15:         if length(output_arrows) > 0 then
16:             Graph_Generator(curr_panel.ID)
17:         end if
18:     end if
19: end for
20: Return()
```

The algorithm 5 refers to the function Define_inp_transitions. Its aim is to define the attribute gstate.input_transitions of the gstate we are generating. It is a vector composed of transitions that are variables defined by the ID of a gstate and its generator panel. In this case, the transition.gstate_ID obviously refers to the previous gstate in the graph. The input of the function is *input_arrows* (row 1) that determines a loop starting in row 2 and ending in row 21. For each iteration, a transition called *curr_transition* may be created by defining its attributes and, then, added to *input_arrows*. Initially, the *curr_transition*.gstate_ID is set to -1 (row 3) and we define as *input_panel* the panel from which the arrow leaves (row 4). In row 5, we check if this panel exists (the arrow may leave an object) and, if the condition is verified, we define as *input_gstate* the gstate generated by the *input_panel* (row 6). The *input_gstate* can already exist (row 7) and so the *curr_transition*.gstate_ID is trivially assigned (row 8), or it has not been created yet (row 9).
In this case, we don't create a gstate (only the Graph_Generator function can do it) but we assign an ID to it. It can be new as described from row 13 to 16 (*available_ID* is a global variable that stores the value of the last assigned ID), or it can be found in the input and output transitions of other gstates as a consequence of an assignment already performed in

the generation process of other gstates. This check is performed by the check_transitions function from row 10 to 12. Finally, in row 18, the attribute *curr_transition*.panel is defined and in row 19 the *curr_transition* is added to the input_transitions vector of the generating gstate. The Define_out_transitions function works analogously.

Algorithm 5 Define_inp_transitions

```
1:  Inputs : input_arrows
2:  for i=1 to length(input_arrows) do
3:      curr_transition.gstate_ID← -1
4:      input_panel←find_panel(input_arrows(i).source)
5:      if input_panel ≠ NULL then
6:          input_gstate←find_gstate(input_panel)
7:          if input_gstate ≠ NULL then
8:              curr_transition.gstate_ID←input_gstate.ID
9:          else
10:             for j=1 to length(gstates_vector) do
11:                 curr_transition.gstate_ID←check_transitions(gstates_vector(i), input_gstate)
12:             end for
13:             if curr_transition.gstate_ID= -1 then
14:                 curr_transition.gstate_ID←available_ID
15:                 available_ID←available_ID+1
16:             end if
17:         end if
18:         curr_transition.panel←input_panel
19:         Add(curr_transition)
20:     end if
21: end for
22: Return()
```

The Assign_ID function instead works as described in algorithm 6. This function defines the gstate.ID attribute of the gstate we are generating. Doing that, it's necessary to pay attention if the ID of the generating gstate has already been assigned. The input of the function is a panel, called *curr_panel*, that corresponds to the attribute gstate.generator of the generating gstate (row 1). From row 2 to row 13, we loop the *gstates_vector* in order to check the input and output transitions of each gstate that has been already created. From row 3 to row 7, the input transitions are analysed: if *curr_panel* is equal to the attribute input_transitions.panel of a specific input transition (row 4), the attribute input_transitions.gstate_ID of the same transition is returned (it is the already assigned ID). From row 8 to row 12, the same analysis is made for the output transitions. If so far no IDs are returned, it means that the generating gstate needs a new ID. Therefore, in row 14, a new one is computed and in row 16 returned.

Algorithm 6 Assign_ID

```
1: Inputs: curr_panel
2: for i=1 to length(gstates_vector) do
3:     for j=1 to length(gstates_vector(i).input_transitions) do
4:         if gstates_vector(i).input_transitions(j).panel=curr_panel then
5:             Return(gstates_vector(i).input_transitions(j).gstate_ID)
6:         end if
7:     end for
8:     for j=1 to length(gstates_vector(i).output_transitions) do
9:         if gstates_vector(i).output_transitions(j).panel=curr_panel then
10:            Return(gstates_vector(i).output_transitions(j).gstate_ID)
11:        end if
12:    end for
13: end for
14: ID←available_ID
15: available_ID←available_ID+1
16: Return(ID)
```

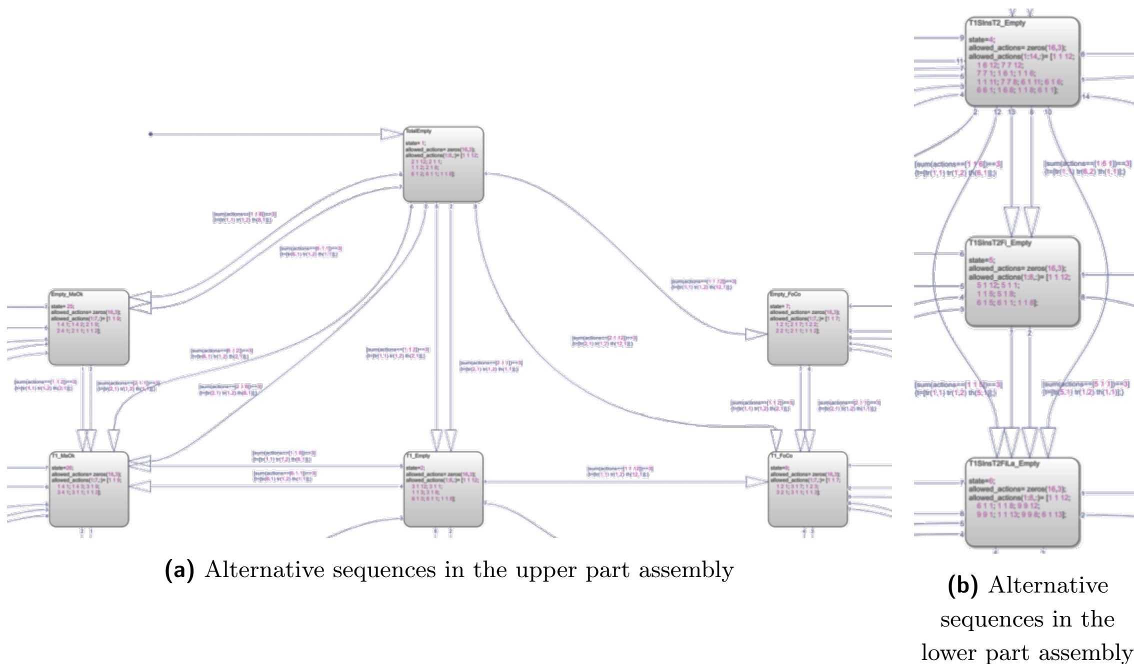
Coming back to algorithm 3, in row 5, the gstates of the graph are all generated. In row 6, we join the gstates having generator panels (i.e. the associated action panels) linked by a connector. These gstates are called connected gstates. The function find_connected_gstates outputs a vector whose elements are sets of connected gstates, while the function Join_gstates joins the connected gstates belonging to the same set. The joined gstates have the following attributes: gstate.ID is new, gstate.generator is a vector containing the generator panels of both the connected gstates, gstate.output_transitions is equal to the one of the connected gstates (they surely have the same output transitions) and gstate.input_transitions is the union of the input transitions of the connected gstates. Beyond creating a new joined gstate, the function Join_gstate manages the input and output transitions of the other gstates in order to keep the graph coherent. Finally, in row 7, the Object_gstates_generator creates particular gstates, called object gstates, that represent objects. They have a specific ID, no generator panels, no input transitions and output transitions that lead to the correct leaves of the graph.

Before proceeding with the explanation of the application "step 3", it is necessary to outline an analysis about the two kinds of parallelism that can be identified into a graph:

- Alternative parallel branches: a parallelism given by two or more incoming arrows into a single graph state that has been generated by some action panels linked with a connector. In this case, the parallelism is among two or more sequences of actions, which are mutually exclusive choices to carry out the same WIP.

- Independent parallel branches: a parallelism given by two or more incoming arrows in a graph state that has been generated by a single action panel. The corresponding action joins two or more semifinished parts, whose creations are totally independent i.e. they do not have any precedence constraints.

This analysis is important because a state of the MDP model is the combination of one or more graph states belonging to independent parallel branches of the graph. So an MDP state, which represents a WIP of the final product (as explained in section 4.4), takes into account also the fact that some processes can be parallelized.
Alternative and independent parallel branches are clearly identifiable in the graph, but they can be noticed also in the MDP whose graphical representation has been already introduced in figure 4.7. It' s possible to notice that it looks like a 2-D matrix since, from almost every state, it is possible to move in two independent directions: the vertical one doing any action regarding the upper part (two caps, vortex and filter) and the horizontal one doing any action regarding the lower part (glass, foot, manual and labels). These main directions correspond with the two independent branches of the graph. For instance, if the independent branches were three, the MDP would be similar to a 3-D matrix. Regarding the alternative parallel branches, below we focus on two clippings of figure 4.7 to highlight some examples.



(a) Alternative sequences in the upper part assembly

(b) Alternative sequences in the lower part assembly

Figure 6.5: Focus on sections of the digital twin representation

As shown in 6.5a, the assembly of the lower part (represented along the horizontal direction) is symmetric with respect to the central column. The main difference between the sequences on the left and the one on the right is the precedence of glass and foot compo-

sition respect to labels and manual one. Also, in 6.5b an example of alternative sequences which evolve along vertical directions is shown. They concern the assembly of the upper part, in particular with vortex and filter that can be placed one by one (two actions case) or together (single action case).

The application "step 3" is the part of the application that converts the graph into the MDP model. It is articulated into three parts: the creation of the MDP states (called states in short) from the gstates, the correct combination of sub-actions in order to carry out the MDP actions (called actions in short) and the process of linking states to other states through the suitable action. All the classes and attributes are summarised in table 6.1 at the end of the chapter.

The algorithm 7 is the first function of the states creation process. Its input is *gstates_vector*, which is the vector containing all the graph states (row 1). In row 2, we define the vector *roots* that contains the gstates that are root of an independent parallel branches (the graph state in which the independent branches merge). They are detected by exploiting the function find_root_parallelism that loops *gstates_vector* picking the gstates that have the length of gstate.input_transition greater than one and the length of gstate.generator equal to one (to reject such gstates that have several input transitions only because they are connected). In row 3, we define a vector of gstates called *ind_gstates* (it may contains also joined gstates and object gstates) and we initially fill it with the overall root of the graph, which is found by using the function find_last_gstate that loops the *gstates_vector* to pick the gstate without output transitions. We also define a gstate variable called *selected*, which is initially set equal to the root gstate (row 4).
In row 5, the Move_among_branch function is launched. The alternated recursive calls of the Move_among_branch function and the Move_along_branch function determine the identification of the gstates that have to be joined in order to create a new MDP state. These gstates, which are from independent branches, are collected in *ind_gstates*. When necessary, *ind_gstates* is passed as input of a Create_state function that properly generates the state.

Algorithm 7 MDP_States_Generator

```
1: Inputs : gstates_vector
2: roots←find_root_parallelism(gstates_vector)
3: ind_gstates(1)←find_last_gstate(gstates_vector)
4: selected← ind_gstates(1)
5: Move_among_branch(ind_gstates, selected, roots)
6: Return()
```

The function Move_among_branch works as explained in the pseudo-code 8. It manages the assignment of the gstate *selected* in order to cross the independent parallel branches. It

can be seen as a movement orthogonal to the direction root-leaves of the graph. This movement is achieved by setting *selected* equal to the successive gstate of *ind_gstates* (function move_selected in row 6) at each recursion of the function Move_among_branch (row 7). When *selected* reaches the last gstate of *ind_gstates*, the function Move_along_branch starts to be called (row 10 and row 12). The function Move_along_branch manages a parallel movement with respect to the root-leaves direction and, if *ind_gstates* is modified, it launches again the function Move_among_branch.
The function Move_along_branch is further explained in the pseudo-code 9. It involves three different scenarios. In the first, *selected* is a root (row 2) since the function is_root, which cycles *roots*, has returned true. In this case, a state is generated (row 4) and *ind_gstates* is modified by the function enlarge_vector. This function replaces the gstate equal to *selected* with its inputs (they are surely more than one since it is a root), which can be found in *selected*.input_transitions. Then, in row 5, the function Move_among_branch is launched again since *ind_gstates* has been changed. In the second scenario, from row 7 to 16, *selected* is not a root but it has one more inputs. It has more inputs only if it is a connected gstate or it is the endpoint of alternative parallel branches. Above all, a state is created (row 8) and then, in row 11, there is a loop whose number of iterations is equal to the number of inputs. Inside the loop, *ind_gstates* is changed by the function modify_vector (row 14). This function replaces the gstate equal to *selected* with its ith input. Then, since *ind_gstates* has been modified, the function Move_among_branch is launched again. In the third scenario (row 17), *selected* has no inputs that means it is a leaf of the graph. In this case a state is created (row 18) and *selected* is set to NULL.
Finally, we specify what is intended as creation of a state. The function Create_state takes as input *ind_gstates* and it simply sets the attribute state.generators equal to *ind_gstates*. Then, this new state is added to a vector called *states_vector* that contains all the already generated states of the MDP.
To sum up, the function Move_among_branch chooses (through the gstate *selected*) in which independent parallel branch the function Move_along_branch has to execute one step toward the leaves. In this way all the admissible combinations of *ind_gstates* occur and, since *ind_gstates* is the input of the function Create_State, all the states are generated.

Algorithm 8 Move_among_branch

```
1: Inputs : ind_gstates, selected, roots
2: while selected ≠ NULL do
3:     if selected ≠ ind_gstates(end) then
4:         mem_ind_gstates←ind_gstates
5:         mem_selected←selected
6:         move_selected(ind_gstates, selected)
7:         Move_among_branch(ind_gstates, selected, roots)
8:         ind_gstates←mem_ind_gstates
```

```
 9:         selected←mem_selected
10:         Move_along_branch(ind_gstates, selected, roots)
11:     else
12:         Move_along_branch(ind_gstates, selected, roots)
13:     end if
14: end while
15: Return()
```

Algorithm 9 Move_along_branch

```
 1: Inputs : ind_gstates, selected, roots
 2: if is_root(selected, roots) then
 3:     Create_State(ind_gstates)
 4:     enlarge_vector(ind_gstates, selected)
 5:     Move_among_branch(ind_gstates, selected, roots)
 6: else
 7:     if length(selected.input_transitions)>0 then
 8:         Create_State(ind_gstates)
 9:         mem_ind_gstates←ind_gstates
10:         mem_selected←selected
11:         for i=1 to length(mem_selected.input_transitions) do
12:             ind_gstates←mem_ind_gstates
13:             selected←mem_selected
14:             modify_vector(ind_gstates, selected, i)
15:             Move_among_branch(ind_gstates, selected, roots)
16:         end for
17:     else
18:         Create_State(ind_gstates)
19:         selected ← NULL
20:     end if
21: end if
22: Return()
```

The creation of the MDP actions and the state connection process are described beginning from the algorithm 10, which is the main function concerning these two parts. Its input is *states_vector*, which is the vector containing all the states just generated (row 1). From row 2 to row 11, we cycle through *states_vector* and for each state the Generate_Actions function outputs all the admissible actions from the current state, called *curr_actions* (row 3). Then, if it exists at least one admissible action, the connection process, which consists in linking two different states through a specific action, starts at row 5. For each action we define a variable called *new_outgoing*, since we set the at-

tribute *new_outgoing*.action equal to the considered admissible action (row 6) and the *new_outgoing*.reached_state equal to the result of Find_State function (row 7). Finally, at row 8, we add the *new_outgoing* variable to the attribute state.followings of the current state.

Algorithm 10 MDP_Actions_Generator

```
1:  Inputs : states_vector
2:  for i=1 to length(states_vector) do
3:      curr_actions←Generate_Actions(states_vector(i))
4:      if length(curr_actions)>0 then
5:          for j=1 to length(curr_actions) do
6:              new_outgoing.action←curr_actions(j)
7:              new_outgoing.reached_state←Find_State(states_vector(i),curr_actions(j))
8:              Add (states_vector(i), new_outgoing)
9:          end for
10:     end if
11: end for
12: Return()
```

The function Generate_Actions is further explained in algorithm 11. It has a state as input (row 1) and, from row 2 to 8, we cycles through all the state generators. Given a specific gstate, which is one of the generators of a state, the function check_output_combinations verifies if the output transition of that gstate can be combined with the output transitions of other gstates in order to generate a new MDP action or, on the contrary, if that gstate is root of a parallelism and so its output transitions cannot be combined with any other ones. The function create_Actions_single_subaction is called at row 5. This function, for each sub-agent of the MDP, cycles all the generator panels of the gstate ($state.generators(i)$) passed as input. Then, it reads the action name associated to each panel and creates all the possible MDP actions (vectors of 3 sub-actions) with only a single activated sub-action (activated means different from “Wait”), which is that one corresponding to the read action name. Finally, in row 9, the function combine_Actions_single_subaction combines all the actions with a single activated sub-action in order to obtain also the actions with two or three activated sub-action, one for each sub-agent. In this way, given a specific starting state, we create all possible and admissible combinations of sub-actions and we return them at row 10.

Algorithm 11 Generate_Actions

```
1:  Inputs : state
2:  for i=1 to length(state.generators) do
3:      if check_output_combinations(state.generators(i))=true then
4:          for j=1 to length(sub_agents) do
5:              temp_actions=create_Actions_single_subaction(state.generators(i))
```

6: **end for**
7: **end if**
8: **end for**
9: state_Actions←**combine_Actions_single_subaction**(temp_actions)
10: **Return**(state_Actions)

The last function, presented in pseudo-code 12 and used in the MDP_Action_Generator function, is Find_State. Its inputs are a state, called *starting_state*, and an *action* (row 1). In row 2, the function identify_subactions_names, which receives as input the action, assigns to a new variable *names* all the sub-actions names. Then, they are used by identify_state_generators_changed function to check which elements of the attribute state.generators of *starting_state* are changed as consequence of performing the sub-actions that corresponds to *names*. These elements are assigned to *state_generators* in row 3. Finally, identify_reached_state function outputs an MDP state generated by all the gstates contained in *state_generators*. This state is assigned to a variable called *reached_state* and, in row 5, returned.

Algorithm 12 Find_State

1: **Inputs** : starting_state, action
2: names←**identify_subactions_names**(action)
3: state_generators←**identify_state_generators_changed**(starting_state, names)
4: reached_state←**identify_reached_state**(state_generators)
5: **Return**(reached_state)

The application "step 4" simulates the interaction between the agent and the environment. It consists in a loop that, at each iteration, has a state variable called *current_state* and an action variable called *current_action* that depends on the policy of the RL algorithms we use. Given these two variables, we determine the state variable called *next_state*. It is found by cycling the vector *current_state*.followings, which contains outgoing variables. We pick the one having an attribute outgoing.action equal to the *current_action*. Given this outgoing variable, its attribute outgoing.reached_state is assigned to the variable *next_state*. From *current_action* we can compute the value of the variable *current_reward*. We loop the three sub-actions of *current_action* and we compare the means of their attributes sub-action.duration. The opposite of the maximum value is assigned to *current_reward* (the RL algorithms may involve a reward shaping that modifies this value). In chapter 7, we also analyse how to determine the attribute sub-action.durations. In this way, we have obtained the digital sample $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$, which is passed as input to the RL algorithms. Finally, the variable *next_state* is assigned to *current_state* and the next iteration of the loop begins.

Workflow		
Class	**Class.Attribute**	**Description**
object	object	An initial piece of the assembly.
	object.ID	A number to identify the object.
panel	panel	It is an action panel, which provides information about the sub-action needed to reach the action panel itself.
	panel.ID	A number to identify the panel.
	panel.name	The name of the sub-action.
	panel.YuMiR	A boolean value that indicates if the sub-action can be performed by YuMiR.
	panel.YuMiL	A boolean value that indicates if the sub-action can be performed by YuMiL.
	panel.Human	A boolean value that indicates if the sub-action can be performed by the human.
arrow	arrow	It links two panels setting a precedence constraint.
	arrow.source	The ID of the panel from which the arrow leaves.
	arrow.target	The ID of the panel in which the arrow arrives.
connector	connector	It links two panels whose sub-actions lead to the same WIP.
	connector.source	The ID of one of the two linked panels.
	connector.target	The ID of the other of the two linked panels.
Graph		
Class	**Class.Attribute**	**Description**
gstate	gstate	It is a state of the graph.
	gstate.ID	A number to identify the gstate.
	gstate.generator	The panel associated univocally to the gstate (it can be a vector of panels if they are linked by a connector).
	gstate.input_transitions	It is a vector of transitions linking the gstate with the previous gstates on the graph.
	gstate.output_transitions	It is a vector of transitions linking the gstate with the next gstates on the graph.
transition	transition	It links two graph states.
	transition.gstate_ID	The ID of the pointed gstate.
	transition.panel	The panel associated to the pointed gstate.
MDP		
Class	**Class.Attribute**	**Description**
state	state	It is a state of the MDP.
	state.generators	The gstates associated to the state.
	state.followings	It is a vector of outgoings linking the state with the next states.
outgoing	outgoing	It links two states.
	outgoing.reached_state	The pointed state.
	outgoing.action	The action that allows the changing of state.
action	action	It is an action of the MDP.
	action.YuMiR	The sub-action performed by YuMiR.
	action.YuMiL	The sub-action performed by YuMiL.
	action.Human	The sub-action performed by the human.
sub-action	sub-action	It is the action performed by a sub-agent.
	sub-action.ID	A number to identify the sub-action.
	sub-action.name	The name of the sub-action.
	sub-action.durations	It is a vector that contains time measures of the sub-action durations.

Table 6.1: Description of the classes and their attributes

Chapter 7

Simulation-based RL solution

In this chapter we outline a simulation-based RL technique that, with the support of a digital twin, can carry out both a static and dynamic operation assignment in HRC manufacturing tasks. We design this technique for our use case and we utilize it to compute a dynamic assignment. Then, a qualitative analysis of the results is presented.

A simulation-based RL technique consists in the application of an RL algorithm to a framework that can simulate the interaction between an agent and the environment. For this purpose, the digital twin of a manufacturing task is a really suitable framework. By exploiting the simulation environment, the optimal scheduling is computed in a very fast way since it doesn't involve a real interaction between the human and the robot. For what concerns our use case, we design a simulation-based RL technique that adopts the two RL algorithms presented in 5.1 and we applied it to the digital twin developed in 6.2.
To carry out the static assignment of the sub-actions the simulation-based RL technique works as the simulations shown in 5.2.2.3. In both the case we simulate the use case and apply the two RL algorithms. Also the resulting optimal scheduling is the same if we feed the digital twin, i.e. we fill the vectors sub-action.durations, with the same estimated durations of the simulations. The optimal scheduling is described in 5.2.3 and the sub-actions durations in 5.1. Anyway, for what concerns the speed of the learning, we have to highlight that we are dealing with two different frameworks: the simulations reproduce an RL framework in which the learning phase is measured in the range of assembled products (or, equivalently, hours of working time), while in a simulation-based RL framework the learning is measured in the range of digital twin iterations, which means almost instantaneous.

Thanks to the instantaneous learning, it's possible to quickly re-compute the optimal scheduling if the human sub-action durations change. So, we can readily react to a situation in which the human gets tired or a working tool breaks and it is replaced with another that leads to slower performances (e.g. an electric screwdriver replaced by a manual one).

In other words, we can carry out a dynamic assignment of the sub-actions. This dynamic assignment consists in repeating the procedure of the static allocation after each assembled product. In order to allow the digital twin to track the human behavior, the attributes sub-action.durations, called memories for the sake of clarity, are continuously updated. They are initialised like the static assignment, then, each time a human sub-action is performed in the assembly, the corresponding memory is updated. The memories have a fixed length, so the filling follows a FIFO logic. So, the digital twin computes the rewards considering estimated durations equal to the moving average of the measured durations.
The synchronization between the simulation-based RL technique and the manufacturing task is the following. At the end of the previous assembly the human memories are updated. Then, simultaneously and in parallel, the simulation-based RL technique and the current assembly start. The two RL algorithms run on the digital twin and achieve the optimal policy earlier than the end of the current assembly. The resulting optimal scheduling will be performed in the next assembly. In this way, a non-stationary human behavior is tracked and the optimal scheduling changes according to it.

The adaptability of the simulation-based RL technique is essential in order to have an optimal dynamic operation assignment. It depends on the length of the memories. We show a qualitative analysis of the adaptability of some simulation-based RL techniques featured with memories having different length. The adaptability of these techniques is evaluated with respect to a dynamic programming algorithm called Policy Iteration (PI) [15], which is an MDP planning algorithm, i.e. an algorithm that process a known MDP model to return the optimal policy. We use PI as reference: at each product we feed it with the MDP structure and the current human sub-action durations (that are unknown in reality and are not available to the digital twin) and, consequently, it outputs the optimal policy synchronized with the changing human behavior. So, we use that policy as a benchmark to evaluate the delay of the simulation-based RL techniques in adapting to changes.
The policy iteration algorithm is shown in algorithm 13.

Algorithm 13 Policy Iteration

1: **Inputs** : MDP=$\langle$S, A, T, R, $\gamma\rangle$
2: Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in$ A arbitrarily $\forall\ s \in S$
3: $\Delta \leftarrow$ arbitrary small value
4: **while** $|V - V_{old}| > \Delta$ **do**
5: $\quad V_{old} \leftarrow V$
6: $\quad T^{\pi} \leftarrow \sum_{a \in A} \pi(s) \cdot T(s'|s,a)$
7: $\quad R^{\pi} \leftarrow \sum_{a \in A} \pi(s) \cdot R(s,a)$
8: $\quad V \leftarrow (I - \gamma \cdot T^{\pi})^{-1} \cdot R^{\pi}$
9: $\quad$ **for all** (s,a) **do**
10: $\quad\quad Q(s,a) \leftarrow R(s,a) + \gamma \cdot \sum_{s' \in S} T(s'|s,a) \cdot V(s')$

11: $\pi(s) \leftarrow \underset{a \in \mathrm{A}}{argmax}\, Q(s,a)$
12: **end for**
13: **end while**
14: **Return**(π)

The policy iteration algorithm is divided into two parts: the policy evaluation part, where the current policy π is evaluated by the state-value function (row 8) and the policy improvement part, where the policy becomes closer to the optimal one by acting greedily (row 11). So, the continuous alternation of policy evaluation and policy improvement leads the optimization to converge to the optimal policy and state-value function. Figure 7.1 recaps this process.

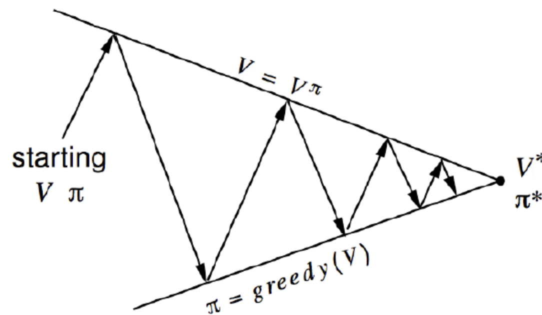


Figure 7.1: Policy iteration alternation between evaluation and improvement phases

A test is organized as follows. We consider three simulation-based RL techniques with memory lengths equal to 8, 10 and 12. Up to the 4th assembly the human behavior is such to lead to an optimal scheduling equal to the one described in 5.2.3. Then, we simulate a change in the human behavior by increasing, as a step, the durations of the performed sub-actions. The new optimal cycle time becomes higher and this increment can be seen as the consequence a worker that gets tired. Obviously, these steps don't perfectly model the evolution in the behavior of a tired worker, however they allow a simple analysis of the dynamic response of the simulation-based RL techniques. The performances of these techniques are shown in figure 7.2. The figure is composed of two plots, one showing the results of the tests in which Q-Learning (with averaged reward) is applied, the other showing the results of the tests using Delayed Q-Learning. On the vertical axis we list the applied techniques: RL algorithms with different memory lengths or the reference PI. On the horizontal axis there is the number of assembling products. The performances are described by coloured circles. Each circle resumes the features of the optimal scheduling returned by a technique at a specific assembled product. The circle size indicates the difference between the optimal scheduling cycle time and the reference cycle time obtained using the PI. These cycle times are given by averaging the results of 10 tests. The circles become larger as the optimal scheduling cycle time gets closer to the reference one. If

the circle is boxed, the cycle time is acceptable i.e. lies within a tolerance interval with respect to the reference one. We have set a tolerance equal to 0.3 s, so the interval is ref_cycle_time ± 0.3s (as a percentage, ref_cycle_time ± 0.65%) The circle colour represents the probability of the technique, calculated over the 10 tests, to output the reference optimal scheduling. The colour changes from red to green as this probability gets higher.

Figure 7.2: Plot of the performances with Q-Learning (top) and Delayed Q-Leaning (bottom). Each circle represents a optimal scheduling returned by a technique. The circle size measures the closeness of the optimal scheduling cycle time to the reference one (larger implies closer). The circle colour measures the probability that the optimal scheduling is the reference one (green implies high probability)

In the figure it's possible to notice that the longer the memories are, the higher the probability is to reach the reference optimal scheduling and the larger the delay is to achieve an acceptable optimal scheduling i.e. the scheduling showing an acceptable cycle time (the boxed circles). Viceversa, the shorter the memories are, the lower the probability is to reach the reference optimal scheduling and the smaller the delay is to achieve an acceptable optimal scheduling. We focus on this last case to explain the overall concept behind these dynamic responses. A short memory quickly forgets the durations referred to the previous human behavior and replaces them with measures of durations referred to

the current human behavior. In this way, we have a small delay to achieve an acceptable optimal scheduling. On the other hand, a short memory implies that the mean of the durations belonging to the memory, which is used to compute the reward and so address the convergence to a specific scheduling, is really subjected to the variance of the duration measures that depends on the human variance. On other words, with a short memory, the mean of the durations does not efficiently filter the human variance. For this reason, a short memory don't accurately model the human behavior even at steady state, hence we have a low probability to reach the reference optimal scheduling. To sum up, the length of the memories determines a trade-off between the speed and the accuracy of the response. Between the two RL algorithms, there are not remarkable differences since, in this framework, they only have to achieve the optimal scheduling, without requirements about the speed of learning.

It doesn't exist a globally correct setting for the memory capacity. It depends on the human variance since, if the human variance is high, a long memory is required, and on the dynamic of the human behavior changes since, if these changes are fast, a short memory is required. Finally, it depends also on the difference, in terms of cycle time, between the reference optimal scheduling and its sub-optimal schedulings. If this difference is low, a long memory is required. This last aspect is mainly due to the product, its workflow and the similarity among the sub-actions necessary to assemble it.

Chapter 8

Conclusions and Future Developments

In this chapter we summarise the obtained results and we propose some possible future developments.

The aim of the thesis has been to solve a typical problem related to the PLC logic: how to choose when two or more actions are simultaneously available. A solution has been found using RL techniques, which allows the learning of the optimal action to execute from a trial-and-error interaction with the environment.
The main steps of our thesis have been the following. Firstly, we have designed a use case. It consists in an industrial assembly task in a HRC domain. We have modeled the use case as an MDP, the framework that underlies the RL techniques. Then, we have chosen two RL algorithms: Q-Learning and Delayed Q-Learning. We have tested them on the use case with the aim of learning the optimal scheduling, which is the one minimizing the time necessary to assemble the product. The first test has been a sensitivity analysis with the objective of tuning the parameters of the algorithms in order to achieve the optimal convergence with the lowest number of episodes. The second test has been a performance comparison between the two RL algorithms in three distinct scenarios, which differ in the number of free actions among which the algorithm chooses. In all the scenarios, the optimal convergence has been achieved faster by Delayed Q-Learning in case of low human variance and by Q-Learning in case of high human variance (except for the first scenario in which, thanks to the simplicity of the MDP, Delayed Q-Learning can perform equally well). We have also evaluated the algorithm performances in terms of working time required to achieve the optimal convergence. In this case, the comparison has been less favorable for the Delayed Q-Learning since its learning phase presents several highly sub-optimal policies. Anyway, it has remained the best algorithm in the case of low human variance.
After that, we have evaluated, the manufacturing task performed following the found optimal scheduling. We have used two standard industrial metrics: the throughput and the

overall equipment effectiveness. They prove that, although the human and robot efficiency can be improved, the performances are satisfying.

The most complex scenario shows that the performances of both the RL algorithms are not suitable for the industrial field since the learning phase is too long. So, in order to speed up the learning, we have developed an application that converts the manufacturing task workflow into its digital twin. This application has been designed to avoid the dependence on computer science specialists, in fact it only requires a drawing of the workflow that can be done by any operator aware of how the manufacturing task works. The graphical elements of the workflow and the functions that constitute the application are deeply explained. The digital twin simulates the interaction between the agent and the environment, from which digital samples are collected. In this simulation-based reinforcement learning framework, we have designed a technique that involves the two previous RL algorithms and the digital twin in order to determine a static and a dynamic assignment of the sub-actions in the face of a non-stationary human behavior. Thus, we have done a qualitative analysis of the adaptability of this technique. We have highlighted how the velocity and the accuracy of the response are affected by the length of the memories that feed the digital twin.

The first future development obviously involves the validation of the RL and simulation-based RL techniques with data coming from the use case in the reality. Then, the thesis work can be extended by integrating other aspects of the HRC. For example, the possibility of assembling more than one product at a time may be modeled. In this way, the human and the robot may exploit the wait time and, consequently, increase the TH (the variable WIP raises) and the OEE index (the efficiencies grow). Another aspect to model may be the risk that an assembly action fails. This risk may be depending on which sub-agent performs the action or on the action itself (e.g. a screwing action can be subjected to a higher risk than a simple insertion). A hint to carry out this variation is the following. A new state may be added in order to represent a “wrong” WIP. Then, the transition function has to be modified in order to allow the reaching of this “wrong” WIP state with a probability greater than 0. An interesting development concerns the reward function that, beyond minimizing the cycle time, may take into account the psychophysical wellness of the human. It can be measured and suitably integrated into the reward function using some metrics such as, for instance, the REBA score that assesses the risk of musculoskeletal disorders or the Heart Rate Variability to evaluate the stress.
Finally, it is possible to further analyse the adaptability of the simulation-based RL technique and elaborate a method to efficiently set the length of the memories. This method has to well manage the trade-off between velocity and accuracy considering the human variance, the dynamic of the human behavior changes and the presence of sub-optimal schedulings with cycle times closed to the optimal one. Then, the analysis can be generalised to more accurate models of the non-stationary human behavior.

Appendix

.1 IRB 14000 YuMi Datasheet

Specification

Robot version	Reach (mm)	Payload (g)	Armload
IRB 14000-0.5/0.5	559	500	No armloads
Number of axes	14		
Protection	Std: IP30 and Clean Room		
Mounting	Table		
Controller	Integrated		
Integrated signal and power supply	24V Ethernet or 4 Signals		
Integrated air supply	1 per Arm on tool Flange (4 Bar)		
Integrated ethernet	One 100/10 Base-TX ethernet port/per arm		

Performance (according to ISO 9283)

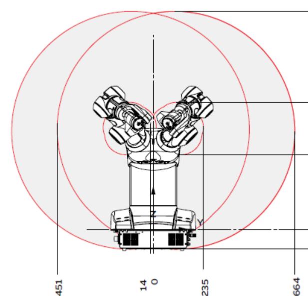
IRB 14000-0.5/0.5	
0.5 kg picking cycle	
25* 300 * 25 mm	0.86s
Max TCP Velocity	1.5 m/s
Max TCP Acceleration	11 m/s*s
Acceleration time 0-1m/s	0.12s
Position repeatability	0.02 mm

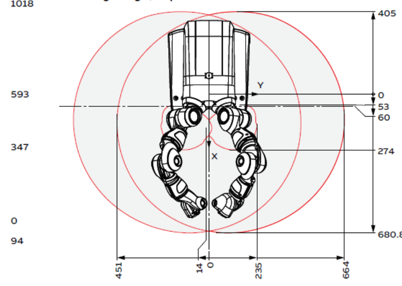
Technical information

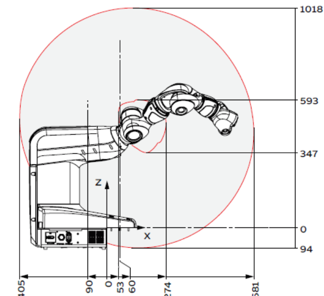
Physical	
Robot base	399 x 496 mm
Robot toes	399 x 134 mm
Weight	38 kg
Environment	
Ambient temperature for mechanical unit	
During operation	+5°C i (41°F) to +40°C (104°F)
During transportation and storage	-10°C (14°F) to +55°C (131°F)
Relative humidity	Max. 85%
Noise level	< 70 dB
Safety	PL b Cat B

Movement

Axis movement	Working range	Axis max. speed
Axis 1 arm rotation	-168.5° to +168.5°	180°/s
Axis 2 arm bend	-143.5° to +43.5°	180°/s
Axis 7 arm rotation	-168.5° to +168.5°	180°/s
Axis 3 arm bend	-123.5° to +80°	180°/s
Axis 4 wrist rotation	-290° to +290°	400°/s
Axis 5 wrist bend	-88° to +138°	400°/s
Axis 6 flange rotation	-229° to +229°	400°/s







Bibliography

[1] Lasi, Heiner Fettke, Peter Kemper, Hans-Georg Feld, Thomas Hoffmann, Michael. (2014). Industry 4.0. Business Information Systems Engineering. 6. 239-242.

[2] Preuveneers, Davy Ilie-Zudor, Elisabeth. (2017). 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.

[3] El Zaatari S., Marei M, Li W., Usman Z., 2019. Cobot Programming for Collaborative Industrial Tasks: An Overview. Robotics and Autonomous Systems 116 (June): 162–180.

[4] Zanchettin, A.M.; Marconi, M.;Ongini, C.; Rossi, R.; Rocco, P.; "A Formal Control Architecture for Collaborative Robotics Applications", Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, 2019.

[5] Huang, Chin-Jung. "Integrate the Hungarian method and genetic algorithm to solve the shortest distance problem." 2012 Third International Conference on Digital Manufacturing Automation. IEEE, 2012.

[6] Malik, Ali Ahmad and Arne Bilberg. "Collaborative robots in assembly: A practical approach for tasks distribution." Procedia CIRP 81 (2019): 665-670.

[7] Antonelli, Dario Bruno, Giulia. (2019). Dynamic distribution of assembly tasks in a collaborative workcell of humans and robots. FME Transactions. 47. 723-730.

[8] Wilcox, Ronald, Stefanos Nikolaidis, and Julie Shah. "Optimization of temporal dynamics for adaptive human-robot interaction in assembly manufacturing." Robotics 8 (2013): 441.

[9] Johannsmeier, Lars, and Sami Haddadin. "A hierarchical human-robot interaction-planning framework for task allocation in collaborative industrial assembly processes." IEEE Robotics and Automation Letters 2.1 (2016): 41-48.

[10] Cividini, Filippo. A scheduling algorithm for human-robot collaborative assembly tasks. Master of Science thesis, Politecnico di Milano, 2017.

[11] Kinugawa, J., Kanazawa, A., Arai, S., Kosuge, K. (2017). Adaptive task scheduling for an assembly task coworker robot based on incremental learning of human's motion patterns. IEEE Robotics and Automation Letters, 2(2), 856-863.

[12] El Makrini, Ilias Merckaert, Kelly De Winter, Joris Lefeber, Dirk Vanderborght, Bram. (2019). Task allocation for improved ergonomics in Human-Robot Collaborative Assembly. Interaction Studies. 20. 103-134.

[13] Akkaladevi, Sharath Plasch, Matthias Chowdhary, Maddukuri Eitzinger, C. Pichler, Andreas Rinner, Bernhard. (2018). Toward an Interactive Reinforcement Based Learning Framework for Human Robot Collaborative Assembly Processes. Frontiers in Robotics and AI. 5. 126.

[14] El-Telbany, Mohammed. (2003). Reinforcement Learning Algorithms for Multi-Robot Organization.

[15] Littman, Michael L.. "Algorithms for sequential decision-making." (1996).

[16] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. Cambridge, MA: MIT Press, 1 ed., 1998.

[17] Bertsekas, Dimitri P. and John N. Tsitsiklis. "An Analysis of Stochastic Shortest Path Problems." Math. Oper. Res. 16 (1991): 580-595.

[18] Bonet, Blai Gener, Hector. (2002). Solving Stochastic Shortest-Path Problems with RTDP.

[19] Kaelbling, L. P.; Littman, M. L.; and Moore, A.W., 1996. Reinforcement learning: a survey. Journal of Artificial Intelligence Research, 4 (1996), 237–285.

[20] Desai, Deven R., Exploration and Exploitation: An Essay on (Machine) Learning, Algorithms, and Information Provision (December 15, 2015). 47 Loyola University Chicago Law Journal, 541 (2015); Georgia Tech Scheller College of Business Research Paper No. WP44.

[21] Shirine El Zaatari, Mohamed Marei, Weidong Li, Zahid Usman, Cobot programming for collaborative industrial tasks: An overview, Robotics and Autonomous Systems, Volume 116, 2019, Pages 162-180.

[22] Robots and robotic devices – Collaborative robots, ISO Standard ISO/TS 15066:2016, 2016.

[23] Watkins, C.J., Dayan, P. Technical Note: Q-Learning. Machine Learning 8, 279–292 (1992).

[24] W. Xia, C. Di, H. Guo and S. Li, "Reinforcement Learning Based Stochastic Shortest Path Finding in Wireless Sensor Networks," in IEEE Access, vol. 7, pp. 157807-157817, 2019.

[25] Strehl, Alexander Li, Lihong Wiewiora, Eric Langford, John Littman, Michael. (2006). PAC model-free reinforcement learning. ICML 2006 - Proceedings of the 23rd International Conference on Machine Learning. 2006.

[26] Strehl, Alexander Li, Lihong Littman, Michael. (2009). Reinforcement learning in finite MDPs: PAC analysis. Journal of Machine Learning Research. 10. 2413-2444.

[27] István Szita and András Lőrincz. 2008. The many faces of optimism: a unifying approach. In Proceedings of the 25th international conference on Machine learning (ICML '08). Association for Computing Machinery, New York, NY, USA, 1048–1055.

[28] D. Schwung, F. Csaplar, A. Schwung and S. X. Ding, "An application of reinforcement learning algorithms to industrial multi-robot stations for cooperative handling operation," 2017 IEEE 15th International Conference on Industrial Informatics (INDIN), Emden, 2017, pp. 194-199.

[29] Ramya Ramakrishnan. Perturbation Training for Human-Robot Teams. Master of Science thesis, Massachusetts Institute of Technology, 2015.

[30] Minati, Marco. Tempi e Metodi, Ipsoa, 1°ed, 2012.

[31] Morioka, M. Sakakibara, S.. (2010). A new cell production assembly system with human–robot cooperation. Cirp Annals-manufacturing Technology - CIRP ANN-MANUF TECHNOL. 59. 9-12.

[32] Ericsson J., 1997, Disruption Analysis – an important tool in Lean Production, Department of Production and Materials Engineering, University of Lund.

[33] Ljungberg O., 1998, “Measurement of overall equipment effectiveness as a basis for TPM activities”, International Journal of Operations Production Management, Vol. 18, Issue 5, pp. 495 – 507.