FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Dynamic Task Graphs for Teams in Collaborative Assembly Processes

Ana Marisa Machado Macedo

DISSERTAÇÃO DE MESTRADO

U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado em Engenharia Informática e Computação

Supervisor: Vítor Hugo Pinto

March 01, 2022

Dynamic Task Graphs for Teams in Collaborative Assembly Processes

Ana Marisa Machado Macedo

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Henrique Lopes Cardoso

External Examiner: Prof. Hugo Proença

Supervisor: Prof. Vítor Pintor

March 01, 2022

Abstract

Collaborative robots are increasingly used in industry as they improve efficiency. Particularly in assembly processes, collaboration, whether Human-Robot or Robot-Robot, expands the possible alternative sequences of operation and ways of team allocation to complete the an assembly task. Since an assembly task can be discretized into simpler and more specific subtasks, building a graph, composed of the possible sequences of operations needed to complete the task and the necessary constraints, is one approach that improves the flexibility provided by team collaboration, where one can easily observe the flow of actions that any team member must execute. Nevertheless, the best sequence must be selected to reduce stoppage times and increase productivity and profit. It is still challenging for a robot to choose the best operations to perform taking into account execution time, partner actions and the process goal.

This dissertation aims to create a novel system composed of three different modules that, in a closed-loop interaction, will allow a robotic agent to correctly plan a task given a set of operations, and optimize the task sequence allocation and scheduling plan, automatically triggering the correct and optimal operation given its partner's actions and the process objective. The first module corresponds to the graph task's representation, using hierarchical task structure combined with a directed graph. The second represents the intelligence component using Machine Learning algorithms for the learning process, namely two off-policy Reinforcement Learning algorithms (Q-learning and Deep Q-learning). Lastly, the third module corresponds to the simulation of the task given the learned sequences, executing the task in a V-REP simulator and updates the graph throughout the execution. The solution was applied to a table assembly task and tested with different parameters and rewards.

The system was able to converge successfully in 3 scenarios: a table with 1 leg, a table with 2 legs and a table with 4 legs, in 400, 27 000, and 20 000 episodes, respectively. Moreover, when training the robot for task allocation, it was possible to perceive the impact of the human operator expertise in the robot's decisions.

The possibility of having an intelligent robot that can collaborate with a partner to complete complex assembly processes would benefit manufacturing industries, helping reduce human workload and improving efficiency. The results and conclusions of this work resulted in a scientific paper submitted to the ICAART conference.

Keywords: Collaborative Robots, Assembly Tasks, Task Allocation and Scheduling, Task Graph, Reinforcement Learning, Deep Reinforcement Learning, Q-learning, Deep Q-learning

Resumo

Robôs colaborativos têm sido cada vez mais utilizados na indústria, uma vez que estes aumentam eficiência. Particularmente nos processos de montagem, a colaboração, quer seja Humano-Robô ou Robô-Robô, expande as possíveis alternativas de sequência de operações e formas de alocação de membros da equipa para completar a montagem de um produto. Uma vez que uma tarefa de montagem pode ser discretizada em subtarefas mais específicas e simples, construir um grafo composto pelas possíveis sequências de operações necessárias para completar a tarefa é uma abordagem que melhora a flexibilidade proporcionada pela colaboração da equipa, onde se pode facilmente observar o fluxo de ações que qualquer membro da equipa deve executar. No entanto, a melhor sequência deve ser selecionada para reduzir os tempos de paragem e aumentar a produtividade e o lucro. Contudo, ainda é desafiante para um robô escolher as melhores operações a executar tendo em conta o tempo de execução, as ações do parceiro e o objetivo do processo.

Esta dissertação visa criar um sistema inovador composto por três módulos diferentes que, numa interacção em circuito fechado, permitirá a um agente robótico planear corretamente uma tarefa dado um conjunto de operações, assim como optimizar a atribuição e agendamento da sequência de tarefas, desencadeando automaticamente a operação correcta e óptima dada a ação do seu parceiro e o objectivo do processo. O primeiro módulo corresponde à representação gráfica da tarefa, utilizando uma estrutura hierárquica de tarefas combinada com um grafo dirigido. o segundo corresponde à componente de inteligência utilizando algoritmos de *Machine Learning* para o processo de aprendizagem (utilizando dois algoritmos de Aprendizagem Reforçada: Q-learning e Deep Q-learning). Por último, o terceiro módulo corresponde à simulação da tarefa dada as sequências aprendidas, executando a tarefa num simulador V-REP e atualizando o grafo durante toda a execução. A solução foi aplicada a uma tarefa de montagem de mesa e testada com diferentes parâmetros e recompensas.

O sistema foi capaz de convergir com sucesso em 3 cenários: uma mesa com 1 perna, uma mesa com 2 pernas e uma mesa com 4 pernas, em 400, 27 000, e 20 000 episódios, respetivamente. Além disso, ao treinar o robô para a atribuição de tarefas, foi possível perceber o impacto da perícia do operador humano nas decisões do robô.

A possibilidade de ter um robô inteligente que possa colaborar com um parceiro para completar processos de montagem complexos beneficiaria as indústrias de manufactura, ajudando a reduzir a carga de trabalho humano e melhorando a eficiência.

Os resultados e conclusões deste trabalho resultaram num artigo científico submetido para a conferência ICAART.

Palavras-chave: Robôs Colaborativos, Tarefas de Montagem, Alocação e Programação de Tarefas, Grafo de Tarefa, Reinforcement Learning, Deep Reinforcement Learning, Q-learning, Deep Q-learning

Acknowledgements

Começo por agradecer ao orientador Doutor Vitor Hugo Pinto e Professor Doutor Gil Gonçalves pelo desafio proposto e oportunidade de percorrer esta etapa com a DIGI2Lab.

Ao Professor Doutor João Reis agradeço o excelente acolhimento, todas as conversas espontâneas e de desanuviamento, e por todo o conhecimento partilhado.

À Liliana Antão pela disponibilidade, confiança, motivação que me dava energia para o resto da semana, por todas as reuniões de *brainstorming* e pela inspiração que dá.

A todos os membros do Digi2Lab pelo confortável ambiente de trabalho criado e por todas as partilhas de ideias, experiências, conselhos e almoços.

À Faculdade de Engenharia da Universidade do Porto pelo serviço e condições prestadas e aos professores que me educaram para chegar até aqui.

Um agradecimento especial à minha família e em especial aos meus pais que sempre trabalharam para me proporcionarem esta oportunidade, que perguntavam todos os dias como ia a tese, que sempre me apoiaram e me educarem para me tornar quem sou hoje. Agradeço ainda ao meu irmão que me inspirou a focar no meu percurso académico.

Aos meus queridos amigos Algodões e Puzzlezitos por todo o suporte, convívios e alegria que me providenciaram. Por estarem sempre lá, mesmo quando à distância.

Não há palavras que expressem o meu agradecimento ao meu namorado e melhor amigo, por sempre me apoiar, por me fazer companhia e me motivar em todos estes meses de trabalho, por me fazer acreditar que tudo é possível e por dar os melhores abraços do mundo.

Ana Marisa Machado Macedo

"To dare is to lose one's footing momentarily. Not to dare is to lose oneself."

Soren Kierkegaard

Contents

List of Figures

List of Tables

Listings

Abbreviations

AI	Artificial Intelligence
AL	Apprenticeship Learning
BoM	Bill of Materials
CC-HTN	Clique/Chain Hierarchical Task Network
CNN	Convolutional Neural Network
CTG	Conjugate Task Graph
DDPG	Deep Deterministic Policy Gradient
DADRL	Dual-agent Deep Reinforcement Learning
DL	Deep Learning
DP	Dynamic Programming
DNN	Deep Neural Network
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
GUI	Graphical User Interface
HRC	Human-Robot Collaboration
HTM	Hierarchical Task Model
HTN	Hierarchical Task Network
IRL	Inverse Reinforcement Learning
MADDPG	Multi-agent Deep Deterministic Policy Gradient
MARL	Multi-agent Reinforcement Learning
MaxEnt	Maximum Entropy
MC	Monte Carlo
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process

ML	Machine Learning
MOMDP	Mixed Observability Markov Decision Process
PG	Policy Gradient
POMDP	Partially Observable Markov Decision Processes
RL	Reinforcement Learning
SARL	Single-agent Reinforcement Learning
TD	Temporal Differences
TLC	Teaching-Learning-Collaboration

Chapter 1

Introduction

In this chapter, the subject of this thesis is contextualized, introducing the problem to be solved, its specifics and how it became the object of the research. Furthermore, the motivation for doing this work is presented, as well as its overall objectives. In the end, the structure of the document is described.

1.1 Context

With the fourth industrial revolution, also known as Industry 4.0 intelligent manufacturing and industrial collaborative robots emerged, which successfully perform repetitive tasks with high precision, boosting efficacy and productivity. However, it is challenging for a robot with limited cognitive abilities to complete complex tasks independently and meet high product variety and mass customization demands [68]. Moreover, due to humans' ability and flexibility, it is impossible to completely replace them with robots. On the other hand, it is also impractical for humans to perform a task in a way that robots encode task models [29]. Thus, new needs emerge with the increasing necessity of collaboration between humans and robots. Human–robot and robot–robot interactions have been identified as a feasible solution, as they join the best characteristics of each element and allow them to complete complex tasks more efficiently while maintaining the quality of the final product. Particularly in assembly processes, different robot and gripper structures are developed to assist workers [61] and in general, most of the tasks focused on holding an object for the person, laying it aside or retrieving it on demand [64], [23].

Moreover, the production engineering society is giving considerable attention to collaborative systems, since they expand the possible alternative sequences of operations to perform and ways of team allocation to complete the assembling of a product, taking advantage of the fact that the tasks can be divided into more specific and discrete operations. Given this, creating a graph that represents the possible sequences of operations necessary to complete a task along with its constraints would provide more flexibility to team collaboration. In such a representation, any team member could easily observe the flow of actions it or its partner must execute, being able to formulate a complete sequence of steps to correctly perform a given task. However, the best sequence of operations planned for a specific product and partner must also take into account aspects like reducing stoppage times and increasing productivity and profit. The problem is that robots do not have the human capacity to plan the best way of completing a task with a partner, namely the best operations to perform in a specific time, given a particular set of process parameters, the partner's actions at that time. Given this, the possibility of having an intelligent robotic agent that, given a task graph, can learn which action to trigger to efficiently achieve the goal task, according to its team partner's actions and task environment, would be highly beneficial.

1.2 Motivation

Research efforts on collaborative robots have been multiple in the past few years. However, there are still significant challenges in the manufacturing processes. Particularly, it is challenging to allocate collaborative subtasks for both human operators and robots since the computation of optical strategies is time-consuming and energy-intensive [18, 68]. Besides, the behavior of a human operator involved in collaboration can be unpredictable in assembly activities, and the robot cannot accurately identify human intentions, which makes collaboration difficult [68].

Despite an active interest in the research community, few advances have been made to get the most out of the efficiency given by team collaboration in manufacturing industries (human-robot, or even robot-robot), and the division of workload between the members of a team is usually based on fixed rules, and both of them mainly perform high-frequency repetitive actions [68].

The high intensity of work given to operators in assembly processes can lead to their increased unavailability to work, affecting the stability of processes and the efficiency of operations [12, 27]. And although robots have high operational efficiency, it is difficult for them to complete the entire assembly process independently. Therefore, it is critical to allocate tasks to the members of a team in a reasonable manner [68].

The primary motivation of the research work developed in this dissertation is to contribute with a novel approach in the field of team collaboration in assembly processes. This approach focuses on adapting an agent's decisions automatically and intelligently, by choosing the sequence of operations to be performed within a given task.

1.3 Objectives

The system must be a simple and modular architecture that acquires data from the graph and sends back decisions to trigger nodes in a closed-loop way. Given the problem and the motivation of this work, we can identify the following objectives:

- Compile key concepts and appropriate methodologies for collaborative tasks execution in industry, within the scope of intelligent agents;

- Identify relevant approaches with similar solutions, regarding the compiled approaches and graph task structures;
- Create a modular system that includes the task representation in a **graph**, Machine Learning methods for **learning** and the task **simulation**;
- The graph must represent the sufficient and necessary information for the agent to efficiently learn;
- Create a system that is applicable and scalable to other scenarios;
- Use **Machine Learning** methods to teach an agent to correctly choose the task sequence in an assembly processes: the robot must automatically trigger the correct and optimal operation (node) given its partner's actions and the process objective;
- Visualize and update the task status in the graph during the simulation, so that the operator can easily visualize the whole assembly process and be guided to perform corresponding tasks.
- Compare the solution's performance between different algorithms, rewards and parameters;
- Validate the solution in a complex and well-defined case study;

1.4 Document Structure

We can divide this paper into four main parts. The first, including Chapters 2 and 3, introduces the reader to the domain of this thesis through a review and analysis of **related and available concepts and projects**. The second part, Chapter 4, presents the **methodology**, namely the case study and the proposed solution in Chapter 4, and the implementation of our proposal in Chapter 5. Chapter 6 presents the **tests and validations**. And lastly but not least, 7 corresponds to the final **conclusions**.

In **Chapter 2** we introduce the fundamental notions and a brief background on the topics mentioned in the subsequent chapters, namely Machine Learning (Machine Learning (ML)), Reinforcement Learning (RL), Q-Learning, Deep Reinforcement Learning (Deep Reinforcement Learning (DRL)), Deep Q-Network (DQN), Deep Deterministic Policy Gradient (DDPG) and Inverse Reinforcement Learning (Inverse Reinforcement Learning (IRL)). This chapter aims to understand the approaches applied and not applied on this thesis and why. **Chapter 3** presents the current knowledge about two main topics: Machine Learning methods applied to collaborative manufacturing tasks, and graph simulation structures for task allocation and scheduling in collaborative tasks.

Chapter 4 is intended to describe the problem of collaborative assembly processes using one case study, as well as the proposed solution, including the methodology and architecture of the system. The algorithms chosen are presented in **Chapter 5**, along with the system structure, the tools and technologies utilized, and the implementation details.

Chapter 6 describes and analysis the tests performed and their corresponding results, gathering a description of the experiment, analysis on reward function and relevant considerations. This chapter also compares the results between the two chosen algorithms and how the hyperparameter's values can impact the training performance.

The final conclusions are given in **Chapter 7**, comprising final reflections and findings in this dissertation and suggestions on future work.

Chapter 2

Background

This chapter presents an overview of the main concepts related to this thesis' scope, starting by the more general one: Machine Learning (ML). ML has three different approaches related to algorithm training: Supervised Learning, Unsupervised Learning and Reinforcement Learning. In this chapter, we focus on learning from reward feedback, RL, and introduce this methodology's main concepts, and a simple and commonly used RL algorithm called Q-learning, essential to understand this work and build the solution.

Later, we introduce an expansion of RL, called Deep Reinforcement Learning DRL, and how its algorithm DQN and DDPG solve some RL's limitations. Furthermore, we give a brief review over a ML algorithm which appears to solve the inverse problem of RL by learning from demonstration: IRL.

All the presented ML techniques can be implemented for assembly tasks in manufacturing. However, some are more suitable than the others for different problems and requirements. A brief background of these concepts is essential to understand the difference between the concepts and which approaches are more suitable to build this dissertation's solution.

The writing of this chapter was based on current literature and well-established definitions.

2.1 Machine Learning

To keep up with the increasing pace and complexity of the environment, manufacturing industries needed solutions that would automate their processes and complex problem-solving. Machine Learning can be applied in this scenarios and has become a high-priority technology, allowing manufacturers to create intelligent machines capable of operating and interacting with other agents in diverse and dynamic environments with minimal human intervention.

According to *Michalski and Anderson* [34], ML is a sub-field of Artificial Intelligence (Artificial Intelligence (AI)) that gives computers the ability to learn from experience without explicitly being programmed, gradually improving their accuracy. Through ML, computers can automatically learn patterns and relationships from observations of the surrounding environment to make reliable decisions and predictions.

To use some Machine Learning algorithms, an input **dataset** must be given at the beginning, containing the necessary information for the algorithm to generate its output. This dataset is gathered and prepared, and from there, a ML **model** uses the data to train itself to extract patterns and produce decisions and predictions. To evaluate how well the algorithm models the data set, ML uses a method called loss function. This method measures how far the expected output is from the algorithm's actual output by measuring the distance between them. The model seeks to minimize this distance.

Based on the provided problem and data set, we can distinguish three main types of ML: **supervised learning**, **unsupervised learning** and **reinforcement learning**. Supervised learning involves learning a function that maps the input data to the target variable. This type of ML requires that the training data contains input-output pairs, which means that the data must be labeled. The model is fit on the training data and used to predict the target variable given new or unseen data. The outputs from the model are then compared to the already known target variables and used to estimate the accuracy of the model.

In unsupervised learning the data is not labeled. Thus, the model needs to detect patterns of interest in data without knowing any outputs or target variables. Such patterns can be for example groups of elements that share the same property or data representations that are converted from high-dimensional spaces into a lower ones, known as clustering and dimension reduction techniques respectively.

In reinforcement learning there is no fixed training dataset. Rather, an agent must learn by trial and error with the objective to maximize a reward. The agent operates in an environment and must learn, knowing the goal he is required to achieve, a list of possible actions to perform, and feedback about its current performance toward the goal. RL is different from unsupervised learning as there is feedback. However, this feedback is not explicit, so it also differs from supervised learning.

In addition to these 3 types of ML algorithms, this subfield offers a fourth type: Deep Learning. Additionally, each of these types comes with multiple specifications and variants, including Inverse Reinforcement Learning (IRL) and Deep Reinforcement Learning (DRL). RL, DRL, and IRL are some of the most commonly used ML techniques in robotics and manufacturing industries and, fpor that reason, this dissertation will explore one of these methods. They have become more prevalent in production and assembly processes, as they help reduce production cost and time.

2.2 Reinforcement Learning

Sutton and Barto [56] describe Reinforcement Learning as a sub-field of ML that involves learning to map states to actions in order to maximize the overall reward of a problem. In RL, we do not teach an agent which actions to make but present it with rewards whether positive or negative based on its decisions. For that, there is a **closed-loop** interaction between an agent and the environment, as illustrated in Fig. 2.1.

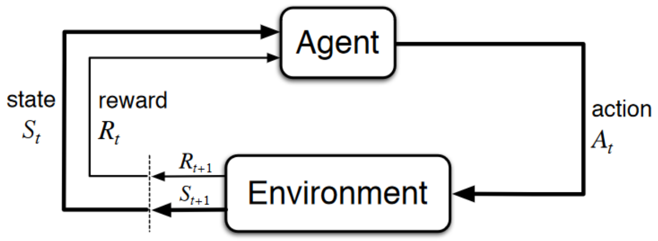


Figure 2.1: Closed-loop agent–environment interface and their interaction through actions, states and rewards. [56]

An **agent** is the learner and decision-maker. The surroundings he perceives and with which he interacts is called the **environment**. A **state** is a combination of observable variables or features on the environment. An example of a feature can be a location. If there is more than one feature, one way to represent the state is by creating a multidimensional array such as S[feature 1, feature 2, feature 3]. The decision an agent makes on the environment is called an **action**. And a **reward** is a feedback given to the agent based on the action it performed.

At each time step t, the agent observes the environment through sensors, and reads the current state s_t in a finite set of possible states S. It selects an action a_t in a finite set of possible actions A at s_t, and takes it using actuators. At time $t + 1$, the environment returns the respective reward r_{t+1} and transitions to the next state s_{t+1}, upon which the agent relies for its future decisions.

There is a difference between a time step t and a state s. A state is defined by the environment and an agent can pass through a state multiple times at each episode, while a time step occurs only once and is defined by the agent. This means that the agent can be in the same state at two different time steps.

2.2.0.1 Problem elements

Reinforcement learning problems include three main elements: a **policy**, a **reward signal** and a **value function**.

A policy is a probability distribution that defines the mapping from perceived states of the environment to actions that the agent must take when they are in those states. A policy can range from simple functions to complex functions. This element is the core of a RL agent in the sense that it alone can determine behavior. Updates on the policy are made based on the results of many experiments regardless of how the task evolves, and when a goal is reached or not, the entire behavior is valued or penalized.

A reward signal, at a given state, is a feedback given by the environment when an agent takes an action. It thus defines which actions contribute or not to achieve the task's goal along the execution. The final goal is to maximise the cumulative rewards by altering the necessary sequence of actions to complete the task. Thus, if an action selected by a policy receives low reward, then the policy may be changed to choose some other action in that state in the future. Whereas the reward signal gives immediate feedback, a value function concerns about achieving the best global solution. This element estimates which states (or actions in a certain state) are promising in the long run, by looking at the expected future reward. For example, an agent may choose an action with a lower immediate reward, but that moves to a more promising state, allowing it to receive a higher reward in the long run.

It can optionally there be a fourth element if the transitions of the environment are known, the **model** of the environment. This model is used to plan actions, by considering possible future situations before actually committing to them. For instance, given a state and action, the model might predict the resultant next state and next reward. Methods for solving RL problems that use models and planning are called **model-based** methods. In these methods, the agent tries to understand the environment and creates the model based on its interactions with the environment. Model-free methods, on the other hand, are trial-and-error learners. The agent will select an action multiple times and will adjust the policy for optimal rewards, based on the outputs.

2.2.0.2 Policy

An RL agent must prefer actions that lead to higher rewards based on its experience in the problem. This technique is called exploitation, since the agent exploits its knowledge in order to maximize the overall reward. On the other hand, if the agent only exploits, it will never discover other actions that may potentially lead to better rewards. The agent must try a variety of actions and explore the environment: exploration. The trade-off between **exploitation** and **exploration** brings a considerable challenges to Reinforcement Learning problems. The two must be **balanced** in order to allow the Agent to both exploit with is experience and repeat the maximum rewarding path it found, but also to explore the environment.

To find the balance, the right policy must be defined. With a **greedy policy**, the agent constantly chooses the action that is believed to outcome the highest expected reward, with a selection probability of 1 (only exploits), which does not give the balance we want. For that, **ε-greedy policy** is often used instead. With this policy, a number ε between [0,1] and a random number (x) also in the range of [0,1] are defined, and prior selecting an action, if x is larger than ε, the greedy action is chosen, otherwise a random action is selected.

Fig. 2.2 portrays an example of all the possible actions in a game, in this case tic-tac-toe, and the sequence of actions chosen by the learning player and the opponent.

The dashed lines represent the possible moves our learning player could choose, and the solid lines represent the moves he actually selected. In most of the states, the player chooses the action with the higher value considering its acquired experience. But, as we can from our second move, the player chose to explore, by preferring an action e over an action ($e*$) with higher value.

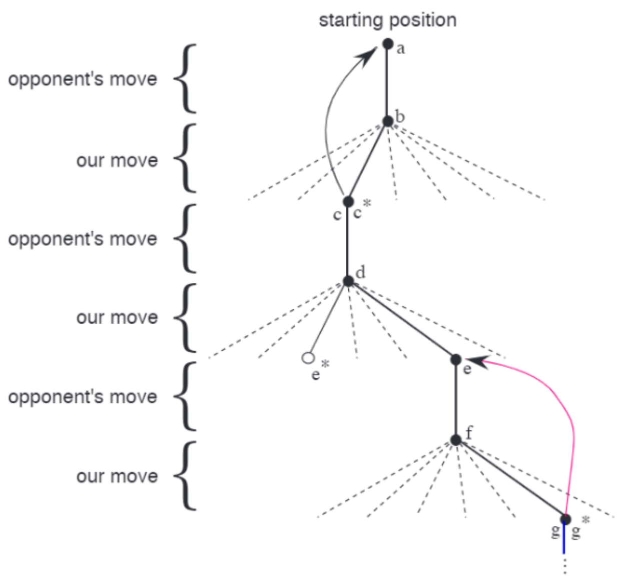


Figure 2.2: A sequence of moves made and considered in tic-tac-toe. [56]

Digging deeper into policies, we can divide them into two types:

- Deterministic: a policy at a given state s will always return the same action a.
- Stochastic: It gives a distribution of probability over different actions.

2.2.0.3 Reward functions

As previously stated, the agent's final goal is to maximize the cumulative reward in the long run. Given a sequence of rewards after time step t $R_{t+1} + R_{t+2} + R_{t+3} + ...$, we seek to maximize the expected reward. Formally, this expected return, denoted as G_t, can be defined as a specific function of the reward sequence. In the simplest scenario it can be defined as the sum of the rewards, as in Eq. 2.1,

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.1}$$

where T is the terminal state. This terminal state exists in tasks where the agent-environment interaction naturally divides into identifiable sub-sequences, which we call **episodes**. and where the number of time steps is finite. These tasks are called **episodic tasks**. On the other hand, there are tasks where the agent–environment interaction never reaches a final state, which we call **continuous tasks**. In this case, the final step would be $T = \infty$, and subsequently; subsequently, the return can also be infinite.

To prevent the total reward from going to infinity, there is a need to use a **discount factor**, γ, where $0 \leq \gamma \leq 1$. This factor defines how important future rewards are to the current state compared to the immediate reward. A reward R that occurs k steps after the current state, is multiplied by γ^k to describe its importance to the current state. If $\gamma = 0$, the agent only maximizes the immediate reward, R_{t+1}. As γ approaches 1, the agent takes future rewards into account more strongly.

In general, both in continuous and episodic task, the expected return, denoted as G_t can be described as Eq. 2.2, where the agent tries to select actions so that the sum of the rewards it receives over the future is maximized.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

2.2.0.4 Markov Decision Process

RL problems are commonly formulated as a **Markov Decision Process** (MDP) [3] in the form of a tuple (S, A, $P_a(s_t, s_{t+1})$, $Ra(s_t, s_{t+1}), \gamma$) where:

- S is a finite set of states;
- A is a finite set of actions;
- $P_a(s_t, s_{t+1})$ is the probability to move to state s_{t+1} when choosing action a in state s_t;
- $R_a(s_t, s_{t+1})$ is the reward r_{t+1} for passing from state s_t to state s_{t+1} after taking action a;
- γ is the discount factor.

A RL task is called a MDP if it is a sequence of states that satisfy the Markov property. This property designs a state signal where the next state s_{t+1} only depends on the immediately preceding state s_t and action a_t, and not on the previous ones, but at the same time is successfully informative. Thus, a state signal that contains all the relevant information to make a decision is said to have the Markov property.

In a finite MDP, where the state and action spaces are finite, given any state s and action a, the probability of each possible pair of next state *s'* and reward *r*, can be denoted by the Eq. 2.3:

$$p(s', r|s, a) = Pr\{R_{t+1} = r, S_{t+1} = s'|S_t = s, A_t = a\} \tag{2.3}$$

From the probability specified by Eq. 2.3, we can compute anything else we might want to know about the environment, such as:

- the expected rewards for a action-state pairs:

$$\begin{aligned} r(s,a) &= \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \\ &= \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a); \end{aligned} \tag{2.4}$$

- state-transition probabilities (probability to transit to a state):

$$p(s'|s,a) = Pr[S_{t+1} = s'|S_t = s, A_t = a] \\ = \sum_{r \in R} p(s', r|s,a); \tag{2.5}$$

- the expected rewards for state–action–next-state triples:

$$r(s,a,s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] \\ = \frac{\sum_{r \in R} r p(s',r|s,a)}{p(s'|s,a)}. \tag{2.6}$$

In these equations, expectation $\mathbb{E}[\cdot]$ is used because the environment's transitions be stochastic.

2.2.0.5 Value functions

The value function, denoted as $v_\pi(s)$, depends on the policy used by the agent to decide on the actions to perform. Thus, to estimate expected future rewards (or expected returns) and achieve the best global solution starting in state s and following a policy π, the value function of a state s, under policy π can be defined as Eq. 2.7,

$$v_\pi(s) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \tag{2.7}$$

where $\mathbb{E}_\pi[\cdot]$ refers to the expected value given that the agent follows π. With Eq. 2.7, we can note that the value-function of the terminal state, if any, is always zero.

Similarly, to estimate the expected return starting from s, taking the action a, and following the policy π, the value of taking the action a at the state s and under the policy π, denoted as q_π, can be defined as Eq. 2.8.

$$q_\pi(s,a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{2.8}$$

Both state-value function and action-value function can be defined using recursive relationships, as in Equations 2.9 and 2.10, respectively. These equations are called the **Bellman Equations** and they are a crucial property of RL.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V_\pi(s')] \tag{2.9}$$

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_a \pi(a|s) q_\pi(s',a')) \tag{2.10}$$

Bellman equations are recursive because they expresses how the state's value can be found using its successor discounted states' values. Fig. 2.3 translates the update or *backup* operations carried by RL techniques, where a state s can get information from its successor states. Starting from a state (or state-action pair) we can look ahead to the possible state-action pairs (or state) and its possible subsequent states (or state-action pairs). The Bellman equations 2.9 and 2.10 takes the average of all the possible outcomes, weighting them according to their probability of occurring.

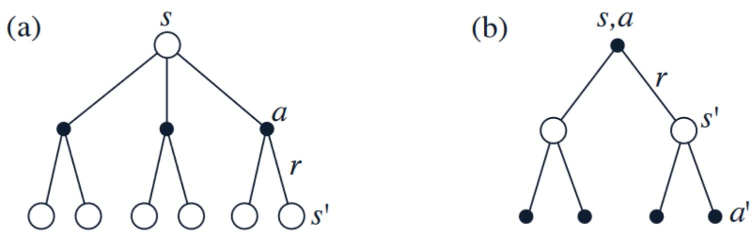


Figure 2.3: Backup diagrams for (a) $v_\pi(s)$ and (b) $q_\pi(s,a)$. [56]

2.2.0.6 Optimal value and action-value functions

Among all possible policies, an RL agent needs to find the one that results in the optimal value function, or in other words, that maximizes the expected return for all states. This policy is called the **optimal policy**. We must note that there may be several optimal policies. However, they all lead to the same state-value function, $v_*(s)$. This value function is called **optimal state-value** function, is unique, and is defined as follows:

$$v_*(s) = max_\pi v_\pi(s) \tag{2.11}$$

Similarly, optimal policies also share the same optimal action-value function, denoted as q_*:

$$q_*(s,a) = max_\pi q_\pi(s,a) \tag{2.12}$$

Both Equations 2.11 and 2.12 can be written as the Bellman Optimality Equations, using the condition of the Bellman Equation:

$$\begin{aligned} v*(s) &= max_{a(s)} q_{\pi*}(s,a) \\ &= max_a \sum_{s',r} p(s',r|s,a)[r +_* (s')] \end{aligned} \tag{2.13}$$

$$q*(s,a) = \sum_{s',r} p(s',r|s,a)[r +_{a'} q_*(s',a')] \tag{2.14}$$

To find an optimal policy we maximize over Eq. 2.14, selecting with probability 1 the action a that gives us the maximum state-action value function ($q*(s,a)$) given a state s. The policy that the optimal state-action value function follows is an optimal policy.

Differently from the backup diagrams for v_π and q_π, in the Bellman Optimality equation's diagrams (2.4), the maximum state-action pair (or state) is chosen over the expected value given some policy.

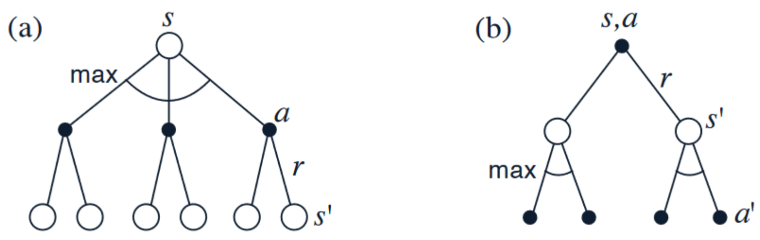


Figure 2.4: Backup diagrams for (a) v_* and (b) q_*. [56]

The Bellman Equation finds the optimal value and action-value functions in most scenarios. However, when we do not know the probability of each possibility, we can not solve the problem. In these scenarios we use Monte Carlo (Monte Carlo (MC)) prediction.

With the MC approach, the value or action-value functions are estimated from experience. It's method relies on, for a policy π, running several episodes with this policy and then calculate the average return for every state or state-action pair. However, MC methods also have a disadvantage: the strategy can only be updated after the entire episode.

2.2.1 Q-learning

According to *Watkins and Dayan* [62], Q-learning is a tabular **model-free** RL method, which main objective is to find the best sequence of actions given its current state s_t, i.e. the sequence of actions that will return the maximum reward. It is tabular because it stores all the Q-values in a table called **Q-table**, where each row represents a state and each column an action. The entry for the table corresponding to state s and action a is denoted Q(s,a).

Q-learning uses **Temporal Differences** (Temporal Differences (TD)) to estimate the value of a state-action pair. Differently from MC methods, TD algorithms update the Q-value at each time step, which takes less time. The basic update rule Q-learning follows is represented in Eq. 2.15, where TDerror measures the difference between the new Q-value and the old Q-value.

$$\begin{aligned} Q(s_t, a_t) &\leftarrow OldQvalue + TDerror \\ &\leftarrow OldQvalue + (newQvalue - oldQvalue) \end{aligned} \tag{2.15}$$

Q-Learning comes up with rules of its own and may not follow the policy π to update the current state's value. Instead, it chooses the greedy action ($max_a[Q(S_{t+1}, a)]$), i.e. the action that gives the maximum reward. This means that there is no actual need for a policy, hence this method is called **off-policy**.

2.2.2 Q-learning Process

Q-learning meets the conditions of the Bellman Equation, as the maximum future reward for a specific action, is the sum of the immediate reward and the maximum reward for taking the next action. This translates to the new Q-value in Eq. 2.15 being equal to $[r_t + \gamma \max_a [Q(s_{t+1}, a)]$. Hence, given a set of states S, a set of actions A, a random policy π, and any initial estimate Q_0, the update rule at each iteration is defines as in Eq. 2.16.

$$Q_{new}(s_t, a_t) \leftarrow Q(s_t, s_t) + \alpha[r_t + \gamma \max_a [Q(s_{t+1}, a)] - Q(s_t, a_t)] \tag{2.16}$$

In Eq. 2.16, the *TDerror* is multiplied by α or learning rate ($0 < \alpha <= 1$), which defines how much the new value overrides the old value. If 0, the agent exclusively exploits prior knowledge, while if 1 the agent considers only explores, ignoring prior knowledge.

The Q-values are iteratively updated using the Bellman Optimality Equation (Eq. 2.12) until the the Q-value function converges to the optimal one, q_*. To converge the Q-value to an optimal one, for a given state-action pair, the loss between the Q-value and the optimal Q-value ($q_*(s,a) - q(s,a)$) are compared at each iteration. If it encounters the same state-action pair later, it updates the Q-value to reduce the loss.

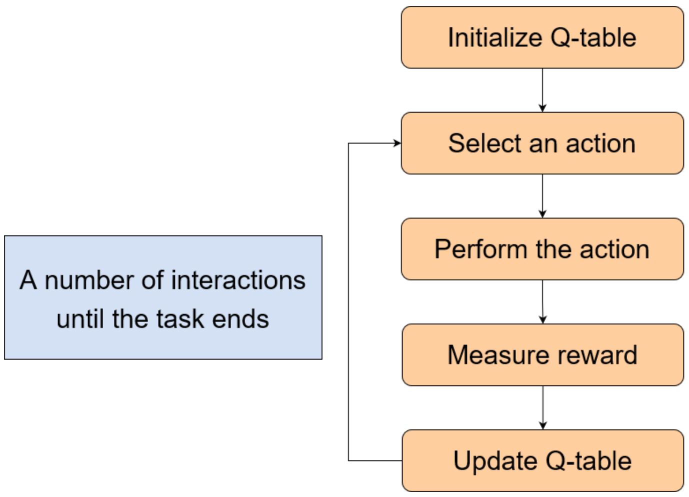


Figure 2.5: Q-learning algorithm flow using the Q-table

At each episode, the Q-table is initialized with null values for all action-state pairs. The agent starts to choose an action a in state s based on the Q-table and performs it. Then, updates the Q-value with the obtained reward using the Eq. 2.16. These steps run in loop until the episode ends or the training is stopped. However, in the beginning, it's difficult for the agent to choose actions with no knowledge about the environment. that's where it comes the ε-greedy policy. The epsilon rate starts with a higher value, and the agent explores the environment and randomly choose actions. As the agent explores the environment this rate decays across the learning process, and it starts to exploit the environment, becoming more confident in estimating the Q-values. The idea is that at the beginning and the agent learns enough information about the environment. Once the agent has the information, it must have an optimal interaction with the environment by exploiting its knowledge. This flow of the Q-learning algorithm is depicted in Fig. 2.5.

2.2.3 Limitations and challenges

RL algorithms can be very useful when the only way to gather information about the environment is through interaction. However, when the decision problems become too complex and the knowledge space becomes high-dimensional, they are no longer feasible solutions. It becomes challenging to learn all combinations of state-action returns and the computation cost get out of control. To overcome this challenge, there are techniques that extend Q-learning and try to use approximate the Q-function from existing knowledge instead of learning the whole Q-table.

Another limitation of RL is that to train an agent, a reward function corresponding to the agent's goal must be well-defined at the very beginning. For complex problems it can be hard to define such function and incorrect rewards may result in bad and unsafe performance, so an agent that learns the reward by observing human's behavior is preferred instead.

2.2.4 Summary

To sum up the review of the RL and Q-learning concepts, the main information we need to retain is the following:

- Differently from supervised learning and unsupervised learning ML paradigms, where the algorithms train with labelled and unlabelled data, respectively, RL learns to react to an environment;
- RL aims to teach an agent to make a sequence of decisions to achieve a goal in an uncertain environment;
- The **agent** is the learner or the decision-maker and the **environment** are the surroundings with which the agent interacts.
- A **state** is a combination of observable variables or features on the environment. The decision an agent makes is called an **action**.
- RL is based on a trial-and-error method, where a favorable output is 'reinforced', and a non-favorable output is 'punished';
- In RL there is a **closed-loop** interaction between the agent and the environment, as the actions taken by the agent based on the current state of the environment influence the next state. At each action made by the agent, the environment responds with a reward and the next state;
- In order to effectively find the optimal sequence of actions, RL must balance between **exploration** and **exploitation**. The agent must exploit the actions that maximize the reward, but also explore new actions that may lead to future better rewards;

- A **policy** is a set of rules and a strategy that the agent uses to make decisions on the actions;
- The **reward function** gives feedback about the current state, by accumulating the future expected rewards. The agent aims to maximize the cumulative reward;
- The **discount factor** (γ) is a parameter used in the reward function that weights the immediate rewards over the future ones;
- When the current state retains sufficient information about the past experience to make successful decisions, the system is is labelled as a **MDP**, lying under the **Markov Property**;
- A **value function** evaluates a state (or action-state pair), mapping it into the corresponding expected reward;
- Among all possible value functions for a state/state-action pair, there is one that leads to the maximum return. This function is called the **optimal value function**;
- For a policy to become the optimal policy, it must satisfy the condition assumed by the Bellman Optimality equation (Equations 2.11 and 2.12);
- The agent estimates the values for states or action-state pairs through experience. One of the most common methods for estimating these values is the Q-value, which relies on doing the average of n observed rewards for a given action a in a state s;
- **Q-learning** is a RL method, where the goal is to learn the best next action given the current state;
- Q-learning uses a **Q-table** to keep the values returned from Eq. 2.16 for each state-action pair. This table allows the agent to calculate the maximum expected future rewards for an action at each state;
- In some decision problems, it is unfeasible to learn all possible sate-action pairs accurately with RL, so function approximators are used instead.

2.3 Deep Reinforcement Learning

To find the optimal policy in a RL task, the agent needs to know the value of each state/state-action pair, in an environment. However, in problems where the search space is too large, tabular methods, such as Q-learning, are insufficient and inadequate. Since the values only have relative importance, that is, the agent only needs to know the relation between the values, RL methods can be extended to new approaches based on approximation and generalization. The aim is to find the value of a state or an action by generalising the estimation of the value of similar and already experienced observations, or in other words, states with similar features, while preserving the relative importance. These approaches save computation time and memory space, as they never find the true value of a state, but an approximation of it instead. The methods that estimate these approximations are called **Function Approximators**.

Deep Learning (Deep Learning (DL)) is a subset of ML whose algorithms use **Deep Neural Networks** as function approximators to solve complex and large-dataset ML tasks with higher accuracy. A Deep Neural Network (DNN) is a set of hierarchical layers used to model high-level abstractions in data, where the knowledge gained from each layer is used as input to the next layer of the hierarchy [51, 24, 11]. The first layer is fed with raw data and as the level gets higher, the concept gets more abstracted. This DNN allows a DL agent to understand and represent a complex concept, discovering patterns and intricate structure in the dataset.

The difference between DL and ML is in the learning process. ML algorithms usually require structured data and are dependent of humans to determine the set of features that differentiate data inputs. DL, on the other hand, attemps to mimic humans when trying to gain knowledge. This subset of ML has the capability of automatically extracting features from the data, enabling the agent to learn from very large datasets faster and easier, eliminating some of the human intervention.

The increasing attraction in Deep learning led researchers to discover how deep neural networks can be useful to learn representations in RL problems [39, 35, 4, 26, 32, 38]. This combination of function approximation and target optimization, that is, Deep Learning and Reinforcement Learning, is called **Deep Reinforcement Learning**.

Similarly to RL, in DRL there is also a closed-loop agent-environment interaction. However, the policy that chooses the action to be taken is represented by a DNN, as Fig. 2.6 portrays.

2.3.1 Deep Q-Network

In 2015, DeepMind researchers developed a novel DRL technique that sucessfully estimates the Q-value of all possible actions for a given state using deep neural networks[36]. This method is called **Deep-Q-Network** or Deep Q-Learning and has become one of the most-popular forms of Q-Learning.

Fig. 2.7 depicts the difference between Q-learning and DQN. While Q-learning uses a Q-table to store every value of a state-action pair (Fig. 2.7a), DQN uses a DNN to approximate the Q-value function (Fig. 2.7b). A state is given as the input to the network, and only a single forward pass through the network is needed to output the Q-value of all possible actions the agent can take in

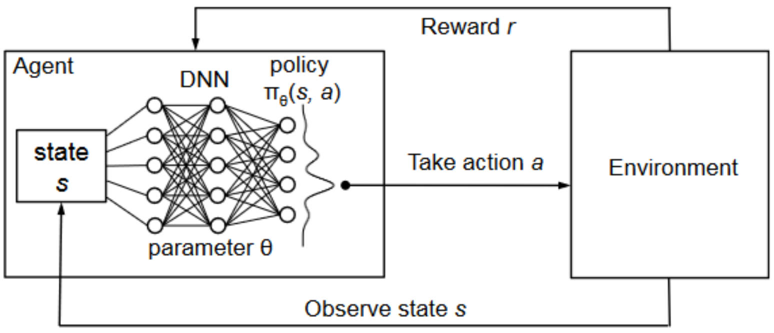
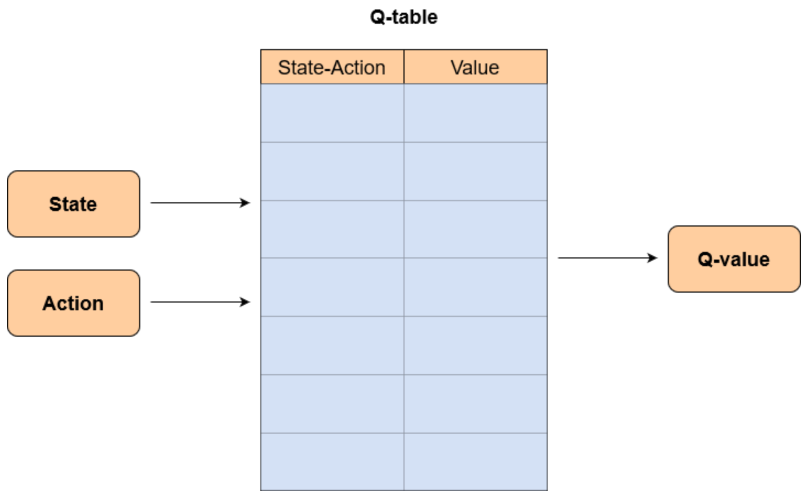


Figure 2.6: Agent-Environment interaction interface with policy represented via DNN [30]

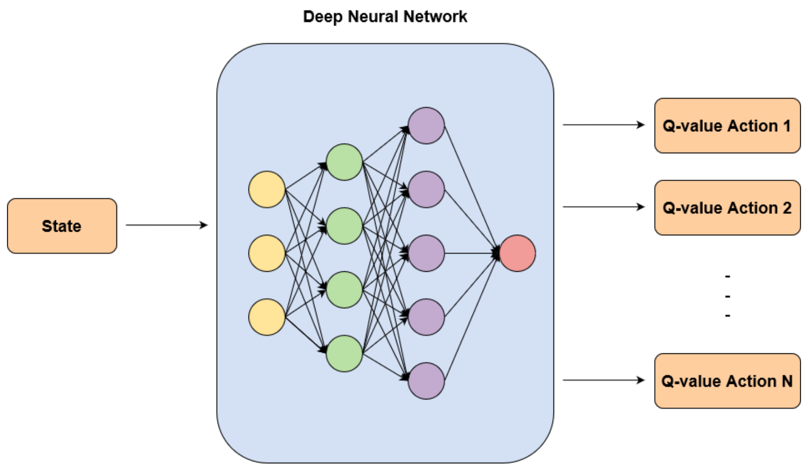
that state with only. Similarly to RL algorithms, DQN seeks to maximize the expected value of the cumulative rewards. Thus, the next action corresponds to the action with the maximum output of the network, that is, the action that leads to higher rewards.

A neural network requires the training data to be independent and have an identical distribution, otherwise it may become unstable and may diverge. However, combining it with model-free RL algorithms provokes some issues. If we take the Q-value update equation (Eq. 2.16), we can argue that the target $[R_{t+1} + \gamma max_a[Q(S_{t+1}, a)]$ value is updated after every iteration, which causes a high correlation between consecutive observations. To break these strong correlations between subsequent iterations, DQN provides a solution called **Experience Replay**. Rather than using the most recent experience to update the Q-network, the system stores the agent's past experiences t into a dataset D, where each experience is a four-tuple, (s_t, a_t, r_t, s_{t+1}), and extract a random mini-batch of these experiences to update the Q-values. The agent then selects an action according to an ε-greedy policy. This approach of taking fully random samples from the system's memory to use in mini-batch updates makes the network learn some experiences multiple times, recall rare observations and, in general converge to the optimal value function.

The stability issues on the network may not be only created by correlation between sequential observations. If we have another look at Eq. 2.16, we can note that $Q(s_t, a_t)$ depends on $Q(S_{t+1}, a)$. This means that slight updates on the Q-value make frequent, and sometimes significant, changes on the policy. This creates instability in the policy and a high correlation between the action-value and the target value. The solution DeepMind [36] proposes is to duplicate the Q-network, creating a copy called **target network**. The difference between the two copies are their parameters (weights) and how they are updated. While Q-network's parameters θ are trained, target network's parameters θ^T are periodically synchronized with the Q-network's parameters. The target is then calculated with the more stable value of the target-network. The idea is that using the target network's values to train the Q-network will decorrelate the action-value with the target value and improve the stability of the network. During the training process, the network tries to optimize the weights θ in order to minimize the loss function. This loss function, $L_i(\theta_i)$, describe in Eq. 2.17, is represented as a squared error of the TD error (predicted and actual Q-values), on time-step t. In



(a) Q-learning using a Q-table to store the Q-value for all state-action pair



(b) Deep Q-Network using a Deep Neural Network to approximate the Q-value of all actions for each state

Figure 2.7: Difference between Q-Learning and Deep Q-Network

DQN, the predicted Q-value is equivalent to the Q-value computed by the target network (target Q-value).

$$L_i(\theta_i) = E_{(S_t,A_t,R_t,S_{t+1})}[(R_t + \gamma max_a[Q(S_{t+1},a_{t+1}|\theta_t^T)] - Q(S_t,a_t|\theta_t))^2] \tag{2.17}$$

DQN introduces two new concepts, batch and epoch. They are often confused, since a batch defines the number of samples in the training database to pass through before updating the model parameters, and the other defines the number of times the training algorithm will pass through the entire input dataset. At the end of a batch, the predictions given my the model are compared to the expected values and an error is calculated. In one epoch, the internal parameters of the model have been updated for each sample in the training dataset.

2.3.2 Policy Gradient methods

Modern Deep Reinforcement Learning algorithms have two most popular building-blocks: Deep Learning version of Q-learning, DQN, and Policy Gradients. DQN's training procedure is based on evaluating each possible action, and select one according to their quality, which requires each state and action to be experimented in a sufficient number. In DQN the learned parameters θ from the training process are used to compute Q-Values, which are then used to decide on the policy π that selects an actions. **Policy Gradient** (Policy Gradient (PG)) algorithms, on the other hand, learn by not estimating the value of each possible move — but by simply evaluating which action should it prefer. These algorithms skip the Q-values calculation and, using a neural network, models the actions probabilities directly from the parameters θ. This makes PG learning strategy more robust, as it does not need to have complete knowledge of all possible actions.

For each agent-environment interaction, the algorithm needs to update the neural network's parameters (θ) so that the actions that lead to higher rewards will be more selected in the future. This update rule can be generally defined using a *performance measure*, $J(\theta_t)$, at each time step t as:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t), \tag{2.18}$$

where α is the learning rate and the gradient in J needs to be ascent as the goal of the learning process is to maximize the performance measure. This process is repeated until the policy π converges to the optimal policy π^*.

The next step is to find the right definition for J. Recalling that the agent's purpose is to maximize the long-term rewards and that the Q-value is a measure of all expected rewards in an episode, a simple and good performance measurement would be something that depends on Q(s,a). J can be then defined as the total accumulated rewards starting on the initial state s_0 and following policy π with parameters θ. This translates to the sum of the probability to select each action $(\pi_\theta(a|s_0))$ on initial state s_0, multiplied by all Q-Values on that state, which is basically the initial state's value function:

$$\begin{aligned} J(\theta) &\doteq v_{\pi_\theta}(s_0) \\ &\doteq \sum_a \pi_\theta(a|s_0) Q_{\pi_\theta}(s_0, a) \end{aligned} \tag{2.19}$$

Having J, *Sutton et al.* (2000) [57] calculate the gradient of the performance measure as shown in Eq. 2.20.

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \ln \pi_\theta(a|s) Q^\pi(s,a)], \tag{2.20}$$

Policy gradient methods have some advantages over value-based methods, such as DQN. First, the policy is simpler to learn than the Q-function, when the reward function is too complex. PG operates directly in the policy space and only needs to learn the parameters. Also, since Policy

Gradients model probabilities of actions, they have the capability of learning stochastic policies, while value-based methods do not, as they converge to a deterministic policy. However, PG methods have the drawback of usually converge to a local optimal, as it only depends on its performance metric and does have knowledge on the value functions, such as Q-learning. The solution to this issue would be to combine both Policy Gradients and Q-learning.

2.3.2.1 Deep Deterministic Policy Gradient

DDPG is an algorithm that solves the local optimality PG problem by simultaneously learning a policy and a Q-function. As stated by *Fan et al. (2021)*, it is an actor-critic algorithm, meaning that it is composed of two networks: actor and critic. The actor is a policy network that takes the state as input and outputs the action to be taken in a continuous action-space. It generates the action directly rather than a probability distribution over actions, which makes the algorithm deterministic. The other, critic, is a Q-value network that learns the Q-value of each state-action and uses that knowledge to evaluate the action chosen by the actor-network and to update the parameters from both networks. The DDPG network architecture is shown in the Fig. 2.8.

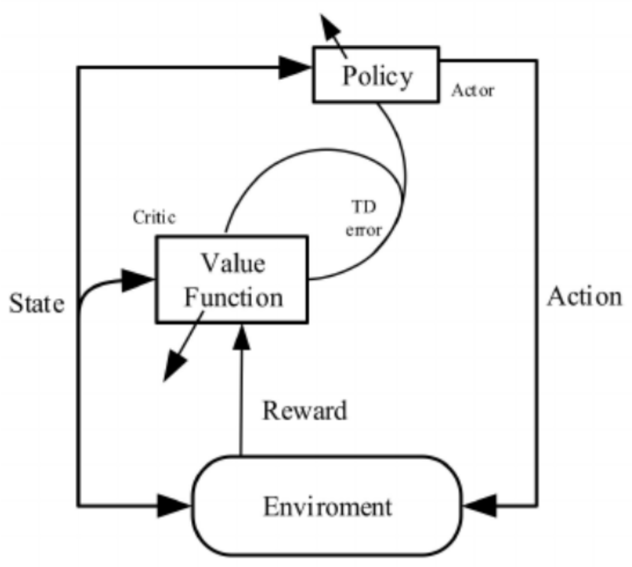


Figure 2.8: Diagram of the DDPG structure. [9]

The critic-network calculates the TD error values and uses them to learn the state-action value function. Being θ^C the weights of the critic-network, its loss function at each time step t can be then defined as:

$$L(\theta^C)_t = [(r + \gamma(Q(s_{t+1}, a_{t+1}|\theta^C)) - Q(s_t, a_t|\theta^C)]^2 \tag{2.21}$$

And the network's optimization goal is to minize this loss function. The update rule of the critic-network weights can be represented as in Eq. 2.18, where θ and α are the weights and the learning rate of its own network, respectively.

The action selected by the actor-network is evaluated of the critic-network. The measure function the network uses to evaluate is computed with the performance measurement defined in Eq. 2.22.

$$J(\theta^A) = Q(s, a|\theta^C)|_{a=\mu(s|\theta^A)} \tag{2.22}$$

The actor-network seeks to learn the optimal policy, so it needs to maximize the performance measurement $J(\theta^A)$. Its update function can be given as follows:

$$\theta^A_{t+1} = \theta^A_t + \alpha_A \nabla_a Q(s_t, a_t|\theta^C_t)|_{a=\mu(s|\theta^A)\nabla_{\theta^A}\mu(s|\theta^A)}, \tag{2.23}$$

where θ^A and α_A are the weights and the learning rate of the actor-network, respectively.

As DDPG uses neural networks, the data needs to be independent and identically distributed. To ensure this, DDPG adopts the same solution as DQN: Experience Replay. At each time-step the actor and critic networks are updated by randomly sampling a mini-batch from the buffer, making the algorithm an off-policy model. To reduce noise during the training phase and avoid the risk of overestimation, DDPG also follows the target network technique from DQN. On both actor and critic networks a copy network is created with different weights. The "regular" network is trained and the target is updated slowly.

2.3.3 Limitations and challenges

The combination of RL and function approximators such as neural networks brings the advantage of solving a wide range of decision making and control tasks that RL could not solve alone. However, as DL needs large amounts of data, application of algorithms that use neural networks have the drawback of data complexity. Even relatively simple problems can require millions of steps to collect data, and complex tasks with high-dimensional samples might need substantially more.

Although neural network algorithms take less time to run tests, the training phase is highly expensive due to the complex data models. Moreover, hardware requirements to ensure efficiency and time consumption decreasing can cause limitations.

2.3.4 Summary

We must keep the following key ideas in mind when discussing DRL and its algorithms DQN and DDPG:

- **Deep Reinforcement Learning** is a subset of ML that combines **Deep Learning** with **Reinforcement Learning**. Deep neural networks of DL are used to learn useful **representation** in RL problems;

- DRL algorithms support noisy, **high-dimensional** and unstructured datasets;

- In DRL there is a closed-loop interaction between the agent and the environment, where the policy is represented via a DNN;
- DQN is a DRL algorithm, developed by DeepMind [36], that approximates Q-values using a DNN. This algorithm is a combination of DL and a variant of Q-learning;
- DQN solves stability issues caused by integrating neural networks in Q-learning with two techniques: experience replay and target network;
- **Experience replay** breaks high correlations between consecutive iterations by extracting random mini-batches from the agent's past experiences at training time. This allows the network to keep remembering rare occurrences and repeat individual experiences;
- A **target network** is a copy of a Q-network whose parameters are periodically synchronized with the Q-network's parameters;
- The purpose of the target network is to use the target network's values to train the Q-network and therefore improve the stability of the network;
- In addiction to value-based methods, such as DQN, DRL also has policy-based methods;
- The main idea of Policy-based methods, or **Policy Gradient** methods, is to learn optimal policy directly from the experiences;
- PG algorithms adjust the policy's parameters so that the performance measurement is maximized. This makes the learning process more robust;
- The performance measure can be defined as the expected reward in the initial state of an episode;
- PG algorithms usually converge to a local optimal, as they do not have knowledge on the value functions;
- **DDPG** is an actor-critic method that combines Policy Gradients and Q-learning techniques;
- The **actor-network** directly generates an action instead of a probability for each action;
- The **critic-network** learns the value function of each state-action pair to evaluate the action given by the actor;
- The critic-network aims to minimize the loss function and the actor-networks to maximize the performance measurement;
- DDPG adopts two techniques from DQN: experience replay and target network;
- DRL algorithms require large amounts of data and high-dimensional steps of data collection;
- DRL methods also have more training complexity and need more expensive hardware.

2.4 Inverse Reinforcement Learning

Abbeel and Ng [1] explain that the reward function is the "most succinct, robust and transferable definition of the task" in RL. However, for many problems, particularly when studying animal and human behavior, it may be difficult to specify a reward function and how its attributes must be combined and weighted. For example, in a running task, the reward function must trade off attributes such as speed, efficiency, stability and strength, in a way that the optimal policy generates a successful running.

IRL is a framework which can efficiently solve this conflict of RL. *Russell* [52] describes IRL as a process of deriving a reward function considering:

- an agent's behaviour over time under different conditions;
- the inputs given by the agent's sensors;
- a model of the environment, if applicable.

Essentially, in IRL, the expert performs a task the agent is trying to learn, the agent then perceives the trajectories (sequence of states) and actions (or the policy) of the expert, and learns a reward function that best explains the expert's behavior. More specifically, the agent tries to find a reward function that the behavior of the expert is optimal for. Once the reward function is learnt, the agent finds a behavior that is optimal with respect to it. Fig. 2.9 illustrates how IRL inverts RL problems.

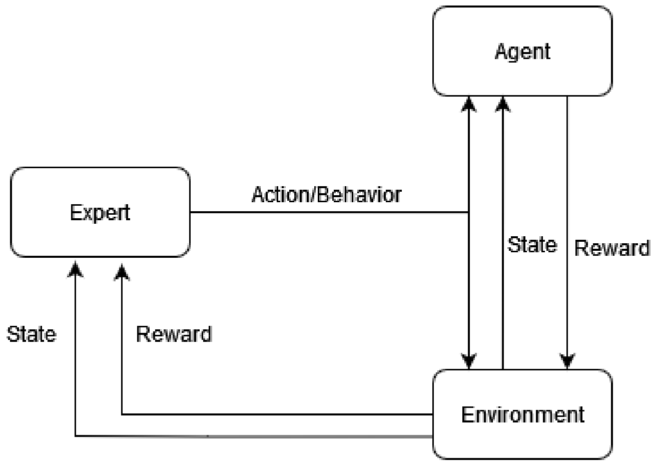


Figure 2.9: Architecture of a IRL problem

IRL is an indirect approach of Apprenticeship Learning (Apprenticeship Learning (AL)) (or learning from demonstration) – learning by observing an expert's behavior. Direct approaches aim to learn the policy as a mapping from states to actions applying a supervised learning method and using a measurement that compares the expert's policy and the policy chosen. In these methods the expert tends to avoid some states, which results difficulties in learning a good policy for such states. In contrast, an indirect method assumes that the expert is doing the optimal behavior. In

IRL, the dynamics of the environment are known (might be unrelated to the expert's actions), but the reward function is not given.

2.4.1 Markov Decision Process and reward functions

In 2004, *Abbeel and Ng* [1] developed a novel algorithm that which generated policy performs almost as well as the expert. For this algorithm, the authors modelled an IRL environment as a finite-state MDP without a reward function, denoted as MDP\R, and in the form of a five-tuple (S, A, T, γ, D) where:

- S is a finite set of states;
- A is a finite set of actions;
- $P_a(s_t, s_{t+1})$ is the probability to move to state s_{t+1} when choosing action a in state s_t;
- γ is the discount factor in the range of [0,1];
- D is the initial-state (S_0) distribution.

Given a vector of features ϕ that represents the different attributes that should be combined in the reward function, and an unknown vector w^* expressing the relative weighting values between those parameters, the authors consider that there is a feature's vector over each state such that $\phi : S \longrightarrow [0,1]^k$ and a true reward function:

$$R^*(s) = w^* \cdot \phi(s), \tag{2.24}$$

where $||w^*||_1 <= 1$ to assure that rewards range between 0 and 1.

In IRL tasks, a policy π maps states to the probability to take each action, and its value can be defined as in Eq. 2.25.

$$\begin{aligned} E_{s_0 \sim D}[V^{\pi}(s_0)] &= \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi] \\ &= \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t w \cdot \phi(s_t)|\pi] \\ &= w \cdot \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \phi(s_t)|\pi] \end{aligned} \tag{2.25}$$

In this equation, the expectation refers to the random sequence of states delineated by starting from the initial state distribution, and taking actions following π.

Defining the feature expectations given policy π denoted as $\mu(\pi)$, where each accumulated vector of features is discounted by the discount factor γ, by Eq. 2.26:

$$\mu(\pi) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \phi(s_t)|\pi] \in \mathbb{R} \tag{2.26}$$

We can conclude that the policy π reformulated as:

$$E_{s_0 \sim D}[V^{\pi}(s_0)] = w \cdot \mu(\pi) \tag{2.27}$$

If we analyse the reward function R from 2.24 we can state that it has a linear relation with features ϕ. Hence, the features expectations $\mu(\pi)$ (Eq. 2.26 will directly translate the expected accumulated discounted rewards obtained after taking actions given the policy π.

In *Abbeel and Ng*'s [1] algorithm, the agent observes the trajectories given by the expert, starting from $s_0 \sim D$ and taking actions following policy π_E. Having an *a priori* knowledge of a MDP\R tuple, a feature vector ϕ and the expert's feature expectations μ_E, the goal is to find a policy $\pi \in \Pi$ whose behavior is similar to the expert's behavior, on the unknown reward function (Eq. 2.24). To achieve this, we need to find a policy $\tilde{\pi}$ such that:

$$|\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi_E] - \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(s_t)|\tilde{\pi}]| \leq \varepsilon \tag{2.28}$$

Following the 2.27, this inequality can be written as:

$$|w^T \mu(\tilde{\pi}) - w^T \mu(\tilde{\pi}_E)| \leq \varepsilon \tag{2.29}$$

Then the IRL problem translates into:

$$||\mu(\pi) - \mu_E|| \leq \varepsilon, \tag{2.30}$$

where is ε is the maximum possible margin between the two feature expectations.

2.4.2 Limitations and Challenges

Although IRL aims to solve the problem of RL, where it is difficult to manually design a reward function for a task, a new challenge then arises when choosing and designing the task's features, which may also be just as complicated. Moreover, the learned reward function may not entirely coincide with the real reward function. Every observed behavior is usually considered optimal by IRL methods. An assumption of this magnitude could return a reward function that does not best explain the expert's behavior.

Another difficulty of IRL is that it is an ill-posed problem, i.e. the optimal solution is not unique for most observations. If R is the fitting reward function, then any xR for $x \geq 0$ is also a reward function that explains the expert's policy. This is caused when the input is a finite and small set of behaviors and many reward functions can return policies that explain the observed trajectories. One solution would be to combine IRL with indirect methods, such as gradient methods, as *Neu & Szepesvári* [42] propose. *Jaynes* [21] also comes up with a solution to avoid the bias introduced to the reward function by the maximum margin that can cause the ambiguity. The author's solution adopts the maximum entropy principle to get a distribution over behaviors.

Generalization is another problem of IRL algorithms. Observed trajectories usually represent a subset of the state and action space. The agent must then estimate the values of new states and actions based on the knowledge of that fraction of information. Hence, the challenge is to generalize correctly reward functions that reflect the expert's goals.

Complexity in solving IRL problems tend to grow disproportionately with the size of the problem. The expert must demonstrate more trajectories as the problem size increases so the training process has sufficient coverage of the task.

2.4.3 Summary

- IRL avoids having to manually design a reward function and combine its attributes, which is the main challenge of RL;
- In IRL, the goal is to learn a reward function that recovers the expert's behavior;
- An expert is someone who is performing the task the agent tries to learn;
- IRL is a technique of AL or Learning from Demonstration – the agent observes the expert's trajectories and assumes the trajectories are optimal;
- The dynamic environments are known, but the reward function is not given;
- The system is an MDP without a reward function;
- When applying an IRL approach, it is essential to previously take into account the following limitations: a good feature selection, ambiguity, generalisation, disproportional complexity growth and data availability.

2.5 Chapter Synopsis

ML is a subfield of AI that allows computers to automatically make predictions from observed patterns and relationships in a given dataset. It emerges in manufacturing industries with the need for intelligent robots with good collaboration and decision-making capabilities.

There are three main types of ML: supervised learning, unsupervised learning and RL. The latter is the most used technique in robotics in the scope of manufacturing industries. In **RL**, a robot learns to perform a given task based on received feedback, called reward. The aim is to **maximise overall rewards**, and to do so, there is a closed-loop interaction between an agent and the environment, where at each time step the agent observes the state of the environment, makes a decision over a set of actions, and receives feedback, whether negative or positive. A **state** is a combination of observable variables or features on the environment. The decision an agent makes on the environment is called an **action**.

To achieve its goal, an RL algorithm tries to balance **exploration** and **exploitation**. In other words, it chooses operations that are believe to outcome the maximum expected future reward, but also discover new actions that may potentially lead to a better solution. To do so, the right policy must be chosen. A policy (π) is a probability distribution that maps the perceived state of the environment (which might not be fully observable) into the possible actions when in that state. If the agent uses a greedy policy, it constantly exploits. However, with ε-greedy policy a random action is selected with probability ε, where $0 <= \varepsilon <= 1$.

Reward functions are another important element to a successful learning process in RL problems, accumulating rewards at each time step. To prevent the total reward from going to infinity, we need to multiply the rewards by a discount factor, $\gamma, where 0 <= \gamma <= 1$. A value function aims to achieve the best global reward, by estimating which states (or state-action pairs) may retrieve the highest expected future reward.

A RL problem usually satisfies the Markov Property, i.e., each state of its sequence of states is informative enough to make decisions and move to the next state. These problems are classified as Markov Decision Processes, and from the probability distribution over each pair of next state and reward it is possible to obtain the respective expected future returns and state and state-action-state transition probabilities.

In a RL task, there is at least one policy that will return the maximum expected reward, called optimal policy. To find it, the agent needs to solve the Bellman Optimality Equation for the state-action value function. The optimal policy is the one that for a state choose the action that returns the maximum $(q*(s,a))$.

Q-learning is an RL algorithm which aims to find the sequence of actions that gives the maximum reward, given a state. It is a tabular method, since it uses a table called Q-table to keep all the action-value functions, or Q-values. Instead of finding all Q-values, it estimates them from experience. Its update rule sums the old Q-value with the TDerror, at each time step, where the TDerror is the new Q-value minus the old Q-value (Eq. 2.16.

For problems where the search space is too large, tabular methods, such as Q-learning, may not be appropriate. We then must use function approximators, which rely on finding an approximation of each state instead of its true value, to save computation space and time. This is possible because the values only have relative importance. DL is a subset of AI whose algorithms use **Deep Neural Networks** as function approximators.

To take advantage of learning through experience, but without worrying about space and time limits, DeepMind [36] researchers created a method that combines both RL and DL, called DQN. This technique estimates the Q-value of all possible actions for a given state using deep neural networks. The combination between RL and DL arise strong correlations between subsequent iterations. To overcome this issue, DQN provides a solution called Experience Replay, that stores the agent's past experiences into a dataset, where each experience is a four-tuple, (s_t, a_t, r_t, s_{t+1}), and extract a random mini-batch of these experiences to update the Q-values. Then the agent selects an action according to an ε-greedy policy. To break the strong correlation, DeepMind [36] researchers propose a solution that is based on duplicating the Q-network, creating a copy called target network.

Another group of methods that takes advantage from both RL and DL are Policy-Gradient methods. The main idea is to learn the optimal policy not by estimating the value of each possible move but by evaluating which action should it prefer. Their algorithms adjust the policy's parameters so that the expected reward in the initial state of an episode is maximized. DDPG is an actor-critic method that combines Policy Gradients and Q-learning techniques. The actor-network directly generates an action instead of a probability for each action, and the critic-network learns each Q-value to evaluate the actor's given action; The critic-network aims to minimize the loss function and the actor-networks to maximize the performance measurement.

For many problems it may be difficult to specify a reward function and how its attributes must be combined and weighted. IRL avoids this issue by learning a reward function, unknowable at the outset, that best explains the expert's behavior, when performing the task the agent tries to learn. To do so, the agent must have a prior knowledge about a set of the task's features and constraints. Then, the agent observes the expert's behaviour and assumes it is optimal. When developing an IRL solution, we must take into account the following challenges: a good feature selection, ambiguity, generalisation, disproportional complexity growth and data availability.

After reading this chapter, we were able to conclude what algorithms are more appropriate for certain problems, depending on the problem's complexity, objective, and available data. Q-learning is sufficient for simple problems where the reward function can be well designed from the problem's goal. DL algorithms, such as DQN and DDPG, on the other hand, have a better performance in complex tasks. These algorithms further differ from each other, considering that the DDPG's learning strategy allows its agents to learn from a larger space of actions in less time and with less memory. As for IRL, its approach is more suitable when the problem is simple, but it is difficult to successfully define a reward function with the knowledge we have about the problem.

Chapter 3

Related Work

Several research related to the main topics on this dissertation have been carried out in recent years. This chapter covers a review on state of the art solutions in the field to establish this thesis's context and explore ways others have solved similar problems. First, this chapter includes an analysis on the use of ML algorithms within RL, DRL and IRL sub-fields, and how they perform in the context of collaborative planning and decision-making. After, we review the state-of-the-art of graph structures for task allocation and scheduling.

An intensive search was conducted using online search tools such as *Google Scholar*, *Science Direct*, *Springer Link* and *Scopus*, and selecting relevant keywords for each topic, among them: **graph simulation**; **task allocation**; **robot**; **cooperation**; **machine learning**; **assembly tasks**.

After selecting the most relevant articles, an analysis is made about their objective and the problems they propose to solve, what kind of results they have obtained, in which use cases they have tested, and, finally, limitations and advantages.

At the end of the chapter, we compare each technique's limitations and challenges to further find and justify the technique that best suits for developing the solution.

3.1 Machine Learning approaches

Smart Manufacturing and Industry 4.0 brought the concept of collaborative robots, which have became a crucial tangible technology for improving process efficiency and flexibility and shortening development times in continually changing manufacturing environments [41, 66, 54]. However, to enable robots to effectively cooperate, it is important to embrace new ML technologies to develop intelligent decision-making agents, enabling them to acknowledge and adapt to human or other robot's unpredictable actions [37, 25, 40] while simultaneously reducing an overall production time.

Many research have been conducted in recent years on the use of task planning algorithms to answer identical research questions to those in this dissertation, and in this sub-chapter we present some of them.

3.1.1 Reinforcement Learning approaches

RL is a main focus of ML and it simplifies collaboration and task planning through experience. A balance between exploitation and exploration allows the agent to discover unseen promising behaviours. It brings many benefits to manufacturing environments, specially for complex processes that are impractical to be manually programmed and performed only by humans or only by robots.

Gradually, DRL has become one of the most active research areas in the manufacturing sector due to its success in solving some complex problems without the need of any *prior* knowledge of the environment during the learning phase. Its algorithms are highly similar to humans learning process, since systems use a neural network to learn from interacting with the environment and getting a reward from a state, at the same time as executing actions that change that state [43, 20].

A reinforcement learning method for human-robot collaboration in assembly tasks (2022)

Zhang et al. [68] implement a RL based Human-Robot Collaboration (HRC-RL) framework that aims to optimize the task sequence allocation scheme in complex assembly processes, considering the dynamic change of human states involved in that collaboration. The HRC-RL framework was designed by extending the DDPG algorithm. In this extension, an Agent-Human with similar characteristics to a real human makes decisions through reinforcement learning and reduces the potential errors in human decision-making. The actor-network of each agent processes the current state and outputs their own actions (A_R, A_H). Then, the critic-network evaluates and calculates the value of actions Q_R and Q_H.

As shown in Fig. 3.1, the agent makes decisions based on the difficulty and time each (or both in collaboration) actor needs to complete each task. Thus, the reward function has a direct relationship with these two values.

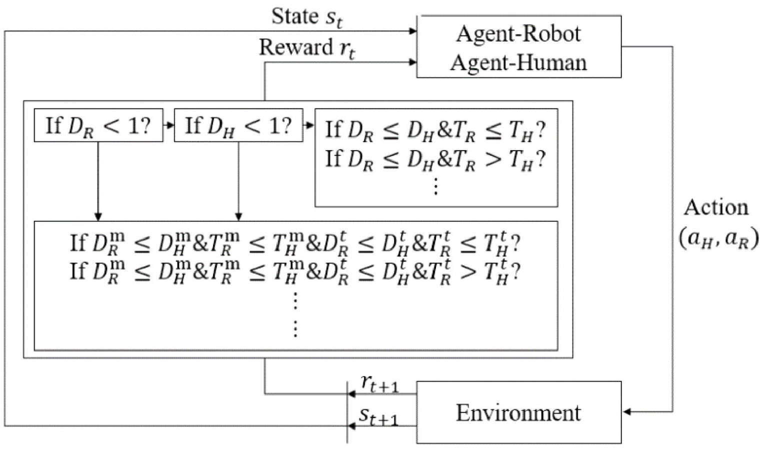
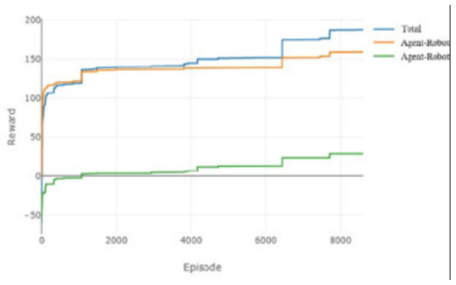
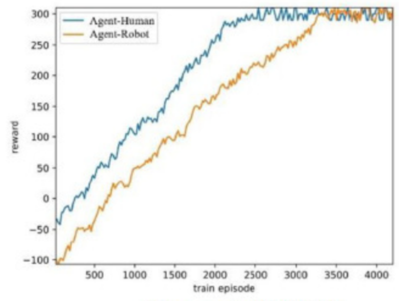

State s_t
Reward r_t
Agent-Robot
Agent-Human
If $D_R < 1$?
If $D_H < 1$?
If $D_R \leq D_H \& T_R \leq T_H$?
If $D_R \leq D_H \& T_R > T_H$?
If $D_R^m \leq D_H^m \& T_R^m \leq T_H^m \& D_R^t \leq D_H^t \& T_R^t \leq T_H^t$?
If $D_R^m \leq D_H^m \& T_R^m \leq T_H^m \& D_R^t \leq D_H^t \& T_R^t > T_H^t$?
Action
(a_H, a_R)
r_{t+1}
s_{t+1}
Environment

Figure 3.1: Decision rules for a collaborative task within RL framework. [68]

Zhang et al. [68] define a reward function for each agent as in Equations 3.1 (human) and 3.2 (robot):



(a) Reward plot of MADDPG [68].



(b) Reward plot of HRC-RL [68].

Figure 3.2: Reward curves of MADDPG and HRC-RL

$$r_H = -\eta + ln(S_E + \varepsilon) + (\frac{10D + T}{T * D}) \tag{3.1}$$

$$r_R = ln(S_E + \varepsilon) + (\frac{10D + T}{T * D}) \tag{3.2}$$

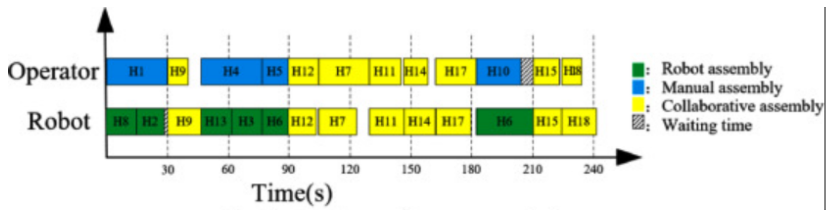
Where η is related to the attenuation factor of human work fatigue affected by time and difficulty. To deal with the dynamic characteristics of a human, the authors add noise to the network. The assembly task is expressed by translating the Bill of Material (Bill of Materials (BoM)) into a vector that corresponds to the structure tree. The operators are guided through this tree that displays the sequence of tasks, to perform corresponding tasks.

The effectiveness of the author's solution is evaluated through experimental analysis of the assembly of an alternator. This alternator includes eighteen structural parts, of which there are seven tasks that can only be completed by the human operator. Seven tasks that are relatively simple and can be performed by both humans and robots individually, such as shaft hole assembly type. The remaining 4 cooperative tasks are selected and completed by dual agents.

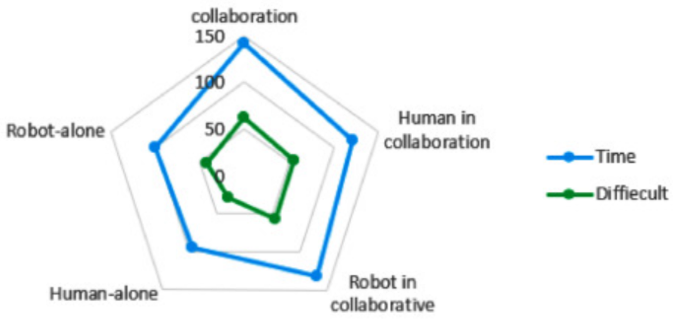
In the training process results, compared with a Multi-agent Deep Deterministic Policy Gradient (MADDPG) algorithm, the proposed method has higher efficiency. From figures 3.2a and 3.2b, the author could conclude that MADDPG focuses more on the total reward than on the collaboration between agents, while with HRC-RL the rewards of both agents have remained close to the maximum value.

The tasks achieved through the collaborative RL network were assigned to the agents in a time sequence, such as the one in Fig. 3.3a. This figure presents the sequence of actions (tasks are represented by H_i, where i is the index of the task), the performer and the time each action takes. Fig. 3.3b shows that, in general, the robot prefers to minimize the overall time of the task, and in collaboration, the total operation time and difficulty of the human are less than than of the robot.

This paper has potential, since it tries to solve the difficulty in task allocation and sequence optimization using reinforcement learning, considering the dynamic assembly process, i.e. the state of the actor, the difficulty the actor has in executing the operation and the corresponding performance time. The human partner and the robot are rewarded for throughout the execution to learn



(a) HRC time sequence for assembling an alternator [68].



(b) Variation in the percentage of difficulty and time [68].

Figure 3.3: Performance of task allocation

how to obtain an efficient task division model. From this paper, one can conclude that, compared with MADDPG, the proposed algorithm has higher efficiency and shows great performance in dynamically dividing human-robot collaborative tasks, replacing human decision making, reducing the manager's workload and avoiding unreasonable sequencing.

Despite its many advantages, this study does not use a graph-based approach to address the optimal collaboration strategies. The computation and update of the actual operating time and resource consumption during the collaborative tasks in real-time are also out of the scope.

Task-level decision-making for dynamic and stochastic human-robot collaboration based on dual agents deep reinforcement learning (2021)

To address the problem of HRC assembly tasks in dynamic and stochastic manufacturing environments, *Liu et al.* [28] propose a novel training approach adapted from the Deep Q-networks method. The method is a Dual-agent Deep Reinforcement Learning (Dual-agent Deep Reinforcement Learning (DADRL)) and aims to teach the robot to be capable of decision-making and task-planning using DRL and taking into account the human partner dynamics.

This work considers both the robot and the human as the agents in the training environment. The robot is trained to learn how to make decisions, while the human agent represents the real human in HRC, showing the dynamic and stochastic collaboration's features.

The training architecture is illustrated in Fig. 3.4. It includes the HRC environment, the agent with a policy and a DQN to select an action, a memory pool to save each iteration's data. One-hot encoding is applied to each observation to increase the mathematical distance between each observation and reduce the disturbance on deep neural network weights when updating parameters.

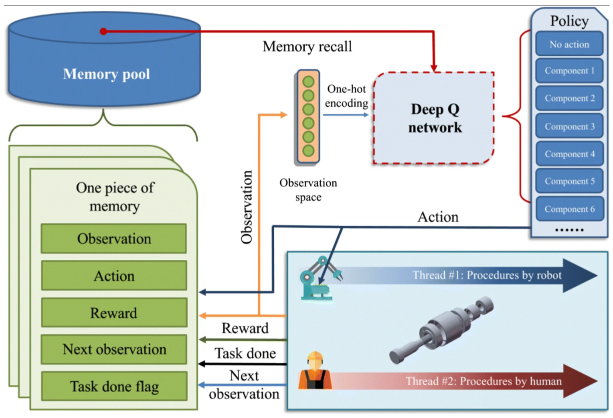


Figure 3.4: System architecture [28].

In the author's method, a task is defined by a set of n sub-procedures. Three buffer sets are also defined as S_W, S_P and S_F, representing the actions waiting to be executed, the ones under performance and those already finished. When the task starts, the two agents can select any of the actions in S_W, and that action is moved from S_W to S_P. Similarly, the procedure is removed from S_P and added to S_F when it is finished. The collaborative task is completed when S_F equals to S.

To evaluate their method, the authors take experiments on two different models to be assembled. One is a bolt-type roller containing six components, while the other is a gear system with eleven sub-components. These products generate a tree file from which the task is loaded to the algorithm, the set and subset of actions, and the constraints among different actions to validate the learning methods. Moreover, the authors also compare the results with and without one-hot encoding.

From the results of the case studies, it is concluded that the proposed solution can drive the robot agent to learn how to make decisions and hence collaborate with the human partner to complete the assembly task even when the human executes unforeseen procedures.

This work contributes to implementing DRL in decision-making of HRC tasks in the manufacturing industry, with an algorithm that extends the tradicional DQN by adding a inherent human agent and two threads for the action occupation and time consumption considering uncertainties of HRC characteristics and the constraints among actions. *Liu et al.* [28] prove that learning from sufficient sampled memories, the agent is able to complete a collaborative task together with a human. Furthermore, combining DRL with HRC in industry brings new opportunities. This solution, however does not approach the advantages that graph structures for collaborative tasks bring, which is one of the main goals of this dissertation.

Optimizing task scheduling in human-robot collaboration with deep multi-agent reinforcement learning (2021)

To optimize the completion time of an HRC assembly task and hence improve the efficiency of manufacturing processes *Yu et al.* [66] develop a Deep-Q-network based MARL (DQN-MARL) algorithm, while conceiving the assembly process as a chessboard game with specific rules determined by the constraints in assembly tasks. This paper aims at contributing to: (1) configuring a classic HRC assembly task as a multi-agent assembly chessboard game with the game rules, and hence simplifying the task scheduling problem; (2) implementing a DQN-MARL algorithm that returns an optimal scheduling policy without domain knowledge or operators intervention; (3) generalizing the proposed solution for other tasks with similar structures.

Fig. 3.5 shows the task scheduling problem HRC being formatted into a Markov game model, in which the robots and the humans are represented as agents and the model inputs the task structure and the agent status as the state and outputs the overall completion time as the reward. The multi-agent algorithm trains the agents in parallell, where at each state s_t, every DQN agent selects its action a_t^n and obtains the corresponding reward. The combination of all selected actions form a joint action a_t, and consequently the current state moves to the next one, according to the transition probability. The agents aim to maximize their own discounted accumulated reward, and for that they choose the action under a policy that corresponds to the probability distribution of the agent's actions.

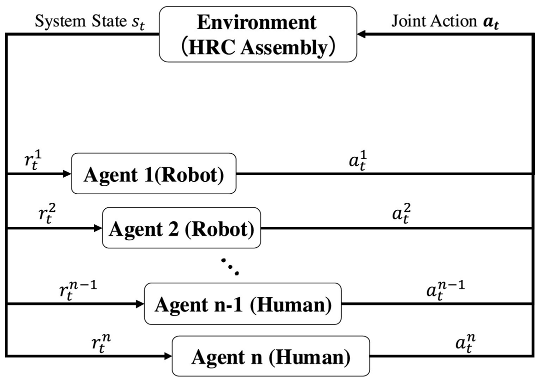


Figure 3.5: MARL framework for task scheduling in HRC assembly. [66].

The authors consider two conclusive measurements that influence the performance of the system: task allocation and task sequencing. To demonstrate the efficiency, an height adjustable standing desk assembly is used with different number of tasks and agents as a case study. The DQN-MARL is compared with the Nash-Q-learning, dynamic programming (DP) and the DQN-based single-agent reinforcement learning (DQN-SARL) method.

Fig. 3.6 shows the comparison between the four algorithms for two products with different number of tasks. Figures 3.6a and 3.6c show the completion time during the training process. When the training phase finishes, the trained policies are used for online simulations to assemble the products, as shown in Figures 3.6b and 3.6d.

The authors also compare the four algorithms with different number of robot agents. From the comparisons, it is demonstrated that more time is needed to find an optimal task scheduling policy with increasing number of agents. However, the proposed DQN-MARL algorithm is performs well when obtaining optimal task scheduling policies for complex tasks. The method can also be generalized to other similar assembly tasks. From this work it was also shown that DQN-based

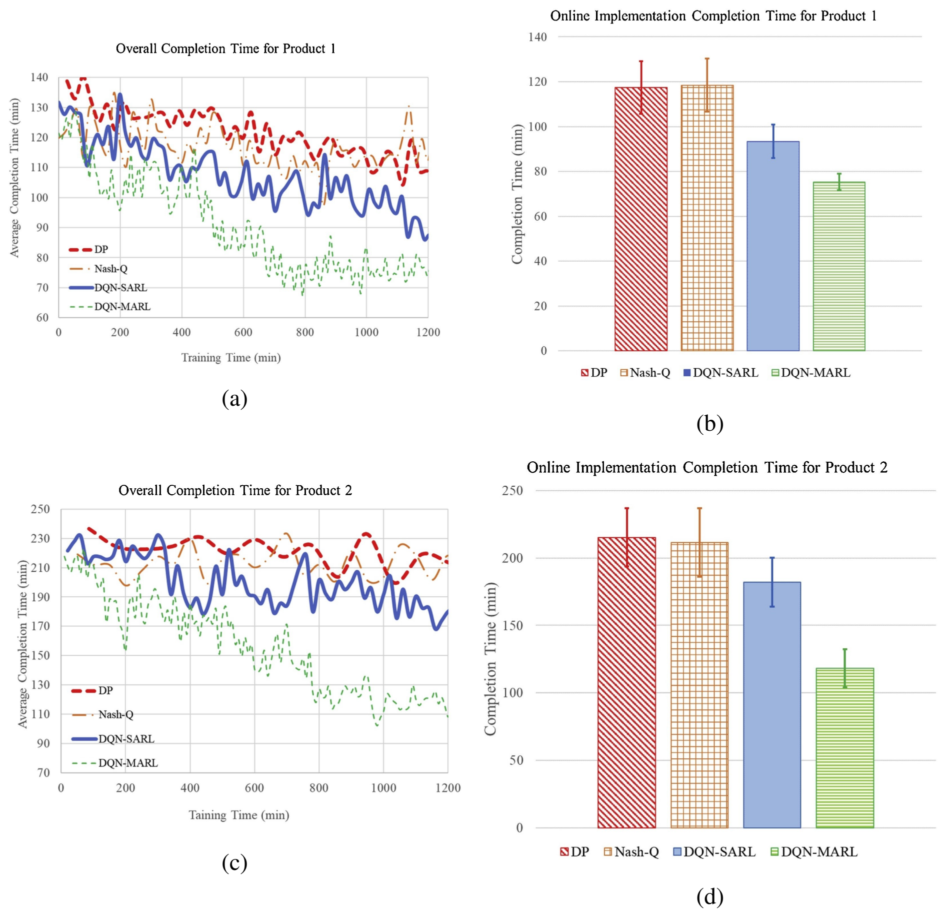


Figure 3.6: Training results. (a) and (c) offline training and corresponding online implementation process (b) and (d) using DP (square dot red line), naive Nash-Q (long dashed orange line) DQN-based SARL (solid blue line) algorithm and DQN-based MARL (dashed green line) algorithm. [66].

algorithms, either single-agent or multi-agent, outperform the naive Nash-Q method and the DP method in terms of both the computational efficiency during the training phase and the optimal completion time during the online simulation.

This paper addresses the problem of task scheduling this dissertation tries to solve using a DRL algorithm to teach a robot to select the best sequence of operations to maximize the working efficiency. However, a limitation of this paper is that they use a chessboard to represent the structure of an assembly task. It is difficult for a human to perceive necessary information of a task on the chessboard, such as the possible alternative sequences, which operations are completed, who performed each operation, and how long they took.

3.1.2 Inverse Reinforcement Learning approaches

IRL resolves the inverse problem of RL in the sense that it avoids a manual design of reward functions and its attributes by automatically generating a reward function through expert's demonstrations (a person who performs the task the agent tries to learn) and knowing the environment [58, 2]. The main focus then shifts to selecting features that represent the task and designing an accurate policy function, which may not correspond to the real one.

IRL method has demonstrated good performance in researches that focus on human-robot collaborative tasks in manufacturing sectors.

Facilitating Human–Robot Collaborative Tasks by Teaching-Learning-Collaboration From Human Demonstrations (2019)

Wang et al. [60] propose a teaching-learning-collaboration (TLC) model that trains a robot to dynamically collaborate and assist its human partner in assembly tasks learning from expert demonstrations. To that end, the human instructs the robot through natural language according to his personal working preferences. Then, the robot uses the Maximum Entropy Inverse Reinforcement Learning (Maximum Entropy (MaxEnt)-IRL) algorithm to learn the optimal assembly plan and update its knowledge on the task.

The architecture of the TLC model developed for human–robot collaborative tasks is presented in the Fig. 3.7, which contains three main parts: human teaching a robot using natural language, robot learning using the MaxEnt-IRL algorithm, and HRC to take decisions and complete the given assembly task.

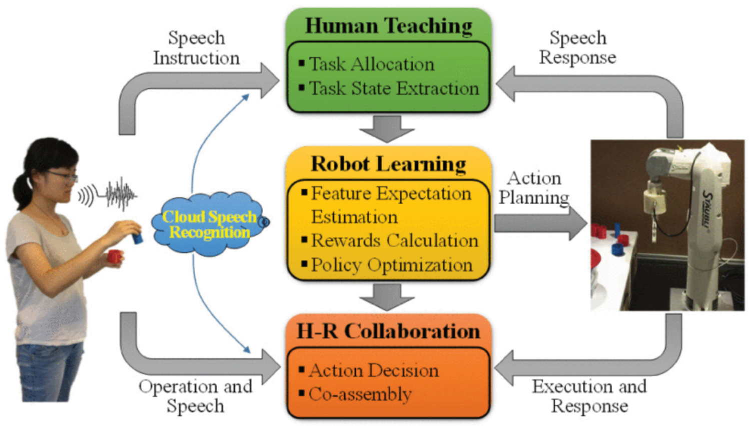


Figure 3.7: Framework of TLC model for HRC [60].

The human starts to efficiently and effectively demonstrate the assembly sequence to the robot using natural language instructions. This instructions are given according to the task customization requirements and the operator's working preferences. At the same time, the model dynamically identifies the operator actions and environment states. And, at the beginning of the learning process, the robot estimates the feature expectations of demonstrations. Moreover, the MaxEnt-IRL algorithm calculates the rewards at each step based on the expectations of the feature. Combining the features and the rewards, the robot is able to generate an optimal policy that is able to plan actions in new dynamic assembly tasks.

As a case study, the team employs a pick-and-place block assembly task. In this assembly there are in total six types of parts for the two agents to pickup in each step. For each assembly of two parts, the relatively small part must be positioned into the relatively large one. Hence, the team has 12 possible actions in each step with all of them independent of each other. As a result, the task presents 12 different state features at each step. The human repeats the task 10 times so the robot can effectively perceive the operator' assembly preferences and product customization.

From the results of the experiments, it could be concluded that the robot is able to assist its human partner in assembling the product according to the human assembly preferences and the product customization requirements, and therefore be effective in human-robot collaboration in assembly tasks. To test the accuracy of the robot collaborative action decision there were used three humans with different preferences for a task, where the robot should make two action decisions to assist the human. Each task was repeated 10 times. The average accuracy of the expected action decision was 95%, which presents a high collaboration capacity between the robot and the human. The team also demonstrates that comparing to the traditional methodologies, the proposed solution demonstrates a competitive efficiency in human–robot collaboration.

Considering the objectives of this thesis, this work also uses machine learning, more specifically IRL, to teach a robot to automatically choose a sequence of operations that completes an assembly task efficiently. However, it does not take the advantages of a graph-based approach for communication between agents and task information useful for the operator, using only speech through natural language.

Inverse Learning of Robot Behavior for Collaborative Planning (2018)

Trivedi and Doshi [59] believe that IRL significantly reduces human effort in manually programming a robot and constantly teaching the robot through behaviours. Therefore, the authors show how IRL can make a robot learn its partner agent's preferences to integrate on decision making and sequence planning, and hence successfully collaborate with the observed agent on a task in the same environment. The team applies MaxEnt to the learning process, nonetheless the solution does not need a specific IRL algorithm, and can integrate other capable methods, yielding similar results.

The team focuses on real-world, "well-defined and repeatable" [59] applications that profit from collaboration, applying their work to a task where a line robot autonomously collaborates with another robot to sort ripe and unripe fruit such as oranges. Fig. 3.8 shows the flow of steps necessary to complete such collaborative task process. A human and a robot agent are deployed in the environment, the firs with the aim of performing the task, and the second with the aim of passively observing the expert's behaviour to learn the expert's preferences on the task. These preferences are then generalized to a collaborative environment, generating a reward function and a plan that will guide the autonomous robot to make correct decisions and planning and effectively collaborate.

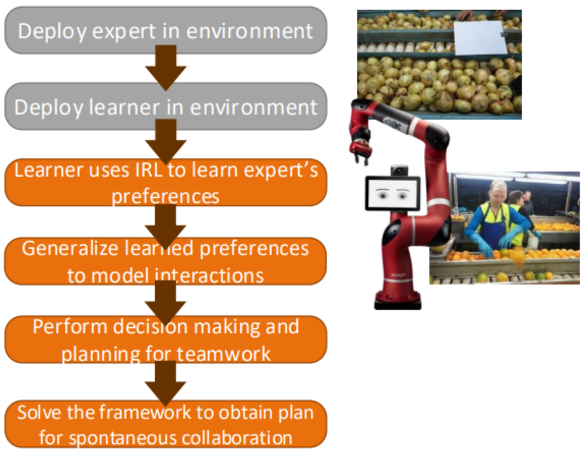
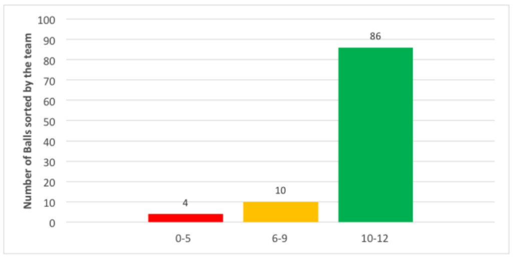

Deploy expert in environment
Deploy learner in environment
Learner uses IRL to learn expert's preferences
Generalize learned preferences to model interactions
Perform decision making and planning for teamwork
Solve the framework to obtain plan for spontaneous collaboration

Figure 3.8: The flow of steps necessary to complete a collaborative task process [59].

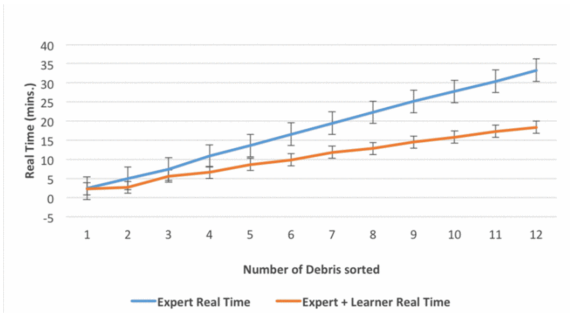
The method was evaluated on a simulated a colored-ball sorting task involving two robot arms, analogous to sorting fruit. This type of tasks only involve pick-and-place movements. The state of a single robot is defined as a tuple (*x, y, inHand, armPos, isTblEmpty*), where *x* and *y* describe the position of the robot, *armPos* the position of the arm, *inHand* the color of the ball in its hand, and the last parameter defines if there are still balls to be sorted. There are eight actions that represent the robot movements and their arms movement. During the sorting process, undesired collisions may occur between the two robots when, for example, attempting to pick up the same ball. Therefore, the authors assign a constant negative penalty (negative value to the reward function) to all collisions that occur at any state.

The effectiveness of the proposed method was demonstrated by how many balls are correctly sorted in two bins and the time taken to performed that. These are compared between a single expert robot and the co-robot system, involving sorting 12 balls across 100 runs. The results of the experiments shown in Figures 3.9a and 3.9b demonstrate that most of the objects were sorted correctly by the agents, and that the collaboration between agents reduce by less than half the time needed to complete the task, comparing to a single expert.

This paper demonstrates that intelligent co-bots can acquire defined and repeatable skills, aligning with observed agents, with customizable autonomy. It is an innovative approach that demonstrates end-to-end how robots can actively participate in tasks that benefit from collaboration, rather than just observing. Yet, this study has not demonstrate to be successful in complex tasks, since they do not adopt Convolutional Neural Network (CNN)-based methods. These research does not also adopt graphs or other similar structures, which is one of the main goals of this dissertation.



(a) Number of balls correctly sorted in each run, in a total of 100 runs.



(b) Time taken (in minutes) to sort the balls for a single expert vs a co-robot team.

Figure 3.9: Diagrams of the performance of the proposed method [59].

Efficient Model Learning for Human-Robot Collaborative Tasks (2014)

Nikolaidis et al. [44] introduce a method that enables a robot to generate a policy for a robust collaboration with a human in performing a task, by expert's joint-action demonstrations. This framework automatically learns the preferred sequences of the human agent taken by the robot when performing the task, without any human intervention, by clustering the sequences using an unsupervised learning algorithm. The system then learns a reward function that represents each preference, through the integration of a IRL algorithm, more specifically a Mixed Observability Markov Decision Process (Mixed Observability Markov Decision Process (MOMDP)).

MOMDP is a structured variant of POMDP, and was chosen by the authors because, according to them, "the number of observable variables for human-robot collaborative tasks in a manufacturing environment is much larger than that of partially observable variables" [44]. The human preferences for the actions of the robot are denoted as a hidden variable, since the intentions must be inferred through observation and interaction. The algorithm tries to maximize the expected total reward, while taking into account the uncertainty of the robot over the human's preferences. This framework is shown in Fig. 3.10 and it contains two main phases: the preprocessing of the training data, and the robot inference over the new human partner's preferences and collaborative execution of the task by both agents according to these preferences.

To evaluate the robustness of the proposed solution, the team applies the proposed framework to two different HRC tasks: a place-and-drill task and a hand-finishing task. For these tasks, humans provided demonstrations on the task, showing the robot how he/she would like the task to

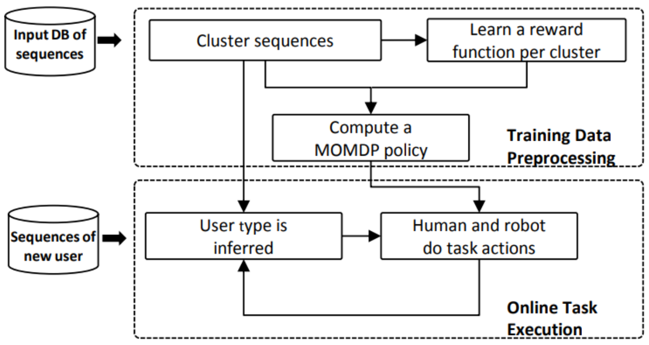


Figure 3.10: Framework flow diagram [44].

be done, and the system collected those preferences. Cross-validation was used at each experiment for the training data, by removing one human agent and using the demonstrated sequences from the other 17 agents.

The results from the experiments demonstrated that that the learned policy generated by the MOMDP algorithm has comparable performance to the one using the hand-coded approach. Moreover, it was possible to prove the robustness in performance of the robot to increasing deviations of human actions from the demonstrated behaviour, compared to other algorithms that reason in state-space. These results reveal that robots can learn robust policies according to the human partner dynamic preferences in HRC by learning these preferences and integrating them into general planning and decision-making.

This paper addresses an ML solution to collaborate to task allocation and scheduling problem in HRC in manufacturing settings. Nonetheless, it does not discusses the interface module, i.e. how the agents communicate with each other and how the agents perceive the status of the task and relevant information. Similarly to the previous paper, this work does not take advantage of CNN-based methods and consequently do not present results in complex tasks.

3.2 Graph Structures for task allocation approaches

Modern robotic systems have already proven to be safe and reliable enough to cooperate with operators in daily tasks. However, there is still a long way to go for robots and humans to achieve the cognitive capabilities of each other. It is both challenging for a robot to understand a task in a level that humans understand, and for a human to reach time and resources efficiency as robots do [29]. In an assembly process, a task can be divided into many smaller assembly subtasks, some of which are suitable to be performed by a robot. However, some tasks are highly flexible and easily deformable or more complex and consequently require an operator to perform some of their operations [68].

With these problems in mind, building a graph that represents the possible sequences of operations needed to complete a process, and that includes which team member executes the operation and other sufficient information about the process, is essential for an operator to perceive the task easily and for the robot to be efficient and effective in its assistance.

Transparent Role Assignment and Task Allocation in Human Robot Collaboration (2017)

Mangin et al. [29] propose a system that transforms high-level hierarchical task representations into low-level policies in Human-Robot Collaboration HRC processes. The aim is to teach the robot to successfully assist the human with supportive actions, such as providing objects, cleaning up the workstation or holding parts. The robot only has partial and sufficient knowledge over the task and the environment. The authors use Hierarchical Task Models (HTM) to share information between the human and the robot and enable transparent communication and mutual understanding. As a result of their level of abstraction, these models can easily communicate with their human peers about the task, since a task is modelled as a person would describe the task, with its proceeding and constraints, and without implementation details.

In Fig. 3.11 there is an example of a representation of a real-world construction high-level task. Each of the nodes can be a subtask or a specific action that can be performed by any of the agents (or both). Each subtask has the information of whether tasks can be done in parallel (||) or sequentially ($\rightarrow$), i.e. if two agents can perform two subtasks at the same time, or they are constrained to a specific sequence of execution. The authors introduced an additional element, alternative operator (V), that represents multiple, exclusive paths with equal probabilities. During task execution, the robot provides feedback to its partner by highlighting the current subtask it is executing (cyan block in Fig. 3.11). The operator can get the feedback, while perceiving the task at various levels of granularity.

Furthermore, the authors model each subtask (each leaf) of a HTM representation as a low-level POMDP, to solve the role assignment and task allocation challenge. Fig. 3.12 depicts the developed POMDP model, where nodes represent states and links represent action-observation pairs. After, the authors use an online Monte-Carlo solver to solve a short-horizon planning.

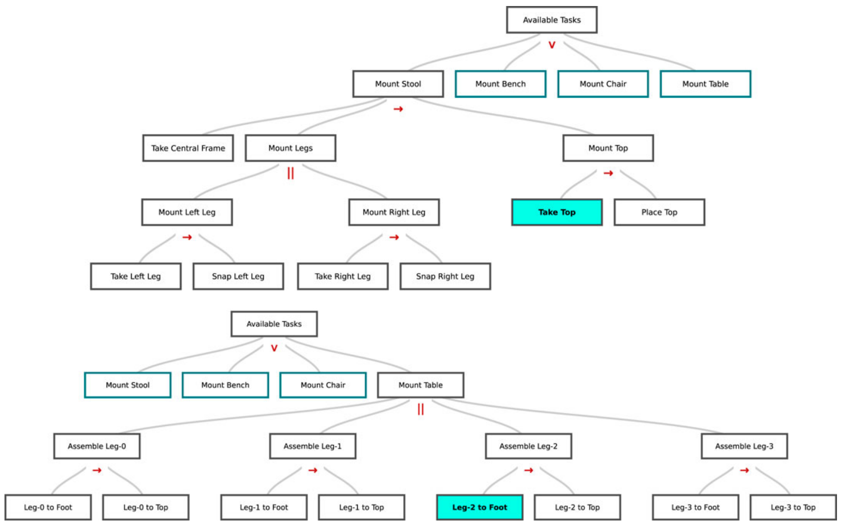


Figure 3.11: HTM for human-robot assembly tasks. [29].

The collaborative scenario the authors use is the construction of a miniaturized table composed of a base, four legs, four brackets, four feet, and 16 screws. While the human is in charge of performing actions that require complex manipulation or perception skills (e.g., screwing), the robot helps with supportive behaviours. *Mangin et al.* [29] evaluate the proposed solution on three task models extracted from three HTMs, which differ in the number of subtasks to solve, and in the type of relational operators between subtasks. The authors then compare the three task models' performance with two hand-coded policies.

The results demonstrate that the policy derived from an HTM matched or outperformed the other policies. Moreover, the proposed task structure has proven to give effective support to a human peer in real-world collaborative tasks, while adapting to his/her preferences. Therefore, we can conclude that this solution can leverage knowledge about the task structure to automatically generate efficient policies for a broad range of tasks.

This paper proposes an appealing task structure that are widely used for high-level task planning [8] and are close to human intuition [50]. According to *Roncone et al.* [50], they enable reusing components and contain the necessary and sufficient information for the robot to be efficient and effective in its support. Depending on the task, an HTM may also encode pre and post-conditions [10], communicative actions [5], and a variety of operators to combine nodes and subtasks [16]. From the results of the experiments, it can be concluded that HTMs are successful in defining policies that support the human to complete a collaborative task, with minimal feedback from the operator. However, from the graph it is not possible to differentiate which operations have already been done and which are still to be done. It is also not possible to see which tasks each team member has executed and the execution time. Moreover, the proposed work was not demonstrated with more complex tasks, considering more complex actions, and the MC algorithm might not be efficient with the increasing problem size.

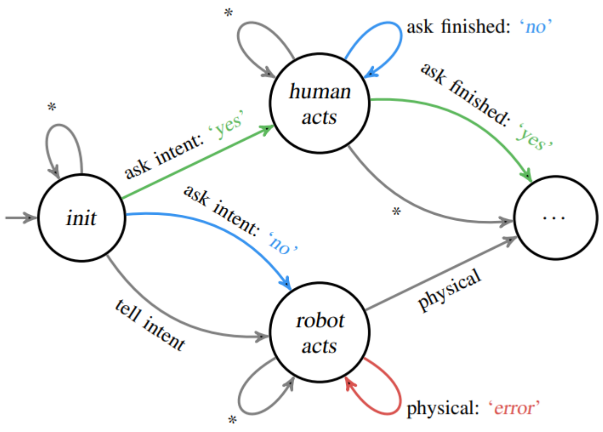


Figure 3.12: HTM subtask converted into a small, modular POMDP [29].

Deployment and evaluation of a flexible human–robot collaboration model based on AND/OR graphs in a manufacturing environment

Another type of hierarchical task structures/networks (Hierarchical Task Network (HTN)) are "and/or" graphs that represent tasks as hierarchies of conjunctions and disjunctions of subtasks. In production systems' planning, an "and/or" graph represents all possible assembly paths (parallel subtasks) of a given product [33] and show the time dependence of those subtasks. It is preferably suited for parallelized assembly.

Murali et al. [41] develop a modular, adaptive and flexible architecture for task planning and representation, sensing and robot control in HRC, representing the collaborative task as an "and/or" graph, and sharing information about the collaboration process between agents through a Graphical User Interface (Graphical User Interface (GUI)). The architecture aims to teach a robot to adapt reactively to unpredictable human operators's actions, while minimizing the overall process time. The motivation is to allow human operators to feel less mentally stressed when using the team's system and collaborate with a robot than manually performing the task.

Fig. 3.13 describes a generic AND/OR graph, where nodes represent a state and hyper-arcs represent the possible actions the agent can perform to reach a specific state. In this graph, h_1 establish an AND relationships between nodes n_1, n_2 and n_3, h_2 an AND relationship between nodes n_3, n_4 and n_1 and h_3 a OR relationship between n_1 and n_5. The root node defines the final goal of the task. At the beginning of the execution, all actions are labelled as unfinished. Then, when an action a is performed successfully, they are labelled as finished. If all actions are finished, then the corresponding h is done. Each node and hyper-arc has a cost or weight, which are assigned based on some parameters such as the difficulty of the respective actions, human operator preferences, or the time needed to complete the action.

As a case study, the authors use a pick-and-place palletization task, where some actions may be executed only by the operator, others only by the robot and others must be performed by both. The AND/OR graph corresponding to this case study is shown in Fig. 3.14a. The root node,

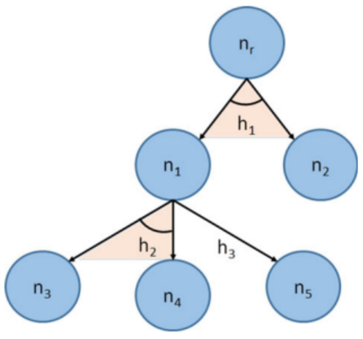


Figure 3.13: A generic AND/OR graph [41]

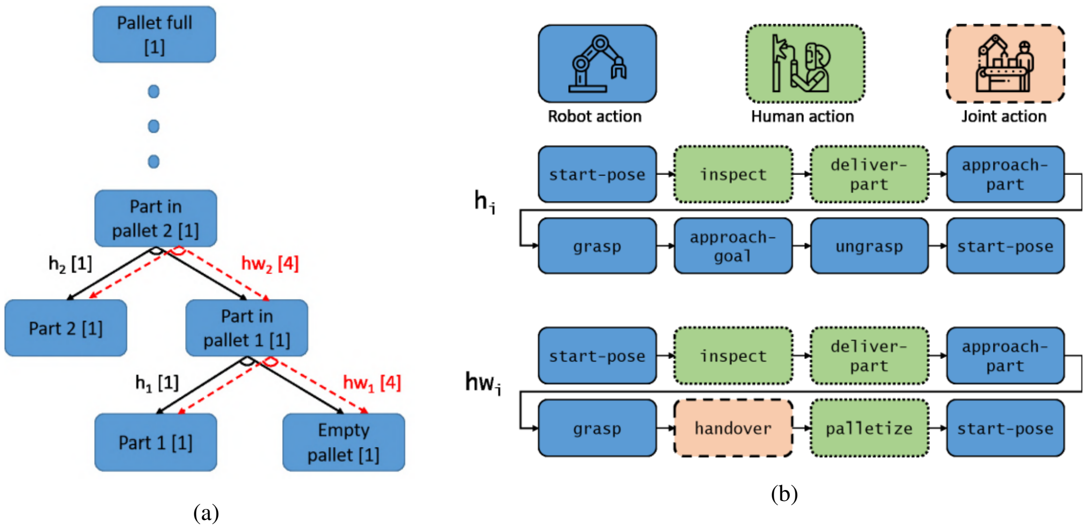


Figure 3.14: (a) The "and/or" graph representing the pick-and-place palletization task with hyper-arcs h_i (solid-black arrows) and hw_i (dashed-red arrows), and the weights of each node and hyper-arc (in brackets). (b) Two sequences of actions each one corresponding to a h-type hyper-arc and hw-type hyper-arc, and the agent assigned to each action. [41]

referred to as pallet-full state, corresponds to the completion of the palletization task consisting of 15 parts, and at each step, there are two possible paths (hyper-arcs h_i and hw_i, where i $\in$ (1, 2, ..., 15)). The difference between h_i and hw_i is that a hyper-arc hw_i involves a human stopping the robot and performing palletization himself. In total there are 2^15 possible paths to complete the process. In Fig. 3.14b, we can see an example of two sequences of actions corresponding to each hyper-arc, and the agent associated with each action.

The authors evaluate the system measuring the operators mental stress, the natural and fluent interaction between the agents that the architecture provides, and the flexibility in the HRC task for human operators. The first and third hypothesis are evaluated by asking volunteers to answer a set of Likert-scale questions before and after each experiment. The second hypothesis was measured using a combination of subjective and objective measures

There were failures during the collaboration that resulted in a lack of robustness in the algorithms. From the results, the team noted that the overall idle time is minimized, and the system ensures fluent interaction in HRC. From the subjective measurements was concluded that interacting with the graph interface helped volunteers understand robot intentions. The results of this research demonstrate how HRC models similar to this one can leverage the flexibility and the comfort of operators in the workplace.

In this study, a graph is used to represent a task and produce task sequences, while simple messages are provided on a GUI suggesting the next action to be executed by one of the agents. The authors, however, do not approach how the optimal task sequences is obtained, considering it to be provided or computed *à priori* considering the agent's capabilities, either to the robot or to the human . Another negative aspect is that this study assumes that the robot does not fail his tasks.

Mastering the Working Sequence in Human-Robot Collaborative Assembly Based on Reinforcement Learning (2020)

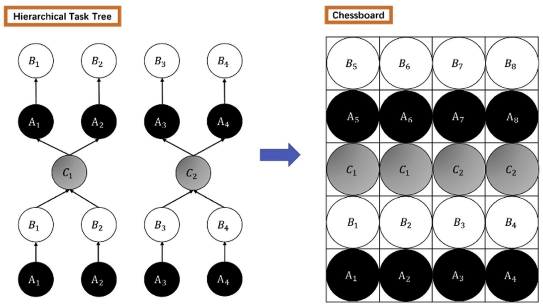
To improve the limited existing task planning approaches for HRC, *Yu et al.* [65] transform a HRC assembly task into a novel chessboard structure, considering specific rules determined by the required task constraints and where the selection of game moves is related to the decision making by both human and robot (assembly sequence).

In this work, the authors train a robot with a Monte Carlo Tree Search (Monte Carlo Tree Search (MCTS)) and integrate a Convolutional Neural Network (CNN) to tackle the dimensional state space issue, to predict the actions probabilities, and to know if a working sequence results in the maximum of the HRC efficiency. This MCTS-CNN algorithm was studied before the same author's more recent work [66] where the robot is trained with a DQN-MARL.

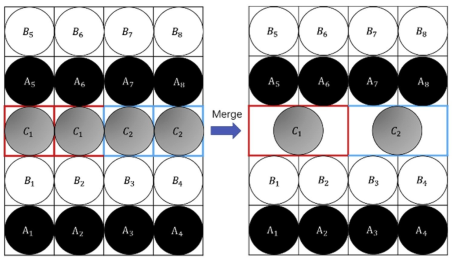
The team uses an height-adjustable desk assembly as a case study to establish the effectiveness of the proposed solution. Fig. 3.15a shows a transformation from this assembly task into a chessboard setting. There are three types of bones: black stones represent tasks that can be done by humans only, white stones express tasks that can be done only by robots and grey stones correspond to task that can be done by both agents. The width of each grid corresponds to the number of parallel tasks and the height corresponds to the number of sequential steps. Tasks with no sequential constraints (on different branches of the tree) are allocated in the same row, whether tasks with sequential constraints are set in the same column with preceding tasks being placed in the lower row.

Each stone occupies each grid of the chessboard, but they can be merging into a unique cell if they represent the same task and are adjacent stones in the same row, as shown in Fig. 3.15b.

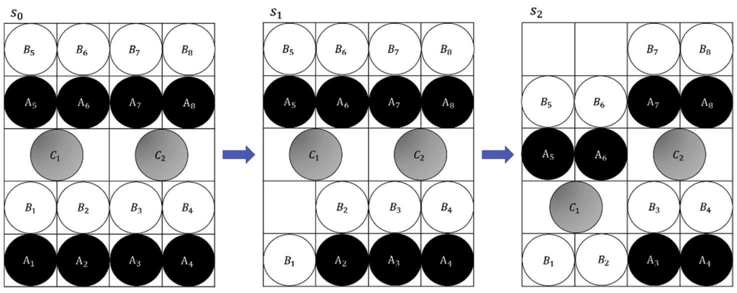
In Fig. 3.15c there is an example of two game moves representing two state transitions. A first task (A_1) is selected and performed by a human operator and the corresponding stone is removed from the chessboard. Consequently, the stone at the upper row (B_1) is moved down to the bottom



(a) Mapping a hierarchical task tree from an assembly task into a chessboard structure [66].



(b) Merging the same tasks in adjacent stones in the same row into a unique cell [66].



(c) Initial state and transitions of states [66].

$$\overset{s_0}{\begin{bmatrix} 999 & 999 & 999 & 999 \\ 1 & 2 & 3 & 4 \\ 2 & -1 & 2 & -1 \\ 999 & 999 & 999 & 999 \\ 1 & 2 & 3 & 4 \end{bmatrix}} \Rightarrow \overset{s_1}{\begin{bmatrix} 999 & 999 & 999 & 999 \\ 1 & 2 & 3 & 4 \\ 2 & -1 & 2 & -1 \\ 0 & 999 & 999 & 999 \\ 999 & 2 & 3 & 4 \end{bmatrix}} \Rightarrow \overset{s_2}{\begin{bmatrix} 0 & 0 & 999 & 999 \\ 999 & 999 & 3 & 4 \\ 1 & 2 & 2 & -1 \\ 2 & -1 & 999 & 999 \\ 999 & 999 & 3 & 4 \end{bmatrix}}$$

(d)

(d) Task structure formulated into a state matrix of the human agent and the corresponding matrix transformation during the game playing [66].

Figure 3.15: Diagrams of the format and transitions of the assembly chessboard [66].

row. Now, B_1 can be executed in parallel with the other black stone tasks at the same row. Stones in upper rows that occupy more cells than the stone that was picked cannot move down.

The authors use the following two performance metrics to evaluate their system: the time needed to finish all tasks for the assembly, and the overall time needed to finish all assembly tasks.

The proposed solution has proven that, using chessboard game rules, most common assembly processes can be modeled intuitively and concisely into an assembly chessboard structure. Using the projected task structure, the completion time of each agent can be easily mapped into a completion time matrix and all matrices can be stacked to give as input to the CNN. The results of the experiments also show that the proposed method is effective in optimizing the collaborative working sequence, and it outperforms the conventional mathematical optimization in terms of the mentioned two performance metrics.

This paper addresses one of the main goals of this dissertation – use a ML algorithm, namely Reinforcement Learning, to teach a robot to select the best sequence of operations to maximize the working efficiency. However, a limitation of this paper is that a chessboard structure might be very restricting for representing a collaborative assembly task conflicts and some task relations.

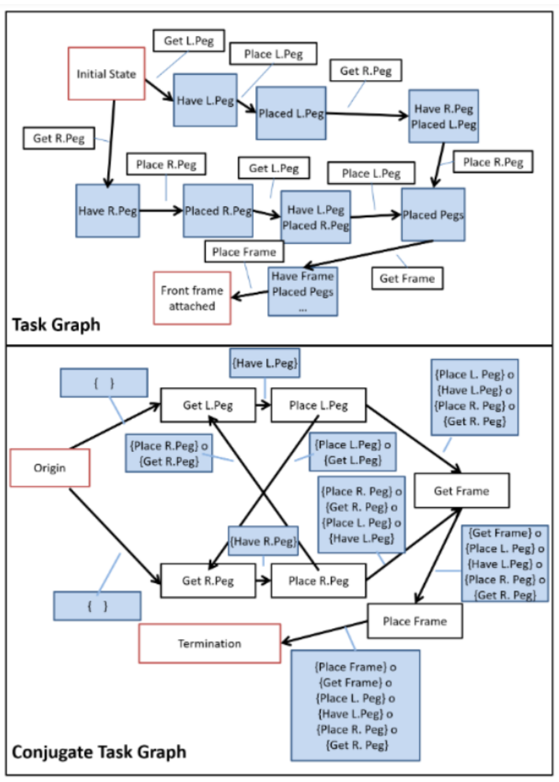
Autonomously constructing hierarchical task networks for planning and human-robot collaboration (2016)

Hayes and Scassellati [17] present a novel HTM type, called Clique/Chain Hierarchical Task Network (CC-HTN), which is autonomously constructed with an algorithm developed by the team that encapsulates properties of a task. This structure can be applied to task planning and the aim is to overcome the problem of typical HTNs that usually require manual specification or available domain knowledge.

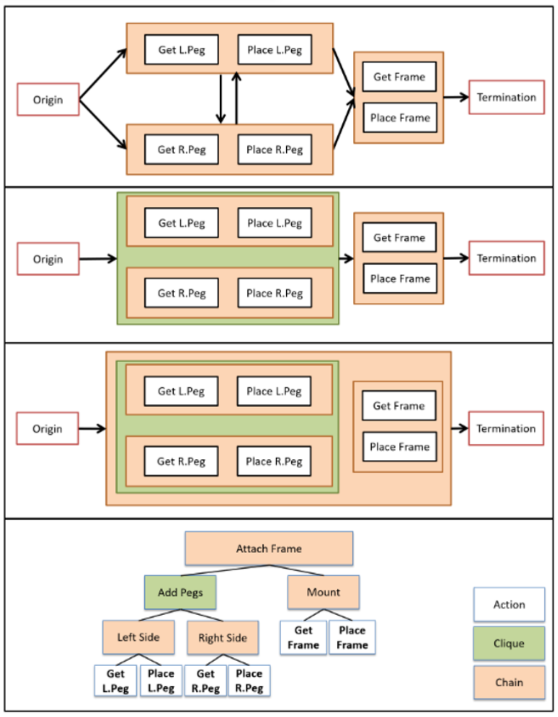
Given an assembly task, *Hayes and Scassellati* [17] start by representing it as a directed graph, where vertices indicate environment states and directed edges correspond to available actions from each state to another. This graph can be shown in Fig. 3.16a (top) as an IKEA furniture assembly task, and it represents all the possible valid task execution paths. However, this structure does not provide features that easily help detect logical sets of actions to abstract into subtasks. To overcome this, the team extends the graph to its conjugate, called Conjugate Task Graph (Conjugate Task Graph (CTG)), via a transformation algorithm. This new structure encodes parameterized actions in nodes as subgoals and environmental constraints encoded on links as compositions of subgoal completions. We can observe this graph in the bottom graph of Fig. 3.16a.

Because hierarchy structures have the benefit of being human interpretable, the CTG is transformed into a novel HTN based on subtask ordering constraints that distinguishes subgraphs of two types: ordered sequences (chains) and unordered sequences (cliques). This classification allows a task hierarchy to be derived only from known transitions. The process of mapping a CTG into a CC-HTN is illustrated in Fig. 3.16b from top to bottom, where a set of operations are computed until either the task consists of a single node, indicating that the hierarchical abstraction was successfully completed, or until the graph prevails unchanged, expressing that the graph is either incomplete or cannot be more condensed.

The tasks were generated using IKEA furniture assembly tasks to demonstrate the scaling capabilities of this work in HRC. The performance was evaluated using two measures: graph size and state estimation computation time. As the results demonstrate, CC-HTNs are useful in



(a) Task graph at the top, where states are represented as vertices and actions as edges, and its conjugate at the bottom, where subgoals are illustrated as vertices and edges represent environmental constraints [17].



(b) Transformation from a CTG (top) to CC-HTN (bottom) [17].

Figure 3.16: The different graph types used for constructing CC-HTNs, applied to part of a IKEA furniture assembly task [17]

segmenting complex tasks, causing a significant reduction in the amount of edges required to represent the task. Comparing with flat task models, CC-HTNs proved to reduce computation time and enable inference at several levels of abstraction.

This paper uses a graph to represent the possible sequence of operations to complete a task in an hierarchical structure, giving flexibility to the sequence planning and allocation. From the graph, one can observe which operations can be performed sequentially and in parallel. However, this study does not address algorithms for optimal task planning and allocation nor task simulation, which are two of the main approaches this dissertation focuses on.

3.3 Chapter Synopsis

Research efforts on collaborative robots have been multiple in the past few years, where the solutions have mainly been using ML techniques when trying to improve efficiency in manufacturing repetitive tasks. Most of the solutions include RL and IRL techniques and, therefore we divided the Machine Learning section into RL approaches and IRL approaches. The RL techniques try to maximize the overall cumulative reward taking into account the actions taken by an agent, while balancing exploration and exploitation to learn a task. However, most of the solution integrate DRL method, which has become a major research area of interest in the manufacturing sector due to its success in solving some complex problems without the need of any prior domain knowledge. IRL approaches can design a reward function without manual specification, given the necessary features and the human's demonstrations.

Table 3.1 summarizes the ML techniques and graph structures adopted from each research studied throughout this chapter.

Research	**ML technique**	**ML algorithm**	**Graph structure/ communication interface**
Zhang et al. [68]	DRL	Multi-agent DDPG	–
Liu et al. [28]	DRL	Dual-agent DQN	–
Yu et al. [66]	DRL	Multi-agent DQN	Chessboard setting and HTN
Wang et al. [60]	IRL	MaxEnt-IRL	–
Trivedi and Doshi [59]	IRL	MaxEnt-IRL	–
Nikolaidis et al. [44]	IRL	MOMDP	–
Mangin et al. [29]	POMDP and Monte-Carlo	Monte-Carlo	HTM
Murali et al. [41]	–	–	AND/OR graph and simple messages
Yu et al. [65]	DRL	MCTS-CNN	HTN and chessboard setting
Hayes and Scassellati [17]	–	–	CC-HTN

Table 3.1: Overview of the state-of-the-art

Zhang et al. [68] have designed a HRC framework to assemble a product. The authors implement a dual-agent algorithm that extends the DDPG algorithm, where the actor-network of each agent processes the current state and outputs their own actions, and subsequently, the critic-network evaluates and calculates the value of the taken actions. The reward function is based on the difficulty and time each agent needs to complete the collaborative assembly task. A structure tree that represents the assembly task, guides the human partner displaying the sequence of tasks assigned to him/her. From the results, where the solution is applied to the assemble of an alternator, the authors could conclude that, compared with the MADDPG algorithm, the proposed method has higher efficiency.

Liu et al. [28] proposed a novel training approach adapted from the DQN method and called DADRL. This algorithms considers both the robot and the human as agents, where the robot is trained to learn how to make decisions, while the human agent represents the real human in HRC, demonstrating the dynamic and stochastic properties of the task. The aim is to teach the robot to be capable of decision-making and task-planning "under the uncertainties brought by the human partner using DRL", as the authors explain [28]. The team applied the solution to a bolt-type roller containing six components, and a gear system with eleven components. These products generate a tree file from which the task is loaded to the algorithm, the set and subset of actions, and the constraints among different actions to validate the learning methods. It was concluded that the proposed solution can drive the robot agent to learn how to make decisions and hence collaborate with the human partner to complete the assembly task even when the human executes unforeseen procedures.

Yu et al. [66] formulate a human-robot collaborative assembly task as a chessboard game with specific assembly constraints determined by the game rules. The robot is trained with an algorithm based on DRL, where a CNN is used to predict the distribution of the priority of move selections and to know whether a working sequence is the one resulting in the maximum of the HRC efficiency. An height adjustable standing desk assembly was used with different number of tasks and agents as a case study. From the experiment results, it was demonstrated that DQN-based algorithms, either single-agent or multi-agent, outperform the naive Nash-Q and the DP methods. The results also proved that the more agents, the more time it takes to find an optimal task scheduling policy. The proposed dual-agent algorithm is successful in obtaining optimal task scheduling policies for complex tasks. The method can also be generalized to other similar assembly tasks.

Wang et al. [60] developed a TLC model that aims to use MaxEnt-IRL to teach a robot to perceive its human partner assembly demonstrations and collaborate in completing the task. The human instructs the robot through natural language according to his/her personal working preferences. As a use case, the team employs a pick-and-place block assembly task. From the results, the authors could demonstrate that the robot correctly assists its human partner in product assembly according to the customization requirements and human assembly preferences. Comparing to the traditional approaches, the proposed approach has proven efficiency in HRC.

Trivedi and Doshi [59] show how IRL can make a robot learn its partner agent's preferences to successfully collaborate with the her/him agent to complete a task in the same environment. The team applies MaxEnt to the learning process, but they enhance that their solution can integrate any IRL capable method. The work is applied to a task where a line robot autonomously collaborates with another robot to sort ripe and unripe fruit such as oranges. The method was evaluated on a simulated a colored-ball sorting task involving two robot arms. The results of the experiments have proven that most of the objects were sorted correctly by the agents, and that the collaboration between agents reduce by less than half the time needed to complete the task, comparing to a single expert.

Nikolaidis et al. [44] introduce a method that enables a robot to generate an optimal policy for HRC, taking advantage of expert's demonstrations. The framework uses MOMDP to automatically learn the preferred sequences of the human agent taken by the robot when performing the task, without any human intervention. To evaluate the robustness of the proposed solution, the authors apply the proposed framework to a place-and-drill task and a hand-finishing task. The results demonstrated that the policy learned using the proposed algorithm has better performance to one using an hand-coded approach. These results reveal that robots can learn robust policies according to the human partner dynamic preferences in HRC by learning these preferences and integrating them into general planning and decision-making.

Both DRL and IRL techniques, according to the researches described in Section 3.1, have demonstrated good performance and success in human-robot collaborative tasks in manufacturing sectors. *Zhang et al.* [68], *Liu et al.* [28] and *Yu et al.* [66] have shown that DQN and DDPG-based algorithms are efficient in solving decision-making and sequence planning reducing problems and are efficient in complex tasks. Most of these works, however, do not approach graph structures and the advantages they might bring to collaboration in manufacturing sectors. *Liu et al.* [28] use a chessboard setting, but they assume this structure has limitations in representing some task constraints and relations. *Wang et al.* [60], *Trivedi and Doshi* [59] and *Nikolaidis et al.* [44] have demonstrate that IRL can be sucessfull in assisting humans in completing tasks that benefit from collaboration, reducing the human's workload. Yet, these studies have not demonstrate to be successful in complex tasks, since they do not adopt CNN-based methods. These research do not also adopt graphs or other similar structures, which is one of the main goals of this dissertation.

Following the ML section, we focused on researches that focus on graph structures that represent the possible sequences of operations needed to complete an assembly task, and that includes the necessary information about the process for both agents in collaboration to perceive and communicate with each other, and hence be efficient and effective in completing the task.

Mangin et al. [29] developed a system that uses HTM to share information between the human and the robot and enable transparent communication and mutual understanding. This high-level hierarchical task representations is then transformed into low-level policies to teach the robot to successfully assist the human with supportive actions. From the experiments taken using a miniaturized table as a case study, it was concluded that HTMs are successful in defining policies that support the human to complete a collaborative task, with minimal feedback from the operator. These models enable easy understanding between the peers, the task is modelled with a level of abstraction as a person would describe the task, with its proceeding and constraints, and without implementation details. HTMs are also scalable and enable reusing components over similar tasks.

Murali et al. [41] proposed an architecture that aims to teach a robot to adapt reactively to unpredictable human operators's actions, while minimizing the overall process time. A collaborative task is represented as an "and/or" graph, where nodes correspond to states and hyper-arcs represent the possible actions the agent can perform to reach a particular state. The team takes evaluate the efficiency of the solution on a pick-and-place palletization task. The results demonstrate that HRC models such as the one the authors propose can bring flexibility and comfort to operators in the workplace. The overall idle time is also minimized.

Yu et al. [65] map a human-robot collaborative assembly task into a chessboard game with specific assembly constraints determined by the game rules, in which the a chess piece move represents the decision made by an agent, considering the task structure and constraints. The authors train the robot with a combination of MCTS and a CNN. Using the chessboard structure, the completion time of each agent is formatted into a completion time matrix, which is given, along with all time matrices, as input to the CNN. The team uses an height-adjustable desk assembly as a case study to establish the effectiveness of the proposed solution. Although there might be some limitations in representing the task conflicts and other complex task relations in a chessboard structure, the proposed representation has shown to be intuitive and concise in largely erratic assembly tasks.

Hayes and Scassellati [17] present a novel structure adapted from a traditional hierarchical structure, called CC-HTN, to represent the possible sequence of operations to complete a task. The goal is to give flexibility to the planning and allocation. From the CC-HTN, one can observe the ordered sequences (chains) and the unordered sequences (cliques). Nodes represent actions as subgoals and edges represent environmental constraints as compositions of subgoal completions. The results were generated using IKEA furniture assembly tasks, which determined that CC-HTNs are useful in segmenting complex tasks, causing a significant reduction in the amount of edges required to represent the task. Comparing with flat task models, the proposed representation has also proven to reduce computation time and enable inference at several levels of abstraction.

All the solutions mentioned throughout Section 3.2 use hierarchical task structures, which have shown to be extremely powerful in supporting the planning of assembly tasks in HRC and co-robot collaboration. These studies, however, have some limitations. *Mangin et al.* [29] do not demonstrate results with complex tasks, not knowing if the MC solver is efficient in collaborative task planning. *Murali et al.* [41] and *Hayes and Scassellati* [17] do not approach how the task sequences and the optimal one are obtained, with the first team considering them to be provided or computed *a priori*. The chessboard structure proposed by *Yu et al.* [66] presents some challenges in representing the task conflicts and other complex task relations.

Chapter 4

Case Study and Proposed Solution

So far, we have identified the problem with collaborative assembly processes for robots, reviewed related techniques, studied the state-of-the-art and compared the different approaches. Now, in this chapter, the solution approach is presented, including the its global architecture and composing modules. Additionally, the case study for implementing and validating the proposed solution is introduced, in order to facilitate the vision of the applicability of our approach.

4.1 Proposed Solution

The goal of this project is to develop a system capable of automatically generating efficient assembly sequences that respect precedence dependencies and task pre-conditions in a collaborative environment, while transmitting knowledge on the task in a graph structure.

This thesis' proposed solution is an architecture divided into three modules. The first represents the translation and visualization of the task into a **graph**. Graph structure solutions must be flexible enough to be re-used for different configurations, whether for different product versions or other tasks involving the same type of operations. The second module expresses the agent's **artificial intelligence**. The agent must be trained with a ML algorithm to generate the sequences and assign a resource. The third module corresponds to the **simulation** of actions in a simulator or in real world.

This architecture can be seen as a **closed-loop interaction** between the modules, since each module influences, directly or indirectly, the performance of the other modules, i.e, the graph module "feeds" the intelligence and the intelligence "feeds" the simulation, which in turn "feeds" back to the graph. The **graph** aggregates information about the task, namely the **subtasks** and their **constraints**, that will be perceived by the agent to define the observation and action spaces on which the agent will run the algorithm, as well as the **possible sequences** that will be taken into account by the reward function of the agent, so it penalizes actions that do not meet precedence dependencies. The graph also reports the **average time** each **operation** takes to be completed by each actor, which will influence the learning process for task allocation, i.e. assign actions to an agent according to the time it takes in average to perform the action with the aim of minimizing the overall task time. The **agent**, in the second module, can now define the **observation and actions spaces** and the **reward function** that will allow it to successfully select the best sequence of actions, using RL algorithms, more specifically **off-policy RL algorithms**. The training process starts with an empty Q-table where the rows correspond to all states and the columns represent the possible actions for each state, and ends with a Q-table that returns the action that generates the maximum accumulated reward for each state. Regarding the third module, the robotic manipulator, at each step, will **simulate** the **optimal action** and update the status of the task in the graph, particularly the actions that are completed, that are being performed and that are still to be done. The simulation also allows an agent to perceive the execution time of each action, as well as its actor. The closed-loop interaction between the three modules is represented in Fig. 4.1.

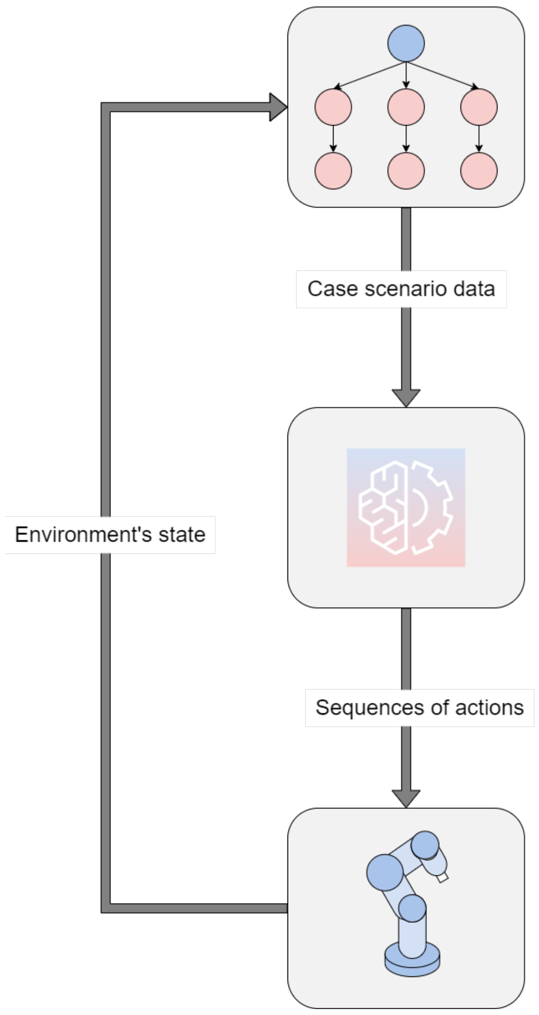


Figure 4.1: Flow of the architecture proposed as a solution

In terms of robot execution within the process, the overall sequence result can be validated using the robotic simulator Coppelia Simulator, formerly known as **V-REP** [49]. This simulator, ideally applicable for multi-robotics, supports Python and can be used through a remote API client or a custom solution, among other programs. It provides various sensors and robot models, making it applicable for multi-robotics. It is used for many features including "factory automation simulations, fast prototyping, and remote monitoring" [53]. By implementing the base actions within the simulator, our graph simulation can trigger each one of these actions and validate that the sequence is indeed correct.

4.2 Case Study

In order to address the problem of this dissertation and implement and validate the solution, a furniture assembly task was chosen, as it depicts a complex process that can result in different final products. In particular, we focus in the task of **assembling tables** as a case scenario. The structure of the table was based on a design presented by *Zeylikman et al.* [67], which constituent parts can be used to create a variety of different objects, such as chairs, shelves and consoles, providing modularity and scalability to our model. Fig. 4.2 shows a 3D rendering of the table.

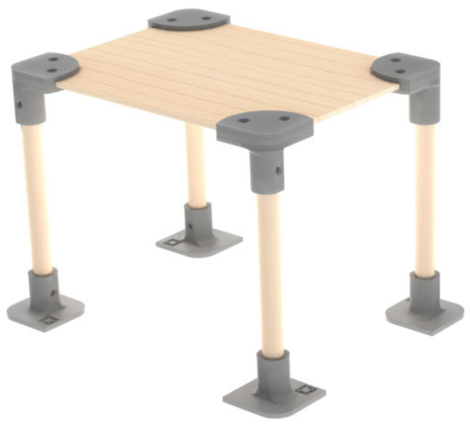


Figure 4.2: Table 3D rendering [67]

The table is comprised in 5 structural parts, **one base** and **four legs**, and composed by 15 components as displayed in 4.1. A table leg consists of a dowel and one F-type bracket, and the base needs one plywood. The T-type brackets are used to assemble the base with the legs, and the screws are the fasteners, which are fastened by a screwdriver.

Given its constituent parts, the task of our case scenario can be easily **discretized** into 27 smaller and simpler problems, called **subtasks**. Each subtask is presented in Table 4.2 along with their amount and its constituent components. The table has only one base; however, this base is composed of four T-type brackets, which do not necessarily require to be built sequentially. Hence, the build base subtask is decomposed into four actions. This division of a task into subtasks is the **key** for a good task representation. It allows a more organized representation of the task on a graph and a better understanding of the different possible sequences in which it can be assembled

Component	Component description	Amount
A	Plywood	1
B	Dowel	4
C	F-type brackets	4
D	T-type brackets	4
E	Screw pack	1
F	Screwdriver	1

Table 4.1: Table components

by any assigned resource. There are more tasks that support discretization, but this scenario is a simple and common in manufacturing and often studied by researchers, which allow us to analyze how others have approached similar problems with the same tasks. Moreover, a chair, in a manufacturing assembly process, may be considered an "extension" of a small table, since it is broken down by the same sub-tasks, but with the addition of a backrest. This sharing of subtasks allows for easier **design** and **reuse** of graphs.

		Components					
Subtask	**Amount**	**A**	**B**	**C**	**D**	**E**	**F**
Bring screw pack	1					1	
Bring screw drive	1						
Bring dowel	4		4				
Bring plywood	1	1					
Bring bracket F	4			1			
Bring bracket T	4				1		
Build leg	4		1	1		2	1
Build base	4		1		1	2	1
Assemble legs and base	4	1	1			2	1

Table 4.2: Table materials

Each subtask must be further decomposed into a sequence of **operations**, containing more relevant and specific information about the task. The table assembly task contains operations such as picking, placing, snapping, positioning, and fastening an object. The first four operations are based in the same basic and common movements a robot may perform in manufacturing: move the gripper to a position, grasp an object and/or release an object. Some operations, such as fastening, require high precision and are too complex for a robot to perform. Based on its performer, the operations may be distinguished in two types:

- Type I - Human only operations, such as fastening screws;
- Type II – Either human or robot operations, such as picking, placing, snapping and positioning objects.

In type I operations, the robot only needs to wait for the operator to finish its operation.

In Fig. 4.3 we can see an example of a sequence of operations to build a leg.These operations must be performed sequentially by one of the agents, starting by picking a bracket and ending in placing the final leg in a predefined position, according to the flow suggested by Figure 4.3. We represent each type of operations in different colors (type I in yellow and type II in gray).

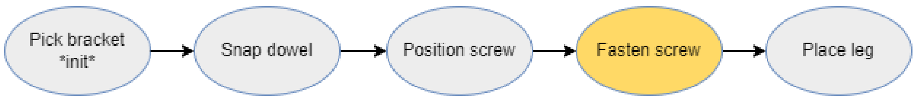


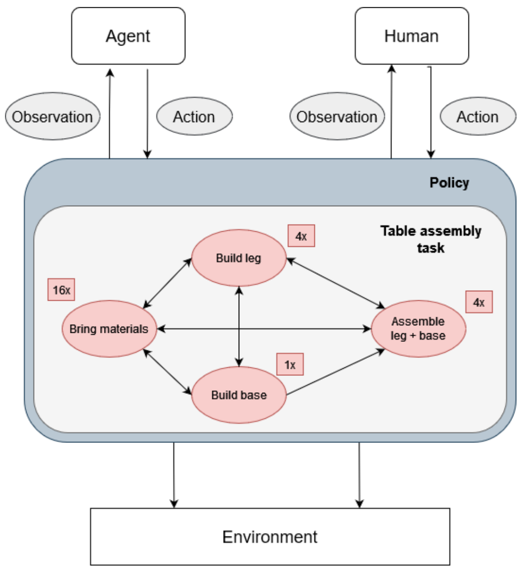
Figure 4.3: Subgraph representing the sequence of operations to build a leg

Given the chosen scenario, we can identify three necessary components for the simulation: an **object**, a **robotic manipulator** and a **gripper**. An object is a tool or a part of the final product. A gripper grasps the objects and is attached to the robotic manipulator, which moves the gripper around.

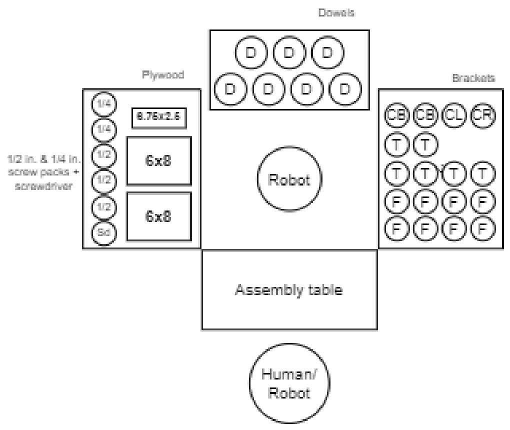
In the manufacturing industry, operators all have different experience and knowledge about a task, which influences the time each one takes to perform an operation, as well as the probability of failure. It is therefore essential that operators can be differentiated by their level of expertise: **beginner**, **intermediate** or **expert**. An expert, in average, performs the tasks with a **lower time and a lower variance**.A beginner, on the other hand, needs more time to complete a task and usually executes the actions in a more random sequence. With this information, the robot can understand the average time it takes the human to perform an operation and thus choose the best sequence.

In Fig. 4.4a we can see the **agent-environment-agent interaction** integrated with the **assembly table task**, based on the RL paradigm. At each iteration, two agents observe a graph and perform actions. The graph is updated based on the environment's state. The robot selects the best sequence of actions according to a policy that must complete the task under certain conditions. Before building a part, the necessary materials must be brought into the workspace. Also, to assemble two or more parts, those parts must be already built. The environment includes a workspace to perform a task and the necessary tools around the agent, as Fig. 4.4b suggests.

For the assembly process to be completed, every subtask must be executed only once, and hence the number of possible assembly sequences is 27!, which is more than 10^{27}. However, the assembly sequence must respect a set of precedence dependencies, which significantly decreases the number of feasibly assembly sequences. For example, a table leg cannot be built without a dowel being on the worktable, that is, without first having an agent execute a subtask to bring a dowel. The assembly base and leg subtask cannot also be performed without the base and a leg being built. Such precedence sequence dependencies are shown in the Table 4.3.



(a) Agent-environment-agent interactions



(b) Possible environment

Figure 4.4: Agent-environment-agent interactions and possible arrangement of the environment for assembling a table

Subtask	Predence subtask
Bring screw pack	None
Bring screw drive	None
Bring dowel	None
Bring plywood	None
Bring bracket F	None
Bring bracket T	None
Build leg	Bring dowel, Bring bracket F, Bring screwdriver, Bring screw
Build base	Bring plywood, Bring bracket T, Bring screwdriver, Bring screw
Assemble legs and base	Build leg, Build Base

Table 4.3: Predence subtasks' dependencies

4.3 Chapter Synopsis

The key aspects to retain about this chapter are the following:

- The goal of this project is to develop a system capable of automatically generating **efficient assembly sequences** that respect **precedence dependencies** and **task pre-conditions** in a collaborative environment, while transmitting information about the task in a **graph structure**;
- The agent must also be able to minimize the **collaboration global time**;
- The proposed solution is an architecture decomposed into three modules: task **representation** on a graph structure, **AI** to train the robot to select the best sequence of actions, and **simulation** of the selected sequence;
- The three modules "feed" each other, directly or indirectly, with information for the module to proceed;
- The case study chosen for this dissertation was the **assembly of a table**;
- When choosing the scenario, the idea was to train and evaluate the system in realistic environments, facilitating the application of the solution to a **real-world environment**;
- The design of the table was based on the design presented by *Zeylikman et al.*; [67], which provides **modularity** and **scalability** to our model;
- The table is comprised in 5 structural parts (four legs and one base) and composed by 15 components;
- The table assembly task can be decomposed into **27 subtasks**. This division is the key for a **efficient task representation**;
- Each subtask can be further **discretized** into a sequence of **operations**, containing more relevant and specific information about the task;
- The task's operations are the following: picking, placing, snapping, positioning, fastening an object, and wait;
- Some operations, such as fastening, require high precision and must be performed **only by the human**;
- There are three essential environment components for the simulation: an object, a robotic manipulator and a gripper;
- Based on his/her experience and knowledge on performing a task, a manufacturing industry operator might be a **beginner**, an **intermediate** or an **expert**. This expertise influences the time each one takes to perform an operation, as well as the probability of failure;

- The assembly task is considered **complete** when **all** the 27 **subtasks** have been executed in the correct order;

Chapter 5

Implementation

Following the definition of the proposed solution and the implementation and validation case study, we are now able do decide which graph structure and ML algorithm might best suit the two case scenarios and describe the implementation of the proposed solution.

The entire system was developed in Python, a programming language that provides useful packages for simple and flexible representations of graphs. For this thesis' solution, we decided to use a powerful package that uses many Python packages to provide more features such as drawing, called *Networkx* [55]. Python also is a popular language to develop ML algorithms, in part because of its libraries, like *Numpy* [15] and *matplotlib* [19], which really makes the learning process easier and the allow us to analyse and evaluate the training statistics.

This chapter aggregates the implementation procedures to build our solution, and it was written with the goal of being able to be used in the implementation of other similar problems related to collaborative tasks that can be composed on discrete subtasks.

We start by describing the chosen graph task structure for our solution, as well as the inputs required for the graph module and consequently our system. Later, we define the ML algorithms adopted, including the environment's observation and action space, the hyperparameters, the reward function and the training algorithm setup in general. At last, we describe the strategy used for our simulation module. The implementation of the whole system was designed to be flexible and scalable to different tasks or products composed of the same base operations.

5.1 Task Graph Module

To represent our collaborative assembly task, given their easy decomposition into simpler and more specific subtasks and operations, we decided to follow a **hierarchical task structure** that, according to its state-of-the-art, is widely used for high-level task planning and enables reusing components over the two tasks. Particularly, we used a structure similar to a simple HTM, adopted by *Mangin et al.* [29]. This structure has proven to be successful, while AND/OR graphs have limitations in representing some task relations and constraints. HTMs are a simple and appealing form, close to human intuition that allows both agents to communicate and understand each other's actions, and that contain sufficient information for the robot to efficiently and effectively complete the task.

For the creation, manipulation and visualization of our graphs, we used **NetworkX** [14], a Python package that allows us to generate networks, analyze network structure, build network models, design new network algorithms and draw networks, as well as load and store graphs in many data formats. In our case, we used **JSON** format to load and build our graphs. The system reads the file containing the necessary information for a task, namely the nodes and the links between each pair of nodes. Networkx then parses the JSON serializable data and generates a graph [22]. Each node is composed by at least the following three elements, which values must be strings:

- **id**: unique node identification;
- **type**: classifies the node as *task*, *subtask* or *operation*;
- **state**: determines whether the node is still to be, is being, or has already been executed.

The state is useful for both agents (operator or robot) to be able to perceive the task's evolution. Operation type nodes, in addition to these, contain four more parameters:

- **operation**: indicates which action the node refers to (*pick*, *place*, *snap*, *position* or *fasten*). Its value must then be a string;
- **object**: indicates which object the node refers to, e.g. screw (string value);
- **time_robot**: Estimated time of operation's execution by the robot (integer value);
- **time_operator**: Estimated time of operation's execution by the robot, depending on his/her expertise (integer value).

An HTM also allows us to encode our tasks' pre and post-conditions, and a variety of operators to combine their sub-tasks. For that reason we were able to add an addition element to our sub-tasks and tasks, called "**sequence**". This operator determines whether a sub-task or task type node is *sequential*, *parallel* or *alternative*. If a task is broken down into sub-tasks that might be done at the same time by two different agents, the task's "sequence" parameter is "**parallel**". If the sub-tasks must be executed by a specific order of execution or exclusively the task is "**sequential**" or "**alternative**", respectively. The same constraints go to sub-tasks that are decomposed into operations. This set of elements demonstrates the complexity of the collaboration, as well as the type of constraints that exist when executing a collaborative task. In our graphs, the parallel operation is represented by $\|$, the sequential by $\rightarrow$ and the alternative by **V**.

In Listings 5.1 and 5.2, we can see parts of a JSON file used to build the assembly table task, with three different nodes (a task, a subtask and an operator) and their data (Fig. 5.1) in JSON format, and a link between two nodes starting from build_table and ending in bring_plywood, meaning that the source node has lower granularity than the second (Fig. 5.2).

```
{
    "id": "build_table",
    "type": "task",
    "state": "todo",
    "sequence": "parallel"
},
{
    "id": "bring_plywood",
    "type": "subtask",
    "state": "todo",
    "sequence": "sequential"
},
{
    "id": "pick_screw"
    "type": "operation",
    "state": "todo",
    "operation": "pick",
    "object": "screw",
    "time_robot": "3",
}
```

Listing 5.1: Example of a node and its data in JSON format

```
{
    "source": "build_table",
    "target": "bring_plywood"
}
```

Listing 5.2: Example of a link between two nodes in JSON format

Just as the name implies, our graphs' structure follows a hierarchy, where the root node represents the task as the lowest granularity, and as the hierarchy moves downwards, the task is divided into subtasks, which in turn are divided into concrete actions. However, contrary to how *Mangin et al.* [29] define the sequences of actions, where each operation's parent is the respective subtask, in our solution, for sequential subtasks, each operation's parent is the previous operation in the sequence. The aim is to have a better and vertical visualization of operations and easily differentiate operations with subtasks.

In Fig. 5.1 there is an example of an alternative subtask of assembling four legs with a base (represented in purple color). It is alternative, because two agents can not be assembling two legs at the same time, as they require the use of the same base. Its subsequent subtasks of building legs (red nodes), on the other hand, require its operations (gray for type I operations and yellow for type II) to be performed by the specified sequence. 5.2

The overall graph structure for the table assembly task is illustrated in Fig. 5.2, simplified for better and understandable visualization.

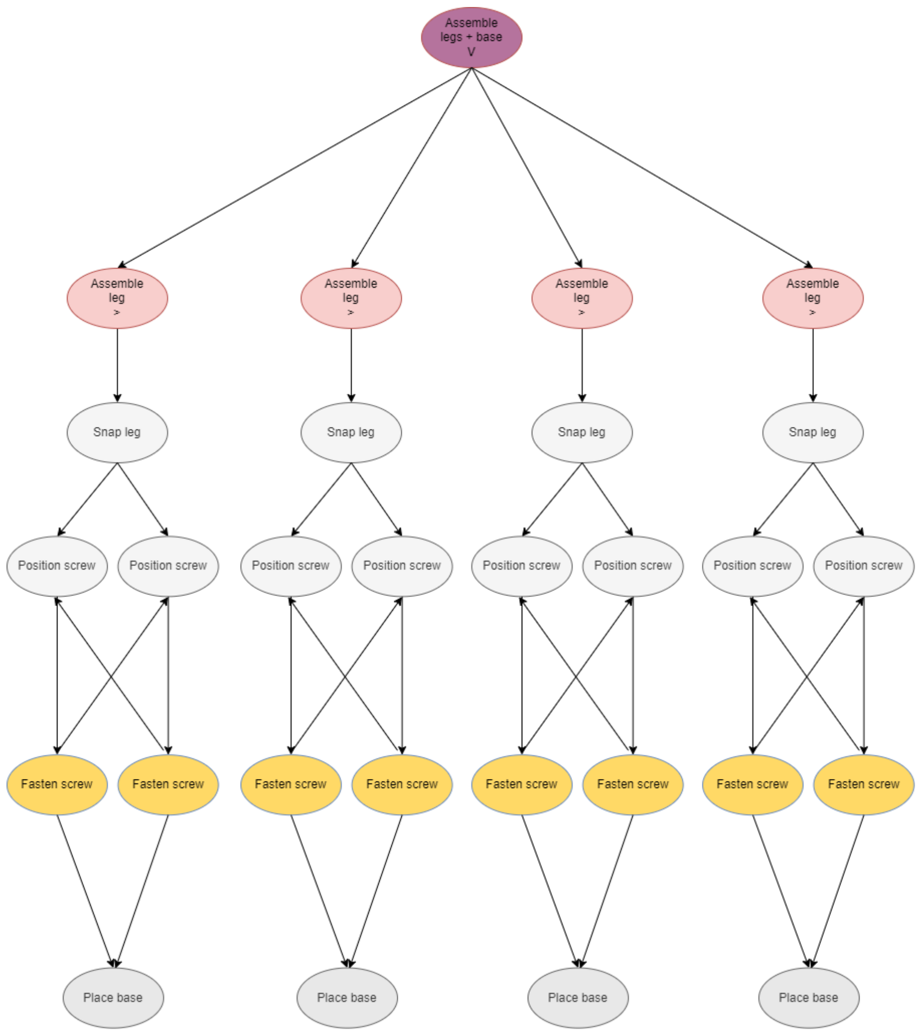


Figure 5.1: Alternative subtasks for assembling four legs with a base of a table

5.2 Robot intelligence Module

Approaches under the scope of RL have revealed good performance in complex tasks without the need of human's demonstrations. We started by focusing on a RL approach based successive interactions with the environment to learn the Q-values and consequently find an optimal policy that maximizes the cumulative reward, more specifically, the **Q-learning** algorithm. This algorithm's Q-values matrix (Q-table) provides us easier analysis and interpretation of the training process results and evolution, as well as how the environment must behave for the algorithm to converge. After, we proceeded to implement DQN to help the agent figure out which action to exactly perform and consequently complete an assembly task with high-dimensional space requiring less time and memory.

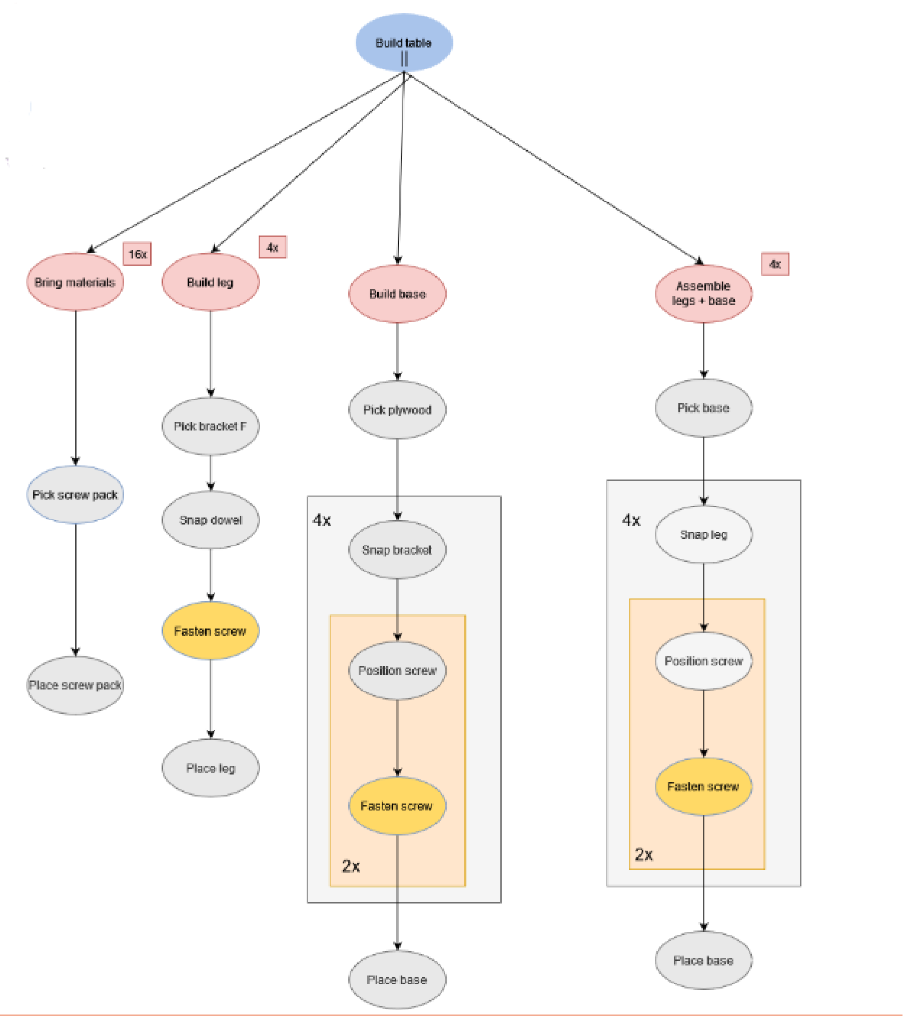


Figure 5.2: Graph structure for the table assembly task

5.2.1 Observation and action spaces

As stated in Chapter 4, the table assembly procedure can be decomposed in 27 different and discrete subtasks, or actions. Moreover, the task can be considered as finished when all these actions have been completed. Consequently, the **states of MDP** can then be defined as the assembly progress for each completed subtask. The initial state would consist in no actions performed, and the final state to all actions completed and the table assembled. Nonetheless, as objects and table parts that are in the workspace influence the feasible sequences of actions, the MDP states must also take into account these elements. Each state is then defined as a **tuple** of two elements: **the number of completed subtasks**, and **a list with the objects/parts in the workspace**. In total, considering that there are 9 different objects/parts and 27 in total, there are **1812473 states** in the table assembly task, where the first state is represented as (0,[]) and the final state as (27, ['screw', 'screwdriver', 'plywood', 'base', 'base', 'base', 'base', 'base_leg', 'base_leg', 'base_leg', 'base_leg', 'dowel', 'dowel', 'dowel', 'dowel', 'bracket_T', 'bracket_T', 'bracket_T',

’bracket_T’, ’bracket_F’, ’bracket_F’, ’bracket_F’, ’bracket_F’, ’leg’, ’leg’, ’leg’, ’leg’]). The ’base_leg’ object represents a leg assembled with the base. Fig. 5.3 illustrates part of the MDP’s states and actions scheme, where actions are represented as number as an example for the purpose of simple visualization.

An action corresponds to a subtask. However, to assign a performer to the subtask and hence reward the action-state pair according to the time the actor takes to complete an action, the action must contain this information. This is why the action space is represented by a tuple composed composed of the subtask and the actor.

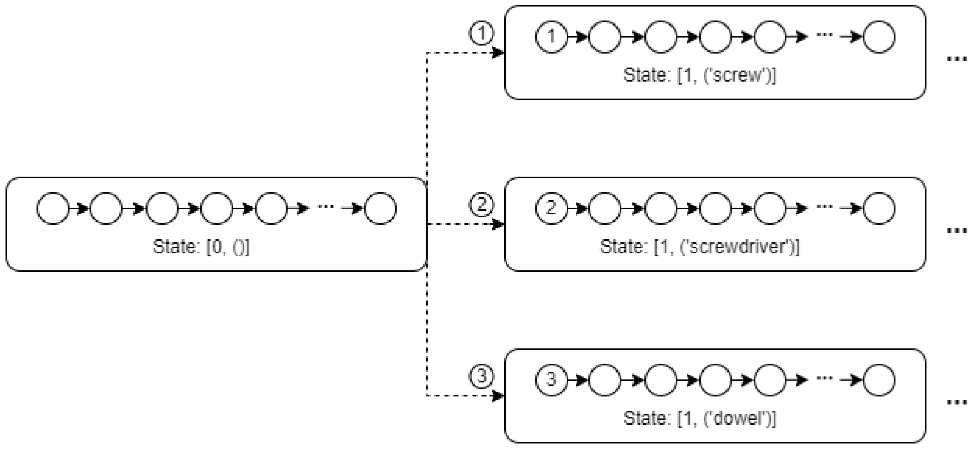


Figure 5.3: MDP’s states and actions scheme.

Some states must not be performed due to task precedence, e.g. (2, [’screw’, ’assemble’]), as a base cannot be assembled with a leg without both being built. However, these states are still considered during the learning process, but penalized, in order to allow the robot to fully comprehend the connections between tasks. The idea is to assign low rewards to those states, making the robot more robust to avoid undesired sequences.

As detailed in Chapter 2, it is time-consuming for the Q-learning algorithm to learn a considerable amount of state-action pairs accurately, as it needs more episodes to pass through all state-action pairs. For that reason, we created one simpler scenario called **two-leg-table assembly task**, where a table only has two legs. This is discretized into **15 subtasks**, and the state space size is reduced to 49557, compared to the original table assembly task.

5.2.2 Q-learning implementation

After defining the state and action space, the Q-Learning algorithm was continuously implemented using Python [63]. The Q-value update rule at each step throughout an episode is defined using the Bellman Equation 2.16. This Equation consists on the weighted average of the old Q-value with the new Q-value, where α represents the learning rate, γ the discount factor, r_t the received reward when transitioning from state s_t to s_{t+1}, Q(s_t,a_t) the old Q-value of taking action a_t in state s_t, Q(s_t,a_t) the new state-action pair Q-value, and max[Q(s_{t+1},a)] the estimate of the optimal Q-value at step *t+1*.

5.2.2.1 Hyper-parameters

The learning rate's value ranges between 0 and 1 and this hyper-parameter controls how the new state-action value influences the old state-action value, where a lower learning rate leads to a longer learning time, and a higher learning rate may lead to sub-optimal results or even divergence. The discount factor determines the significance of future rewards, where the lower its value the less meaningful they will be. This means that, if the discount factor is 0, only the immediate reward is considered, and as it approaches 1 the future rewards are more taken into account.

The selection of an action by a robot agent is made using an ε-greedy policy, i.e. the agent selects a random action with probability ε, otherwise selects the action with the highest Q-value value. The value of ε decays exponentially at each iteration based on a decay rate known as exploration rate decay or epsilon decay.

The training phase takes place over a number of several episodes (a sequence of states, actions and rewards), each one starting with the environment set to an initial state and the agent's reward reset to zero, and ending when it reaches a terminal state. In our scenario, an episode starts with no actions executed, and ends when all subtasks have been successfully completed (27 iterations for the table task and 15 for the two-leg-table) or when the maximum number of iterations has been expired.

The learning process needs to run until the agent is capable of solving the task in the testing phase. The number of episodes required to complete each training phase must be then dictated by experiments. Its value is represented by the maximum number of episodes. At the end of the train, the agent must be able to generate a feasible sequence of actions based only on the Q-Values, selecting at each state the action that corresponds to the maximum Q-value. There are more than one possible solution, meaning that several trains can retrieve different sequences, but each train generates always the same learner assembly sequence. Such sequence can be for example starting by bringing all objects, followed by building legs and base and ending in assembling all legs.

5.2.2.2 Reward function

The agent's final goal is to maximize the cumulative reward in the long run, i.e. maximize the sum of Q-values of all state-action pair triggered from the selected assembly sequence. Furthermore, the system must generate only realistic scenarios, i.e. assembly sequences with no repeated actions and that satisfies precedence dependencies. Therefore, at each iteration during the training phase, when an action is triggered, the algorithm must verify if this action meets the required conditions and if not, subtract a penalty to the reward value r_t.

To ensure that the task progresses, the agent must use a progress function $\ln(S_E)$ of the global assembly task to guide itself through the decision-making. In this way, the agent is encouraged to pay closer attention to the progress of the global assembly task. Whenever the Q-Learning agent selects a repeated action, the algorithm considers that the current state does not change, meaning that the sequence is penalised to require exactly the number of sub-tasks in the graph.

Taking all mentioned factors into account, the final reward function is given in Equation 5.1:

$$r_t = \ln(S_E + bias) - PrecedencesPenalty - RepeatedPeanalty \tag{5.1}$$

The RepeatedPenalty represents the penalty received for a selected repeated action, PrecedencesPenalty corresponds to the penalty given for not satisfying precedence dependencies, and ε is the bias of S_E, with a value of 0.1, to avoid the problem caused by null progress at the beginning of the task, just as mentioned by *Zhang et al.* [68].

To penalize repeated actions, the system stores a list of actions that were already executed and verifies if a selected action is already on this list. To certify that an action meets the precedence dependency condition, the algorithm looks at the list from the current state's tuple which stores the objects in the workspace, and hence know if the action can be executed. The progress state of the task S_E is calculated by $\frac{N_{actions\_done}}{N_{total\_actions}}$, where $N_{actions\_done}$ is the number of unique actions executed from the beginning of each episode and $N_{total\_actions}$ the total number of sub-tasks in the table assembly task.

The default values of the reward variables for both scenarios are presented in Table 5.1.

Hyperparameter	**Value**
PrecedencesPenalty	1
RepeatedPenalty	1

Table 5.1: Default reward values of the Q-learning algorithm

The defined reward function does not assure time efficiency in the collaboration. For that, we must add a new parameter, SubtaskTime, that penalizes the total time of the subtask, as in Equation 5.2. This parameter is a sum of the time, in seconds, the given actor takes to execute each operation contained in the subtask, as follows:

$$SubtaskTime = \begin{cases} \sum o(time\_operator), \forall o \in S, & \text{if actor is operator} \\ \sum o(time\_robot), \forall o \in S, & \text{if actor is robot} \end{cases}$$

Where S is the subtask's operations and o is an operation.

$$r_t = \ln(S_E + bias) - PrecedencesPenalty - RepeatedPeanalty - SubtaskTime \tag{5.2}$$

The value returned from the reward function is used in the r_t attribute from the Bellman Equation 2.16.

5.2.3 Deep Q-Network

After implementing Q-learning, to help the agent learn which action to perform and consequently complete the assembly task with less time and memory, we implemented DQN, a DRL algorithm which combines DL with RL. DQN in an extension of Q-learning, using a CNN to represent the policy in the close-loop interaction and approximate the Q-value function. The state is given as the input to the network, and the Q-value of all possible actions is returned as the output. This algorithm supports high-dimensional inputs better than Q-learning.

As explained in Chapter 2.3.1, DQN uses two techniques to solve stability issues brought by the integration of a CNN: experience replay and target network. Experience replay breaks high correlations between consecutive iterations by extracting random mini-batches from the agent's past experiences during learning time. This allows the network to keep memory of rare occurrences and repeat individual experiences. A target network is a copy of a Q-network whose weights are periodically synchronized with the Q-network's parameters, improving the stability of the network.

5.2.3.1 Environment setup

To develop the DQN, we resorted to the **Gym** toolkit [6]. Gym is a simple open-source Python library (`https://github.com/openai/gym`), created by OpenAI, for developing and testing RL algorithms [13] and that has been significantly adopted among the ML community. It provides a standard API and a standard set of environments that are easy to set up and are dedicated to testing and validating learning agents. The main purpose is to create standardization in papers and a universal benchmark, increasing reproducibility on different tasks and scenarios. This framework allows researchers to focus more on implementing the algorithm rather than the implementation details of a specific environment [48].

Although OpenAI Gym provides us a pack of environments, it also gives us the opportunity to **create a custom environment** for our solution, through the Env class. It is a class developed in Python that implements a simulator that executes the environment we want to train our agent in.

The observation and action spaces in a Gym environment can be formatted in several data structures called "Spaces" [47], depending on the scenario. In our implementation, both the observation and action spaces are represented as an **OpenAI Gym Discrete Space** [46], a discrete space described by a Discrete(n) box containing a finite number of values between 0 and n-1. In our scenario, each element of the observation space represents a tuple containing the number of completed subtasks and a list with the objects/parts in the workspace, as previously defined in Section 5.2.1. The n thus corresponds to the total number of observation states. Each element of the action space, on the other hand, represents each action/subtask, and n corresponds to the total number of subtasks.

5.2.3.2 Network Architecture

Our network architecture defines the main and target networks as Sequential models from Keras [7]. Keras is a simple, flexible and powerful API written in Python that focuses on deep learning and runs on top of the end-to-end ML library TensorFlow [31].

The networks consist of 6 layers, with the first being an Embedding layer, a layer that turns positive indexes into fixed size vectors. The input to a neural network of DQN are the possible states of the environment. Hence, the dimension of the first layer's input refers to the number of states and output_dimensions refers to the vector space we are squishing it to.

In our implementation, we have 2807, 49 557 and 1 812 473 possible states, for the different assembly tasks, and we want them to be represented by 10 values. The length of the input for a discrete environment in Gym is 1 because the new state returned by the environment given an action taken in the current state is a single number.

The next layer is a Reshape layer, where we take the output from the previous layer and reshape it to a a one-dimensional vector. In our case, a one dimensional array with 10 values. The model is followed by 3 consecutive dense layers. Keras provides us an helpful tool to find the best hyperparameter values for our model, the kerasTuner [45], saving us time in the optimal hyperparameter search. We used KerasTuner to find the optimal number of units and the activation function to use in our model's Dense layers. The goal of the tool was to minimize the loss value.

In Deep Q-learning, the output of the network must be the action to be taken. Therefore, its last layer is a Dense layer with a linear activation and with the same output size as our action space size.

We compile the model using Adam optimizer. In the Deepmind paper [36], the authors compile the model with a RMSProp as the model optimizer. However Adam improves training time, since the parameter updates are estimated using running averages. As a loss, we use the Huber loss function rather than Mean Squared Error to achieve better performance, since Huber weighs outliers less.

5.2.3.3 Training Algorithm Setup

We start our training algorithm setup by initializing our main and target neural networks, as well as our environment. In addition to the observation and action spaces, the Gym platform also defines the environment **reset**. At the beginning of the training process and after each termination step, either because the episode was successfully finished or because the maximum number of iterations was reached, the environment is reset to its initial state, i.e. the progress state is set to 0 and the observation state is set to (0,[]).

The agent proceeds the training phase by **selecting an action** using an $\varepsilon - greedy$ policy. If a random number in the range of [0,1] is smaller than the value of the predefined ε hyperparameter, the agent chooses a random action from the action space. Otherwise, it exploits the best known action. To find such action, the two networks predict the Q-values for all actions in the current state and the maximum predicted Q-value is returned. After selecting an action, the agent can perform the action and use the Bellman Equation (Eq. 2.16) to update the main and target networks and transition the environment to a new state. This process is know as the environment **step** and it returns, along with the new state, a reward and a boolean value informing either the episode successfully reached the end or not, i.e. (next_state, reward, done).

To take advantage of the Experience Replay technique, the algorithm stores each episode's data into a memory buffer which size is predefined. This data includes the current state, the taken action and all the outputs returned by the step method, i.e. (current_state, next_state, reward, done). After, the agent shuffles the memory buffer and selects a batch size of experiences. Every N steps, the weights from the main network are copied to the target network, leading to a more effectively learning.

Having the right model parameter update has influence on the learning performance. If we update the weights too often (N is to small), the algorithm will learn very slowly when not much has changed. In our case, just like in Deepmind's implementation [36], we sample and train the main network samples on a batch of previous experiences every 4 steps, helping the algorithm to run faster. We also need to make sure that the target network is updated in the right frequency. We chose to copy copied the main network weights to the target network every 1000 steps, which turns out to be more stable than at every episode, since each episode can have a different number of steps. After all the algorithm's described steps, the model can be trained using the predicted Q-values and Q-target.

5.3 Simulation Module

After the training stage, the system can now simulate the sequence of actions with highest Q-values that the intelligence module returns. Within the simulation module there are also 2 sub-modules, the motion layer and the pose layer.

At each given action, the motion layer breaks down each operation within the action into one or several specific and discrete movements for the robot agent to perform. For example, for the pick dowel operation, the robot must move and/or rotate its arm to the object position, grab the object with the gripper, moving the joint, and lastly move the object to the desired position. These movements are sent to a previously implemented API connected, which decodes the movements and sends them, through sockets, to the V-REP simulator. For operations that does not require the robot to move, e.g. operations that are assigned to the other agent, the API "tells" the robot to wait for an undefined time, until the operator finishes his execution or until it is assigned to a subtask.

When the robot is executing a movement, it must know to where it must move. For that, there is a text file for each assembly task with each object's current pose and final pose. This final pose is the desired pose for the object relatively to the other objects. For example, one leg is supposed to be assembled on the base, and therefore its pose must be such that, given the current position of the base, whatever it is, the two pieces are assembled. Each pose is a 6-float coordinates (pose and orientation) and is recognized by the pose layer to send to the API along with the movements. Every time a movement is executed, the pose layer updates the object's current position on the file.

When the two agents perform two operations in a row of the same subtask, there are always a few seconds or milliseconds of agent swapping. This time is counted towards the total time of the task execution and displayed in the graph as a 'wait' node. Fig. 5.4 shows an example of wait nodes in a building base subtask, where the robot performs the position screw operation, the performer of the task then swaps to the human, executing fasten screw operations, and at the end it swaps back to the robot, placing the base.

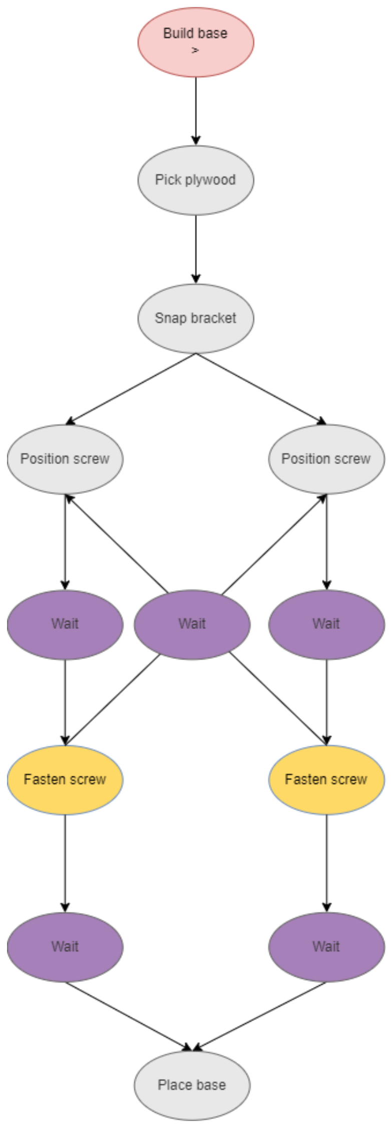


Figure 5.4: Example of a subtask with wait actions

During the simulation, for the operator to better perceive the status of the task, the graph nodes are "painted" with different colors depending on the respective task, subtask or operation status. If the task/subtask/operation correspondent to the node has not been performed, the node is colored by gray, if it is in the process of execution, it is illustrated by yellow, and green if it is finished. The actor who performed the task is also shown in the graph, along with the time taken to perform the operation or subtask related to the executed node.

During the simulation, the time_robot and time_operator parameters contained in the JSON file are also updated after every operation execution with the real time taken to execute it.

Given all the factors mentioned above, we can notice that the simulation module "feeds" the graph module with information to update the graph visualization.

At the end of the simulation, a log file is generated with all the information about the simulation, including:

- the total assembly process time;
- the number of operations performed by each agent;
- the total time of each agent's activity;
- sequence of executed operation;
- the resource assigned to each operation;
- the time taken to execute each operation;
- possible errors occurred during the simulation;

The overall activity time taken by each agent can be useful, for example, for the human to know if it is overloading the robot.

There are three ways of performing the table assembly task: manually, automatically or collaboratively. If manually, the task is completely performed by the human agent. In the automatic option, the task is completely executed by the robot (with the exception of the fastening operation, which only can be done by the human operator). In the collaboratively option, the two agents (robot-robot or human-robot) work together to collaboratively complete the task, using the implemented artificial intelligence.

In the manual and automatic options, when an agents fails the performance of an operation or there is an external error, the graph suggests the agent to repeat the action. In the collaborative option, failures are not addressed.

5.4 Chapter Synopsis

We started this chapter by explaining the **graph module's implementation**. Similarly to *Mangin et al.* [29], we define our task's graph structure as a **hierarchical task structure** called HTM, which is a simple and appealing form, close to human intuition that has shown good performance in allowing the agents to communicate and understand each other's actions. This representation of our collaborative assembly tasks is possible given their easy decomposition into smaller subtasks.

Our graphs are composed of nodes and links, where the root is the task and, as the hierarchy moves downwards, the task is divided into subtasks, which in turn are divided into concrete actions. Each node of our graph is composed by the following three elements:

- **id**: unique node identification;
- **type**: classifies the node as *task*, *subtask* or *operation*;
- **state**: determines whether the node is still to be, is being, or has already been executed. It allows the agents to perceive the task's evolution;

Operation type nodes, in addition to these, contain four more parameters:

- **operation**: indicates which action the node refers to (*pick*, *place*, *snap*, *position* or *fasten*). Its value must then be a string;
- **object**: indicates which object the node refers to, e.g. screw (string value);
- **time_robot**: Estimated time of operation's execution by the robot (integer value);
- **time_operator**: Estimated time of operation's execution by the robot, depending on his/her expertise (integer value).

The sub-task and task type nodes may also include an additional element, **sequence**, that expresses whether the node is *sequential*, *parallel* or *alternative*, i.e. whether a task's subsequent sub-tasks or a sub-task's subsequent operations might be done at the same time by two different agents (**parallel**), must be executed by a specific order of execution (**sequential**) or must be performed exclusively (**alternative**). This set of elements demonstrates the type of constraints that may exist when executing a collaborative task along with the complexity of the collaboration in a human-robot or a robot-robot team. In our graphs, the parallel operation is represented by $\|$, the sequential by $\rightarrow$ and the alternative by $\mathbf{V}$.

Our graph's data is built and loaded in JSON format. NetworkX [14], a Python library used for the creation, manipulation and visualization of our graphs, parses the JSON serializable data and maps it into a Networkx graph [22].

A MDP environment is composed of an observation space and an action space. Our observation space corresponds to the evolution of the task's status, i.e. each space is a tuple composed of **the number of completed subtasks** and **a list with the objects/parts in the workspace**. The last element is important, as objects and parts that are in the workspace influence the feasible sequences of actions. The initial state corresponds to no actions executed and no objects/parts in the workspace and the final state to all actions executed and all objects/parts in the workspace. An action in our MDP environment corresponds to a tuple composed of a sub-task and the actor who will perform the sub-task.

For the robot intelligence module, we started by implementing **Q-learning** algorithm, since the Q-table provides us easier analysis and interpretation of the environment's behaviour and the overall training process results and evolution. We created one simpler scenarios called **two-leg-table assembly task**, where a table only has two legs and is decomposed in **15 subtasks**. The idea is to create tasks that are less time-consuming for the algorithm.

Q-learning takes into account a list of hyperparameters, which values will influence the results of the training process. We started by defining default values, shown in Table 5.2, and will test some variations until the algorithm converges and returns a possible sequence of actions. The system must generate only realistic scenarios, i.e. assembly sequences with no repeated actions and that satisfy precedence dependencies. Therefore, and since the agent's final goal is to maximize the cumulative reward in the long run, the system must penalize the sequences that do not meet the required conditions. On the other hand, to ensure that the task progresses, the environment must reward the status of the global assembly task to guide the agent through the decision-making. The states that do not satisfy precedence dependencies are still considered during the learning process, but penalized. The aim is to assign low rewards to those states, making the robot more robust to avoid undesired sequences.

Hyperparameter	**Value**
Learning rate (α)	0.9
Discount factor (γ)	0.9
Exploration rate (ε)	1
Exploration rate decay	0.0001
Min exploration rate	0.1
Max iterations per episode	18
Max number of episodes	50 000

Table 5.2: Q-Learning hyperparameters' values

We followed the robot intelligent module's implementation by training the agent with **DQN** algorithm, as this algorithm supports high-dimensional inputs better than Q-learning. We now define the MDP environment using the OpenAI's **Gym** [6] framework. Our observation and action spaces were represented as an **OpenAI Gym Discrete Space** [46] with n elements. In the observation space n corresponds to the total number of observation states, and in the action space n corresponds to the total number of subtasks.

The main and target networks of our DQN are Sequential models from Keras [7] that consist of 6 layers. The input to a neural network, in DQN, must be the possible states of the environment and output the action to be taken. Therefore, the first layer's input has the dimension corresponding to the size of the observation space, and the last layer as the output dimension corresponding to our action space size. In the middle of these layers, the networks contain a Reshape layer followed by 3 Dense layers. Some optimal hyperparameters were found using a Keras' tool named kerasTuner [45].

In the end, the robot intelligence module returns an algorithm capable of providing the action with the highest cumulative reward (highest Q-value) for each given state to the **simulation module**. The agents simulates the sequence of actions for the given collaborative assembly task. But for that, it needs the **poses of each object**, which are given in a text file. Each operation the robot must perform is divided into one or several **specific and discrete movements** and sent to a previously implemented API connected, which decodes the movements and sends them, through sockets, to the V-REP simulator.

The simulation module introduces a new node type to the graph, the **wait node**, which represents the time "lost" between an exchange of actors in the execution of a subtask. During the simulation, the graph nodes are colored with different colors depending on the node status, i.e. the respective task/subtask/operation has not been performed, is in progress, or is finished.

In addition to a collaborative assembly, our tasks can also be performed manually or automatically, i.e. the task is completely performed by the human agent or executed only by the robot (with the exception of the fastening operation, which only can be done by the human operator). In these two options, when an agents fails the performance of an operation or there is an external error, the graph suggests the agent to repeat the action. In the collaborative option, failures are not addressed.

In the end of our simulation, we have a system composed of three modules, where each module feeds the others, directly or indirectly, with the necessary data for they to successfully run.

The complete graph for the chosen scenarios are depicted in Appendix A, where, in the table task graph, we can see an example of a subtask with wait nodes.

Chapter 6

Tests and Results

After describing the MDP environment and the algorithm specification in Chapter 5, we have everything we need to start testing and validating our solutions in our defined case study (table assembly task: four-legs and two-leg-table), first with the Q-learning algorithm and later with the Deep Q-learning technique. Our tests were organized in two different experiments: (i) agent has the objective of learning the complete feasible sequence of the tasks, with no prior knowledge, and (ii) the agent tries to learn the sequence while also trying to minimize the global time to complete the task.

6.1 Q-learning

6.1.1 Feasible assembly sequences

We started by testing the **two-leg-table assembly** with the first implemented algorithm, **Q-learning**. The experiments were conducted with the hyperparameters defined in Table 5.2. The experiments were repeated **100 times**, where in each one of the experiments one assembly sequence is learned. We compare the efficiency of the agent in learning **feasible assembly sequences** through the running average reward evolution and the percentage of **possible assembly sequences** learned.

In this scenario, as we can see from Fig. 6.1, the running average reward **converges** until the last episode, which is the desired behaviour. However, from the generated assembly sequences, it was visible that the agent could not converge and learn any feasible sequence until the task was completed, and 53% of the times, it could only learn the first 5 actions with the maximum Q-value.

These results indicated that the agent needed more time to learn. Therefore, we gradually increased the number of episodes by 10 000 until, until we reached 100 000 episodes. However, as Fig. 6.2 illustrates, the running average reward **did not converge** anymore. In 100 experiments, the agent does not successfully learn any feasible assembly sequence, and in 46% of the trials it only learns a desirable sequence until the 7th action.

This stagnation was the result of the reward not changing its value significantly after a certain number of actions, which did not guide the robot for convergence. Also it was clear by analyzing

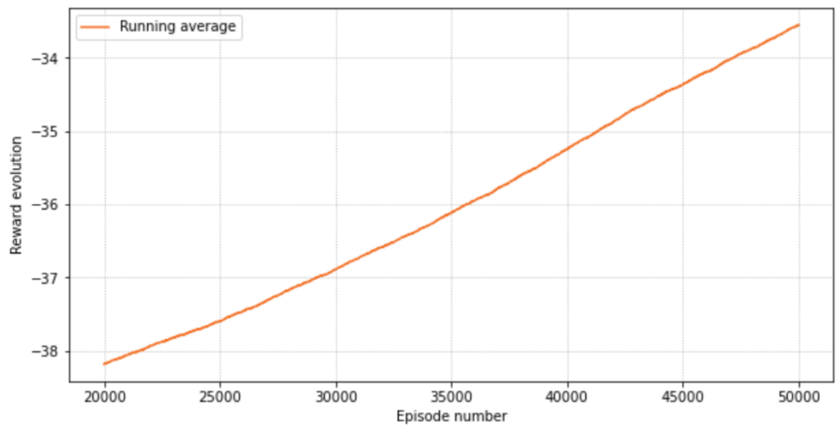


Figure 6.1: Running average reward evolution for the two-leg-table assembly task with 50 000 episodes and the default hyperparameters.

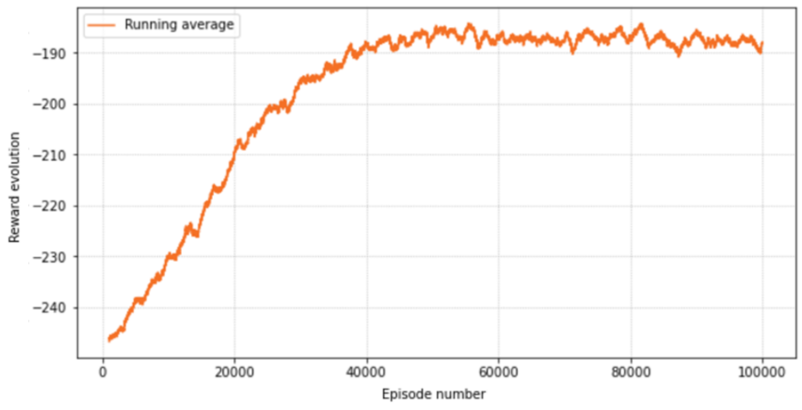


Figure 6.2: Running average reward evolution for the two-leg-table assembly task with 100 000 episodes and the default hyperparameters.

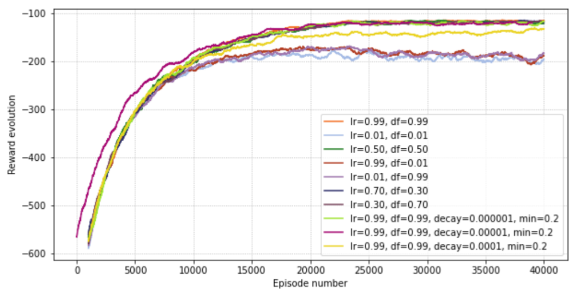
the choices of actions that the agent needed to explore more and thus the epsilon decay needed to be larger, where the values of the learning rate and/or the discount factor also have some impact.

To better understand the behaviour of the algorithm, as Q-learning is still expensive in terms of time and memory for 100 000 episodes, we created a third scenario, **one-leg assembly task**, that contains only one leg and, which requires only one bracket T and one dowel, reducing even more the number of actions to **9** and the **state space** to the size of **2807**. The complete graph for the this scenarios is exemplified in Appendix A.

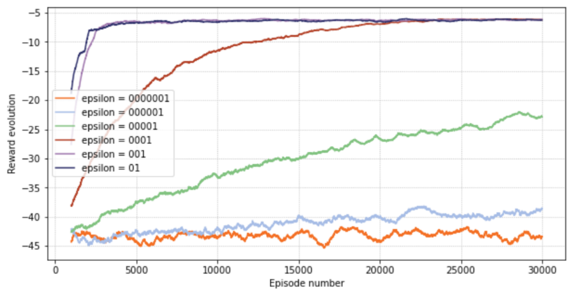
We tested the scenario in several experiments with different parameters, as shown in Fig. 6.11.

First, we analysed the reward evolution on different exploration rate decays. This value will influence how much the exploration rate decays at each time step, and consequently how much the agents will explore or exploit. The lower its value, the slower the ε will decay and with higher probability the agent will explore rather than exploit. As we can see from Fig. 6.3b, high ε values led to quicker convergences, as well as higher rewards.

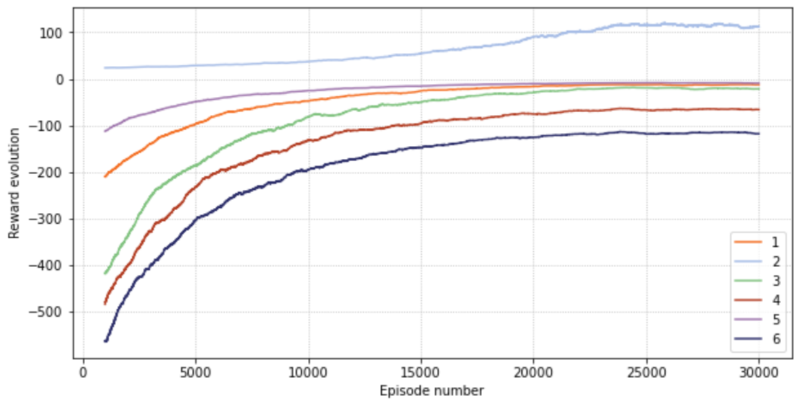
After, we examined the running average reward evolution for different sets of reward weights



(a) Different sets of learning rates (lr) and discount factors (df).



(b) Different values of exploration rate decay (epsilon)



(c) Different reward values. (1) PrecedencesPenalty and RewardPenalty are multiplied by 5. (2) ProgressState is multiplied by 10. (3) PrecedencesPenalty and RewardPenalty are multiplied by 10. (4) PrecedencesPenalty and RewardPenalty are multiplied by 20. (5) PrecedencesPenalty and RewardPenalty are multiplied by 20 and ProgressState is multiplied by 10. (6) PrecedencesPenalty and Reward-Penalty are multiplied by 20 and ProgressState is multiplied by 20.

Figure 6.3: Impact of the RL hyperparameters and reward values on the running average reward evolution for the one-leg-table assembly task with 40 000 episodes

and the results are presented in Fig. 6.3c. As expected, the trial with a progress state with a weight of 10 was not well succeed, as the agent was not able to learn which state-actions pairs lead to higher rewards or not. The processes trained with higher reward penalties return lower rewards in the first episodes. However, the overall convergence curve is sharper and the evolution is higher.

At last, we explored different sets of learning rates and discount factor. As we can see from the results in Fig. 6.3a, with the sets values of [$\alpha = 0.01$, $\gamma = 0.99$], [$\alpha = 0.99$, $\gamma = 0.01$] and [$\alpha = 0.99$, $\gamma = 0.01$] the agent started to diverge its average running reward from the others from approximately episode number 10 000, slowing down the its evolution. This experiments were not successful in finding possible assembly tasks. We can conclude from this results that learning rates or discount factors with lower values lead to worst results from the training process.

Furthermore, we introduced a **learning rate decay**, which main idea is similar to the exploration rate decay. Initially, the agent trains the network with a large learning rate, then it is gradually reduces it until a local minima is reached. Both optimization and generalization usually benefit from it empirically. We applied an exponential decay function so α decays exponentially over time. The results show that the integration of a decay function did not cause much change in the reward curve, and as the decay rate increased, the reward converged less. This means that our algorithm behaves better with high learning rates and discount factors.

After exploring the values, we ran 100 experiments with a reward where each variable is multiplied by 20 and an exploration rate decay of 0.00001 obtaining the results presented in Fig. 6.4. In this scenario, the agent learned a feasible solution in all 100 experiments, and the running average reward evolution converges.

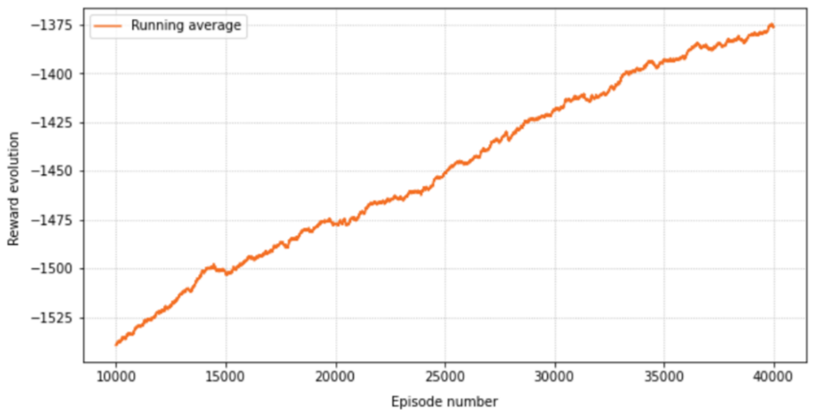


Figure 6.4: Running average reward evolution for the one-leg-table assembly task with 40 000 episodes and higher reward values

However, 40 000 episodes was still a high number of episodes for 2 807 states, and as we can see from the previously results where the exploration rate decay was 0.001 and 0.01, the reward converged before the episode 5000 and stagnated until the end, while generating feasible assembly sequences. We then increased the exploration rate decay to 0.001 and obtained the running average reward evolution illustrated in Fig. 6.5. In 100 runs, the agent could learn 100% of feasible solutions.

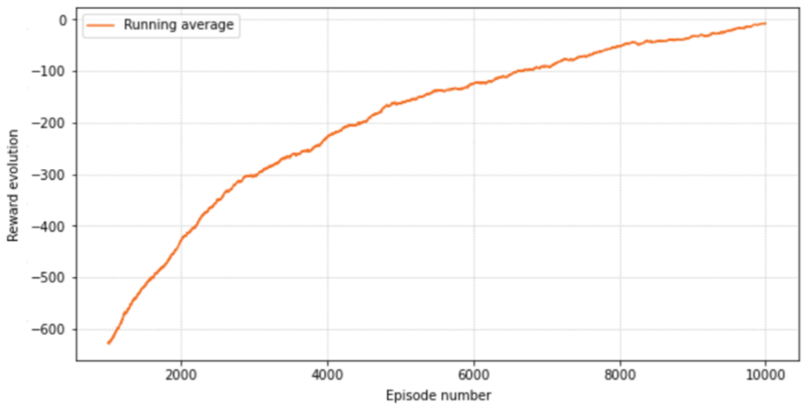


Figure 6.5: Running average reward evolution for the one-leg-table assembly task with 10 000 episodes and higher reward values and an ε decay of 0.001

Still trying to reduce the number of episodes needed to converge, we increased even more the exploration rate decay to 0.01 and reduced the number of episodes until the percentage of feasible assembly sequences learned was less than 100%. We reached 400 episodes with the reward evolution shown in Fig. 6.6.

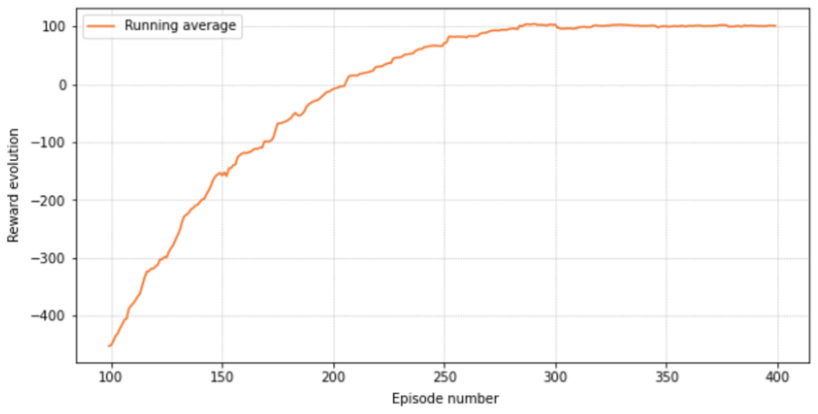


Figure 6.6: Running average reward evolution for the one-leg-table assembly task with 400 episodes, higher reward values and an exploration rate decay of 0.01.

After all the trials and comparisons, the set of hyperparameters that achieved the optimal reward evolution and results with the one-leg-table assembly task is the one expressed in Table 6.1. Table 6.2 lists the reward values and weights that lead to the **optimal results**. In 100 runs, the agent was able to learn feasible assembly sequences 100 times (≅ **100%**).

Hyperparameter	**Value**
Learning rate (α)	0.99
Discount factor (γ)	0.99
Exploration rate (ε)	1
Exploration rate decay	0.01
Min exploration rate	0.01
Max iterations per episode	12
Number of episodes	400

Table 6.1: Optimal Q-Learning hyperparameters' values for One-leg-table assembly task

Reward variable	Value
Progress State	20
Discount precedences	20
Discount repeated	20

Table 6.2: Optimal weights for the reward function's parameters

Returning to the **two-leg-table assembly task**, we adapted the previously obtained optimal parameters to this case study, increasing the exploration rate decay to 0.00001 and maintaining the number of episodes to 50 000, compared to the default values in Table 5.2. The reward converged, as shown in Fig. 6.2 with 100% of the expected sequences generated.

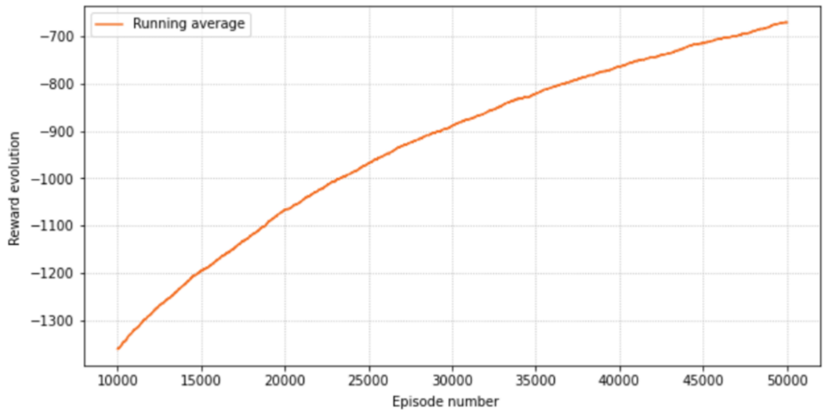


Figure 6.7: Running average reward evolution for the two-leg-table assembly task with 50 000 episodes with higher reward values and an exploration rate of 0.00001.

We continued to explore the behaviour of the algorithm by increasing the exploration rate decay to 0.00005, from which the agent obtained the desired results training in 35 000 episodes. The reward evolution is illustrated in Fig. 6.8.

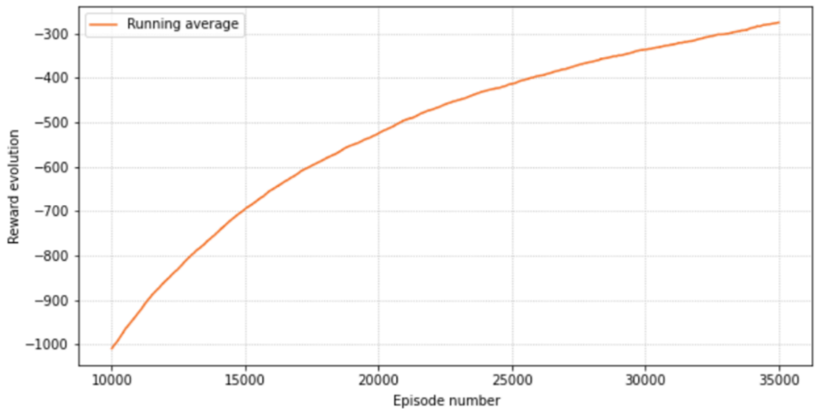


Figure 6.8: Running average reward evolution for the two-leg-table assembly task with 35 000 episodes and a exploration rate decay of 0.00005.

Finally, as shown in Fig. 6.9 the minimum number of episodes from which we can train the agent to optimize task sequence was **27 000 episodes**, with an exploration rate decay of **0.001**.

Table 6.3 summarizes the **set of hyperparameters that lead to optimal results** with the possible minimum number of episodes for the two-leg-table assembly task.

For the **table assembly task**, we used the parameters presented in Table 6.4. From Fig. 6.10 we can see that the reward is **converging**. The agent could learn a feasible sequence until the action number **17** in a total of 27 actions. Unfortunately, as Q-learning is an algorithm not suitable for high-dimensional problems, it takes days to run in this scenario, so we could not obtain results for more episodes.

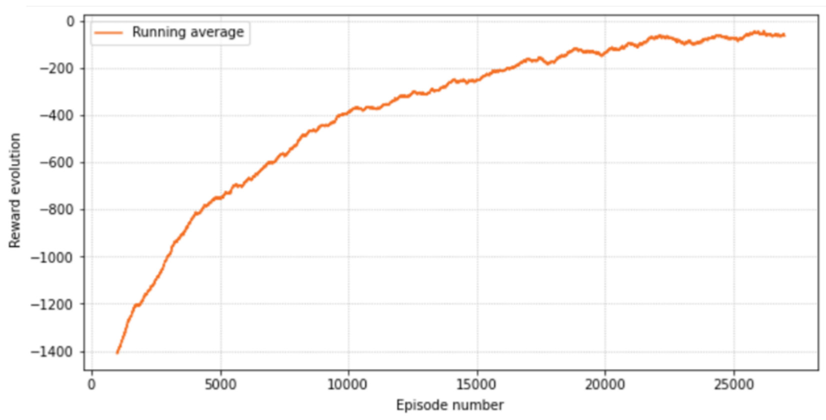


Figure 6.9: Running average reward evolution for the two-leg-table assembly task with 27 000 episodes and a exploration rate decay of 0.001.

Hyperparameter	**Value**
Learning rate (α)	0.99
Discount factor (γ)	0.99
Exploration rate (ε)	1
Exploration rate decay	0.001
Min exploration rate	0.01
Max iterations per episode	18
Number of episodes	27 000

Table 6.3: Optimal Q-Learning hyperparameters' values for Two-leg-table assembly task.

Hyperparameter	**Value**
Learning rate (α)	0.99
Discount factor (γ)	0.99
Exploration rate (ε)	1
Exploration rate decay	0.00001
Min exploration rate	0.01
Max iterations per episode	30
Number of episodes	40 000

Table 6.4: Optimal Q-Learning hyperparameters' values for Table assembly task

6.1.2 Subtask allocation

As explained in Chapter 4, the **time** it takes for a human operator to execute an operation depends on his/her expertise. An **expert**, in average, performs the tasks with a **lower time and a lower variance**. A **newbie**, on the other hand, needs **more time to complete** a task and usually that time **varies more**. An **intermediate** takes **more time** with higher variance than an expert, but **less time** with lower variance than a beginner. A task graph carries the time it takes for an operator, given its expertise, to perform an action. With this information, the agent is able to **allocate the tasks to actors** by penalizing the time in the reward function. The goal is to **reduce the overall task time**.

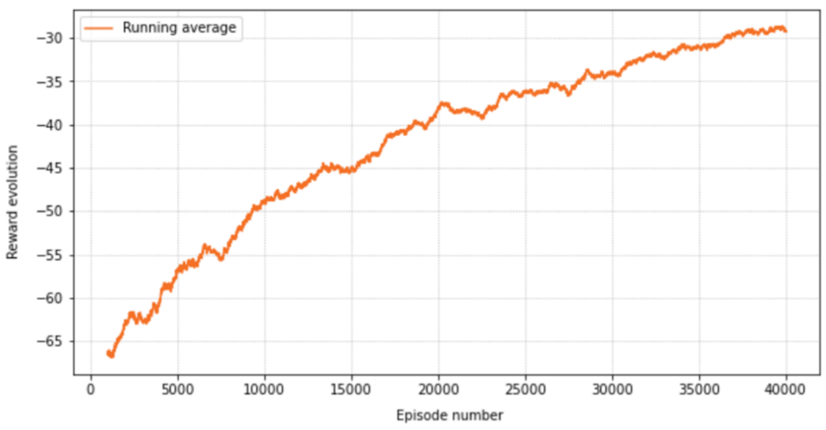
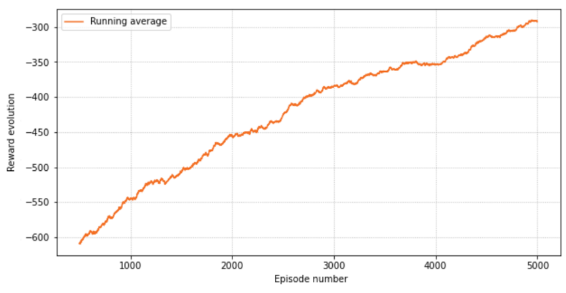
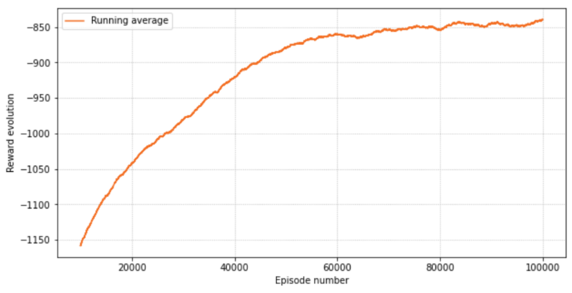


Figure 6.10: Running average reward evolution for the table assembly task with 40 000 episodes.



(a) 5 000 episodes and an ε of 0.0001.



(b) 100 000 episodes and an ε of 0.000001.

Figure 6.11: Running average reward evolution considering task allocation for (a) the one-leg-table and (b) the two-leg-table assembly tasks.

The reward function used is the one given by Equation 5.2. However, as we still want the agent to learn feasible sequences and assure progress, the SubtaskTime variable's value can not be higher than the other variable's values at each time step. And as the time of each action is in seconds with a range of [4,80], we need to scale it down. We multiplied by values in the range of [0, 1] and received good results with 0.01 for the task with 9 actions and 0.0001 for the task with 15 actions. We also had to decrease the exploration rate decay to 0.0001 and 0.00001 and increase the number of episodes to 5 000 and 100 000. Good results in this case means converging the average running reward, while keeping to generate feasible sequences. Figures 6.11a and 6.11b demonstrate the reward evolution for both scenarios.

To evaluate the algorithm's decisions, three performance metrics are used:

- The time required to finish the global assembly task;
- The operator's working time on the assembly task;
- The number of operations assigned to the operator.

We ran 300 experiments, in which for every 100 executions the operator has a different **expertise** (beginner, intermediate and expert). We compared the percentage of actions assigned to each actor and the results are shown in Table 6.5.

As expected, even considering the time taken to switch between actors when the operator is required to fasten screws, the robot was assigned to more actions if its human partner is a beginner, since the robot takes, on average, less time to complete the task. An intermediate operator has a robot-like assembly speed and hence both agents have a similar percentage of subtasks allocated. An expert, on the other hand, has more experience and ease in performing assembly tasks and is assigned to more actions.

Operator's Expertise	**Actor**	**Percentage of actions allocated**
Beginner	Human	≅ 0.071%
	Robot	≅92.87%
Intermediate	Human	≅ 50.9%
	Robot	≅49.1%
Expert	Human	≅96.0%
	Robot	≅0.04%

Table 6.5: Percentage of actions allocated for each actor at each operator's expertise.

Table 6.6 details the **mean** and **standard deviation** of the **global task** completion time with or without the robot learning task allocation, and also considering each operator's expertise. We can see that without learning task allocation, the global assembly duration is higher and has more variance than with the agent learning task allocation. This happens because without considering the time, the agent assigns the actor randomly, instead of assigning the one that executes the actions in less time. As the expertise of the operator increases, the mean duration of the task, considering optimal task allocation, decreases, as expected. The standard deviation of task time in the experiments with a beginner operator and with learning task allocation is lower, because the operations are mostly assigned to the robot, which is more constant, due to its automation, than a human in performing tasks.

Operator's Expertise	**With/without learning task allocation**	**Mean**	**Standard deviation**
Beginner	With	135.2	0.23
	Without	143.22	3.46
Intermediate	With	126.58	0.42
	Without	128.04	1.26
Expert	With	73.25	0.42
	Without	80.7	5.32

Table 6.6: Mean and standard deviation of the global task duration, in seconds

An interesting result we obtained for the two-leg-table assembly task was the one which running average reward is illustrated in Fig. 6.12, where from around the 15 000 episode, the agent stopped learning and started to drop in reward value, with an exploration rate decay of 0.00001 and a multiplication of 0.01 by the time of each action. This results may be due to the fact that from episode 15 000 the agent only explored actions and states which progress was not compensating for the given penalization (task allocation, repeated sequence and precedence dependencies).

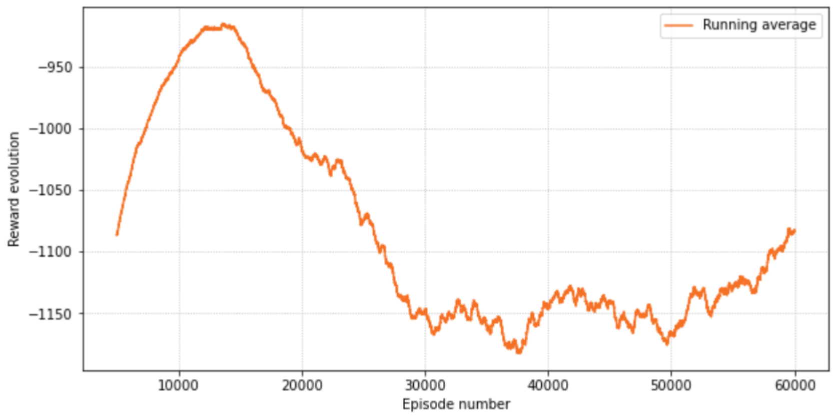


Figure 6.12: Experiment the two-leg-table assembly task where the running average reward evolution decreased.

Unfortunately we could not obtain results for the table assembly task with 27 actions due to the limitations of Q-learning within high-dimensional spaces.

6.2 Deep Q-Network

To find the best parameters for the Dense layers of our model, we used a Keras tool called kerasTuner. In 10 trials of 200 episodes each, kerasTuner returned 384 for the optimal number of units and relu for the activation function to use in our model's Dense layers. This values returned the minimal loss value.

For the one-leg-table assembly task, with the parameters found in Q-learning for the same task (defined in Table 6.1), the algorithm generated a running average reward illustrated in Fig. 6.13. As we can see the reward converges in 5 000 episodes.

With 15 actions (two-leg-table assembly task), with the optimal parameters defined in Table 6.3, we obtained the reward evolution presented in Fig. 6.14, converging in 40 000 episodes.

The expected results, given the high-space dimensionality of the case studies (2807 for the one-leg-table task, 49 557 for the two-leg-table task and 1 812 473 for the table task) would be a faster convergence with the DQN algorithm. However, this was not verified, even after optimization of parameters using the kerasTuner tool. From our intuition, this happened due to the complexity and number of parameters that are needed to optimize, in the DQN network, which due to time constraints it was not possible to evaluate the impact of various combinations of parameters, thus

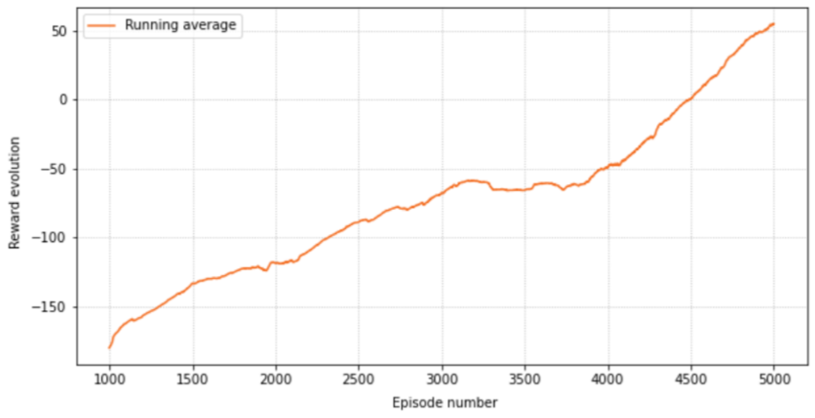


Figure 6.13: Running average reward evolution for the one-leg-table assembly task with Deep Q-network

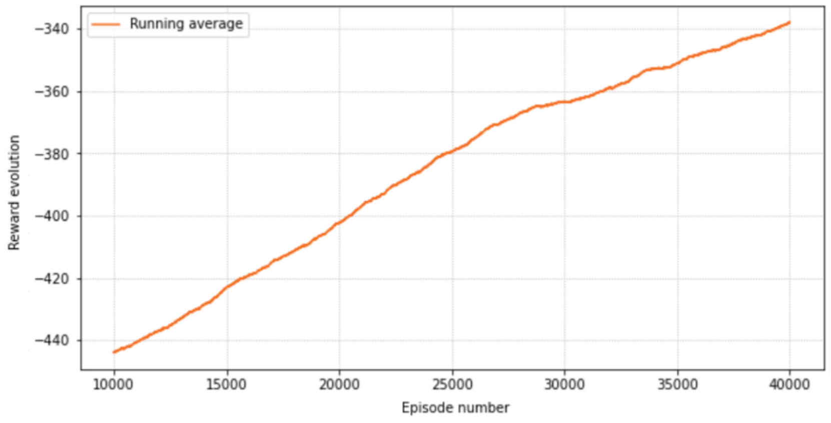


Figure 6.14: Running average reward evolution for the two-leg-table assembly task with Deep Q-network

not being possible to achieve the optimal network that would lead to a faster convergence by this algorithm.

For the table assembly task, with 27 actions, we obtained the results from Fig. 6.15, where the reward seems to be converging until around the episode number 20 000, although with more episodes it reduces the reward value. Given this we made used early stopping to achieve the best DQN model weights, before it started overfitting.

6.3 Graph and Simulation

The simulation ran as expected, with the robot simulator performing the correct movements for each operation, and updating the graph at each step with the task status, i.e. which operations are being performed and which ones are completed. If an operation is being executed its correspondent node changes its color to yellow. If it completed is changes to green.

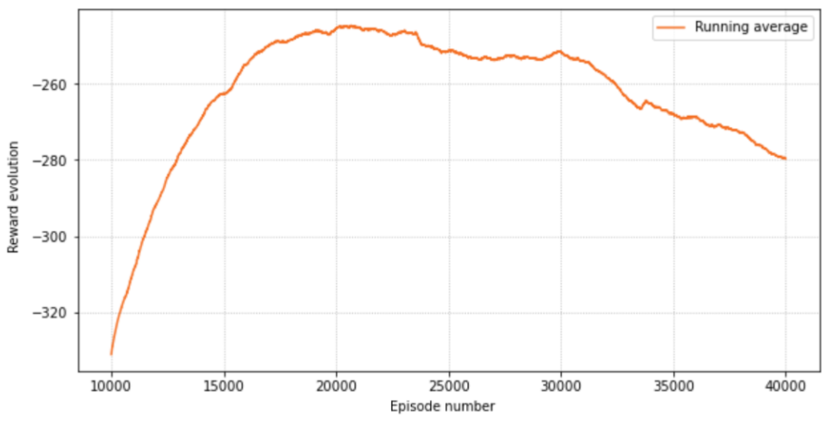


Figure 6.15: Running average reward evolution for the table assembly task with Deep Q-network.

At each step, if an operation requires the movement of an object, the simulation module gets the object's pose from a file, and updates it after the simulation execution. The time it takes to perform an operation is also updated.

In manual and automatic modes, an operator fails the performance of an operation with a probability, depending on its expertise and the operation itself. When a failures occurs, the operator must repeat its execution.

Fig. 6.16 shows part of a graph running during a simulation of the table assembly scenario. The subtask is to build a leg, and the figure presents its operations. Except for the *wait*-type node, each node contains information about its *type*, the *operation*'s name, the *actor* and the *time* the performance of the action last. The node of *wait*-type node only needs to inform about the waiting time between the switching of actors.

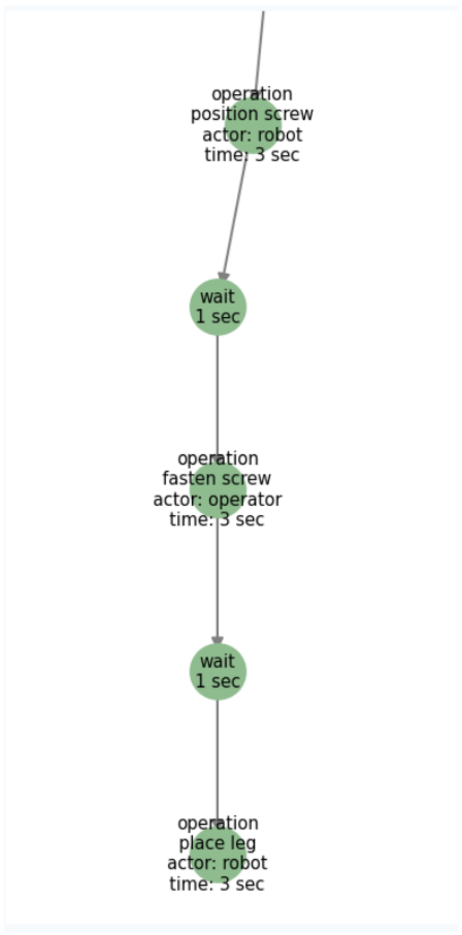


Figure 6.16: Part of a graph running during a simulation of the table assembly scenario. Each node contains information about its *type*, the *operation*'s name, the *actor* and the *time* the performance of the action last.

Chapter 7

Conclusions and Future Work

This chapter is reserved for conclusions and final reflections of the work of this dissertation. We also present the future work to be implemented.

Conclusions

Manufacturing processes, when complex, can be hard for both human operators and robots to complete independently. Humans do not have the model encoding capability that robots have, and robots can nor reach the level of thinking and understanding that robots do. A system capable of enabling humans and robots to collaborate and take advantage of each other's capabilities for task allocation and sequence optimization, using artificial intelligence and graph approaches, in collaborative processes would be benefit for manufacturing industries. However, there are still some challenges to overcome to achieve success and there are still few solutions that are applicable to the real world.

We started this long project by understanding concepts essential to the build our solution, namely concepts related to ML. We formulated the learning from trial and error methodology, RL, and how DRL can deal with complex tasks with high-dimensional spaces better than traditional RL algorithms. We also have seen how demonstrations from an expert and IRL methods, where the reward function is learned, can be useful for a learning agent. The analysis continues with some examples and similar solutions regarding ML approaches for collaborative task optimization and scheduling, as well as how graph structures can be integrated to improve collaboration.

We defined a case scenario of a a table assembly task, where the goal is to assemble a table with 5 structural parts and 15 components, performing 27 actions in total without repetitions and satisfying precedence constraints. Such precedence constraints must be given by the first module, the graph. Later, we defined two more scenarios: two-leg-table assembly task and one-leg-table assembly task. These scenarios are composed by a smaller number of actions and possible sequences, compared to the first specified case study. Each task has alternative sequences of subtasks, which in turn have sequences of operations, such as picking, placing, pinching, snapping, and positioning an object. At each iteration of a sequence, the graph is updated based on the environment. According to the algorithm paradigm, the agent decides the sequence and the entity that should perform each operation based on the observed state of the graph and a certain policy.

The simulation module used a UR5 industrial manipulator, a gripper and an object. The object pose must be given *à priori*, and as the simulation occurs, the poses are updated. The task status must also be updated in the graph, so that the agents are aware of each other's actions and the task progress. This poses can later be given by a vision system, as our solution is ready to easily incorporate the data.

We implemented two RL algorithms: Q-learning and Deep Q-learning. Q-learning is a model-free algorithm that aims to find the sequence of actions that gives the maximum reward, given a state. It uses a matrix of Q-values, providing us easier analysis and interpretation of the training process results and evolution. Later, we implemented DQN to help the agent figure out which action to exactly perform and consequently complete an assembly task with high-dimensional space taking less time and memory.

Our tasks were represented as hierarchical structures combined with a directed graphs. The graph, composed of nodes and links, have the task as the root and, as the hierarchy moves downwards, the task is divided into subtasks, which in turn are divided into concrete actions. Nevertheless, the sequence of actions is not structured as a hierarchy, but rather as a directed graph, where each action is connected to the following, within the given sequence, by a directed link.

The modelling of this work was developed with the aim of being flexible and scalable enough to be reproduced in different tasks or products with the same base operations and that can be composed on discrete subtasks.

The algorithm was trained on different rewards and hyperparameters. Two types of experiments were performed: the first with the aim of generating feasible sequence of actions to complete the task given precedence dependencies, and the second with goal of task allocation to an actor. The system was able to converge successfully in 2 scenarios for the first experience with the Q-learning algorithm: a table with 1 leg and a table with 2 legs, in 400 and 27 000, respectively. Due to the limitations of Q-learning within high-dimensional spaces, we could not obtain results for a task with 27 actions and 1 812 473 spaces.

With Deep Q-learning, the agent was successful in converging the reward for the 3 scenarios: the task with 9 actions in 5 000 episodes, the task with 15 actions in 40 000 episodes, and the task with 27 actions in 20 000, the last with early stopping to avoid overfitting. Given the high-space dimensionality of the case studies, the DQN algorithm was expected to converge faster. However, that did not happen, even after optimization of parameters using the kerasTuner tool. This may have happened due to the complexity and number of parameters that are needed to optimize, in the DQN network, which due to time constraints it was not possible to evaluate the impact of various combinations of parameters, thus not being possible to achieve the optimal network that would lead to a faster convergence by this algorithm.

Moreover, when training the robot for task allocation, it was possible to perceive the impact of the human operator expertise and the operation's average duration in the robot's decisions. An expert, with lower time performance than a robot was assigned to 92.87% of the actions, decreasing the mean time from 80.7 seconds to 73.25. And a beginner, with higher time performance, was allocated to 0.071% of the actions, reducing the global task mean time from 135.2 seconds to 143.22 seconds. An operator with an intermediate expertise, on the other hand, as a mean time performance similar to the robot, and therefore the actions were allocated for the two agents with a similar percentage. The impact of the training process on the global task mean time was hence less visible.

From this dissertation we could obtain a system that acquires data from the graph and sends back decisions to trigger nodes. In this graph, we should be able to visualize the sequences triggered by the robot, from the beginning of the process to its conclusion. We obtained an algorithm that makes decisions about sequences of operations and allows the team to complete the task collaboratively and obtain the final product. Both the time taken to perform whole task, as well as the time taken by the human are less than if the task were performed fully manually or automatically. Last but not least, We have a simulation through which we can validate if the process is correct.

The results and conclusions of this work led to the development of a scientific paper, which was submitted to the 15th International Conference on Agents and Artificial Intelligence (https://icaart.scitevents.org).

7.1 Future Work

In the future, the model developed in this dissertation can be extended to multi-agent planning, which will allow simultaneous execution of sequences between the two agents, and consequently improve the solution's efficiency and application in real-life scenarios.

Additionally, we will further address failures within collaboration, making the robot capable of compensating for an error in a sequence, i.e. adapt the algorithm to efficiently complete the task despite the error that occurred. The failures will include errors in the performance of the operations, as well as external errors. In our solution, failures are only address in automatic and manual modes, suggesting the agents to repeat the action.

Moreover, we intend to improve the robot agent's intelligence to be able to **adapt** the algorithm to unforeseen **variability** within the sequence, e.g. an operation performed unexpectedly by an external agent (robot or human) after a sequence has already been chosen.

Our solution would have more valuable results if it was tested with a real UR5 collaborative robot arm, increasing the transferability to real-world applications.

The robustness and scalability of our solution could also have a more vigorous validation if it was tested with different scenarios and types of sub-tasks.

Bibliography

[1] Pieter Abbeel and Andrew Y. Ng. "Apprenticeship Learning via Inverse Reinforcement Learning". In: *Proceedings of the Twenty-First International Conference on Machine Learning*. ICML '04. Banff, Alberta, Canada: Association for Computing Machinery, 2004, p. 1. ISBN: 1581138385. DOI: 10.1145/1015330.1015430. URL: https://doi.org/10.1145/1015330.1015430.

[2] Saurabh Arora and Prashant Doshi. "A survey of inverse reinforcement learning: Challenges, methods and progress". In: *Artificial Intelligence* 297 (2021), p. 103500. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2021.103500. URL: https://www.sciencedirect.com/science/article/pii/S0004370221000515.

[3] RICHARD BELLMAN. "A Markovian Decision Process". In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684. ISSN: 00959057, 19435274. URL: http://www.jstor.org/stable/24900506 (visited on 07/13/2022).

[4] Wendelin Böhmer et al. *Autonomous learning of state representations for control: An emerging field aims to autonomously learn state representations for reinforcement learning agents from their real-world sensor observations - ki - künstliche intelligenz*. Mar. 2015. URL: https://link.springer.com/article/10.1007/s13218-015-0356-1.

[5] Jake Brawer et al. "Situated human–robot collaboration: predicting intent from grounded natural language". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 827–833.

[6] Greg Brockman et al. "OpenAI Gym". In: *CoRR* abs/1606.01540 (2016). arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540.

[7] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[8] Kutluhan Erol, James Hendler, and Dana S Nau. "HTN planning: Complexity and expressivity". In: *AAAI*. Vol. 94. 1994, pp. 1123–1128.

[9] Luqin Fan et al. "Optimal Scheduling of Microgrid Based on Deep Deterministic Policy Gradient and Transfer Learning". In: *Energies* 14 (Jan. 2021), p. 584. DOI: 10.3390/en14030584.

[10] Ilche Georgievski and Marco Aiello. "HTN planning: Overview, comparison, and beyond". In: *Artificial Intelligence* 222 (2015), pp. 124–156. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2015.02.002. URL: https://www.sciencedirect.com/science/article/pii/S0004370215000247.

[11] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. MIT Press, 2016. ISBN: 9780262337373. URL: https://books.google.pt/books?id=omivDQAAQBAJ.

[12] L. Gualtieri et al. "Safety, Ergonomics and Efficiency in Human-Robot Collaborative Assembly: Design Guidelines and Requirements". English. In: *Procedia CIRP*. Vol. 91. 2020, pp. 367–372. URL: www.scopus.com.

[13] *Gym is a standard API for reinforcement learning, and a diverse collection of reference environments*. URL: https://www.gymlibrary.dev/.

[14] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[15] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[16] Bradley Hayes and Brian Scassellati. "Autonomously constructing hierarchical task networks for planning and human-robot collaboration". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2016, pp. 5469–5476.

[17] Bradley Hayes and Brian Scassellati. "Autonomously constructing hierarchical task networks for planning and human-robot collaboration". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 5469–5476. DOI: 10.1109/ICRA.2016.7487760.

[18] A. M. Howard. "Role allocation in human-robot interaction schemes for mission scenario execution". English. In: *Proceedings - IEEE International Conference on Robotics and Automation*. Vol. 2006. Cited By :12. 2006, pp. 3588–3594. URL: www.scopus.com.

[19] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[20] F. Ishida et al. "Reinforcement-learning agents with different temperature parameters explain the variety of human action-selection behavior in a Markov decision process task". English. In: *Neurocomputing* 72.7-9 (2009). Cited By :5, pp. 1979–1984. URL: www.scopus.com.

[21] E. T. Jaynes. *Information theory and statistical mechanics*. May 1957. URL: https://doi.org/10.1103/PhysRev.106.620.

[22] *JSON*. URL: https://networkx.org/documentation/stable/reference/readwrite/json_graph.html.

[23] J. Krüger, T.K. Lien, and A. Verl. “Cooperation of human and machines in assembly lines”. In: *CIRP Annals* 58.2 (2009), pp. 628–646. ISSN: 0007-8506. DOI: https://doi.org/10.1016/j.cirp.2009.09.009. URL: https://www.sciencedirect.com/science/article/pii/S0007850609001760.

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. *Deep learning*. May 2015. URL: https://www.nature.com/articles/nature14539.

[25] Jay Lee, Behrad Bagheri, and Chao Jin. “Introduction to cyber manufacturing”. In: *Manufacturing Letters* 8 (2016), pp. 11–15. ISSN: 2213-8463. DOI: https://doi.org/10.1016/j.mfglet.2016.05.002. URL: https://www.sciencedirect.com/science/article/pii/S2213846316300049.

[26] Sergey Levine et al. “End-to-End Training of Deep Visuomotor Policies”. In: *CoRR* abs/1504.00702 (2015). arXiv: 1504.00702. URL: http://arxiv.org/abs/1504.00702.

[27] K. Li et al. “Sequence planning considering human fatigue for human-robot collaboration in disassembly”. English. In: *Procedia CIRP*. Vol. 83. Cited By :21. 2019, pp. 95–104. URL: www.scopus.com.

[28] Zhihao Liu et al. *Task-level decision-making for dynamic and stochastic human-robot collaboration based on Dual Agents Deep Reinforcement Learning - the International Journal of Advanced Manufacturing Technology*. June 2021. URL: https://link.springer.com/article/10.1007/s00170-021-07265-2.

[29] Olivier Mangin, Alessandro Roncone, and Brian Scassellati. “How to be Helpful? Supportive Behaviors and Personalization for Human-Robot Collaboration”. In: *Frontiers in Robotics and AI* 8 (2022). ISSN: 2296-9144. DOI: 10.3389/frobt.2021.725780. URL: https://www.frontiersin.org/article/10.3389/frobt.2021.725780.

[30] Hongzi Mao et al. *Resource Management with deep reinforcement learning: Proceedings of the 15th ACM Workshop on hot topics in networks*. Nov. 2016. URL: https://dl.acm.org/doi/pdf/10.1145/3005745.3005750.

[31] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[32] Jan Mattner, Sascha Lange, and Martin Riedmiller. “Learn to Swing Up and Balance a Real Pole Based on Raw Visual Input Data”. In: *Neural Information Processing*. Ed. by Tingwen Huang et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 126–133. ISBN: 978-3-642-34500-5.

[33] Luiz Homem de Mello and Arthur C. Sanderson. “AND/OR graph representation of assembly plans”. In: *IEEE Trans. Robotics Autom.* 1990.

[34] Ryszard S. Michalski and John R. Anderson. "Machine learning - an artificial intelligence approach". In: *Symbolic computation*. 1984.

[35] Volodymyr Mnih et al. *Human-level control through deep reinforcement learning*. Feb. 2015. URL: https://www.nature.com/articles/nature14236.

[36] Volodymyr Mnih et al. *Human-level control through deep reinforcement learning*. Feb. 2015. URL: https://www.nature.com/articles/nature14236.

[37] Volodymyr Mnih et al. *Human-level control through deep reinforcement learning*. Feb. 2015. URL: https://www.nature.com/articles/nature14236.

[38] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[39] Seyed Sajad Mousavi, Michael Schukat, and Enda Howley. "Deep Reinforcement Learning: An Overview". In: *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016*. Ed. by Yaxin Bi, Supriya Kapoor, and Rahul Bhatia. Cham: Springer International Publishing, 2018, pp. 426–440.

[40] Debasmita Mukherjee et al. "A Survey of Robot Learning Strategies for Human-Robot Collaboration in Industrial Settings". In: *Robotics and Computer-Integrated Manufacturing* 73 (2022), p. 102231. ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2021.102231. URL: https://www.sciencedirect.com/science/article/pii/S0736584521001137.

[41] Prajval Kumar Murali, Kourosh Darvish, and Fulvio Mastrogiovanni. "Deployment and Evaluation of a Flexible Human-Robot Collaboration Model Based on AND/OR Graphs in a Manufacturing Environment". In: *CoRR* abs/2007.06720 (2020). arXiv: 2007.06720. URL: https://arxiv.org/abs/2007.06720.

[42] Gergely Neu and Csaba Szepesvári. "Apprenticeship Learning using Inverse Reinforcement Learning and Gradient Methods". In: *CoRR* abs/1206.5264 (2012). arXiv: 1206.5264. URL: http://arxiv.org/abs/1206.5264.

[43] R. Nian, J. Liu, and B. Huang. "A review On reinforcement learning: Introduction and applications in industrial process control". English. In: *Computers and Chemical Engineering* 139 (2020). Cited By :50. URL: www.scopus.com.

[44] Stefanos Nikolaidis et al. "Efficient Model Learning for Human-Robot Collaborative Tasks". In: *CoRR* abs/1405.6341 (2014). arXiv: 1405.6341. URL: http://arxiv.org/abs/1405.6341.

[45] Tom O'Malley et al. *KerasTuner*. 2019. URL: https://github.com/keras-team/keras-tuner.

[46] Openai. *Gym/discrete.py at master · Openai/Gym*. June 2022. URL: https://github.com/openai/gym/blob/master/gym/spaces/discrete.py.

[47] Openai. *Gym/gym/spaces at master · openai/gym*. URL: https://github.com/openai/gym/tree/master/gym/spaces.

[48] Armando Plasencia et al. "Open Source Robotic Simulators Platforms for Teaching Deep Reinforcement Learning Algorithms". In: *Procedia Computer Science* 150 (2019). Proceedings of the 13th International Symposium "Intelligent Systems 2018" (INTELS'18), 22-24 October, 2018, St. Petersburg, Russia, pp. 162–170. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2019.02.031. URL: https://www.sciencedirect.com/science/article/pii/S1877050919303758.

[49] E. Rohmer, S. P. N. Singh, and M. Freese. "CoppeliaSim (formerly V-REP): a Versatile and Scalable Robot Simulation Framework". In: *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*. 2013. URL: www.coppeliarobotics.com.

[50] Alessandro Roncone, Olivier Mangin, and Brian Scassellati. "Transparent role assignment and task allocation in human robot collaboration". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 1014–1021. DOI: 10.1109/ICRA.2017.7989122.

[51] Nicole Rusk. *Deep learning*. Dec. 2015. URL: https://www.nature.com/articles/nmeth.3707.

[52] Stuart Russell. "Learning Agents for Uncertain Environments (Extended Abstract)". In: *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*. COLT' 98. Madison, Wisconsin, USA: Association for Computing Machinery, 1998, pp. 101–103. ISBN: 1581130570. DOI: 10.1145/279943.279964. URL: https://doi.org/10.1145/279943.279964.

[53] Mirella Santos Pessoa de Melo et al. "Analysis and Comparison of Robotics 3D Simulators". In: *2019 21st Symposium on Virtual and Augmented Reality (SVR)*. 2019, pp. 242–251. DOI: 10.1109/SVR.2019.00049.

[54] Michael Sharp, Ronay Ak, and Thomas Hedberg. "A survey of the advancing use and development of machine learning in smart manufacturing". In: *Journal of Manufacturing Systems* 48 (2018). Special Issue on Smart Manufacturing, pp. 170–179. ISSN: 0278-6125. DOI: https://doi.org/10.1016/j.jmsy.2018.02.004. URL: https://www.sciencedirect.com/science/article/pii/S0278612518300153.

[55] *Software for complex networks*. URL: https://networkx.org/documentation/latest/.

[56] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. The MIT Press, 2020.

[57] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: NIPS'99. Denver, CO: MIT Press, 1999, pp. 1057–1063.

[58] Zhaorong Tao, Zhichao Chen, and Yanjie Li. “Sensitivity-Based Inverse Reinforcement Learning”. In: *2013 32ND CHINESE CONTROL CONFERENCE (CCC)*. Chinese Control Conference. 32nd Chinese Control Conference (CCC), Xian, PEOPLES R CHINA, JUL 26-28, 2013. Syst Engn Soc China; NW Polytechn Univ; Chinese Assoc Automat, Tech Comm Control Theory; Chinese Acad Sci, Acad Math & Syst Sci; China Soc Ind & Appl Math; Xian Jiaotong Univ; Xian Univ Technol; IEEE Control Syst Soc; Soc Instrument & Control Engineers Japan; Inst Control Robot & Syst Korea. 2013, 2856–2861.

[59] Maulesh Trivedi and Prashant Doshi. “Inverse Learning of Robot Behavior for Collaborative Planning”. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 1–9. DOI: 10.1109/IROS.2018.8593745.

[60] Weitian Wang et al. “Facilitating Human–Robot Collaborative Tasks by Teaching-Learning-Collaboration From Human Demonstrations”. In: *IEEE Transactions on Automation Science and Engineering* 16.2 (2019), pp. 640–653. DOI: 10.1109/TASE.2018.2840345.

[61] Xi Vincent Wang et al. “Human–robot collaborative assembly in cyber-physical production: Classification framework and implementation”. In: *CIRP Annals* 66.1 (2017), pp. 5–8. ISSN: 0007-8506. DOI: https://doi.org/10.1016/j.cirp.2017.04.101. URL: https://www.sciencedirect.com/science/article/pii/S0007850617301014.

[62] Christopher J. C. H. Watkins and Peter Dayan. “Q-learning”. In: *Machine Learning*. 1992, pp. 279–292.

[63] *Welcome to Python.org*. URL: https://www.python.org/.

[64] H. A. Yanco and J. Drury. “Classifying human-robot interaction: An updated taxonomy”. English. In: *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*. Vol. 3. Cited By :221. 2004, pp. 2841–2846. URL: www.scopus.com.

[65] Tian Yu, Jing Huang, and Qing Chang. “Mastering the Working Sequence in Human-Robot Collaborative Assembly Based on Reinforcement Learning”. In: *IEEE Access* 8 (2020), pp. 163868–163877. DOI: 10.1109/ACCESS.2020.3021904.

[66] Tian Yu, Jing Huang, and Qing Chang. “Optimizing task scheduling in human-robot collaboration with deep multi-agent reinforcement learning”. In: *Journal of Manufacturing Systems* 60 (2021), pp. 487–499. ISSN: 0278-6125. DOI: https://doi.org/10.1016/j.jmsy.2021.07.015. URL: https://www.sciencedirect.com/science/article/pii/S0278612521001527.

[67] Sofya Zeylikman et al. “The HRC Model Set for Human-Robot Collaboration Research”. In: *CoRR* abs/1710.11211 (2017). arXiv: 1710.11211. URL: http://arxiv.org/abs/1710.11211.

[68] Rong Zhang et al. "A reinforcement learning method for human-robot collaboration in assembly tasks". In: *Robotics and Computer-Integrated Manufacturing* 73 (2022), p. 102227. ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2021.102227. URL: https://www.sciencedirect.com/science/article/pii/S0736584521001095.

Appendix A

Assembly tasks graphs

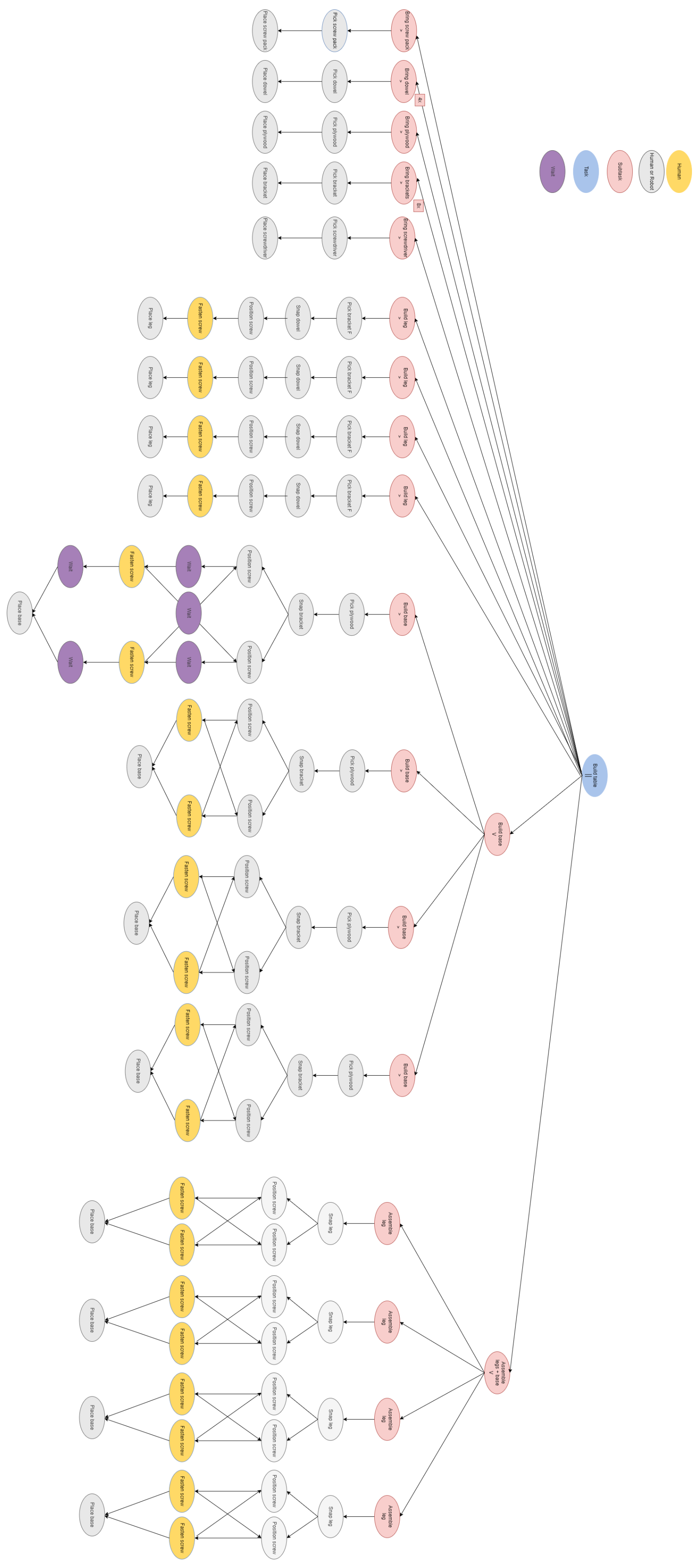


Figure A.1: Graph for a table assembly task and its possible sequence of actions

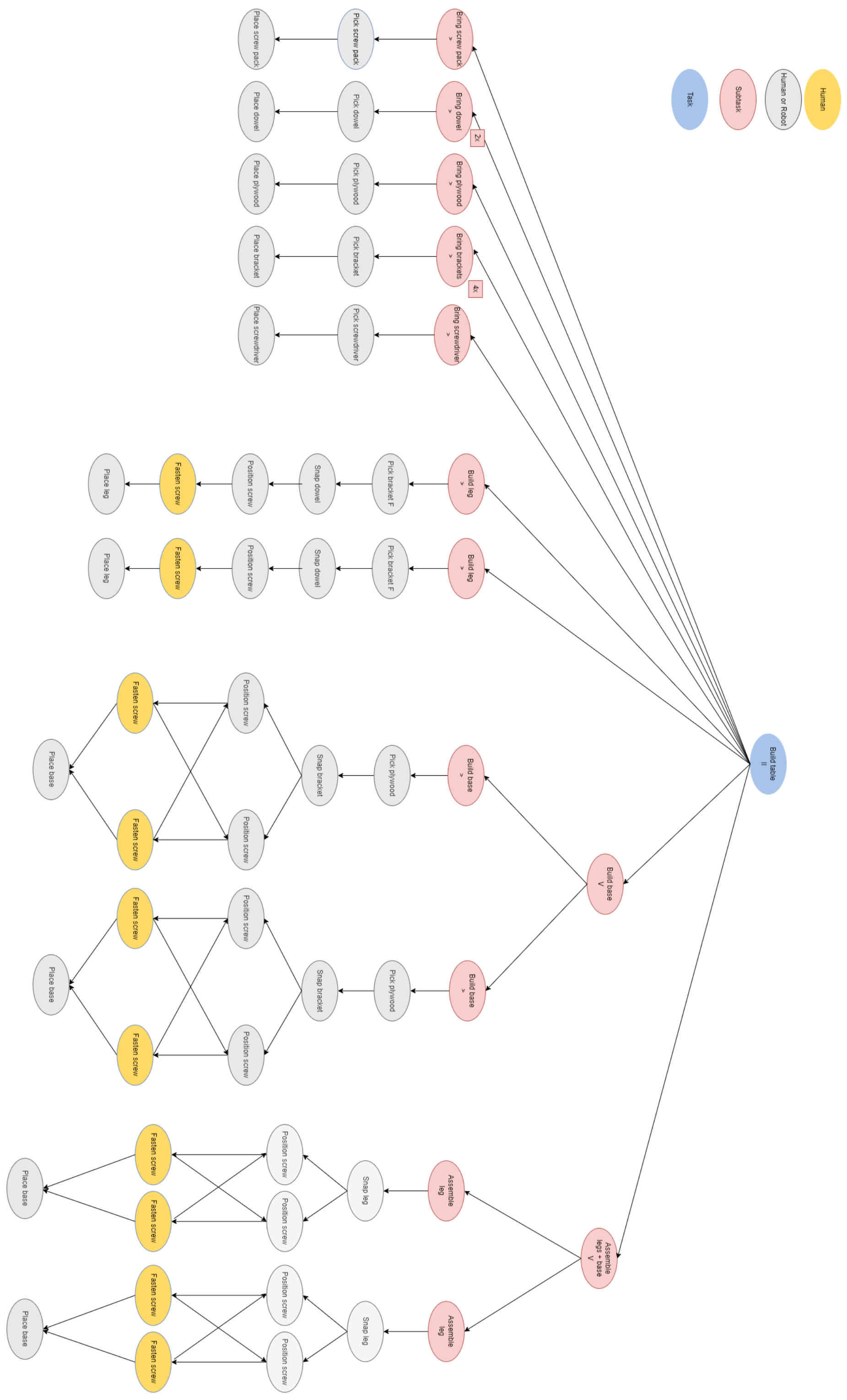


Figure A.2: Graph for a two-leg-table assembly task and its possible sequence of actions

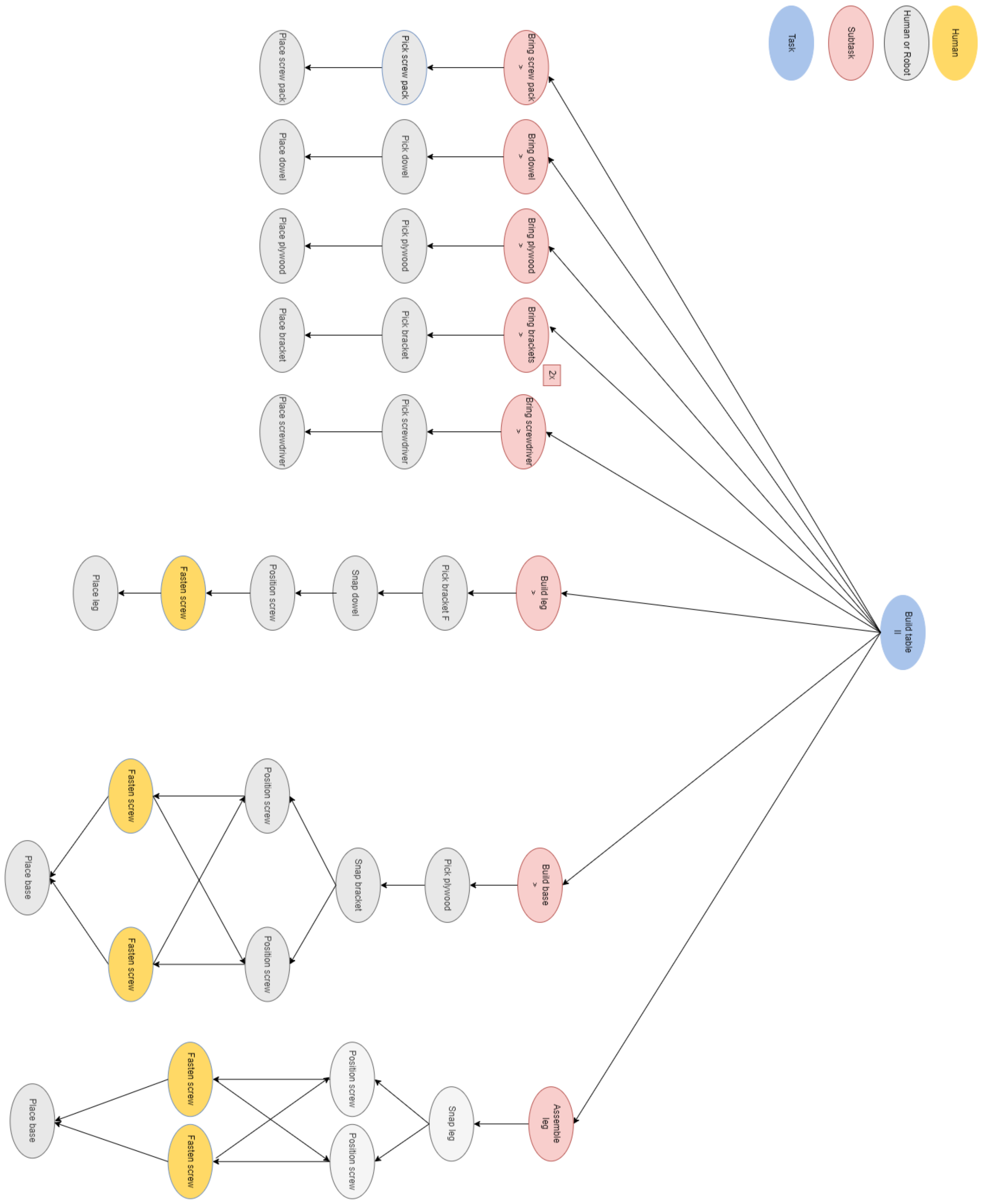


Figure A.3: Graph for a one-leg-table assembly task and its possible sequence of actions