This research is supported by European Union's Horizon 2020 programme under the ReMAP projet, grant No 769288.

# Agradecimentos

Aos meus orientadores e colegas de estágio.
À minha família. Aos meus amigos.
Aos efémeros e perenes.

E acima de tudo, e para sempre, aos meus pais.

This page intentionally left blank.

## Abstract

Currently, maintenance in aviation is mostly pre-scheduled or done reactively. These techniques may prompt to over-maintenance, more time and resources spent, less aircraft availability and high expenses. Real-time Condition-based Maintenance for Adaptive Aircraft Maintenance Planning (ReMAP) intends to solve this issue by introducing a proposal that seeks, by using sensors and state-of-the-art algorithms, to apply condition-based maintenance (CBM) in commercial aircraft, allowing real-time health monitoring of the aircraft's systems to be integrated with a decision making reinforcement learning (RL) agent to formulate an optimal maintenance scheduling plan. The proposal integrates different work packages (WPs), each of them with a different role in the project: from collecting data from the aircraft's systems to defining a maintenance plan for a whole fleet. This internship was embedded in the WP responsible for the development of the maintenance decision support tool (MDST), in charge of planning and assigning maintenance tasks for a fleet of aircraft. The final version of the tool is envisioned to receive information about each aircraft's periodical maintenance checks plus any unexpected maintenance needs that might arise from the aircraft's health prognostics. These maintenance tasks will be assigned to time-frames in order to create a feasible maintenance schedule. Nevertheless, as an introductory concept proof, the MDST must organize a maintenance schedule for periodic tasks, only. Accordingly, the internship's goal was to use RL to assign periodical maintenance checks for a fleet of 51 aircraft over 6 years. Our results show that the RL algorithm converges quickly and performs significantly better than a greedy solution.

**Keywords**: aircraft maintenance, condition-based maintenance, Q-learning, reinforcement learning, task scheduling.

This page intentionally left blank.

# Resumo

Atualmente, as manutenções na aviação são maioritariamente pré-programadas ou reactivas. Estas técnicas podem levar a manutenção excessiva, mais tempo e recursos gastos, menor disponibilidade de aeronaves e despesas altas. Real-time Condition-based Maintenance for Adaptive Aircraft Maintenance Planning (ReMAP) pretende solucionar este problema, para isso introduzindo uma proposta que procura, fazendo uso de sensores e algoritmos de última geração, aplicar condition-based maintenance (CBM) em aeronaves comerciais, possibilitando a integração da monitorização em tempo real dos sistemas de aeronaves com um agente reinforcement learning (RL) de tomada de decisões, capaz de formular um plano de manutenções ótimo. A proposta integra várias work packages (WPs), cada uma delas com um papel diferente no projeto: desde a extração de dados dos sistemas das aeronaves à definição de um plano de manutenção para uma frota inteira. Este estágio foi embutido na WP responsável pelo desenvolvimento da maintenance decision support tool (MDST), encarregue de planear e atribuir tarefas de manutenção para uma frota de aviões. A versão final desta ferramenta irá receber informação de cada manutenção periódica de cada aeronave juntamente com pedidos de manutenção inesperados que possam surgir dos prognósticos de saúde das aeronaves. O intuito será distribuir essas tarefas por um prazo pré-definido de maneira a criar um calendário de manutenções praticável. Todavia, inicialmente, como prova de conceito, a MDST irá criar um planeamento que apenas inclua manutenções periódicas. Portanto, o objetivo do estágio foi usar RL de maneira a calendarizar trabalhos de manutenção de uma frota com 51 aviões para o prazo de 6 anos. Os nossos resultados demonstram que o algoritmo de RL converge rapidamente e tem um melhor desempenho do que uma solução gananciosa.

**Palaras-chave**: manutenção de aeronaves, condition-based maintenance, Q-learning, reinforcement learning, calendarização de tarefas.

This page intentionally left blank.

# Contents

# Acronyms

**ACARE** Advisory Council For Aeronautical Research In Europe 10

**CBM** Condition-based Maintenance v, vii, 1, 3, 4, 5, 6, 10, 11, 12, 14, 63

**DL** Deep Learning 28

**DP** Dynamic Programming 14

**DQL** Deep Q-learning 4, 7, 26, 44, 68, 70

**DQN** Deep Q-network xv, 26, 44, 45, 68, 70

**DRL** Deep Reinforcement Learning 7, 27, 29, 30, 41, 44, 57, 64, 68

**DY** Calendar Days 8, 29, 34, 52

**EDD** Earliest Due Date First 12

**ERP** Enterprise Resource Planning 13, 14

**FC** Flight Cycles 8, 9, 34

**FH** Flight Hours 8, 9, 34

**FIFO** First-in, First-out 12

**GA** Genetic Algorithm 30, 73, 78, 81

**GENMOP** General Multi-objective Parallel Genetic Algorithm 13

**ICLR 2018** 6th International Conference On Learning Representations 28

**IJCAI2018** 27th International Joint Conference On Artificial Intelligence 18

**JSP** Job-shop Scheduling Problem 12

**LSTM** Long Short-term Memory 27

**MDP** Markov Decision Processes xiii, xv, 18, 19, 20, 21, 23

**MDST** Maintenance Decision Support Tool v, vii, xiii, 2, 7, 8, 29, 63, 64, 67

**MRI** Maintenance Required Item 8, 9

**MRO** Maintenance, Repair And Overhaul 14

**MRP** Material Requirements Planning 14

**MRPII** Manufacturing Resource Planning 14

**NASA** National Aeronautics And Space Administration 12

**NN** Neural Network 26, 69

**OEM** Original Equipment Manufacturer 4

**QL** Q-learning 12, 24, 68

**RC** Remote Controlled 26

**ReLU** Rectified Linear Unit 44, 69

**ReMAP** Real-time Condition-based Maintenance For Adaptive Aircraft Maintenance Planning v, vii, 2, 3, 4, 5, 6, 7, 8, 10, 29, 57, 63

**RL** Reinforcement Learning v, vii, xiii, xiv, 3, 4, 7, 12, 13, 17, 18, 22, 24, 26, 27, 28, 30, 37, 38, 39, 40, 41, 43, 44, 46, 47, 48, 49, 50, 51, 52, 53, 54, 57, 58, 67, 68, 73, 77

**RUL** Remaining Useful Life 5, 6, 7, 29, 30, 67

**SE** Simulated Environment xiii, 3, 37, 38, 39, 43, 54

**SEQ2SEQ** Sequence-to-sequence 27

**SoA** State-of-the-art 10, 26, 27

**SPT** Short Processing Time First 12

**TD** Temporal Difference 12

**WP** Work Package v, vii, 2, 4, 5, 7, 8

# List of Figures

# List of Tables

This page intentionally left blank.

# List of Algorithms

This page intentionally left blank.

# Nomenclature

**methodologies:**

| | |
|---|---|
| $A$ | set of possible actions |
| $S$ | set of all non-terminal states |
| $A_t$ | action at time $t$ |
| $S_t$ | state at time $t$ |
| $R_t$ | reward at time $t$ |
| $s, s'$ | states |
| $r$ | a reward |
| $t$ | discrete time step |
| $\pi$ | a policy |
| $\mathbb{E}[X]$ | expectation of variable $X$ |
| $\gamma$ | discounting factor |
| $v_\pi(s)$ | value of state $s$ under policy $\pi$ |
| $q_\pi(s, a)$ | value of taking action $a$ in state $s$ under policy $\pi$ |
| $Q$ | array estimates of action-value function $q_\pi$ or $q_*$ |

**research approach:**

| | |
|---|---|
| $T$ | set of all tasks |
| $D$ | set of all days |
| $C$ | set of all tasks in a conflict |
| $H$ | set of all hangars |
| $\|X\|$ | number of elements in a set $X$ |
| $t$ | a task |
| $l$ | a length of a task |
| $dd$ | a due date of a task |

| | |
|---|---|
| $d$ | day in which a task was scheduled |
| $k$ | discrete number usually representing the position of an element in a set |
| $T_k$ | task number $k$ |
| $D_k$ | day number $k$ |
| $C_k$ | conflict number $k$ |
| $d_k$ | scheduled day of task $T_k$ |
| $dd_k$ | due date of task $T_k$ |
| $l_k$ | length of task $T_k$ |

# Chapter 1

# Introduction

Industrial equipment is expected to have high reliability and maximum availability possible. However, product deterioration is inevitable when dealing with equipment operating in a real environment, bound to different types of burden and load (Jardine, Lin, & Banjevic, 2006a) and, in order to follow the technology growth over the last few decades, maintenance operations were forced to evolve.

It is considered maintenance actions those which can restore equipment to a serviceable condition, such as servicing, repair, modification, overhaul, inspection and determination of condition (PeriyarSelvam, Tamilselvan, Thilakan, & Shanmugaraja, 2013). There are two types of maintenance techniques: **preventive** maintenance and **reactive** maintenance. In the latter, the equipment is repaired after it has failed. This brings obvious problems such as high maintenance costs and less equipment availability, in of itself leading to even bigger costs. In the preventive strategy maintenance activities are done prior to equipment breakdown and, in most cases, using a fixed interval approach. This interval is decided either by manufacturer recommendations or by staff experience (Ahmad & Kamaruddin, 2012). To follow the manufacturer recommendations seldom will serve as a solution for minimizing operation costs because the conditions each machine encounters throughout its lifetime may be different, which leads to different levels of deterioration (Labib, 2004). We can observe that in the case of aircraft, for example, where some flights present rougher conditions than others for the same type of aircraft, leading some planes to need maintenance sooner than others. Relying on staff experience can be more precise, giving that the technicians will already know what kind of burden each equipment is subject to, but also risky as technicians may leave the company or not be able to be present every time there is a maintenance problem. Besides, by applying a preventive approach, equipment might be subjected to maintenance before needed, thus leading to over-maintenance.

**Condition-based maintenance (CBM)** comes to avoid the problems allied to both of these techniques, making use of real-time monitoring and diagnose of equipment. About 99% of the times it is possible to predict a piece of equipment is going to fail by certain indications that a breakdown is going to happen (Bloch & Geitner, 1983). CBM allows for this verification to happen constantly by making use of sensors that can measure vibration, noise, temperature, etc.

## 1.1 Motivation

Equipment reliability has a substantial impact on companies' profitability, therefore a maintenance plan which can make as few interventions as possible while keeping the equipment in its maximum productivity is one of great interest for a company (Peng, Dong, & Zuo, 2010). According to Holmberg et al. (2004), due to disturbances or failures that result in production shutdown, the overall losses by industries in Europe are around 140 billion Euros annually and, with systems growing in their complexity level, equipment diagnosis becomes harder to execute with precision.

Aircraft systems, components, and structures are no exception to the rule. Currently, aircraft maintenance mostly takes place periodically, based on a fixed interval approach or reactively, when a piece of equipment has a breakdown. According to PeriyarSelvam et al. (2013), maintenance costs usually correspond to around 10-20 percent of aircraft related operating costs and current systems have around 40 percent chance for false equipment removal, leading to high downtimes and expenses. Furthermore, the maintenance costs contribution to direct operating costs almost did not decrease over the last few decades (Papakostas, Papachatzakis, Xanthakis, Mourtzis, & Chryssolouris, 2010). In 2016 alone, airlines had an average of \$318M spent on direct aircraft maintenance (*IATA's Airline Maintenance Cost Executive Summary*, 2017). In an ideal scenario, maintenance would only be performed when needed while avoiding equipment breakdown. This is the goal of Real-time Condition-based Maintenance for Adaptive Aircraft Maintenance Planning (ReMAP).

On a short-term impact ReMAP's proposal estimates savings of around 270 thousand euros per year on a single large passenger aircraft (*ReMAP's Proposal*, 2017). This comes from the reducing of both unscheduled and periodic maintenance checks, parts removals and delay events, resulting in an extra 1.2 days of work plus a reduction of 140 minutes on delays per year for a single aircraft. This estimation applied to all the 4600 commercial aircraft in the European market results in potential benefits of over 700 million Euros per year.

## 1.2 Aim and Scope

The internship initiated within the first out of a total of four years of the expected project's duration. It was set on an initial phase of deliberation in which an introductory and partial objective was constructed given the overall goal, in order to alleviate the problem and provide introductory formulations and concept proofs. The internship was embedded in the project's work package (WP) in charge of developing the project's maintenance decision support tool (MDST). The tool's goal is to deal with the uncertainty associated with the aircraft's health status and to assist maintenance planners in their day-to-day operations related to aircraft maintenance scheduling. The MDST must, as the final output, return a feasible schedule by assigning time-frames to maintenance checks.

Although the final goal is to schedule both periodic inspections, which each aircraft has assigned to it, and unexpected maintenance jobs, which might originate from the aircraft's health prognostics, our WP was only focused on the further for the time period of the internship. As such, our goal was to schedule periodic maintenance tasks in a given period of time, keeping each task as close to its due date (limit day for maintenance) as possible.

Because we were not granted access to data immediately, we began by creating a set of **toy problems** which would mimic the real problem at hand. We then developed a solution capable of creating a feasible maintenance calendar, which we called **Conflict-Solver**, in which a calendar is created by solving possible conflicts that might arise from scheduling a group of tasks in a limited time-span. A greedy algorithm was implemented to decide on what to do in each conflict, by selecting the action which brings be the best immediate solution. A "random-action" formulation was also used, in which each decision is made, as the name points out, randomly. Following these algorithms being completed and feasible calendars started being outputted, a **reinforcement learning agent** was developed, capable of gradually learning better ways of solving each conflict in our algorithm. A **simulated environment (SE)** was also formulated in order to make possible the interaction between the RL agent and our solution. After **real aircraft maintenance data** was received, we were prepared to use it in our designed solution. The results from both the real data and the toy-problems were then compared with the greedy and random-action implementations. This whole process adds up to seven main steps done throughout the internship:

1. Develop Toy-Set Problem Formulation for Scheduling
2. Design Conflict-Solver
3. Implement a greedy solution
4. Implement a random-action solution
5. Implement RL solution
6. Utilize real aicraft maintenance data in the previous designs
7. Compare results from steps 3, 4 and 5

## 1.3 Challenges

One of the main challenges for the ReMAP project as a whole is regulatory issues. Regulatory aviation authorities have rigid rules regarding safety and reliability which may conflict with an effective CBM execution. As a result, CBM must be accredited to work on each aircraft system, being thoroughly tested in each one of them for reliability and effectiveness. ReMAP's ambition is to test and validate its solution on a real environment, in a setting involving 12 aircraft with 2 different aircraft types, over a time-span of 6 months.

Another big challenge is that of getting access to data. Operational data generated by on-board systems is owned by the airlines and generally, they do not want to share the raw data related to aircraft health management. Adding to that, there are strict rules related to sharing flight data in order to protect the pilot's privacy. ReMAP agrees that airline operators are the owners of the data and will comply with Airlines and Pilots unions' agreements in regards to data protection. The obstacles on getting access to data directly affected the internship being that, for most of the internship's extent, toy-problems had to be used together with artificially created data.

As of today, original equipment manufacturers (OEMs) still do not have defined standards to manage the combination of aircraft data with data from maintenance operations. Given the various stakeholders involved in the project, data interoperability is an important aspect of the project, being that maintenance decisions derive from information concerning the aircraft systems and structures health. ReMAP has integrated into its team a member of the KLM Royal Dutch Airlines, which is working as the middle man between ReMAP's stakeholders and the SAE International HM-1 "Integrated Vehicle Health Management" committee to tackle this issue.

## 1.4 Thesis Outline

This report is divided into seven chapters in total. The next chapter details what is the ReMAP project and its global goals as well as the focus of the WP in which this internship is embedded in, responsible for the planning support tool. The third chapter reflects on some of the work already made on the subjects here portrayed (reinforcement learning, condition-based maintenance, scheduling, and aicraft maintenance). Fourth chapter thoroughly details the method (reinforcement learning) that is going to be used in our solution together with a description of the specific algorithm, deep Q-learning. A more complete description of the problem and the approaches taken in order to solve is given in chapter five, where the first six previously detailed work points are going to be carefully explained. Results and discussion are presented in chapter six. The seventh and final chapter contains the conclusion to this report.

# Chapter 2

# ReMAP Framework

In this chapter, a description of the project's scope and how it is assembled is going to be given. The task this internship is embedded in is only a small portion of the extent of the whole ReMAP structure and of what the project represents. Therefore giving an overall view of the project will also give the reader a better understanding of the goals of our task and the work package it is embedded in. Likewise, a description of what CBM represents is given.

## 2.1 Condition-Based Maintenance

The goal of condition-based maintenance (CBM) is to identify failures before they occur, thus making it possible to significantly reduce the number of unscheduled maintenance jobs (Jardine, Lin, & Banjevic, 2006b). Using this approach it is possible not only to spot upcoming failure in advance and to schedule maintenance accordingly but also to fully exploit the systems remaining useful life (RUL), which is the time for which a system is still operational before needing maintenance. In CBM there are three main steps (Jardine et al., 2006a):

- Data Acquisition
- Data processing
- Maintenance Decision Making

In data acquisition, useful data is gathered using sensors designed to collect information from the physical parts being monitored, usually related to temperature, oil, sound, etc. In data processing, tools are used to clean the data which is then analyzed for better understanding of the information collected. In the last step, maintenance decision making, an efficient maintenance action is employed. According to Jardine et al. (2006a), there are two categories of techniques for maintenance decision support: prognostics and diagnostics. Prognostics occur prior to faults and are in charge of detecting upcoming failures and estimate on when they are likely to happen while diagnostics occur after a problem occurs and detect the essence of the issue. These two techniques complement each other as diagnostics can help in creating more accurate data for future health prognostics.

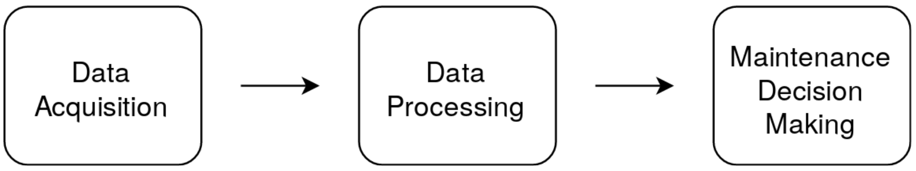


Figure 2.1: CBM Steps

ReMAP's project implements these steps and adds another one to the process, which is that of creating a maintenance schedule.

## 2.2 ReMAP Project

ReMAP is a Horizon 2020 (*H2020 ReMAP*, n.d.) project whose goal is to create a strategy that makes it possible to perform maintenance only when aircraft parts are damaged or close to breakdown, thus reducing aircraft down-times, the consuming of resources, and costs associated. This will be achieved by making use of the number of sensors already embedded in modern aircraft and more, combined with probabilistic algorithms for damage monitoring and RUL estimation in order to create real-time adaptive maintenance plans for a whole fleet. Given this strategy, ReMAP aims to fully exploit the benefits given by implementing a CBM policy.

As of the time that the project began, the efforts made concerning the health monitoring of modern aircraft had been mainly related to obtaining diagnostics and prognostics for specific systems (Byington, Roemer, & Galie, 2002; W. Wang & Zhang, 2005). But there is still not an efficient approach capable of taking advantage of the health diagnostics and prognostics in regards to an efficient maintenance plan, and most airlines still transfer information manually into spreadsheets. ReMAP wants to develop an integrated approach by coordinating the health monitoring of aircraft with data management, data analysis, and maintenance performance.

In order to build a stable implementation of the CBM methodology, the project is built upon five technology blocks:

- Data acquisition
- Data collection, storage and processing
- Systems prognostics
- Structures prognostics
- Decision support tool

In "data acquisition" sensors will be used to collect damage information regarding aircraft systems. The block regarding "data collection, storage and processing" will be responsible for data management, processing, visualization, and sharing. "Systems prognostics" will in charge of damage monitoring and RUL estimation, making use of machine learning techniques. "Systems prognostics" will deal with

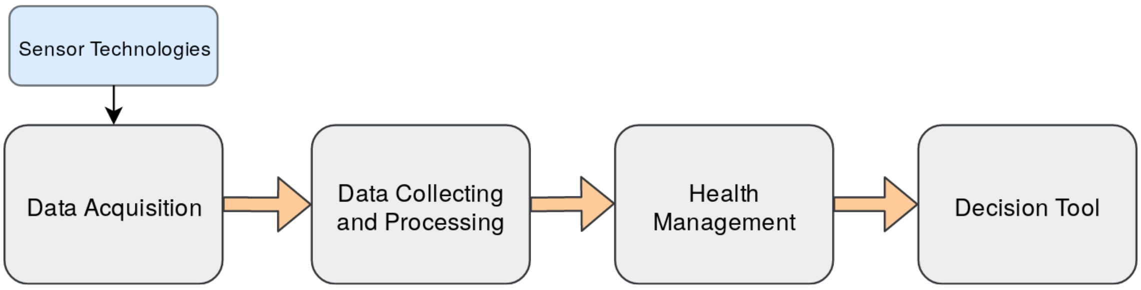


Figure 2.2: ReMAP's Structure Representation

identifying and predicting system degradation or faults, achieving this by making use of machine learning methods, data analytics, and signal processing algorithms. Also in this step, physics-based models will be used to determine the causes of the respective degradation or failure. Lastly, we have the "decision support tool" block, responsible for creating an adaptive fleet maintenance plan, taking into account the diagnostics and prognostics received from before. Figure 2.2 shows a high-level representation of the several stages of the solution, in which systems and structures prognostics are shown aggregated into "Health Management".

ReMAP project has several partners with vast experience and knowledge in the fields incorporated by the proposal: sensing technology, IT platform, structural data analysis, system data analytics and simulation and optimization tools. Furthermore, ReMAP involves partners capable of validating and demonstrating the technologies in a real and relevant environment, namely *KLM Royal Dutch Airlines* and *Embraer* (aircraft manufacturer). The ReMAP team is therefore divided into several WPs, each one of them with their particular role in the project. The WP in which this internship is embedded on is responsible for developing the **maintenance decision support tool (MDST)**.

## 2.2.1 Maintenance Decision Support Tool

The overall goal of the MDST is, using the information concerning the structures and systems prognostics of a whole fleet, to assist the maintenance planners on delivering a plan capable of reducing costs while assuring safety. The big challenge will be to deal with the uncertainty given by the extensive set of possible health scenarios. For that, MDST will make use of deep reinforcement learning (DRL) algorithms, specifically deep Q-learning (DQL). Besides this being an experimental approach, ReMAP believes that RL, being an adaptive optimal control (Sutton, Barto, & Williams, 1992) allied to "the powerful function approximation and representation learning properties of deep neural networks" (Arulkumaran, Deisenroth, Brundage, & Bharath, 2017), will be able learn the system model efficiently.

As a definitive goal, MDST wants to use RUL estimations in the maintenance planning process together with the available material resources and human resources, hangar availability, flight plans, etc., in order to create fleet maintenance plans capable of maximizing aircraft availability and minimizing costs. This will come to replace the current system used by most airlines regarding aircraft maintenance which still relies on manually developed spreadsheets. By achieving this, ReMAP expects to reduce by 8% the number of maintenance checks.

The role of validating not only this tool but the whole project is attributed to different WPs, in charge of identifying hazards and safety barriers in each element that composes the ReMAP proposal. This evaluation will be made by the use of rare event simulations, using Monte Carlo methods (Rubino & Tuffin, 2009). Our tool, in particular, will be evaluated in terms of the degradation of the aircraft's systems and structures, the duration of the maintenance tasks and the impact of the maintenance procedures.

The input data for the tool has been a crucial point of discussion between the members responsible for the MDST. Throughout the span of the internship, a **data landscape** has, therefore, been organized, containing information of what we think is the key data needed to plan maintenance on a fleet level.

### 2.2.2 Data Landscape

The data landscape contains thirteen sheets of information. The first four contain information about **periodic tasks**, four other are of **snapshot data** containing information about current open tasks, available slots for maintenance, etc. and the rest of the sheets contain **historical data** on maintenance tasks completed, fleet utilization, ground times, etc. Below follows an overall view of each one of these three information groups.

#### Periodic Tasks

The time intervals in the periodic tasks related sheets are given in flight hours (FH), flight cycles (FC) or calendar days (DY). flight hours (FH) is the amount of time an aircraft spends flying, one flight cycles (FC) is defined by one takeoff and one landing and one DY corresponds to a 24 hour period. In periodic tasks, we have maintenance required items (MRIs) which are obligatory to perform for each aircraft and are composed by a group of sub-tasks. The MRI's are then clustered, forming **blocks** of maintenance, in what is called packaging. The packaging is done in a way to optimize as most as possible the maintenance management, according to the MRI's:

- Periodic intervals, in order for each MRI to be executed as close to its due date as possible.
- Resources required, so the required labor for the whole block is similar.
- Similarity of tasks, in order to only remove certain components just once. For example, if the same panel must be removed in different MRIs makes sense to group those.

There are two types of blocks: **A-checks** and **C-checks**. An A-check is a lighter maintenance process and is performed every 100-150 FH. C-checks are more time-consuming and are performed around every 12-18 months or every certain number of flight hours determined by the manufacturer (Zhu, Gao, Li, & Tang, 2012).

In our data landscape, each MRI contains information about its interval (periodicity), the block it belongs to and the sub-tasks it contains. Equally, each block also has its interval described in the data landscape but, in practice, the due date for each block is the same as the earliest due date between its MRI's. Each sub-task has information about its cost, duration, required skills and required materials. Figure 2.3 shows how a Block is composed in regards to this data landscape.

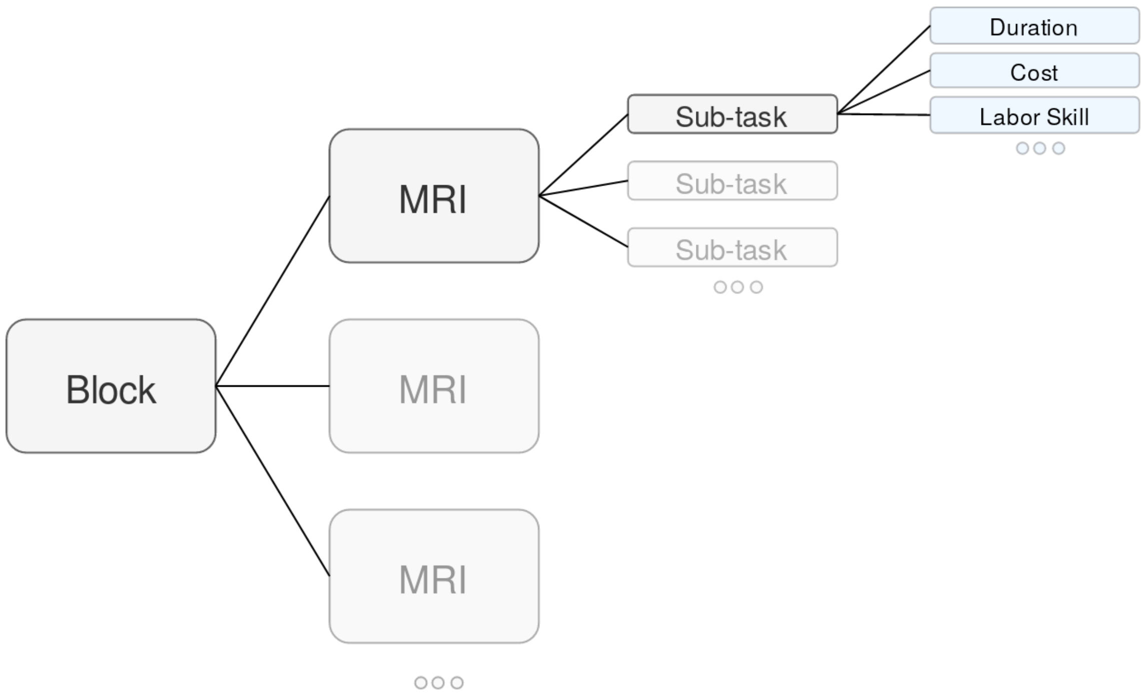


Figure 2.3: Composition of a block in the data landscape

### Snapshots

The landscape also contains information about currently open maintenance tasks, which may be maintenance blocks, out-of-phase MRI's, etc. Open tasks are not scheduled yet but contain information about its due date (the limit for when it can be scheduled). There is also a sheet with information on open faults, which are tasks that originated from crew complaints or inspections. A list of assigned and unassigned ground times (slots in which it is possible to schedule maintenance) is also described so we can know what the maintenance opportunities are, in which there are different types of slots for different aircraft and maintenance types. The final snapshot sheet is of “aircraft usage” in which the FH and FC for each aircraft since the last task performed are detailed, so we can calculate how much time is left until the next task for an aicraft needs to be performed.

**Historic**

Historic sheets are straight forward. We have information about all tasks performed, all faults resolved, historical FH and FC usage during several periods for each aircraft, and a list of how past ground times were used. The last sheet in the historic section contains an estimated utilization for each aircraft type (for each trimester), which allows us to convert FH and FC into calendar days. This estimation will be used to calculate the exact calendar day corresponding to the due date of each maintenance job.

## 2.3 Conclusion

Although there are still regulatory obstacles regarding using CBM as the standard approach for aircraft health management and maintenance planning, Advisory Council for Aeronautical Research in Europe (ACARE) foresees that by the year of 2035, CBM will be accepted as such and by 2050 all new aircraft and systems will be designed for CBM (*Strategic Research and Innovation Agenda - Volume 2 — Acare*). ReMAP puts together an innovative solution, contributing to this forecast by developing a solution which will demonstrate cost and safety benefits on implementing CBM.

Beyond the technical goals previously detailed, ReMAP wants to ensure the exploitation of CBM by involving European aviation stakeholders in the project, contributing to the discussion of how the current aircraft maintenance methods may be gradually changed to include the benefits offered by CBM. Because the proposal also foresees demonstrating the benefits of the solution on real environments, this will help validate CBM as a cost-saving and safe alternative to the current practices. This is useful not only for the ReMAP project itself but also on a research point of view, by extending the current knowledge on practical CBM implementation and also contributing to the state-of-the-art (SoA) in fields such as sensing technology, structures prognostics, systems prognostics and diagnostics, and maintenance planning and decision support.

ReMAP comprises an innovative idea, where multiple fields of knowledge are combined for the first time in order to implement a reliable CBM approach in what is a challenging environment. The expected impacts of this project are very beneficial not only from a business point of view but also as a scientific and technological ramp for future assessments.

# Chapter 3

# Background

This chapter will serve as a way for the reader to get a notion of what was the main focus of the study, given the problem to be approached. Besides the main reading on CBM and aircraft maintenance, in order to get acquainted to this type of problem and what is usually the state of mind in order to solve it, an extensive study was also made on planning and scheduling overall, the goal being to deviate from this particular scope and look at it from an outside perspective.

## 3.1 Condition-Based Maintenance

To improve engine reliability and availability, CBM is applied to gas turbine engines by using its health diagnostics and prognostics (Y. Li & Nilkitsaranont, 2009). According to the authors, engine maintenance is usually carried out at fixed intervals, regardless of each individual engine's health. After testing the prognostic approach, they concluded it provided valuable estimations in regards to the equipment health, being possible to apply them in a CBM approach.

Tian and Liao (2011) compare the costs between a proposed CBM policy for multi-component systems and another one where the policy is implemented only for single units. The thought behind it being that, because there are fixed maintenance costs such as sending a repair team to the site, repairing a single unit becomes costly. The proposed solution had the goal of replacing multiple pieces of equipment at the same time, thus reducing maintenance costs. Using two examples with data from real environments, they concluded that a multi-component CBM approach was more effective than a single-unit solution.

In a similar study, a CBM solution is proposed that extends to multi-component systems in order to replace the previous policy directed to mono-component systems (Castanier, Grall, & Bérenguer, 2005). According to the authors, an optimal decision for one unit may not be the best one for the whole system. They concluded that when the set-up costs are not null or close to zero, the proposed policy is preferred to the mono-component systems.

Grall, Bérenguer, and Dieulle (2002) measured the performance and cost-saving results on a CBM replacement for a system that is continuously deteriorating. They concluded the system adjusted well to the uncertainty of the environment, achieving good performance results on a lower cost than the classic maintenance approach.

A CBM approach was implemented in a large power plant and its benefits evaluated (Yam, Tse, Li, & Tu, 2001). The results showed that the approach resulted in reduced maintenance costs by allowing the problems to be detected before becoming critical, being possible to solve them without losing production value. The authors also stated that the earlier prediction of an incoming failure allowed to make a proper maintenance plan.

## 3.2 Scheduling/Planning

In a study that came as the inspiration for the solution utilized in our problem, Zhang and Dietterich used a temporal difference (TD) algorithm in a job-shop scheduling problem (JSP) problem (Zhang & Dietterich, 1995). A JSP consists of $j$ jobs, $n$ machines and $o$ operations. Each operation belongs to a job and requires a certain amount of time. Their goal was to create a solution that minimized constraint violations and the schedule time-span. The problem domain was a NASA (National Aeronautics and Space Administration) space shuttle payload processing, which had tasks before and after the launch. The starting state was a critical path schedule which was constructed by assigning each task to be performed before the launch as later as possible and each task to be performed after the launch as early as possible. Having the starting state structured, the algorithm would then give a negative reward for each violation the schedule had. The violations could be resolved by moving a scheduled job or assigning them to a different "pool" of resources. The results obtained outperformed the previous algorithms for scheduling space shuttle payload processing jobs.

Y.-C. Wang and Usher composed a Q-learning (QL) algorithm to schedule jobs on a single buffer, using dispatching rules (earliest due date first (EDD), short processing time first (SPT) and first-in, first-out (FIFO)) to select the next job to be processed (Y.-C. Wang & Usher, 2005). In this formulation, the number of jobs in the buffer and estimation of total lateness/tardiness is used as a computational representation of the environment's state in three different case scenarios. For example, in the first scenario, where the objective is to minimize the maximum lateness, if the tardiness of the completed job is greater then the maximum tardiness, a negative reward is given to the agent. If not, a positive reinforce happens. They concluded the agent was able to adapt and learn the best rules for different systems.

In a problem also related to an aircraft fleet, Mattila and Virtanen (2011) constructed reinforcement learning (RL) formulations to schedule aircraft maintenance, but in this case fighter aircraft under conflict. In this problem, maintenance is performed regularly with a window of tolerance, being that an airplane goes automatically to maintenance when that window is surpassed. There are two actions possible for the agent to perform: to start maintenance or to delay it. This decision is made only for aircraft that reached the window of tolerance and is also for the aircraft with the most elapsed flight hours. The state chosen was $s = (m, h)$, $m$

being the number of aircraft being maintained and $h$ corresponding to the aircraft with most flight hours exceeded the feasible window. The first formulation's goal was to maximize the average availability of the aircraft and the second was to maintain a target of availability while performing a maximum number of maintenance jobs. The algorithm found efficient policies in both formulations.

Knowles, Baglee, and Wermter combine Q-Learning and simulated scenarios to create a decision-making module to decide if a maintenance job should take place or not (Knowles et al., 2011). The simulated environment offers different scenarios to the reinforcement learning module and retrieves a reward according to the module's decision. The rewards are given accordingly to the costs associated with the maintenance jobs and breakdowns. If there is no maintenance nor failure, a positive reinforce happens, if a maintenance job takes place there is a negative reward and if a breakdown happens an even bigger negative numerical answer is reinforced. They used four different reward formulations and, even using a limited scenario, RL proved to have several benefits, having always achieved convergence in the different formulations.

### 3.2.1 Aircraft Maintenance

Samaranayake and Kiridena (2012) use a single integrated framework to plan and schedule maintenance for aircrafts. This framework was created in response to some limitations of using standalone functional applications currently available in Enterprise Resource Planning (ERP). According to the authors, the accessible generic solutions on ERP were not capable of dealing with the complexity of more intricate aircraft scheduling problems. Samaranayake and Kiridena concluded that the integrated framework was better capable of dealing with unpredictable maintenance activities.

Kleeman and Lamont (2005), in order to solve the problem of scheduling periodic routine maintenance tasks for aircraft engines, developed a General Multi-objective Parallel Genetic Algorithm (GENMOP) attempting to reduce the time each engine stays in the shop. Each engine arrives at the shop to be maintained for one of two reasons: unscheduled maintenance or routine maintenance. The authors' goal was to decrease the time each engine was in the shop. Kleeman and Lamont concluded that the algorithm could solve the problem effectively.

A scheduling problem of a military aircraft fleet is formulated as a mixed-integer mathematical programming model which is then solved by making use of a Branch-and-Bound method (Safaei, Banjevic, & Jardine, 2011). The work was created on the basis that managing aircraft availability in order to suffice the needs of the fleet flying program was an arduous task. The formulation simulated the flows of aircraft between missions, hangars, and the repair shop. The algorithm correctly described the problem and found solutions for achieving high aircraft availability.

In (Alfares, 1999), the author studies aircraft maintenance labor in an aviation department to determine how to meet the growing labor requirements with a minimum cost associated. After switching the work week from five days to seven, they defined an integer programming formulation which allowed for the aviation department to have no increase in the workforce, resulting in savings of about 13 percent ($100,000) annually.

El Moudani and Mora-Camino (2000) proposed a solution for solving the problems of assigning planes to flights using a dynamic programming (DP) approach and of fleet maintenance operations scheduling using a heuristic technique. The solution was adapted to a charter airline, which is the business of renting an entire aircraft. The authors deduced that in the various flight plans and maintenance constraints tested, the solutions obtained achieved a gain of 2.5% of total flight hours related to the manual procedures.

## 3.3 Conclusion

Scheduling of aircraft maintenance is a problem that has been around for a long time, with many studies done on how it could be improved. The problem is very complex as it is not only a matter of allocating maintenance jobs between hangars, in what is usually called service scheduling. Aircraft planning must take into account material and human resources, spare-parts available, operational needs, different types of aircraft, etc.

Several approaches created in the past as a solution to this kind of problem are available on the Enterprise Resource Planning (ERP) system, which contains various applications to aid companies on resource planning implementations, with the ability to access several systems through the same application, supplied by a relational database. According to Samaranayake (2006), the earlier solution used by many systems for global scheduling and planning was based on putting together several individual applications available on ERP such as material requirements planning (MRP), manufacturing resource planning (MRPII), etc. to compose an overall solution. This generic solutions, however, did not present all the services required for more complex maintenance problems (Samaranayake & Kiridena, 2012). Trying to overcome these problems, Maintenance, Repair and Overhaul (MRO) was created, which is also available in ERP, intending to provide specialized functionalities for aircraft maintenance scheduling. But, as Samaranayake and Kiridena (2012) pointed out, there are some problems with this system, being the most worrisome the lack of capabilities for: finite loading of resources, simultaneous planning of materials and activities, sequencing of tasks and planning of materials and resources.

While reading previous papers on CBM, common goals stand out on why different studies are being made. The goal is usually to save money and increase equipment availability and the problem mostly consists of unexpected failures and equipment being maintained before needed, bringing too many expenses for a company. We also acknowledged that CBM usually comes as a cost-saving solution for a maintenance system. We can recognize why is that by looking at a real world example about the costs between maintaining a piece of equipment prior to failure or after the breakdown, more specifically on wind turbines when replacing a generator or gearbox (*What is Your Return on Condition-Based Maintenance?*, n.d.). In it, the cost of the latter is around $400K while performing maintenance prior to breakdown is around $50K. It is, therefore, of great interest for companies that the study on

CBM continues to be done as it is a method that has already demonstrated good results and continues to do so regarding its cost-saving potential.

This page intentionally left blank.

# Chapter 4

# Reinforcement Learning

Reinforcement learning (RL) is learning by trying different actions in a certain environment and discover which ones bring a bigger reward at each moment. Unlike in supervised learning, where an agent learns from a training set with labeled data, or unsupervised learning, in which an agent tries to create groups out of data that is not labeled, in RL an agent maps his environmental observations to actions to maximize the **reward** it gets. In RL there is no prior information on what actions to take in what situations. All a learning agent knows is the **state** it is in (what the environment looks like), the possible **actions** he can perform and its **goal**. In some cases, actions do not only affect the immediate reward but also all the following ones (Sutton & Barto, 1998), meaning that the agent will have to "see" the big picture of its actions regarding the final goal. Being that an agent's goal is to have the biggest reward possible, it will choose actions it knows were efficient at achieving them in the past. We can say it is the most intuitive type of learning, as it is the closest to what we imagine when we think of *learning*, which is by trying and learning from our errors.

We can think, for example, of a dog learning how to sit by command (figure 4.1). The dog, in this case, corresponds to the learning **agent** and the person teaching it is the **environment**. The **goal** of the agent (dog) is to receive as many biscuits as possible. Let's say that the person saying "SIT" is the current state of the environment. Now let's say the dog can make one of two **actions**: the first is to sit ("SIT") and the second one is to stand ("STAND"). If the dog sits, the person will give it a biscuit, which in computational terms can be translated to "+1", for example, and if the dog stays in the same position the person will do nothing or, in numerical terms, "0". The dog, by trying both actions will discover that siting will bring it a bigger reward and will start to repeat that action every time the state of the environment is the person saying "SIT". The dog didn't know beforehand which was the correct action for each state of the environment but will learn it by trial and error.

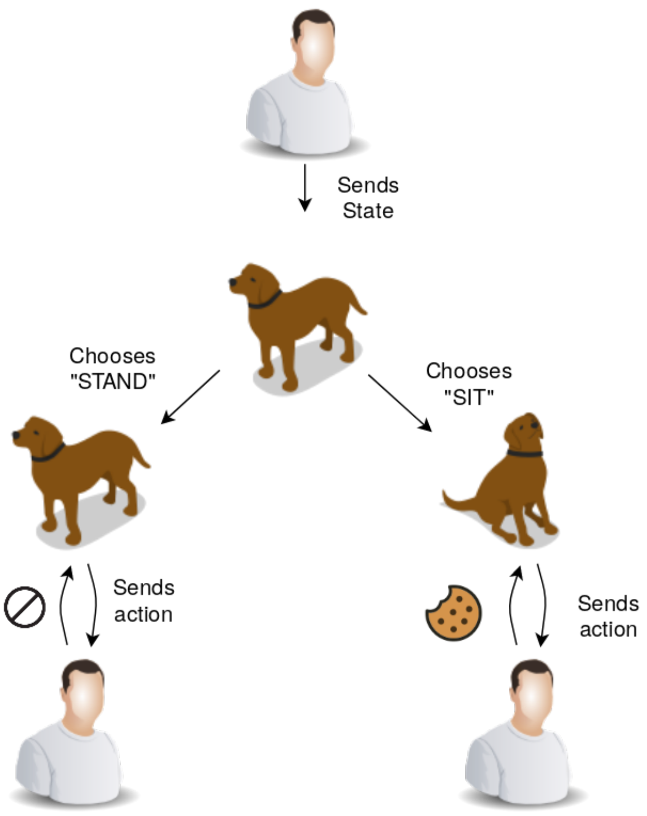


Figure 4.1: Dog learning to sit by reinforcement

Usually, it is said that one of the main characteristics of this method is, as just said, the "trial-and-error search" or, as Prof. Andrew G. Barto stated as being more suited in a presentation at the 27th International Joint Conference on Artificial Intelligence (IJCAI2018), "trial-and-evaluate". That is because, in order to discover what actions were effective in the past in achieving the bigger reward, the RL agent must experiment with the most actions possible, trying those he did not choose before. Therefore there must be a trade-off between **exploration** and **exploitation**. Exploration meaning the agent must try as many actions as possible and exploitation being the agent choosing a behaviour he already knows and getting a reward close to what he expects. This is not a trivial problem and it has been studied for many decades by mathematics. One method commonly used in RL for this problem is the $\epsilon$-greedy strategy in which we have a variable $\epsilon$ whose value declines through time. In this strategy, the agent will follow a greedy strategy with a probability of 1-$\epsilon$ and choose a random action with a probability of $\epsilon$ (Mnih et al., 2013), this way preferring random actions at the beginning of training and gradually opting for the greedy option (figure 4.2).

In short, in RL we have an agent that is in an environment it has no information on how it behaves. It only knows what is its goal and what are the possible actions to perform. The agent then interacts with the environment while learning from the experiences collected (mapping actions and rewards to states), gradually learning how to behave to get the bigger reward possible. This type of problem defined by states, actions, and goals can also, and more formally, be described as Finite Markov Decision Processes.

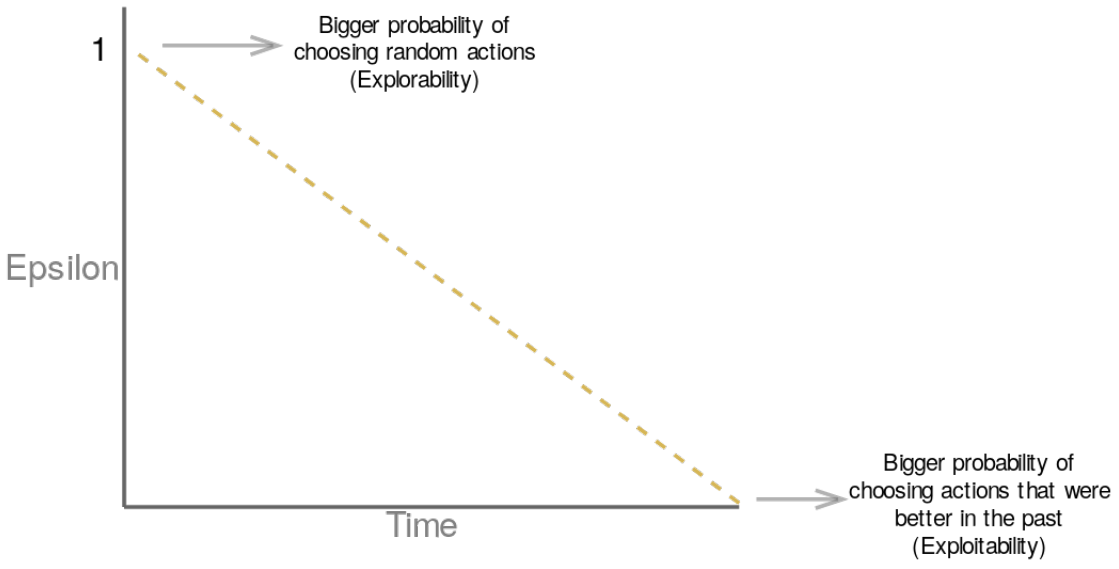


Figure 4.2: Example of a possible $\epsilon$-greedy strategy

## 4.1 Finite Markov Decision Processes

Finite MDPs are a common setting to describe reinforcement learning problems. In a finite Markov Decision Process or MDP we have finite set $S$ of states, each of them with a set of $A$ possible actions. We have several time steps $t = 0, 1, 2...$, having in each one of them the agent making a decision of what action to take, and the environment responding to it. This interaction will cause the agent to be on another state, one step later, receiving a respective numerical reward $R_{t+1}$.

This iterative interaction between the agent and the environment results in a sequence of events as shown in figure 4.3, where we see that triggering an action in a certain state causes the environment to change. Afterwards, the agent receives information about the new state its in and the respective reward for performing action $a$ while in state $s$.

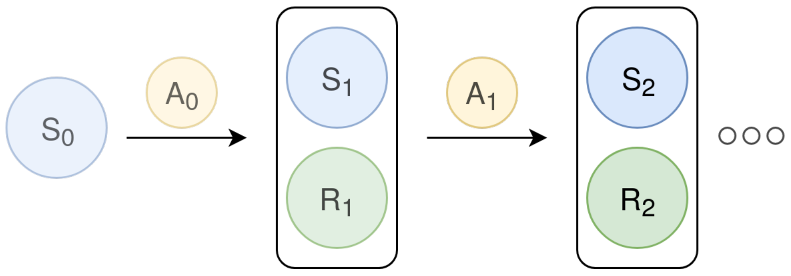


Figure 4.3: Sequence of events in a MDP

The dynamics of a finite MDP are defined by a set of probabilities. Having the agent performing action $a$ in state $s$ at time $t-1$, there is a probability for $s' \in \mathbb{S}$ and $r \in \mathbb{R}$ to occur in time-step $t$:

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}, \\ \text{for all } s', s \in \mathbb{S}, r \in \mathbb{R}, \text{ and } a \in \mathbb{A}. \tag{4.1}$$

These probabilities are often used in a graph to represent the dynamics of a finite

MDP, in order to summarize the problem. In figure 4.4 we see an example of that representation where we have 3 states, $|S| = 3$, and two actions, $|A| = 2$. Table 4.1 shows the transition probabilities for each state-action pair. If a pair does not appear on the table, it means it has a null probability.

| s | a | s' | p(s'—s,a) |
|---|---|---|---|
| S0 | A1 | S1 | $\beta$ |
| S0 | A1 | S2 | 1 - $\beta$ |
| S0 | A2 | S0 | 1 |
| S1 | A1 | S0 | 1 |
| S1 | A2 | S2 | 1 |
| S2 | A1 | S0 | $\alpha$ |
| S2 | A1 | S1 | 1 - $\alpha$ |
| S2 | A2 | S1 | $\theta$ |
| S2 | A2 | S2 | 1 - $\theta$ |

Table 4.1: Transition probabilities for the example's MDP

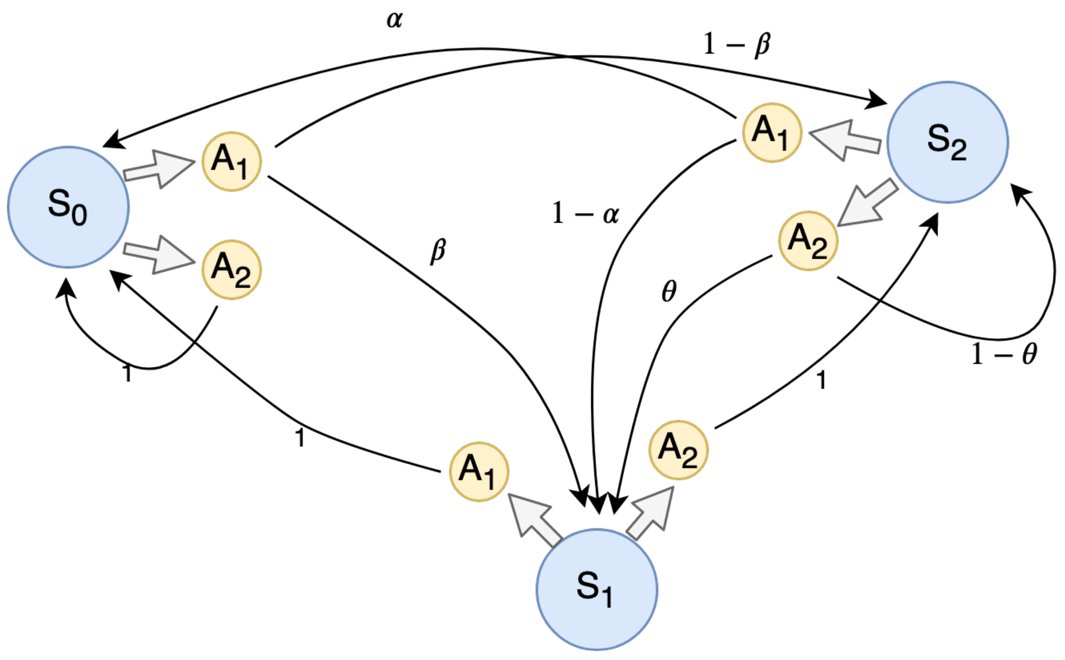


Figure 4.4: Transition graph given the example's transition probabilities

Knowing how to calculate state-transition probabilities we can also calculate the expected reward for having performed action $A_{t-1}$ in state $S_{t-1}$, for each state-action pair:

$$r(s, a) = \mathbb{E}\Big[R_t | S_{t-1} = s, A_{t-1} = a\Big] = \sum_{r \in \mathbb{R}} r \sum_{s \in \mathbb{S}} p(s', r | s, a). \tag{4.2}$$

Given this information, the question that remains to answer is how to solve a problem represented by a MDP. How does an agent find the best solution for a given environment? This is where **value functions** are introduced. But to understand what value functions are, we must know about policies first. A *policy*, $\pi$, is a mapping assigning each state to a set of probabilities of choosing each action at each time step. We can say that $\pi(a|s)$ is the probability of the next action being $a$ if the current state is $s$. In short, a policy is a way for the agent to act.

### 4.1.1 Value Functions

Value functions estimate on how good it is for the agent to be in a certain state $s$. "How good" meaning the expected reward for the agent, having it starting from state $s$.

In MDP, there are two types of value functions. The first is the *state-value function* which tells us the value of being in a state and then following a policy $\pi$ and is described by:

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t = s\right], \text{for all } s \in \mathcal{S}, \tag{4.3}$$

where $\mathbb{E}_\pi$ represents the expected value given a current policy, $\pi$, $\gamma$ is the discounting factor, $R_t$ the reward at time step $t$ and $S_t$ the state at time-step $t$.

The *action-value function*, however, defines the value of taking an action $a$ in a state $s$ and consequently following policy $\pi$ and it's defined by:

$$q_\pi(s,a) = \mathbb{E}_\pi\Bigg[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\Bigg|S_t = s, A_t = a]. \tag{4.4}$$

Adding to these, we can relate the returns given by successive time steps using a *Bellman* equation. This will allow us to calculate the value of a state using the value of the following state. The Bellman equation for $v_\pi$ is defined as:

$$v_\pi(s) = \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big], \text{ for all } s \in \mathcal{S}, \tag{4.5}$$

where $p(s',r|s,a)$ reads the probability of the next state being $s'$ and reward being $r$ when taking action $a$ in state $s$.

Knowing how to calculate the value of one state given the value of the next states, we start to understand how it is possible for the the values to be constantly updating in order to find a solution. The next step will be to make use of value functions to search for the **optimal** policy. Among all possible policies there is at least one that will be optimal, meaning a policy that is equal or better than all the others. There are two *Bellman optimality equations*:

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_*(s')\Big]. \tag{4.6}$$

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \underset{a'}{max}\ q_*(s',a')\Big]. \tag{4.7}$$

The first equation (4.6) is the Bellman optimality equation for $v_*$ and the second (4.7) is the Bellman optimality equation for $q_*$. They express that, in an optimal policy, the value of a state will be equal to the expected return for the best action to take from that state. To find the optimal policy is to solve a reinforcement learning problem.

Now it is left to know how to update our policy in order to achieve an optimal one. For that we have two methods: **Policy Iteration** and **Value Iteration**. Let's start with the further one. We saw that with equation 4.5 we can calculate the value of a state-value function for a policy $\pi$. By using that equation as an update rule, we can calculate, for each state, an approximation of its value:

$$v_{k+1}(s) = \sum_{a} \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_k(s')\Big], \tag{4.8}$$

where the approximation for $v_{k+1}$ is calculated using the value from $v_k$ and by applying it to each state, $s \in S$, we are performing iterative **policy evaluation**.

In order for us to know whether it would be better or not to change policies, we must calculate what is the value of choosing a different action $a$ in state $s$ and after that following policy $\pi$, which is done using:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]. \tag{4.9}$$

And according to the **policy improvement** theorem, if the value for this new policy $\pi'$ is bigger than the value from the old policy, $\pi$ or, formally:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s), \tag{4.10}$$

then $\pi'$ is as good or better than $\pi$ and the new policy can be updated by, from now on, always following action $a$ in state $s$.

By combining both steps above, *policy iteration* and *policy improvement*, we have policy iteration. This method starts with a random policy and goes on improving it from there by a series of iterations. According to Sutton and Barto (1998), policy iteration usually converges to an optimal policy in few iterations.

The value iteration method is similar in a lot of ways to policy iteration with a small difference in the updating function, where the policy evaluation is truncated into the update function and is calculated for all $s \in S$, using:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_k(s')\Big], \tag{4.11}$$

where the main difference between equation 4.11 and the policy iteration update is the use of the *maximum* function. By truncating the policy evaluation step, the computation required for each step is not as heavy as in the policy iteration method.

Knowing this we know the mathematical way of solving a MDP. But when applying a MDP framework to a problem there is a level of abstraction to it, left to the programmer to clarify. In computational terms, when solving a problem represented by a MDP we have to define how a state is described (**state representation**), it can be a matrix, an array, etc. Also we have to define our possible **actions** and, lastly, we have do define our **reward functions**.

As stated by Sutton and Barto (1998), the MDP scheme is very flexible and it can be adapted to many different problems in various ways. That adaption is a matter of how the problem is interpreted given that this framework is very abstract. For example, in a game of chess the state could be represented by a matrix containing each position of each piece of the game and the actions would probably be each possible move for the player. In a self-driving taxi, the state of the environment would have to contain information about everything around him. It would have to represent the distances to the cars around it, the speed limit for the road its in, which traffic signs it sees, the color of the traffic lights, the direction the taxi is going, etc. And the possible actions could be: accelerate, brake, move the wheel left, move the wheel right, etc. It is up to the person solving the problem to think of what makes sense for the the state representation and possible actions to be.

In these two examples, only the states and actions were described, with one third and final element left to talk about, also crucial in the design of a formulation to this type of problem, and that is the **goal** of the agent. The agent's goal will be decided by the reward functions created by the programmer. For example, in the game of chess, a positive reinforce can be given every time the agent wins a game and a negative one every time it loses. In the taxi environment, a negative reinforce may be given each time an infraction is committed and a positive reward when the taxi reaches its destiny, etc. Although it seems trivial, the reward definition can also be a delicate process as Sutton and Barto point out also using chess as an example. Coming across the chess problem we could think of defining the rewards by giving a positive reinforcement to the agent each time it captures a piece from the opponent and, in the opposite situation, give the agent a negative reward. The problem with this formulation, in particular, is that the chess agent might start losing games but still achieving big rewards because it would not be interested in winning the game but only on capturing the opponent's pieces. While, instead, giving it a positive reward only at the end of the game, whenever it wins, might prompt the agent to even sacrifice its own pieces in order to achieve victory. This is another interesting aspect regarding the agent's behavior, the fact that it maximizes not only the immediate reward but the cumulative one it receives throughout the whole run. Putting it simply, the agent will always try to maximize its reward. It is up to the programmer to implement reward functions in which maximizing the reinforce obtained equals achieving the goal intended.

## 4.2 Q-Learning

Q-learning (QL) is a RL tabular method in which an agent can learn directly from the environment by constantly updating its state-action pairs values (Q-values). The iterative process is done by constantly updating the QL table using the Q-function (Watkins, 1989):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[ R_{t+1} + \gamma \, \underset{a}{max} \, Q(S_{t+1}, a) - Q(S_t, A_t) \Big], \tag{4.12}$$

where $\alpha$ represents the learning rate, $R_{t+1}$ the immediate reward for taking an action, $\gamma$ is the discount rate and $Q(S_t, A_t)$ the Q-value of the state-action pair corresponding to state $S$ and action $A$.

In this algorithm, we start with a table of Q-values with one element for each state-action pair possible, all initialized at zero. In each action the agent performs, the respective state-action pair is updated using equation 4.12. This way the agent keeps track of what actions are the most valuable.

To better understand how the algorithm works we'll go through a quick example where figure 4.5 represents our environment and figure 4.6 shows the Q-table with all state-action pairs initialized at zero.

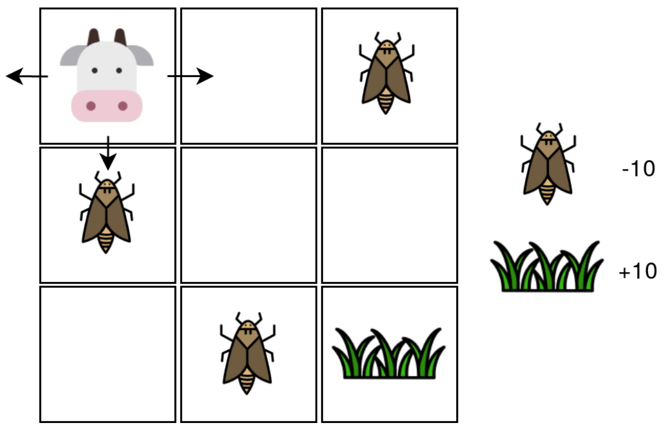


Figure 4.5: Example's environment and agent

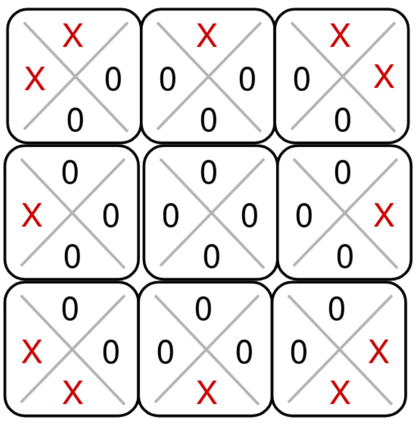


Figure 4.6: Example's Q-Table

In our environment there are nine tiles (states), in which some of the tiles contain bugs and one of them has grass. The set of possible actions, $A$, for our agent are "LEFT", "RIGHT", "UP" or "DOWN" and its goal is to get to the grass while avoiding the bugs. As mentioned earlier, one of the jobs for the programmer is to decide on a reward function that will represent the agent's goal. In this case let's define the reward for going to a bug tile as being "-10" and reaching the grass as a positive reinforcement of "+10".

Let's now see how the updates to the Q-table are made. Imagine the agent would make the move depicted in figure 4.7. The Q-value for the respective state-action pair would then be updated using equation 4.12. For this problem let's say $\alpha = 0.1$ and $\gamma = 0.99$. We get:

$$Q(S_t, A_t) = 0 + 0.1[10 + 0.99 * 0 - 0] = 1. \tag{4.13}$$

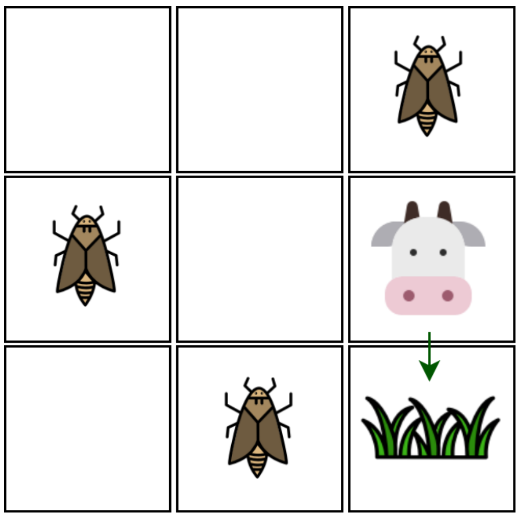


Figure 4.7: Agent moving to goal

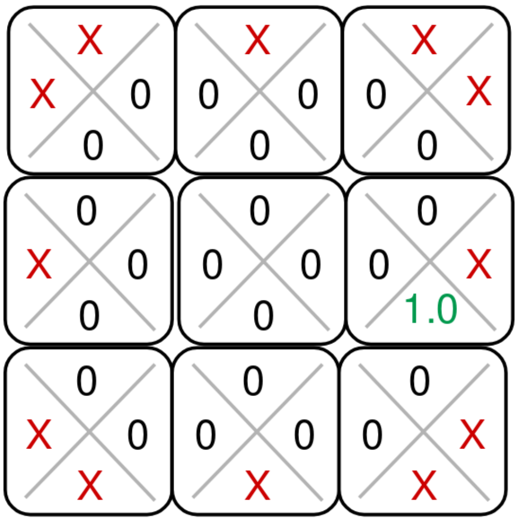


Figure 4.8: Respective state-action value updates

Now whenever the agent moves to the state that was just updated, the state from which it moves is updated using the new Q-value. For example, by performing the move shown in figure 4.9, the respective state-action pair value would be updated with:

$$Q(S_t, A_t) = 0 + 0.1[0 + 0.99 * 1 - 0] = 0.099, \tag{4.14}$$

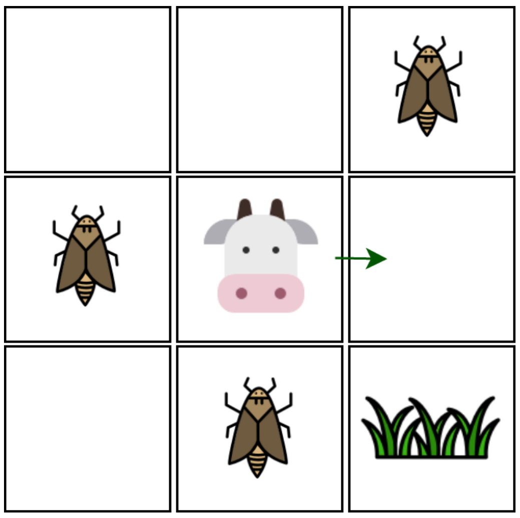


Figure 4.9: Agent moving to previously updated state

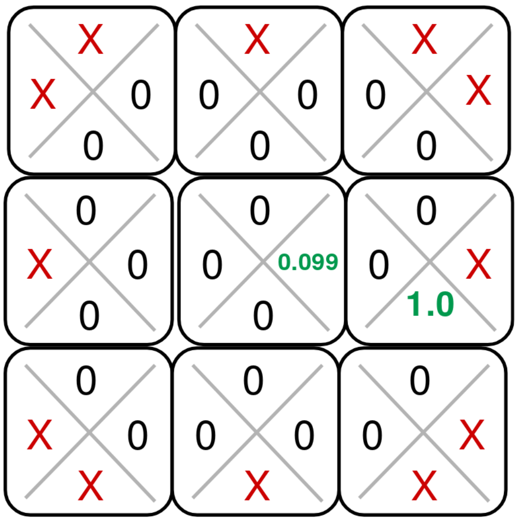


Figure 4.10: Respective state-action pair being updated

resulting in the Q-table shown in figure 4.10.

It is important to point out the importance of the discount factor, $\gamma$. This variable is defined by the programmer with values between 0 and 1 and will determine the emphasis we give to the value of the next state. If $\gamma$ is zero, for example, the value of $Q(S_t, A_t)$ will be equal to the immediate reward, meaning that the agent will have "short vision", not being able to maximize its reward in the long run. If $\gamma$ is one, the Q-values will be calculated by giving a significant importance to future actions.

We also see the importance of the exploration and exploitation trade off in the agent's trial-and-error search. If the agent doesn't explore the state space correctly, it might not achieve the best path to its goal. So it is crucial that, in a certain point of the algorithm, our agent tries different actions for different states in order to discover the best behaviour possible.

### Deep Q-Learning

Contrary to Q-Learning where we use a matrix to save the state-action pairs' values, in Deep Q-Learning a Neural Network ( DQN) is used instead. This comes to solve problems in which the state space is too big and using a matrix becomes technically infeasible.

A DQN usually will have a large number of inputs which enables, together with reinforcement learning, the learning of a successful policy (Mnih et al., 2015). With this method, we are not limited to a small state space and can use reinforcement learning in a much more complex environment.

Commonly, when new DQL algorithms are introduced, they are tested in video-games, more precisely in the *Atari 2600* domain. As a way of getting acquainted with the method, one of the tasks performed in this internship was one of mimicking the implementation presented by (Mnih et al., 2015). A detailed explanation of this process is presented in annex section A.2.

## 4.3 Current Approaches

One of the most known articles about Deep Q-Learning is (Mnih et al., 2015). In the paper, the authors give a thorough description of the algorithm they used from the neural network architecture to all the hyper-parameters used. The algorithm was tested in dozens of Atari games having surpassed the performances of all the previous algorithms at the time.

The Asynchronous Actor-Critic (A3C) algorithm is introduced in (Mnih et al., 2016), presenting a different approach where several agents would run asynchronously, each one of them having his own neural network (NN). In the algorithm, each agent's particular NN is then used to update a global one. Having different agents with their own NN ensures that the state space is well explored, being that different agents will acquire different observations. The algorithm was also tested in Atari games having surpassed the results from prior state-of-the-art algorithms.

In (Abbeel, Coates, Quigley, & Ng, 2007) RL is used successfully in the task of having a remote controlled (RC) helicopter autonomously performing four acrobatic maneuvers.

Bakker, Zhumatiy, Gruener, and Schmidhuber implement a RL agent with memory (long short-term memory (LSTM) network) so that a robot, which relies on sensory inputs to extract information about the environment, remembers relevant information from the past to achieve it's goals (Bakker et al., 2003). The Robot had to go through a T-maze and had two goal positions. It rapidly learned what to do after only collecting twelve episodes in which the Robot was hand-controlled through the maze. In another robotics application, Kober, Muelling, and Peters successfully use RL in order to teach a robot to hit a ball in table tennis (Kober et al., 2012).

There are several variations to the Deep Q-Learning formula. Van Hasselt, Guez, and Silver and Z. Wang et al. present two adaptations to the algorithm, the first being Double Q-Learning (Van Hasselt et al., 2016) and the second Dueling Network Architectures (Z. Wang et al., 2015). In the Double Q-Learning, a new neural network is added, called the target-network, responsible for calculating the target values for the online-network's update. The target-network is updated every $t$ steps, copying the weights of the online-network. This solution was presented to solve the issue of the Deep Q-Learning algorithm overestimating action values under certain conditions. Dueling network architecture is similar to the Deep Q-Learning algorithm, the difference being that after the last convolutional layer, instead of a fully-connected layer the stream is divided into two, for value and advantage estimations. These two values are then combined to calculate the Q-values. The motivation for this solution was the fact that many times the actions performed do not have significant effects on the environment's state and this way it's possible to measure the importance of each action.

In a recent study, DRL was applied in chat-bots to model their dialogue rewards. The new approach was compared in terms of diversity, length and human judges with the sequence-to-sequence (SEQ2SEQ) model (J. Li et al., 2016). The final result had the RL model giving longer conversation before start giving "dull" responses (length), more diverse outputs and produced more interactive and sustained conversations.

Mao, Alizadeh, Menache, and Kandula (2016) developed a DRL based solution for a multi-resource cluster scheduling problem, called *DeepRM*. The authors wanted to test if machine learning could replace human-generated heuristics in the problem of resource management. Mao et al. stated that their algorithm didn't need prior information about the environment and supported various goals by using different reward functions. Their experimental results showed that *DeepRM* performed as good or better than SoA heuristics.

## 4.4 Conclusion

Reinforcement learning (RL) is a method that has been in constant evolution. Just in the last few years, there was a great number of papers on DRL, several of them even outshining the performance of previous SoA algorithms.

Probably the most known application for RL algorithms is in the games department where it had impressive achievements. AlphaGo (Silver et al., 2016) and AlphaGo Zero (Silver et al., 2017) became very famous by beating professional *Go* players and AlphaGo Zero even became the world's top player. Recently OpenAI Five, a Dota 2 team of video-game bots which is programmed with RL, was able to beat for the first time, live, the Dota 2 world champions. These are two very complex games, very hard to master. And while the board game Go is played 1vs1, Dota 2 is played in teams of 5, so the agents had to cooperate in order to win.

But not only in the gaming world has RL been successful. RL has shown to promising results in robotics (*This Factory Robot Learns a New Job Overnight - MIT Technology Review*, n.d.), health-care (Gottesman et al., 2019), resource management (Tesauro, Jong, Das, & Bennani, 2006), chemistry (Zhou, Li, & Zare, 2017), etc. As Koray Kavukcuoglu, the director and researcher at DeepMind, stated while giving a talk at 6th International Conference on Learning Representations (ICLR 2018), combining reinforcement learning, which is a general framework for sequential decision making, together with deep learning (DL), which has the best set of algorithms for learning representations, is the best answer we have at the moment to in order to learn state representations and solve challenging problems not only in ”toy problems” but also on a real-world environment.

# Chapter 5

# Scheduling Maintenance Tasks with Reinforcement Learning

In this chapter, the problem description is going to be detailed alongside the formulations built to represent it. In order to solve those formulations, a deep reinforcement learning (DRL) solution was also developed which is also going to be explained together with its results.

In the last month of the internship, we were given access to relevant data which allowed us to adapt our solution to a real world problem. However, there was no access to a concrete problem with real data throughout the majority of the internship's time-span, thus being necessary for an academic approach to be taken to fulfill the needs of this work. This chapter details both the real data adaptation and the toy-problems created.

It is worth noticing that the final goal of the maintenance decision support tool (MDST) has been stripped down to a partial objective for this initial work phase. ReMAP's ultimate goal is to use the aircraft's health information given by RUL estimations to create an adaptive maintenance plan, aimed to help maintenance operators in their day-to-day operations. This maintenance plan will combine both periodical (A-checks and C-checks) and non-periodical (unexpected) maintenance jobs (figure 5.1). Still, for now, only periodical tasks are to be taken into account.

## 5.1 Problem Description

In our three different artificial formulations, we have a set of tasks, $T$, to be distributed between a set of $D$ days, making use of $H$ hangars. Each task has a length $l$ and a due date $dd$ and each hangar holds at maximum one task per day. We adopt the vernacular days and hangars but could equally use, for example, "slots" and "resource groups", as the challenge and final results would be the same. Nevertheless, as a matter of consistency, this is the language we are going to use throughout the rest of the report.

Using the real world data, the problem holds similar with minor distinctions. In

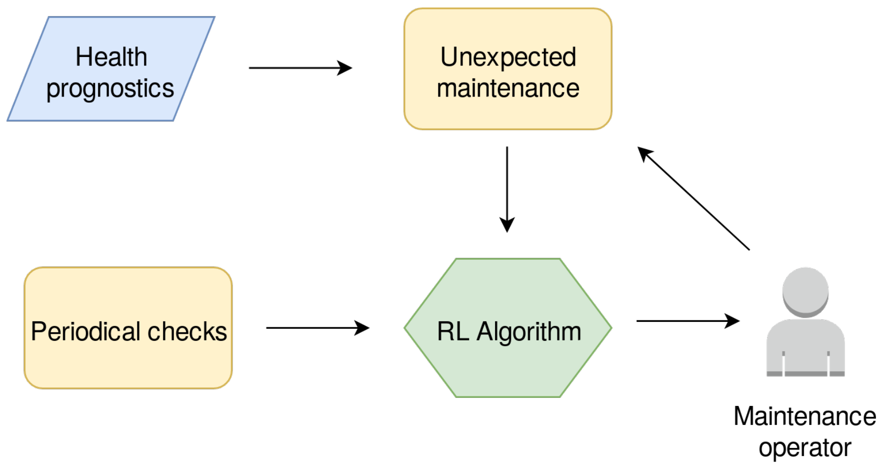


Figure 5.1: MDST's high level architecture

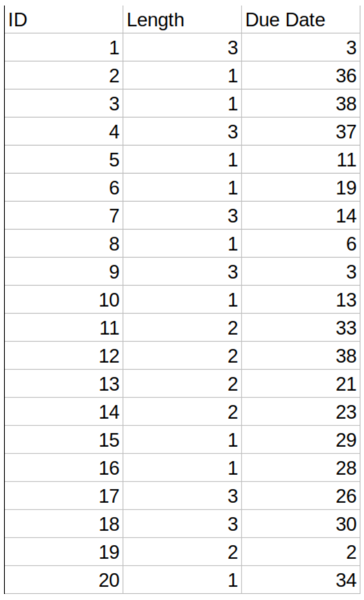
it, we have one A-check and one C-check assigned to each aircraft. We will call "task" to the act of performing an A-check or C-check and refer to these as simply "checks". In the data file, the checks do not have assigned due dates but appointed intervals instead. These intervals tell us the periodicity of each check. Therefore we obtain information about the due date of a task by subtracting the interval of the respective check to the day it was last performed. So, for example, if we have one check with an interval of 2 calendar days to be scheduled for a period of 10 days, the algorithm will probably have to schedule 5 maintenance tasks. The real data formulation ends up being similar to the toy-problems because each check will be converted to a group of tasks, $T$, each of them with an assigned due date, $dd$, and a length, $l$. In this problem each day can take up to 3 C-check type maintenance jobs and 1 A-check.

The goal for each formulation is to schedule each task as close to its due date as possible while respecting the maximum possible maintenance jobs per day. The idea behind this goal is to fully take advantage of the assigned intervals between maintenance jobs. If we think about the real case scenario, by doing so not only we will take better advantage of each system's RUL, but also there will be fewer maintenance checks thus increasing aircraft availability.

## 5.2 Set of Toy-Problems

Three problem formulations were created to test the reinforcement learning (RL) algorithm in the problem of scheduling. Besides using DRL, a greedy approach was also implemented for each formulation as a way of comparing results. At a point in time, it was also decided to implement a genetic algorithm (GA) as a benchmark tool. Although implemented with results obtained, the GA idea was then dropped, but the concept, as well as some results, are documented in annex B.

The data used for these exercises was artificially created and decided together with the project's partners who briefed us that this abstract data was in a similar format as that of future data, regarding periodic tasks. The data was created using a *python* script in which it is possible to choose the number of tasks to be created and the number of days the tasks are to be distributed in. A *xlxs* file is then returned with a number $|T|$ of tasks with lengths, $l$, between two values at choice and due dates, $dd$, between $l$ and $|D|$. In figure 5.2 an example of an input is shown with $|T|$=20, $|D|$=40 and $1 \leq l \leq 3$.



| ID | Length | Due Date |
|---|---|---|
| 1 | 3 | 3 |
| 2 | 1 | 36 |
| 3 | 1 | 38 |
| 4 | 3 | 37 |
| 5 | 1 | 11 |
| 6 | 1 | 19 |
| 7 | 3 | 14 |
| 8 | 1 | 6 |
| 9 | 3 | 3 |
| 10 | 1 | 13 |
| 11 | 2 | 33 |
| 12 | 2 | 38 |
| 13 | 2 | 21 |
| 14 | 2 | 23 |
| 15 | 1 | 29 |
| 16 | 1 | 28 |
| 17 | 3 | 26 |
| 18 | 3 | 30 |
| 19 | 2 | 2 |
| 20 | 1 | 34 |

Figure 5.2: Example of a table with a group of tasks to be scheduled

As for the output, a scrollable window is returned showing which tasks are scheduled on which days by coloring the respective slot (figure 5.3). If the color of a slot is green it means that the task is scheduled on its due date, if it's yellow it's scheduled before the due date and if it is red it is set to after the due date. To more detailed information a *xlxs* file is also returned detailing, for each task, the difference between its due date and the day it was scheduled (figure 5.4).

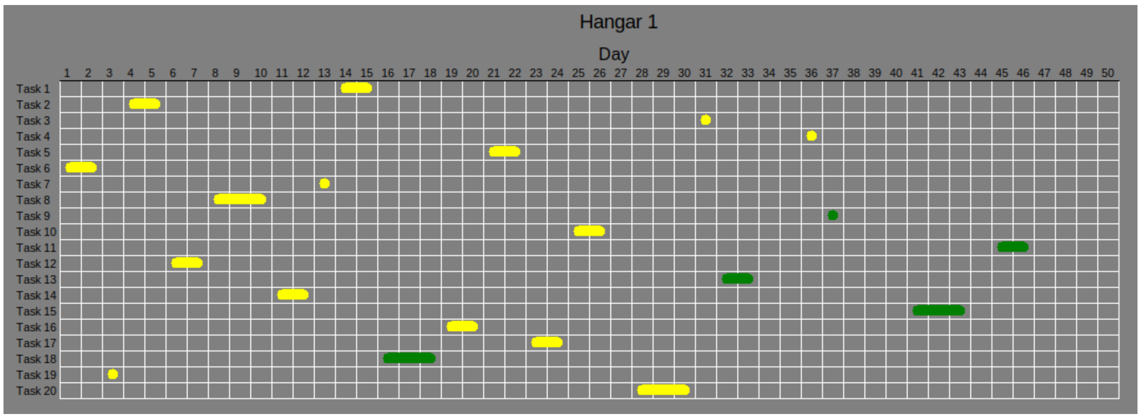
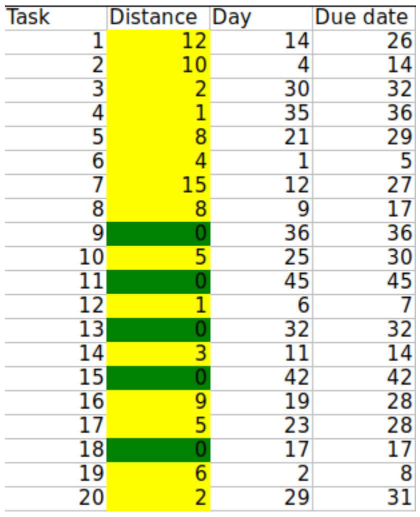


Figure 5.3: Output of a calendar with colored slots

### 5.2.1 Formulation 1

This was the initial thought formulation and it is the simplest of the three. The idea is to take a set, $T$, of tasks as shown in figure 5.2 and arrange them without



| Task | Distance | Day | Due date |
|---|---|---|---|
| 1 | 12 | 14 | 26 |
| 2 | 10 | 4 | 14 |
| 3 | 2 | 30 | 32 |
| 4 | 1 | 35 | 36 |
| 5 | 8 | 21 | 29 |
| 6 | 4 | 1 | 5 |
| 7 | 15 | 12 | 27 |
| 8 | 8 | 9 | 17 |
| 9 | 0 | 36 | 36 |
| 10 | 5 | 25 | 30 |
| 11 | 0 | 45 | 45 |
| 12 | 1 | 6 | 7 |
| 13 | 0 | 32 | 32 |
| 14 | 3 | 11 | 14 |
| 15 | 0 | 42 | 42 |
| 16 | 9 | 19 | 28 |
| 17 | 5 | 23 | 28 |
| 18 | 0 | 17 | 17 |
| 19 | 6 | 2 | 8 |
| 20 | 2 | 29 | 31 |

Figure 5.4: *xlxs* file containing output information

overlaps between $D$ days and $H$ hangars, creating a feasible calendar.

Not only in this but in the other formulations, tasks can be scheduled after the due date, although this does not happen unless there is no other option available. It is also possible for a task not to be scheduled but, generally, only exercises where all tasks can be scheduled are used.

This formulation was inspired by the base concept of the problem in which, simply putting it, we have several tasks and need to arrange them in an interval of time.

### 5.2.2 Formulation 2

In this formulation we have a group of tasks just like in the first one but, unlike before, the days advance in time. Every time a feasible calendar is created with all the tasks scheduled, the time moves forward one day. Whenever a task is completed, it is rescheduled again to a new due date. This new due date is the same as the original with a random variation between -3 and 3 days. In this exercise, the maximum days used were 100 and the viability of the solution was examined by calculating the mean of the rewards of the calendars obtained in each day. Also, tasks can not be rescheduled to the first 10 days in order to avoid reschedules being made too close to the current day.

This interpretation was based on the fact that we wanted to introduce some randomness into the problem as the due dates may vary for tasks belonging to the same system, depending on the previous maintenance day or some issue with the system.

In figures 5.5 and 5.6 we can see an example of this formulation behaviour. In the first image, we see the sixth task, $t6$, close to being completed and in the second figure, when a day advances, the task is rescheduled.

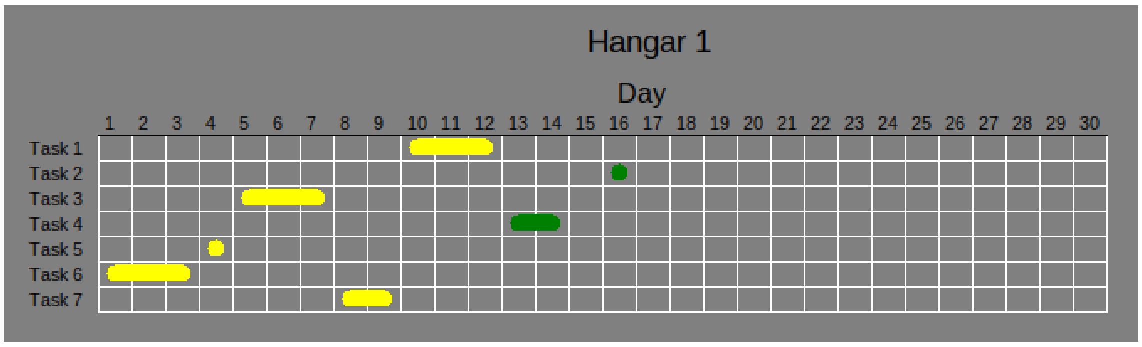


Figure 5.5: Task 5 almost maintained

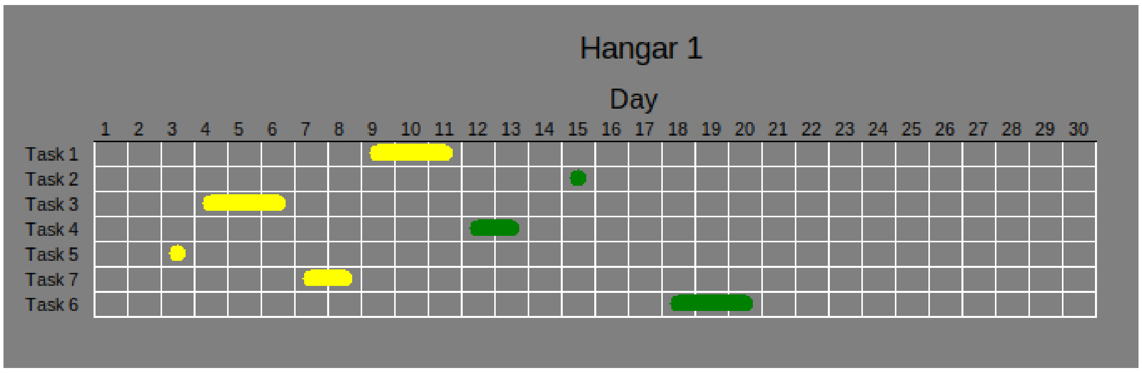


Figure 5.6: Task 5 after re-scheduling

### 5.2.3 Formulation 3

In the last artificial definition, we have different types of slots and tasks. Each task must only be scheduled to a slot specific to it. This formulation is inspired by different types of checks of different aircraft models having slots specified for them. This way a company knows that, in that specific slot, there are human and material resources ready for that maintenance job. Currently, maintenance teams already know that, for example, on Monday an aircraft of type $A$ checks into maintenance while on Tuesdays an aircraft of type $B$ is maintained. This way it is known in advance what is the labor required for each day.

So, for example, if a task's due date in on the twentieth day, $dd = 20$, but the nearest possible slot is on day 15, this is the slot where the task is going to be scheduled. Because this is slot-based, all tasks have length $l = 1$.

In figure 5.7 we can see three different types of tasks and days represented by different colors. All the tasks are scheduled on slots assigned specifically for that type.

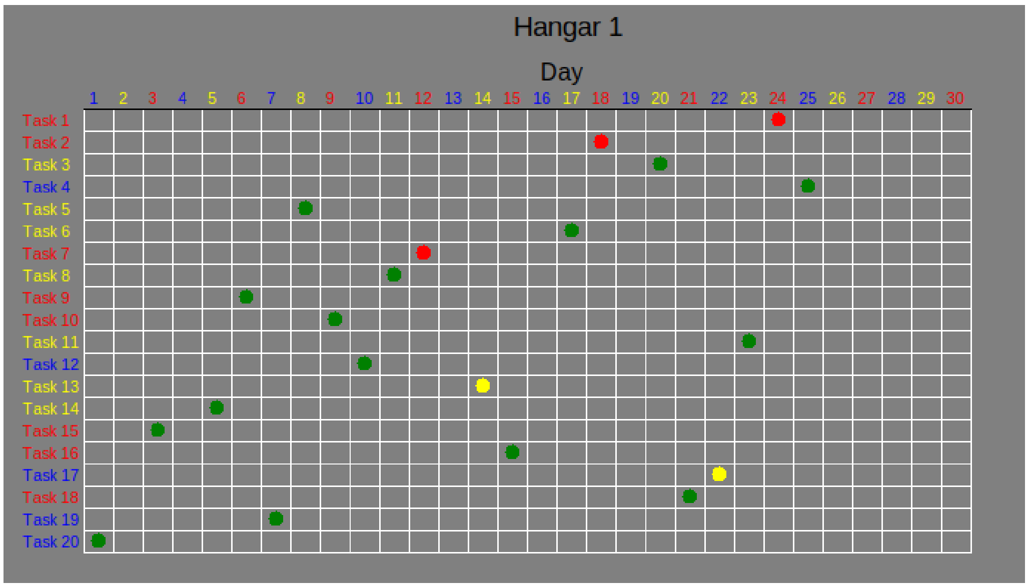


Figure 5.7: Example of a calendar with different slots and task types

## 5.3 Aircraft Real Data Formulation

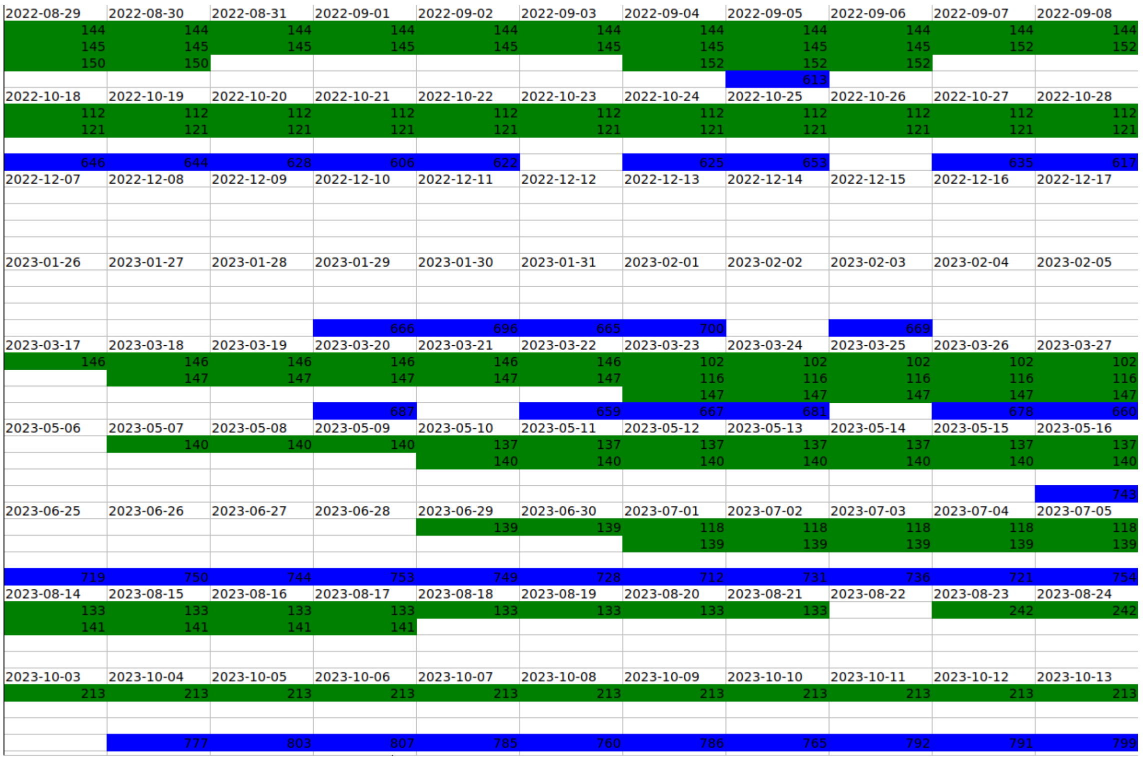
This formulation is that in which relevant aircraft maintenance data is used. The data-set contains information on 51 aircraft, with details about the corresponding A-Check and C-Check for each one of them (*Annex C*). Throughout the data file, every piece of data that involves time-related information is given in three different usage parameters: flight hours (FH), flight cycles (FC), and calendar days (DY).

For each A-Check and C-Check we have the following knowledge:

- Elapsed time since previous check
- Check's interval
- Tolerance (amount of days after the due date in which it is still allowed to schedule a specific task)

There is also information on what is the amount of time the schedule must cover, which in this case is 6 years, equivalent to 2190 days. There are seven C-Checks with an interval of 1096 DY, the rest having an interval of 730 DY. A-Checks have an interval of 120 DY. This all adds up to 919 A-Checks and 116 C-Checks, in a total of 1035 tasks to be distributed between the 2190 days. Each A-Check has a length of 1 DY and C-Checks duration varies between 10 and 30, that information being on the data file. Because we counted weekends as possible days for maintenance, we added 5 days to each C-check, in order to hamper the problem. Each day can take up to 3 C-Checks and 1 A-Check.

The output for this formulation is a *xlxs* which replicates the calendar. In the file, each day has 3 available slots for C-checks (green colored) and one for A-checks (blue colored). Figure 5.8 shows a fraction of a possible calendar in which the numbers inside the colored slots represent the respective task's number.



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 2022-08-29 | 2022-08-30 | 2022-08-31 | 2022-09-01 | 2022-09-02 | 2022-09-03 | 2022-09-04 | 2022-09-05 | 2022-09-06 | 2022-09-07 | 2022-09-08 |
| 144 | 144 | 144 | 144 | 144 | 144 | 144 | 144 | 144 | 144 | 144 |
| 145 | 145 | 145 | 145 | 145 | 145 | 145 | 145 | 145 | 152 | 152 |
| 150 | 150 | | | | | 152 | 152 | 152 | | |
| | | | | | | | 613 | | | |
| 2022-10-18 | 2022-10-19 | 2022-10-20 | 2022-10-21 | 2022-10-22 | 2022-10-23 | 2022-10-24 | 2022-10-25 | 2022-10-26 | 2022-10-27 | 2022-10-28 |
| 112 | 112 | 112 | 112 | 112 | 112 | 112 | 112 | 112 | 112 | 112 |
| 121 | 121 | 121 | 121 | 121 | 121 | 121 | 121 | 121 | 121 | 121 |
| | | | | | | | | | | |
| 646 | 644 | 628 | 606 | 622 | | 625 | 653 | | 635 | 617 |
| 2022-12-07 | 2022-12-08 | 2022-12-09 | 2022-12-10 | 2022-12-11 | 2022-12-12 | 2022-12-13 | 2022-12-14 | 2022-12-15 | 2022-12-16 | 2022-12-17 |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| 2023-01-26 | 2023-01-27 | 2023-01-28 | 2023-01-29 | 2023-01-30 | 2023-01-31 | 2023-02-01 | 2023-02-02 | 2023-02-03 | 2023-02-04 | 2023-02-05 |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | 666 | 696 | 665 | 700 | | 669 | | |
| 2023-03-17 | 2023-03-18 | 2023-03-19 | 2023-03-20 | 2023-03-21 | 2023-03-22 | 2023-03-23 | 2023-03-24 | 2023-03-25 | 2023-03-26 | 2023-03-27 |
| 146 | 146 | 146 | 146 | 146 | 146 | 102 | 102 | 102 | 102 | 102 |
| | 147 | 147 | 147 | 147 | 147 | 116 | 116 | 116 | 116 | 116 |
| | | | | | | 147 | 147 | 147 | 147 | 147 |
| | | | 687 | | 659 | 667 | 681 | | 678 | 660 |
| 2023-05-06 | 2023-05-07 | 2023-05-08 | 2023-05-09 | 2023-05-10 | 2023-05-11 | 2023-05-12 | 2023-05-13 | 2023-05-14 | 2023-05-15 | 2023-05-16 |
| | 140 | 140 | 140 | 137 | 137 | 137 | 137 | 137 | 137 | 137 |
| | | | | 140 | 140 | 140 | 140 | 140 | 140 | 140 |
| | | | | | | | | | | |
| | | | | | | | | | | 743 |
| 2023-06-25 | 2023-06-26 | 2023-06-27 | 2023-06-28 | 2023-06-29 | 2023-06-30 | 2023-07-01 | 2023-07-02 | 2023-07-03 | 2023-07-04 | 2023-07-05 |
| | | | | 139 | 139 | 118 | 118 | 118 | 118 | 118 |
| | | | | | | 139 | 139 | 139 | 139 | 139 |
| | | | | | | | | | | |
| 719 | 750 | 744 | 753 | 749 | 728 | 712 | 731 | 736 | 721 | 754 |
| 2023-08-14 | 2023-08-15 | 2023-08-16 | 2023-08-17 | 2023-08-18 | 2023-08-19 | 2023-08-20 | 2023-08-21 | 2023-08-22 | 2023-08-23 | 2023-08-24 |
| 133 | 133 | 133 | 133 | 133 | 133 | 133 | 133 | | 242 | 242 |
| 141 | 141 | 141 | 141 | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| 2023-10-03 | 2023-10-04 | 2023-10-05 | 2023-10-06 | 2023-10-07 | 2023-10-08 | 2023-10-09 | 2023-10-10 | 2023-10-11 | 2023-10-12 | 2023-10-13 |
| 213 | 213 | 213 | 213 | 213 | 213 | 213 | 213 | 213 | 213 | 213 |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | 777 | 803 | 807 | 785 | 760 | 786 | 765 | 792 | 791 | 799 |

Figure 5.8: Example of a formulation's output

## 5.4 Conflict Solver

The solution formulated for our problem is based on solving conflicts. In it, each task will be scheduled to the furthest day possible and, whenever two tasks overlap each other, a **conflict** is said to happen, which will then be solved by choosing which task must be rescheduled. This move can create yet another conflict and, if it does, the process repeats itself (figure 5.9). This procedure is repeated until all tasks are scheduled and there are no more conflicts.

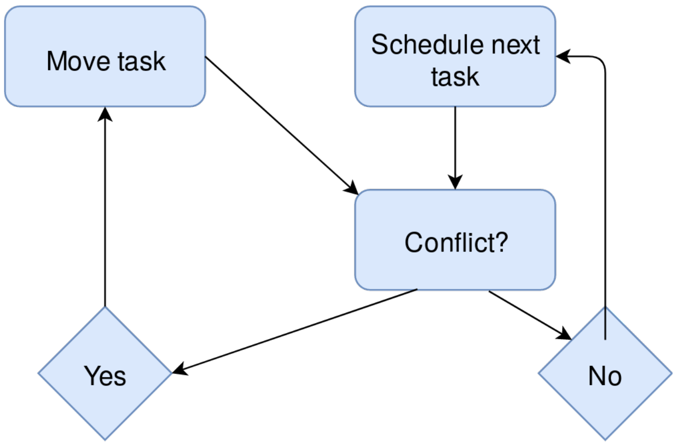


Figure 5.9: Conflict Solver behaviour loop

This solution was inspired by the already detailed work made by Zhang and Dietterich (1995) and on the fact that the process of solving conflicts is what maintenance operators apply in the real environment in order to create a feasible maintenance schedule (Deng, Santos, & Curran, 2019). The problem with the manual approach is that it is very time consuming, therefore making it very difficult to create a calendar that is optimal (Vanbuskirk et al., 2002).

Let's go through a quick example demonstrating the algorithm formulated.

Suppose that we have 3 tasks to be distributed between 6 days and only one hangar. Therefore we have $|T| = 3$, $|D| = 6$ and $|H| = 1$. Figure 5.10 details the the parameters for each task. Throughout this example, tasks scheduled without any conflict will be shown in green, conflicts in red and rescheduled tasks in yellow.

| Task ID | Length | Due Date |
|---|---|---|
| 1 | 2 | 6 |
| 2 | 2 | 4 |
| 3 | 2 | 3 |

Figure 5.10: Table with each task details

Let's start by scheduling t1 to its due date. By doing this, days 5, $d5$, and 6, $d6$, become occupied with $t1$. We check for conflicts but, because there is only one task scheduled, there are none. We then schedule $t2$ to as further as possible, which is its due date. We see that both tasks are scheduled without creating conflicts as figure 5.11 shows, so we can continue and schedule the next task.

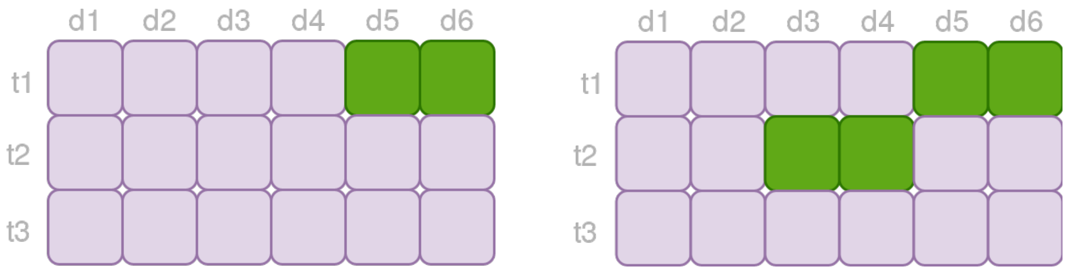


Figure 5.11: Scheduling of tasks $t1$ (left) and $t2$ (right)

By scheduling the third task, however, to its due date, $d3$ becomes occupied by both $t2$ and $t3$ (figure 5.12). Consequently, we say a **conflict** has been found between tasks $t2$ and $t3$.

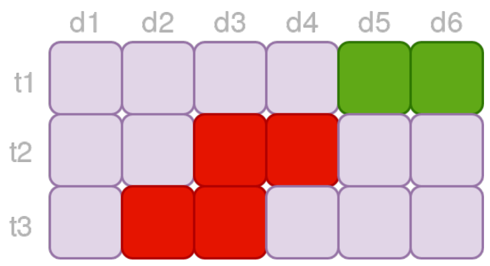


Figure 5.12: Conflict between $t2$ and $t3$

Now a decision must be made on which task is going to be rescheduled. This is a job for the RL agent and is going to be explained thoroughly in the next sections. For now, let's say $t3$ will be the task chosen to be moved. $t3$ is therefore rescheduled for the next day possible, resulting in the calendar shown in figure 5.13. We then check again for conflicts but this time there are none. Because all tasks are scheduled and no conflict is taking place, we can say we have a **feasible calendar**. The pseudo-code for the algorithm is presented in 5.1.

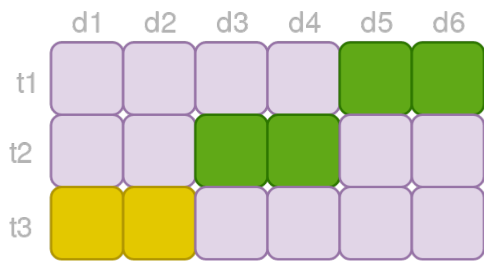


Figure 5.13: Final calendar

**Algorithm 5.1** Pseudo-code for Conflict Solver

```
$T$ = set of tasks
$C$ = queue with all the conflicts
while $t < T$ do
    while $t < T$ and not $C$ do
        schedule task $t$ to the furthest day possible
    end while
    while C do
        pop $C[-1]$
        $t_m$ = task chosen by $RL$ to be moved
        move $t_m$
    end while
end while
```

## 5.5 Simulated Environment

The simulated environment (SE) is as crucial for the algorithm as the RL agent. It is this implementation that allows for a bridge to be done between the Conflict Solver and the RL agent. It is this structure that will schedule each task until a conflict occurs, send the state to the RL agent, act on its decision and return the respective reward and new state obtained. As mentioned before, it is the reward function that will dictate the goal for the agent, thus conducting the agent's behavior.

Figure 5.14 displays this behavior, in which blue squares represent SE operations and the pink circle represents the RL agent.

Different operations to happen with the algorithm like rendering, task's statistics, etc. are usually also computed on the environment. In general, in the SE four main things must be defined:

- Scheduling functions
- Reward function
- State representation
- Action functions

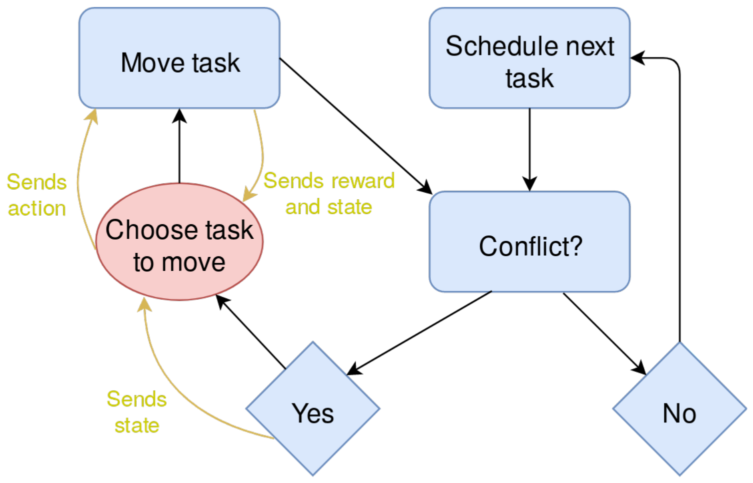


Figure 5.14: Relation between the SE and RL agent

As mentioned before, the definition of these functions is up to the programmer, based on its interpretation of the problem. Although the algorithm used (Conflict Solver) is the same for all the formulations, the functions defined on the SE are sometimes different for particular problems in order to better adapt to each one of them. The functions created and used for each formulation are detailed in the next sections.

### 5.5.1 Scheduling functions

The initial idea was, when solving a conflict, to reschedule one of the tasks to the next available day, meaning to a day that would create no conflicts. This idea had to be dropped because the calendars created would not be the ideal. We concluded that sometimes a conflict might be needed in order to create a better calendar.

Therefore when moving a task, it is rescheduled to the next day which avoids only the current conflict. Moving a task does not avoid conflicts with all the tasks, only with the tasks which initially created the ongoing conflict. For example, being $t1$ the task to be moved, with length $l_1$ and scheduled on $d_1$ and $t2$ the task that is not going to be moved, with length $l_2$ and scheduled on $d_2$:

- if a task is to be moved to a previous day:

$$\text{new day} = d_2 - l_2 \tag{5.1}$$

- if a task is to be moved to a later day:

$$\text{new day} = d_2 + l_1 \tag{5.2}$$

In this definition, there is a tricky part when there are multiple hangars because there might be a day closer to a certain due date on a different hangar, more suited for moving to. To solve this issue each task keeps a history of the last day it was scheduled on each hangar. In the moment of moving a task, the SE checks the task's history and decides on what is the best hangar for rescheduling.

This reward function is used for Formulation 1 and Formulation 2. For Formulation 3, which is based on slots, the task is moved to the next slot available which is of the same type as the respective task.

In the Aircraft Real Data Formulation the scheduling function is similar to the above but in that case, the function must take into account the fact that the conflict can happen between more than two tasks. In it, if the task is to be moved to an earlier day, the new day chosen is one before the earlier scheduled day of the tasks in conflict. The opposite happens if a task is to be moved to a later day. The pseudo-code for this process is described below (algorithm 5.2).

**Algorithm 5.2** Pseudo-code for the reward function in the Aircraft Real Data Formulation

```
$t$ = task to be moved
$d_t$ = Day of the calendar where $t$ is or will be scheduled
$d_{new}$ = New day for $t$
$C$ = Contains all the tasks which belong to the conflict
$d_{min}$ = Earliest scheduled day
$d_{max}$ = Tardiest scheduled day
if task is to be moved to an earlier day then
    $d_{min}$ = min day in $C$
    $d_{new} = d_{min} - 1$
else
    $d_{max}$ = max day in $C$
    $d_{new} = d_{max} + 1$
end if
$d_t = d_{new}$
Schedule $t$ on $d_t$
```

### 5.5.2 Reward function

After the RL agent is called to decide on what to do in a certain conflict and the environment acts upon that decision, the respective reward is calculated and returned to the agent. Because our goal is to schedule each task as close to the due date as possible, the reward function will express that. The function calculates the distance of each task (the one that was moved and the ones that were not) to their original due date and returns that distance as negative reinforcement. If a task is scheduled after its original due date, the negative reinforcement is even greater. Being $R$ the reward, $d$ the day a task is scheduled and $dd$ the due date of a task, the reward is calculated for each task as:

- if $d <= dd$

$$R = dd - d \tag{5.3}$$

- if $d > dd$

$$R = 5 * (d - dd) \tag{5.4}$$

"Calendar reward" is a term broadly used throughout this report. That is because, when a feasible calendar is obtained, the process represented above is repeated for all tasks $t \in T$, summing all the values, in order to calculate the calendar's full reward.

This definition is the same for all four formulations.

### 5.5.3 State representation

The state will be the input of the RL agent, it will be what he "sees", it is what it needs to know to make a decision. In our exercise, the agent needs to know what the calendar looks like at a certain moment, so that is what the input will represent. There are many possibilities regarding what could be the state representation for the RL agent, and that can be something to be better explored in the future but, for now, these were the solutions delineated.

Initially, a state representation was formulated to be applied in the first three formulations (5.2.1, 5.2.2, 5.2.3). However, after receiving the real data, a new state representation was composed, and ended up being used also for formulations 5.2.1 and 5.2.2. In all representations used, a slot being assigned "1" means that the task corresponding to that row is assigned to the day that column represents. A "0" means that the respective day is empty.

In our initial concept, each hangar was composed of a matrix with one row for each task and one column for each day. In figure 5.15 we can see that, for example, $t1$ is scheduled on the fifth and sixth day ($d5$ and $d6$).

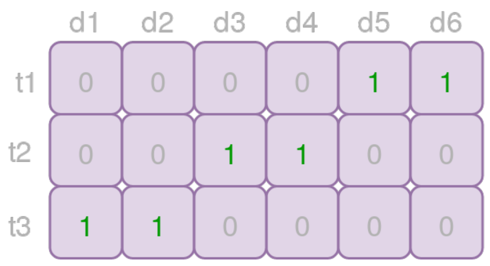


Figure 5.15: Representation of one hangar

The combination of the representations for all the hangars would compose the final state. In the example pictured in figure 5.16, we have $|H| = 2$, $|T| = 3$ and $|D| = 6$, which will result in the final shape of *(2, 3, 6)*.

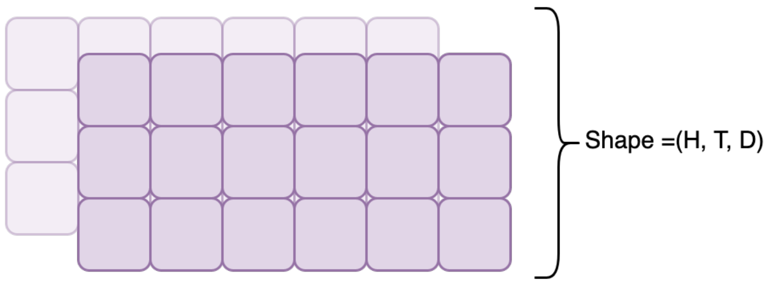


Figure 5.16: Final input of the RL agent

For the Aircraft Real Data Formulation, a different state representation was defined. As mentioned before, in it each day can take up to 3 C-checks and 1 A-check, therefore the algorithm will not be dealing with hangars directly. Using the real data, we are dealing with 2190 days, which would create too big of a state, slowing down significantly the DRL algorithm. To deal with this issue only a partial calendar is sent to the RL agent, enough for it to make a decision. In figure 5.17 there is a simple example of this management. In that example, each column represents one day, a slot colored green signifies a task scheduled on the respective day and a red slot symbolizes a task scheduled on a certain day and belonging to a conflict. In a situation like this, the algorithm will select a number of days to the left and right of the conflict (in our case we use 100 days total) and use it as the representation of the current calendar.

Also, a different approach is taken on how to represent the tasks scheduled in the calendar. We are going to use a matrix with 10 rows and 100 columns (shape = (10,100)). The first rows are used to represent the scheduled days for each of the tasks that belong to the conflict. If there are 4 tasks in the open conflict, the first 4 rows will be representative of each one of them. A single row after is then used to describe all the other tasks which belong to the partial calendar that are not in conflict. With this definition, some rows of the state might stay empty, but the 10 number of rows was chosen to encompass situations where there might be a large number of tasks in a conflict. For example, let's say we have 4 tasks in a conflict, in a problem with 9 tasks, that the number of days chosen for the partial calendar is 10,

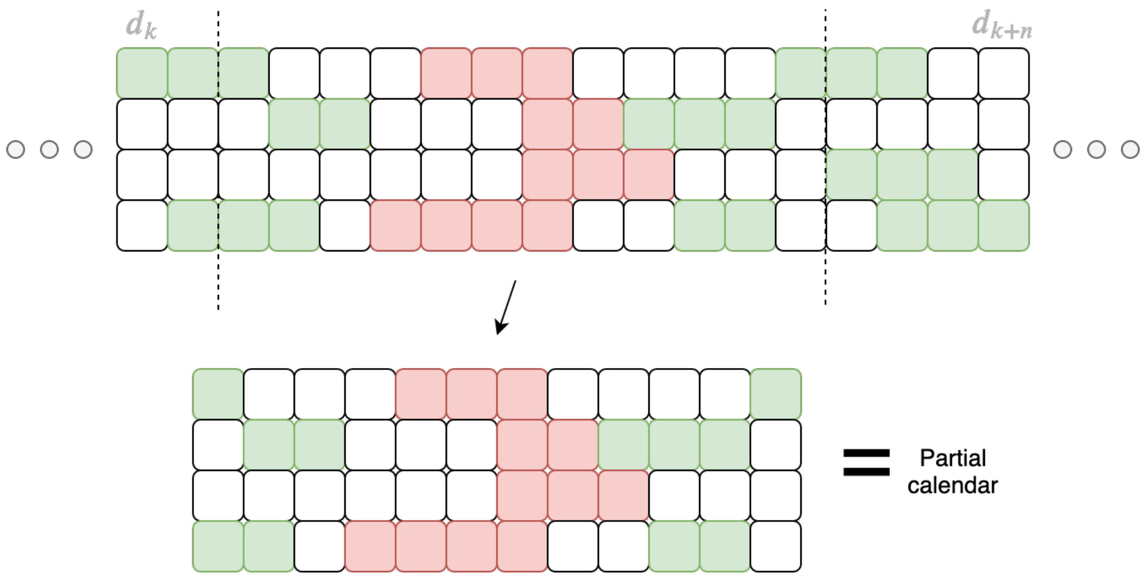


Figure 5.17: Example of the selection of a partial calendar

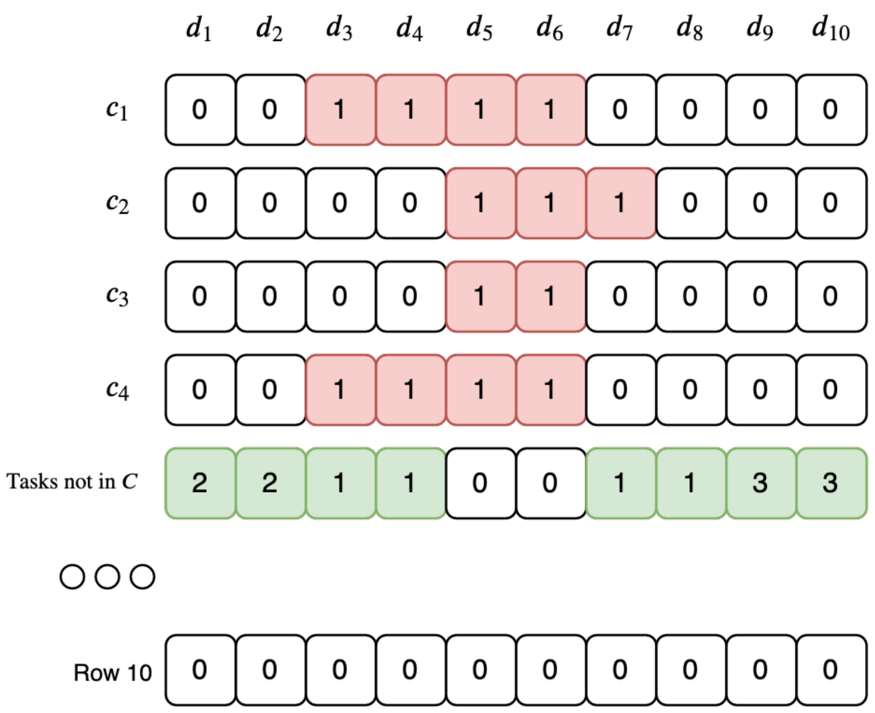
and that the current details for each task are detailed in table 5.1. In our approach, we represent a conflict by $C$ and each task on that conflict using $c_1, c_2, c_3$...

| ID | Scheduled day | Length | In conflict C |
|---|---|---|---|
| **1** | **d6** | **4** | **Yes** |
| **2** | **d7** | **3** | **Yes** |
| 3 | d4 | 4 | No |
| **4** | **d6** | **2** | **Yes** |
| 5 | d4 | 2 | No |
| **6** | **d6** | **4** | **Yes** |
| 7 | d10 | 2 | No |
| 8 | d10 | 2 | No |
| 9 | d10 | 4 | No |

Table 5.1: Tasks' details for the example

In this example, tasks $t1$, $t2$, $t4$ and $t6$ (in conflict) will be represented on the first four rows, respectively, and tasks $t3$, $t5$, $t7$, $t8$ and $t9$ are all represented on the fifth row, by counting the number of tasks in each day. Just as a reminder, a value of "1" means the task corresponding to that row is scheduled on that respective day and "0" signifies the day does not have a task scheduled in it. The difference here is in the row representing the tasks that are not in conflict (in this case, the fifth row). In it, each number represents the number of tasks scheduled on the day corresponding to that column. Because the first four rows, in this case, are reserved for the 4 tasks in the conflict, we will use the next one, the fifth row, to represent all the tasks in the partial calendar that do not belong to the conflict. As we do not need to know specifically which those tasks are, just where they are scheduled, that matrix's row will give information on the number of tasks that are in each day. Figure 5.18 shows what would be the state representation for our example given the info given in table

5.1. In it, we can see that only the first five rows were used. That is because there are 4 tasks in this conflict and, as just mentioned, only one row is used to represent all the tasks that are not in the conflict.



$d_1$ $d_2$ $d_3$ $d_4$ $d_5$ $d_6$ $d_7$ $d_8$ $d_9$ $d_{10}$
$c_1$ 0 0 1 1 1 1 0 0 0 0
$c_2$ 0 0 0 0 1 1 1 0 0 0
$c_3$ 0 0 0 0 1 1 0 0 0 0
$c_4$ 0 0 1 1 1 1 0 0 0 0
Tasks not in $C$ 2 2 1 1 0 0 1 1 3 3
Row 10 0 0 0 0 0 0 0 0 0 0

Figure 5.18: Task representation for the partial schedule

### Action functions

For Formulation 1, Formulation 2 and Aircraft Real Data Formulation, there are four possible actions for the RL agent to choose from, each one of them being one of the following dispatching rules:

- ESD: Earliest Scheduled Day
- TSD: Tardiest Scheduled Day
- SLF: Shortest Length First
- LLF: Longest Length First

"Scheduled Day" means the day in which a certain task is scheduled at the moment and "Length" corresponds to the length, $l$, of a task. In the algorithm, the RL agent will choose an action between 0 and 3 and the SE will translate that response to the respective dispatching rule, then moving the corresponding task.

In Formulation 3, however, all conflicts arise from tasks with the same length and same type. And, because the tasks have the same length, $l = 1$, two tasks conflicting means they also start on the same day. Therefore, the dispatching rules specified above are not suited to it. To get around this issue, instead of dispatching rules we take advantage of the fact that the conflicts are always between two tasks and we simply use following actions:

- Move first task
- Move second task

By using dispatching rules, the decision making can be more versatile. We could use always two actions, the first being to move the first task and the second action moving the second task. However, this interpretation would only be prepared to decide between two tasks. And because the number of actions cannot change throughout the running of the algorithm, by using dispatching rules the agent is prepared to choose between a changeable number of tasks. Also new and different dispatching rules can be added or replace others. In the future, if dealing with costs and/or resources, dispatching rules taking into account those variables can be possible actions for the RL agent.

## 5.6 Reinforcement Learning Model

The DRL implementation follows the work made in (Mnih et al., 2015), where the authors used DRL to play Atari 2600 games, achieving better results than previous methods on 43 of the games used. This is the same implementation done in the first semester of the internship in order to learn the algorithm, also tested on the Atari 2600 environment (annex A.2).

### Neural Network

For each one of the formulations earlier described, the DQN receives as input its respective state representation. Following the input layer, there are two fully-connected layers. The first one with 400 neurons and the second with 100, both using a Rectified Linear Unit (ReLU) as the activation function. The final DQN layer has a single output for each of the possible actions (figure 5.19).

### Algorithm

Just as detailed in annex A.2, the algorithm makes use of a **replay buffer** to store the environment's observations. During the training process and every four steps, a random batch is selected from the buffer and used for the network's update. Each time the RL agent is called to make a decision we call that a step.

The DQL algorithm starts by storing several observations in the replay buffer, by having the agent interaction with the environment making random decisions. After that, the agent begins to interact with the environment using the $\epsilon$-greedy strategy in order to choose which action to make, with $\epsilon$ decreasing linearly from 1.0 to 0.05 over the maximum number of steps chosen. Every time a new interaction is made

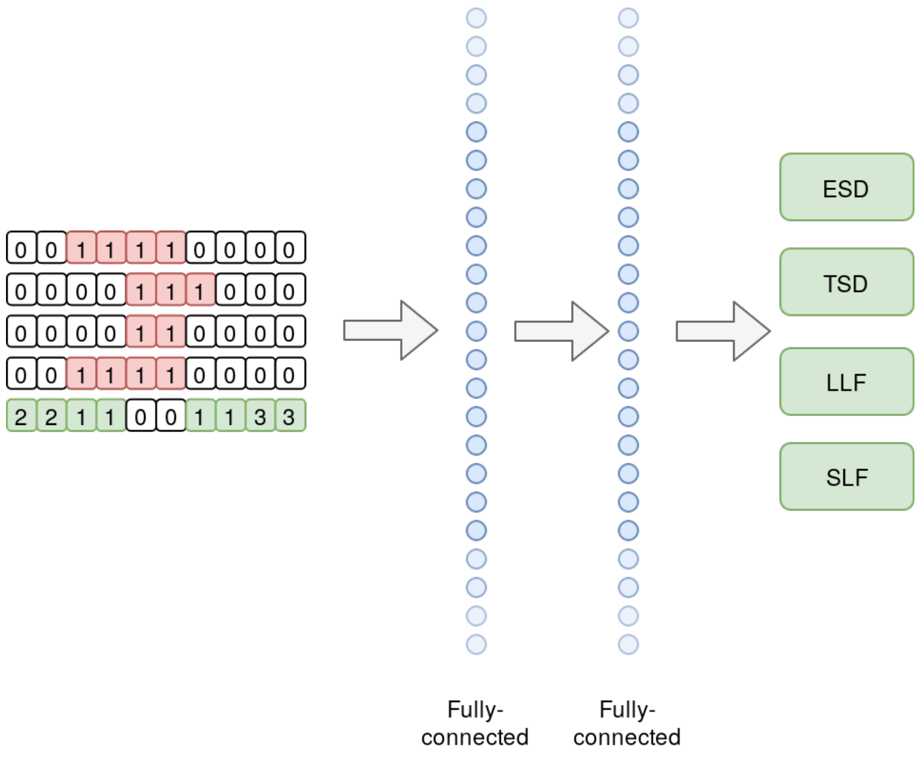


Figure 5.19: Deep Q-Network representation

with the environment (step), the new observation $(s_t, s_{t+1}, r_t)$ is stored in the replay buffer for future network updates.

Whenever the agent finds a feasible calendar, meaning a calendar without conflicts and with all the tasks scheduled, an episode is said to happen and a reset is done to the environment. Each time an episode is concluded, the final reward of the calendar is stored along with the number of conflicts that arose in that episode. The pseudo-code for the algorithm can be found in algorithm A.1 and the algorithm's hyper-parameters are detailed in table 5.2.

| **Hyperparameter** | **Value** | **Description** |
|---|---|---|
| Batch size | 32 | Number of samples taken from the replay buffer to train our agent |
| Replay buffer size | 200000 | Size of the queue that saves the agent's observations |
| Pre-population size | 20000 | Number of random observations stored before training |
| Discount factor | 0.99 | Discount factor used in Q-learning |
| Learning rate | 0.00025 | Learning rate used by the network's optimizer |
| Max frames | 100000-500000 | Maximum number of frames the agent saw during training |
| Initial exploration | 1 | Initial value of $\epsilon$ |
| Final exploration | 0.05 | Final value of $\epsilon$ |
| Target Update | 10000 | Frequency with which the target-network is updated |

Table 5.2: Hyper-parameters used for the DQN agent

## 5.7 Conclusion

Having access to real aircraft maintenance data was essential in order to better understand the problem and envision a possible solution. We can acknowledge that by noticing that the state representation initially created was, for the most part, dropped after a new one was conceived for the Aircraft Real Data Formulation. This is, mainly, because there was o lot of abstractness in building the artificial formulations. It was a loose process, with few rules to follow. Therefore the solution was left to our perception of the problem, at the time. After the data being received, the problem became much more clear and a solution was formulated with much more confidence and guidance.

The Conflict Solver, however, which was created at the very beginning of this work, is an adequate solution for both the toy-problems and the formulation which uses the real aircraft maintenance data. The reason for that, we believe, is in the fact that it is a very human-like manner of solving the problem. If we imagine ourselves creating a maintenance calendar with a lot of tasks and the possibility of not being able to schedule each task on its due date, this might be how we contemplate the dilemma. And being that RL, in several applications, by emulating a human-like behavior is able of achieving good results, we believe that this is was an adequate frame of mind in order to find a solution to the problem. After finding an answer to the question of "how would I solve this issue with a pen and paper?", Conflict Solver came from converting that answer into a computational version of it.

# Chapter 6

# Experiments and Results

In this chapter, the results for the different formulations will be shown. For each exercise, throughout the running of the RL agent, the higher calendar reward was reported as the result obtained. Those results are here compared with a **greedy** algorithm which, in each conflict, instead of calling the RL agent to decide on what task to move, simulates moving the different tasks belonging to that conflict and then selects the move that brings the bigger reward (algorithm 6.1). Each formulation was also compared with the results of an algorithm that decided what to do in each conflict randomly (random-action). The results from the random-action formulation were recorded after running the program 100 times and saving the mean of all the calendar rewards from all the runs. The comparisons between these three formulations will be detailed for each one of the exercises.

For each formulation two exercises will be displayed, a) and b), together with its results. In each of the exercises, it is going to be specified the respective number of tasks, $|T|$, number of days, $|D|$ and the range of possible task lengths, $l$.

**Algorithm 6.1** Pseudo-code for the greedy formulation

```
$C$ = conflict
$max_r$ = bigger reward
$t_{move}$ = task to move
for $c$ in $C$ do
    simulate moving $c$
    $r_c$ = reward of the simulation
    if $r_c > max_r$ then
        $max_r = r_c$
        $t_{move} = c$
    end if
end for
reschedule $t_{move}$
```

The algorithm was trained using *Google Colab*, a Jupyter notebook environment, free of charge, supported by Google.

# Formulation 1

In this formulation we have, for each exercise, a static set of tasks, $T$. The goal is to go through each task, assigning it to the corresponding time-frame within the number of days, $|D|$. For both exercises 2 hangars were used ($|H| = 2$). As soon as all tasks are scheduled and there are no conflicts a feasible calendar is said to be constructed.

(a) **200 tasks, 1000 days**

- $|T| = 200$
- $|D| = 1000$
- $1 \leq l \leq 5$

| | **Reward** | **Tasks before** | **Tasks after** | **Conflicts** |
|---|---|---|---|---|
| **RL** | -73 | 37 | 0 | 169 |
| **Greedy** | -72 | 33 | 0 | 145 |
| **Random** | -144 | 33 | 0 | 173 |

Table 6.1: Results with 200 tasks and 1000 days

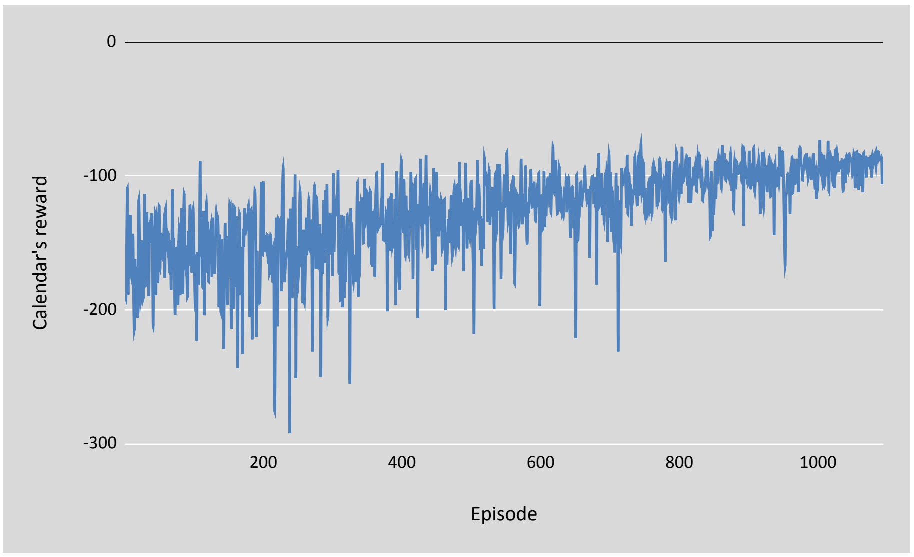


Figure 6.1: RL behavior for Formulation 1 with 200 tasks and 1000 days

(b) **50 tasks, 200 days**

- $|T| = 50$
- $|D| = 200$
- $2 \leq l \leq 8$

| | **Reward** | **Tasks before** | **Tasks after** | **Conflicts** |
|---|---|---|---|---|
| **Reinforcement Learning** | **-168** | 28 | 0 | 140 |
| **Greedy** | -260 | 24 | 2 | 163 |
| **Random** | -486 | 28 | 2 | 206 |

Table 6.2: Results with 50 tasks and 200 days

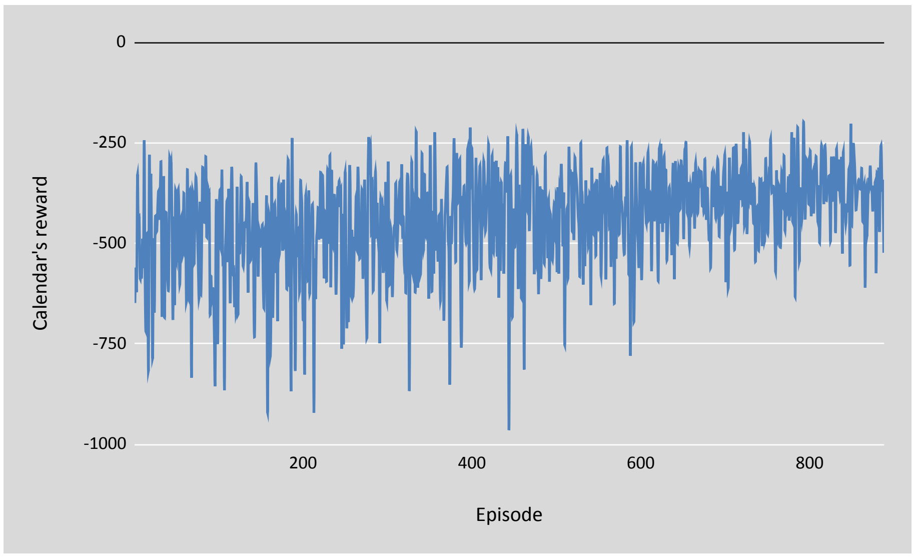


Figure 6.2: RL behavior for Formulation 1 with 50 tasks and 200 days

## Formulation 2

As mentioned before, in this formulation every time a feasible calendar is obtained, the time advances one day. Just as before, we also have a static number of tasks but, whenever a task is completed (the day for when it was scheduled goes by), it is rescheduled with a random variation (between -3 and 3 days) from its due date. In this formulation's results, we chose not to use the number of tasks before and after the due date given the randomness in the exercise's due dates throughout the running of the algorithm. Just as in the previous formulations, we use two hangars for both exercises.

(a) **20 tasks, 50 days**

- $|T| = 20$
- $|D| = 50$
- $1 \leq l \leq 3$

| | **Mean reward** | **Mean no. conflicts** |
|---|---|---|
| **Reinforcement Learning** | -89 | 405 |
| **Greedy** | **-66** | 418 |
| **Random** | -276 | 490 |

Table 6.3: Results with 20 tasks and 50 days

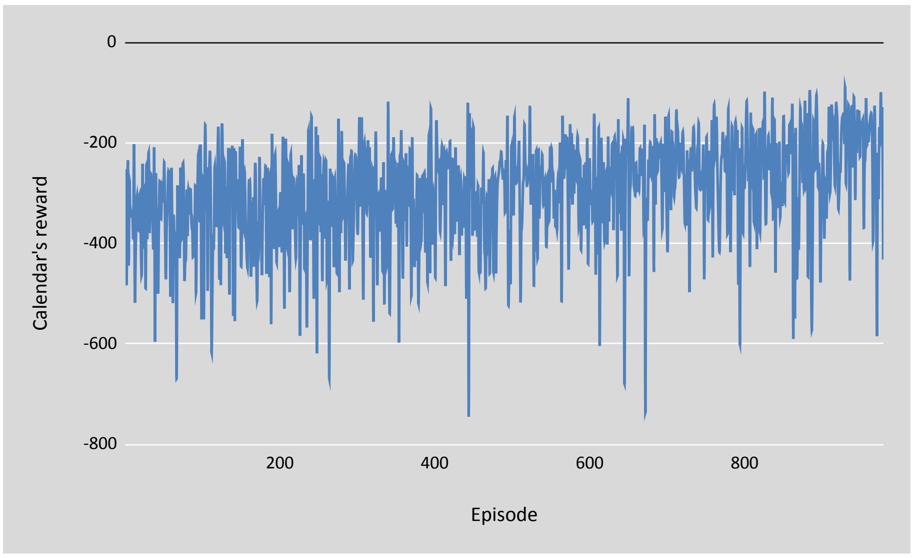


Figure 6.3: RL behavior for Formulation 2 with 20 tasks and 50 days

## (b) 50 tasks, 250 days

- $|T| = 50$
- $|D| = 250$
- $1 \leq l \leq 8$

| | **Mean reward** | **Mean no. conflicts** |
|---|---|---|
| **Reinforcement Learning** | -503 | 545 |
| **Greedy** | **-208** | 781 |
| **Random** | -1182 | 759 |

Table 6.4: Results with 50 tasks and 250 days

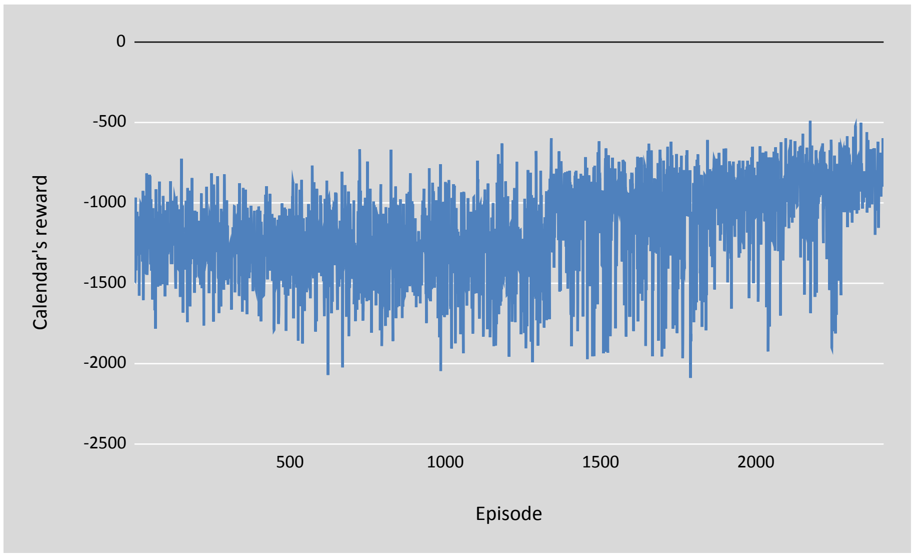


Figure 6.4: RL behavior for Formulation 2 with 50 tasks and 250 days

## Formulation 3

This formulation is slot-based, meaning that all tasks have length, $l$, equal to 1. In both exercises, there are three different types of tasks and, likewise, three different types of slots. Accordingly, each task is only scheduled to a slot specific to it and only 1 hangar is used in both exercises.

(a) **50 tasks, 55 days**

- $|T| = 50$
- $|D| = 55$

| | **Reward** | **Tasks before** | **Tasks after** | **Conflicts** |
|---|---|---|---|---|
| **RL** | **-367** | 45 | 5 | 137 |
| **Greedy** | -457 | 37 | 13 | 173 |
| **Random** | -1064 | 36 | 8 | 156 |

Table 6.5: Results with 50 tasks and 55 days

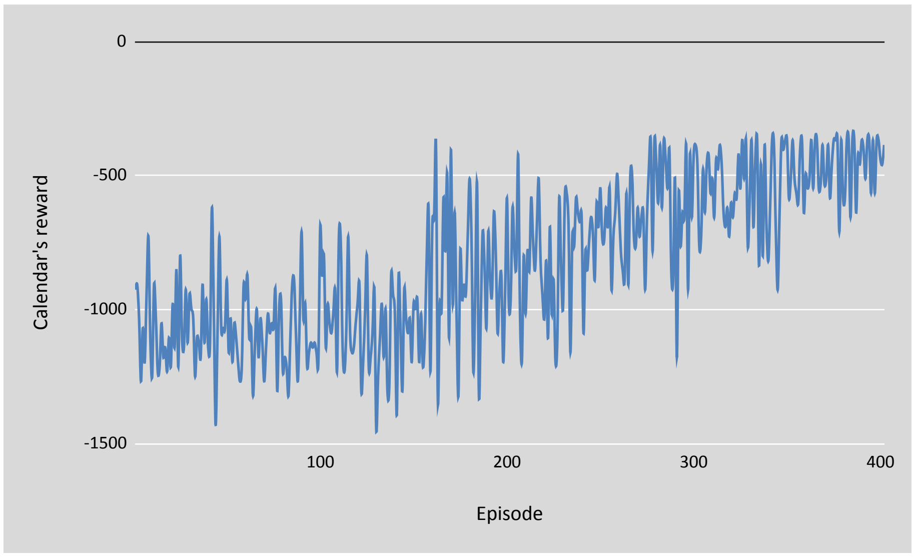


Figure 6.5: RL behavior for Formulation 3 with 50 tasks and 55 days

(b) **100 tasks, 150 days**

- $|T| = 100$
- $|D| = 150$

| | Reward | Tasks before | Tasks after | Conflicts |
|---|---|---|---|---|
| **RL** | -420 | 97 | 3 | 98 |
| **Greedy** | -420 | 37 | 8 | 91 |
| **Random** | -635 | 95 | 5 | 66 |

Table 6.6: Results with 100 tasks and 150 days

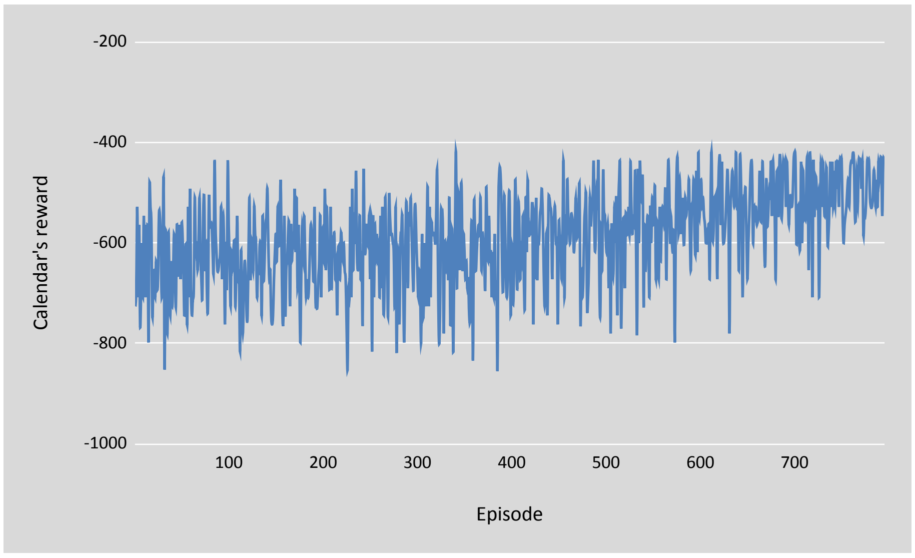


Figure 6.6: RL behavior for Formulation 3 with 100 tasks and 150 days

## Aircraft Real Data Formulation

As mentioned before, the data-set used for this formulation contains 2 maintenance checks for each of the 51 aircraft described in it, to be scheduled throughout 6 years which, for our exercise, results in 1035 tasks and 2190 days. All A-checks have lengths, $l$, equal to 1 DY and C-checks' lengths vary between 15 and 35 days, depending on the information in the data set.

| | **Reward** | **C-checks before** | **C-checks after** | **A-checks before** | **A-checks after** | **Conflicts** |
|---|---|---|---|---|---|---|
| **RL** | **-359** | 16 | 2 | 47 | 0 | 127 |
| **Greedy** | -832 | 25 | 2 | 52 | 0 | 208 |
| **Random** | -1301 | 32 | 2 | 49 | 0 | 264 |

Table 6.7: Results with real aircraft maintenance data

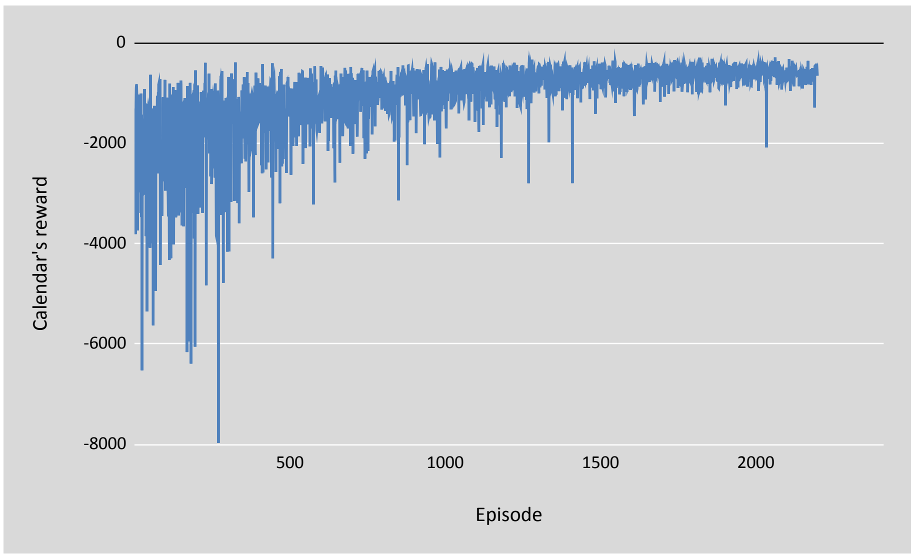


Figure 6.7: RL behavior for Aircraft Real Data Formulation

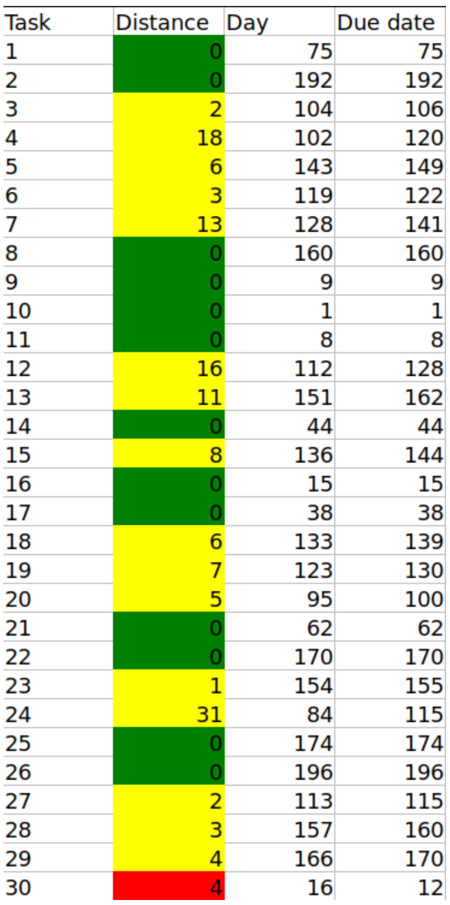
With Formulation 1, when using 200 tasks, the results between RL and greedy were virtually the same, although the learning curve for the agent is very steep. On exercise b), RL had significantly better results, being able to schedule all tasks before or on their due dates. On the contrary, the greedy algorithm had 2 tasks scheduled after their due dates and, as established by the reward functions, this results in a bigger negative reinforcement, which had a significant impact on the disparity of rewards.

Formulation 2 was where worse results were obtained as the greedy algorithm ended up having significantly better results in both exercises. We believe that this occurred because, in this formulation, the RL agent has no control about the random changes that happen on a task's due date whenever it is completed and rescheduled. There is also a big number of conflicts in both exercises which can hamper the learning for the agent. Still, there is a visible tendency for the agent to learn through time and, for that reason, it may be that with more episodes the results could be better.

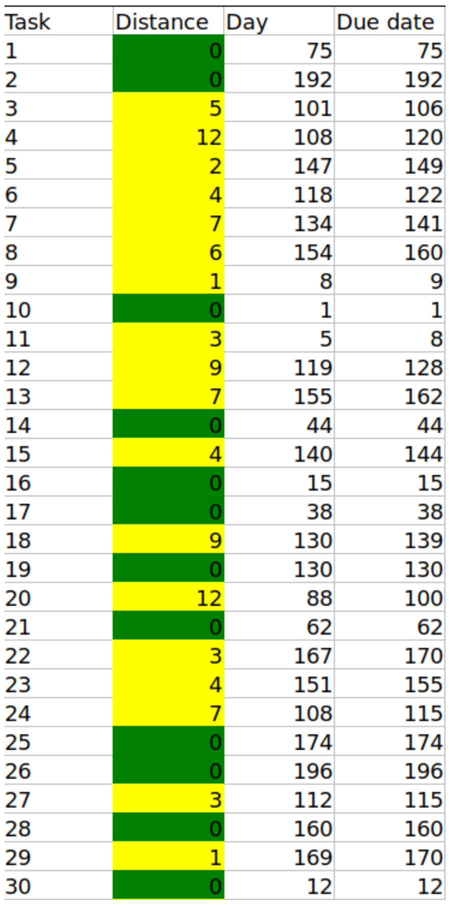
Formulation 3 presented valuable results. On exercise a) the results were considerably better when using RL and, in addition to that, the agent's progress is very noticeable. On the second exercise, both the RL and greedy had a similar calendar reward but, interestingly enough, they had a very distinct number of tasks before and after their due dates. That, alongside with the fact that, throughout the training, the maximum value was scored several times (figure 6.6) may be an indication for -420 being the maximum possible reward for that specific exercise.

The most satisfying results were achieved in the Aircraft Real Data Formulation. In it, RL had a highly better result than the greedy formulation. Although the difference in the number of tasks scheduled prior and after their due dates is not great, RL is able to schedule them much closer to their due dates. This result makes sense when we think of the fact that the RL agent has full control of the exercise. It is not as in Formulation 2, where new due dates are calculated by the simulated environment (SE) with random modifications. In this exercise, when a decision is made and a maintenance task (of a check) is moved, the due date for the next task of that check is also changed based on that movement. This means that, in order to solve this exercise, only thinking about what is better for solving an ongoing conflict is not enough. There must be an awareness of the impact of each decision on future tasks. That is something that the greedy formulation can not predict and, contrariwise, reinforcement learning is known to excel at.

Figure 6.8 shows how different calendars with distinctive rewards are. The figure shows a comparison of the better result from both the greedy (picture on the left side) and RL algorithm (picture on the right side) related to exercise b) of Formulation 1, for the first 30 tasks. Both pictures represent the same information. The column "Task" refers to the task's number. "Distance" signifies the difference, in days, between each task's due date and the day it was scheduled. "Day" is the calendar day in which the task was scheduled. "Due date" is the task's due date. We can see that the greedy formulation, besides scheduling tasks further away from their due dates, has one of the tasks, $t_{30}$, scheduled after its due date. The RL algorithm, on the other hand, is able to schedule all tasks prior to and closer to their due dates.



| Task | Distance | Day | Due date |
|---|---|---|---|
| 1 | 0 | 75 | 75 |
| 2 | 0 | 192 | 192 |
| 3 | 2 | 104 | 106 |
| 4 | 18 | 102 | 120 |
| 5 | 6 | 143 | 149 |
| 6 | 3 | 119 | 122 |
| 7 | 13 | 128 | 141 |
| 8 | 0 | 160 | 160 |
| 9 | 0 | 9 | 9 |
| 10 | 0 | 1 | 1 |
| 11 | 0 | 8 | 8 |
| 12 | 16 | 112 | 128 |
| 13 | 11 | 151 | 162 |
| 14 | 0 | 44 | 44 |
| 15 | 8 | 136 | 144 |
| 16 | 0 | 15 | 15 |
| 17 | 0 | 38 | 38 |
| 18 | 6 | 133 | 139 |
| 19 | 7 | 123 | 130 |
| 20 | 5 | 95 | 100 |
| 21 | 0 | 62 | 62 |
| 22 | 0 | 170 | 170 |
| 23 | 1 | 154 | 155 |
| 24 | 31 | 84 | 115 |
| 25 | 0 | 174 | 174 |
| 26 | 0 | 196 | 196 |
| 27 | 2 | 113 | 115 |
| 28 | 3 | 157 | 160 |
| 29 | 4 | 166 | 170 |
| 30 | 4 | 16 | 12 |



| Task | Distance | Day | Due date |
|---|---|---|---|
| 1 | 0 | 75 | 75 |
| 2 | 0 | 192 | 192 |
| 3 | 5 | 101 | 106 |
| 4 | 12 | 108 | 120 |
| 5 | 2 | 147 | 149 |
| 6 | 4 | 118 | 122 |
| 7 | 7 | 134 | 141 |
| 8 | 6 | 154 | 160 |
| 9 | 1 | 8 | 9 |
| 10 | 0 | 1 | 1 |
| 11 | 3 | 5 | 8 |
| 12 | 9 | 119 | 128 |
| 13 | 7 | 155 | 162 |
| 14 | 0 | 44 | 44 |
| 15 | 4 | 140 | 144 |
| 16 | 0 | 15 | 15 |
| 17 | 0 | 38 | 38 |
| 18 | 9 | 130 | 139 |
| 19 | 0 | 130 | 130 |
| 20 | 12 | 88 | 100 |
| 21 | 0 | 62 | 62 |
| 22 | 3 | 167 | 170 |
| 23 | 4 | 151 | 155 |
| 24 | 7 | 108 | 115 |
| 25 | 0 | 174 | 174 |
| 26 | 0 | 196 | 196 |
| 27 | 3 | 112 | 115 |
| 28 | 0 | 160 | 160 |
| 29 | 1 | 169 | 170 |
| 30 | 0 | 12 | 12 |

Figure 6.8: Calendars' details for the greedy formulation (left) and RL (right) for exercise b) of Formulation 1 (first 30 tasks)

This page intentionally left blank.

# Chapter 7

# Conclusion

In this work, several challenges were created and then solved with deep reinforcement learning (DRL) in order to prove that the algorithm is capable of achieving good results on constructing a maintenance schedule for an aircraft fleet. In the three artificial problems created, RL already shown decent results which can be used as a concept proof for the problem. Fortunately, throughout the internship, it was still possible to use real world aircraft maintenance data and, making use of this data, DRL proved that, by maximizing future rewards, the agent is capable of making the right decisions in order to create solutions significantly better than the greedy algorithm. Given that, in the Aircraft Real Data Formulation, the due dates are calculated using the previous maintenance day plus the respective interval, past decisions have a big impact on the final calendar's quality, being that each decision directly affects the next's. This ability for a RL agent to choose actions that will have a more positive impact in the long run is what ReMAP believes it will have a chain of effects while composing a maintenance schedule, resulting in a positive solution.

## 7.1 Challenges

The uncertainty on our task's scope was one of the main challenges of this internship. In the first semester, a whole different approach was assembled to be used in a different problem and throughout the internship. While in the second semester, most of the work done in that sense had to be changed. Nevertheless, we still decided to include that initial work on the report as it was a significant part of this internship (annex A.1).

The second semester brought some transparency to the problem and what was our goal in the project. Still, there were some doubts on how the problem would be approached and how the method would be integrated with a possible solution. Also, the lack of real data and the inability to find relevant data online was a problem we had to mitigate by making use of artificial data in toy-problems. After receiving the real data, the problem became much more clear and a new solution was conceived that fitted the problem much better.

As a more practical challenge, we had the use of Google Colab. There were some issues with disconnects throughout the making of the tests. This was partially due to the service but also to blame is the environment and the network used when making the tests. Therefore the training for each of the exercises was not very long, usually between one and two hours. Still, it was enough to show that the RL agent was able to 'learn' in this particular problem, and gradually make better decisions and calendars in each one of the exercises used.

## 7.2 Future Work

Plans for the future involve including unexpected maintenance jobs in the problem. Additionally, a future solution will have to take into account resources available and the materials needed for each maintenance in a more meticulously manner. As a personal assessment, the intern believes that finding the optimal solution for this problem passes by visiting the airline and talking to the personnel involved in charge of developing the maintenance schedule. It is important to understand what is the line of thinking of the people who have been dealing with this problem for years and to discern the details of their approach to it. If we look into RL implementations used in games, for example, there are not new decisions created. The RL agent is not identifying another procedure to play the game. It simply learns to make better decisions and it is capable of doing them faster, while still playing the game the same way a human does.

Previously, in the report, it was mentioned that a lot of times the manual approach does not allow enough time to create an optimal calendar, thus the maintenance operator settling for a feasible one. Given this information, the speed can be said to be one of the main drawbacks to the manual method. Computers are certainly capable of mitigating that speed deficit and RL has the ability to learn what are the best decisions to make in different scenarios. Also, RL is able to choose an action that will allow for a better reward in the future, not only worrying about the immediate decision. Thus, the method can mitigate the speed problem plus take into account more variables into the decision process.

# References

Abbeel, P., Coates, A., Quigley, M., & Ng, A. Y. (2007). An application of reinforcement learning to aerobatic helicopter flight. In *Advances in neural information processing systems* (pp. 1–8).

Ahmad, R., & Kamaruddin, S. (2012). An overview of time-based and condition-based maintenance in industrial application. *Computers & Industrial Engineering*, *63*(1), 135–149.

Alfares, H. K. (1999). Aircraft maintenance workforce schedulinga case study. *Journal of Quality in Maintenance Engineering*, *5*(2), 78–89.

*Annex c.* (2019). (confidential information: unpublished)

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017, nov). *Deep reinforcement learning: A brief survey* (Vol. 34) (No. 6). Institute of Electrical and Electronics Engineers Inc. doi: 10.1109/MSP.2017.2743240

Bakker, B., Zhumatiy, V., Gruener, G., & Schmidhuber, J. (2003). A robot that reinforcement-learns to identify and memorize important previous observations. In *Intelligent robots and systems, 2003.(iros 2003). proceedings. 2003 ieee/rsj international conference on* (Vol. 1, pp. 430–435).

Beraldi, P., De Simone, F., & Violi, A. (2010). Generating scenario trees: A parallel integrated simulation–optimization approach. *Journal of Computational and Applied Mathematics*, *233*(9), 2322–2331.

Bloch, H. P., & Geitner, F. K. (1983). *Practical machinery management for process plants. volume 2: Machinery failure analysis and troubleshooting* (Tech. Rep.). Exxon Chemical Co., Baytown, TX.

Byington, C. S., Roemer, M. J., & Galie, T. (2002). Prognostic enhancements to diagnostic systems for improved condition-based maintenance [military aircraft]. In *Proceedings, ieee aerospace conference* (Vol. 6, pp. 6–6).

Castanier, B., Grall, A., & Bérenguer, C. (2005). A condition-based maintenance policy with non-periodic inspections for a two-unit series system. *Reliability Engineering & System Safety*, *87*(1), 109–120.

Defourny, B., Ernst, D., & Wehenkel, L. (2012). Multistage stochastic programming: A scenario tree based approach to planning under uncertainty. In *Decision theory models for applications in artificial intelligence: concepts and solutions* (pp. 97–143). IGI Global.

Deng, Q., Santos, B. F., & Curran, R. (2019). A practical dynamic programming based methodology for aircraft maintenance check scheduling optimization. *European Journal of Operational Research*.

El Moudani, W., & Mora-Camino, F. (2000). A dynamic approach for aircraft assignment and maintenance scheduling by airlines. *Journal of Air Transport Management*, *6*(4), 233–237.

Gottesman, O., Johansson, F., Komorowski, M., Faisal, A., Sontag, D., Doshi-Velez, F., & Celi, L. A. (2019). Guidelines for reinforcement learning in healthcare. *Nat Med*, *25*(1), 16–18.

Grall, A., Bérenguer, C., & Dieulle, L. (2002). A condition-based maintenance policy for stochastically deteriorating systems. *Reliability Engineering & System Safety*, *76*(2), 167–180.

*H2020 remap.* (n.d.).

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the ieee international conference on computer vision* (pp. 1026–1034).

Holmberg, K., Komonen, K., Oedewald, P., Peltonen, M., Reiman, T., Rouhiainen, V., . . . Heino, P. (2004). Safety and reliability technology review. *Res Rep BTUO43-031209. VTT Industrial Systems, Espoo.*

*IATA's Airline Maintenance Cost Executive Summary.* (2017). Retrieved 2019-08-17, from `https://www.iata.org/whatwedo/workgroups/Documents/MCTF/MCTF-FY2016-Report-Public.pdf`

Jardine, A. K., Lin, D., & Banjevic, D. (2006a). A review on machinery diagnostics and prognostics implementing condition-based maintenance. *Mechanical systems and signal processing*, *20*(7), 1483–1510.

Jardine, A. K., Lin, D., & Banjevic, D. (2006b). A review on machinery diagnostics and prognostics implementing condition-based maintenance. *Mechanical systems and signal processing*, *20*(7), 1483–1510.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kleeman, M. P., & Lamont, G. B. (2005). Solving the aircraft engine maintenance scheduling problem using a multi-objective evolutionary algorithm. In *International conference on evolutionary multi-criterion optimization* (pp. 782–796).

Knowles, M., Baglee, D., & Wermter, S. (2011). Reinforcement learning for scheduling of maintenance. In *Research and development in intelligent systems xxvii* (pp. 409–422). Springer.

Kober, J., Muelling, K., & Peters, J. (2012). Learning throwing and catching skills. In *2012 ieee/rsj international conference on intelligent robots and systems* (pp. 5167–5168).

Labib, A. W. (2004). A decision analysis model for maintenance policy selection using a cmms. *Journal of Quality in Maintenance Engineering*, *10*(3), 191–202.

Li, J., Monroe, W., Ritter, A., Galley, M., Gao, J., & Jurafsky, D. (2016). Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541*.

Li, Y., & Nilkitsaranont, P. (2009). Gas turbine performance prognostic for condition-based maintenance. *Applied energy*, *86*(10), 2152–2161.

Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. In *Proceedings of the 15th acm workshop on hot topics in networks* (pp. 50–56).

Mattila, V., & Virtanen, K. (2011). Scheduling fighter aircraft maintenance with reinforcement learning. In *Simulation conference (wsc), proceedings of the 2011 winter* (pp. 2535–2546).

Meng, Q., & Wang, T. (2011). A scenario-based dynamic programming model for multi-period liner ship fleet planning. *Transportation Research Part E:*

*Logistics and Transportation Review*, *47*(4), 401–413.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928–1937).

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... others (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529.

Papakostas, N., Papachatzakis, P., Xanthakis, V., Mourtzis, D., & Chryssolouris, G. (2010). An approach to operational aircraft maintenance planning. *Decision Support Systems*, *48*(4), 604–612.

Peng, Y., Dong, M., & Zuo, M. J. (2010). Current status of machine prognostics in condition-based maintenance: a review. *The International Journal of Advanced Manufacturing Technology*, *50*(1-4), 297–313.

PeriyarSelvam, U., Tamilselvan, T., Thilakan, S., & Shanmugaraja, M. (2013). Analysis on costs for aircraft maintenance. *Analysis on Costs for Aircraft Maintenance*, *3*(3), 177–182.

Powell, W. B. (2009). What you should know about approximate dynamic programming. *Naval Research Logistics (NRL)*, *56*(3), 239–249.

*Remap's proposal.* (2017). (internal document: unpublished)

Rubino, G., & Tuffin, B. (2009). *Rare event simulation using monte carlo methods.* John Wiley & Sons.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Safaei, N., Banjevic, D., & Jardine, A. K. (2011). Workforce-constrained maintenance scheduling for military aircraft fleet: a case study. *Annals of Operations Research*, *186*(1), 295–316.

Samaranayake, P. (2006). Current practices and problem areas in aircraft maintenance planning and scheduling–interfaced/integrated system perspective. In *Industrial engineering and management systems, proceedings of the 7th asia-pacific conference in bangkok, uws, sidney* (pp. 2245–2256).

Samaranayake, P., & Kiridena, S. (2012). Aircraft maintenance planning and scheduling: an integrated framework. *Journal of Quality in Maintenance Engineering*, *18*(4), 432–453.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... others (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... others (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354.

*Strategic Research and Innovation Agenda - Volume 2 — Acare.* (n.d.). Retrieved 2019-07-31, from `https :// www .acare4europe .org / acare -db ?keys = cbm{&}field{_}cluster{_}enablers{_}text{_}value = {&}field{_}enabler{_}value={&}field{_}capability{_}value=`

Sutton, R. S., & Barto, A. G. (1998). *Introduction to reinforcement learning* (Vol. 135). MIT press Cambridge.

Sutton, R. S., Barto, A. G., & Williams, R. J. (1992). Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems Magazine*, *12*(2), 19–22.

Tesauro, G., Jong, N. K., Das, R., & Bennani, M. N. (2006). A hybrid reinforcement learning approach to autonomic resource allocation. In *2006 ieee international conference on autonomic computing* (pp. 65–73).

*This Factory Robot Learns a New Job Overnight - MIT Technology Review.* (n.d.). Retrieved 2019-08-22, from https://www.technologyreview.com/s/601045/this-factory-robot-learns-a-new-job-overnight/

Tian, Z., & Liao, H. (2011). Condition based maintenance optimization for multi-component systems using proportional hazards model. *Reliability Engineering & System Safety*, *96*(5), 581–589.

Vanbuskirk, C., Dawant, B., Karsai, G., Sprinkle, J., Szokoli, G., Currer, R., et al. (2002). Computer-aided aircraft maintenance scheduling.

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Aaai* (Vol. 2, p. 5).

Wang, W., & Zhang, W. (2005). A model to predict the residual life of aircraft engines based upon oil analysis data. *Naval Research Logistics (NRL)*, *52*(3), 276–284.

Wang, Y.-C., & Usher, J. M. (2005). Application of reinforcement learning for agent-based production scheduling. *Engineering Applications of Artificial Intelligence*, *18*(1), 73–82.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*.

Watkins, C. J. C. H. (1989). Learning from delayed rewards.

*What is Your Return on Condition-Based Maintenance?* (n.d.). Retrieved 2019-08-27, from https://www.nrgsystems.com/blog/what-is-your-return-on-condition-based-maintenance/

Yam, R., Tse, P., Li, L., & Tu, P. (2001). Intelligent predictive decision support system for condition-based maintenance. *The International Journal of Advanced Manufacturing Technology*, *17*(5), 383–391.

Zhang, W., & Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *Ijcai* (Vol. 95, pp. 1114–1120).

Zhou, Z., Li, X., & Zare, R. N. (2017). Optimizing chemical reactions with deep reinforcement learning. *ACS central science*, *3*(12), 1337–1344.

Zhu, H., Gao, J., Li, D., & Tang, D. (2012). A web-based product service system for aerospace maintenance, repair and overhaul services. *Computers in Industry*, *63*(4), 338–348.

# Appendix A

# Preliminary Study

## A.1 Problem Description

Our task is embedded on the ReMAP's MDST. This tool must schedule maintenance for a fleet of aircraft, distributing each job between the resources available while obeying a set of constraints and following an objective function. Before scheduling, however, a decision must be made on whether the maintenance is to be carried out or not. This decision must be made in regards not only to the current condition of the plane but also to the state of an entire fleet, offering a global solution. In order to achieve that, a number of variables must be taken into account. For example, if we're trying to minimize the average of aircrafts grounded, the algorithm could use the information about the number of aicrafts already grounded together with their health prognostics to delay some maintenance jobs or anticipate others trying to achieve that goal. Or let's imagine we are scheduling maintenance for just one plane and our algorithm works towards maximizing plane availability. One possible approach is to have information about all the aircraft flights so to not overlap them with the maintenance job (if it's possible). In order to make a decision regarding maintenance, the MDST will receive a set of inputs from prognostics and diagnostics among others related to the aicraft utilization and resources available. One of the ideas is to allow users to provide inputs using an interface, including current occupied maintenance slots, number of slots available, etc. But for now we will not worry about that.

After the inputs being received we shall have gathered information about the urgency of maintenance, all the resources available for the job, the flights scheduled for the fleet in question and the costs associated with each operation. A threshold must be carefully defined, using all this information, regarding the need for maintenance, as an optimistic threshold would eventually lead to failures while, on the other hand, a threshold leading to maintenance before needed would result in over-maintenance and the whole point of the CBM approach would be missed (Knowles et al., 2011).

With all this information a decision must be made regarding maintaining the airplane or delaying the maintenance job.

The output of the solution must provide an easy to read output for the end-user, returning an *CSV* file plus a graphical layout with the results obtained.

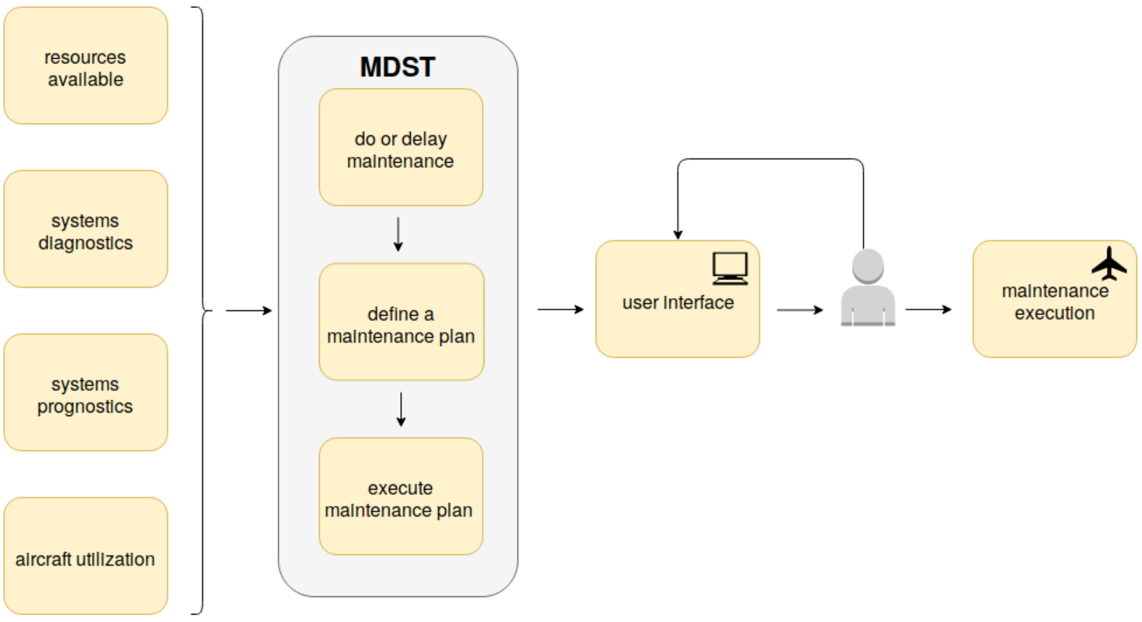


Figure A.1: High-Level Architecture

### A.1.1 Main Goals

The overall goal of the decision support tool is to establish adaptive fleet maintenance plans according to the resources available and each aircraft's condition and maintenance costs while assuring safety. Our task is to develop a deep reinforcement learning (DRL) algorithm able to return a decision on whether to perform maintenance or not giving the vast set of possible health scenarios.

### A.1.2 Problem Formulation

The data to be used and how we are going to use it is still being discussed by the MDST team. For now we are going to focus on the initial approach proposed to this problem, which will be described in the following sections. Giving this initial proposition we defined a first formulation as the the starting point to our work.

In our first approach, we are going to make a decision regarding maintenance according to the data of just one plane. This will be the first step, being that in the future the goal is to receive information on the whole fleet to decide whether to perform maintenance on a particular aircraft or not. We remember from the previous chapter that Reinforcement Learning has three main attributes:

- State: Computational representation of the environment in it's current condition.

- Action: What the agent can do. In this case it will decide whether to perform maintenance or not.
- Rewards: Positive or negative reinforce according to the agent's actions.

In the next section we are going to define how this three attributes are going to look like in response to the formulation to be approached.

### A.1.3 Reinforcement Learning Model

In figure A.2 a computational representation to a whole fleet is depicted. In it each row is representative of a plane and each plane is represented as an array containing all the inputs related to it, listed in table A.1. As said before, for now we are going to focus in receiving only the information related to one aircraft and make a decision according to it, so our state is going to be the same as one row of the figure.

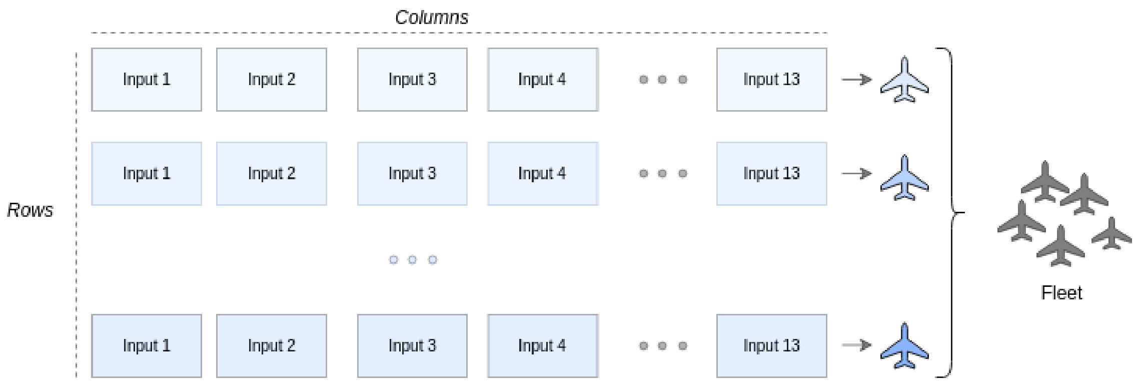


Figure A.2: Computational representation of a Fleet

Giving the state, the agent will, for now, choose between two possible actions:

- Perform maintenance
- Delay maintenance 1 weak

After choosing an action, a reward will be enforced. The rewards are not easy to assign and only testing will dictate if the rewards chosen suit the problem or not.

That being said, the rewards here defined are going to be simple yet carefully constructed. When deciding on whether to execute or not an maintenance job there are four possible outcomes.

1. Maintenance performed while being needed (True positive)
2. Maintenance performed without being needed (False positive)

| ***Data Group*** | ***Data*** | ***Input Number*** |
|---|---|---|
| **Resources Available** | Hangar Capacity | 1 |
| | Maintenance stands capacity | 2 |
| | Human resources | 3 |
| | Possibly inventory available, etc. | 4 |
| **Systems and Strucutre Diagnostics** | Indication of sub-systems failures | 5 |
| | Structural damage | 6 |
| **Systems and Structure Prognostics** | RUL for systems being analyzed | 7 |
| | RUL for structures being analyzed | 8 |
| **Aircraft Utilization** | Average flight hours (FH) | 10 |
| | Average flight cycles (FC) | 11 |
| | Elapsed calendar days (DY) since last C-Check* | 12 |
| | Flight plans | 13 |

Table A.1: List of Inputs in consideration

3. Maintenance not performed while being needed (False negative)
4. Maintenance not performed without being needed (True negative)

With this in mind we are going to decide on the numerical rewards. Let:

- $M_0$ = Not performing maintenance
- $M_1$ = Performing maintenance
- $TP$ = True Positive
- $FP$ = False positive
- $TN$ = True negative
- $FN$ = False negative
- $R_1, R_2, R_3, R_4$ = Numerical rewards

The reward table A.2 shows how the numerical reinforcements are going to be distributed, positively reinforcing the agent for true negatives and true positives and offering a negative reward otherwise.

| ***Action*** | ***Result*** | ***Reward*** |
|---|---|---|
| **M_0** | TN | $+R_1$ |
| | FN | -$R_2$ |
| **M_1** | TP | $+R_3$ |
| | FP | -$R_4$ |

Table A.2: Reward table

Several tests will be made regarding the reinforces to be applied to our agent. A possible solution is to always give +1 or -1 as rewards. Another one is having the agent receive a bigger punishment for false negatives than for false positives, being that the cost of false negatives in the real world will most likely be bigger. There is no obvious solution to this but the core idea will be as that described in the previous table.

## A.1.4 Training Scenario

The goal of the MDST is to create an algorithm capable of dealing with all the uncertainty associated with this problem given the broad set of possible health condition scenarios. A scenario where dynamic programming algorithms, like Reinforcement Learning, have proven to be capable of dealing with optimization problems dealing with uncertainty (Powell, 2009). A stochastic model which will be used for the learning process will be created using Scenario Trees. A scenario tree is a representation of the branching process caused by observation (Defourny, Ernst, & Wehenkel, 2012). Beraldi, De Simone, and Violi (2010) describe each possible outcome of a scenario tree to be a node $n$ at a level $t$ while having $a(n)$ as an unique predecessor to $n$, as shown in figure A.3 (taken from (Beraldi et al., 2010)). Scenario trees are a structure capable of discretizing the uncertainty space and are commonly used with dynamic programming to formulate stochastic optimization problems (Meng & Wang, 2011; Defourny et al., 2012; Powell, 2009). Using RUL probability distribution estimations, future realizations of maintenance requirements will be simulated by the MDST team. After that, a deterministic optimization model will be created by the MDST taking into account operational maintenance requirements, resource limitations and maintenance execution policies. In this model a transition function will be created which describes how a state evolves if a certain action is adopted.

The idea is to use this model to create a simulated environment which will then be used in our task to train the reinforcement learning algorithm. The RL algorithm will receive a state $S$ as input, select an action and, using the model created before, receive an numerical reward depending on the new state. The simulated environment will behave as illustrated in figure A.4.

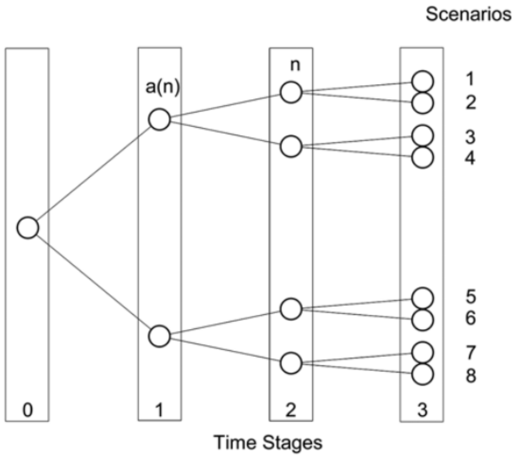


Figure A.3: Scenario tree

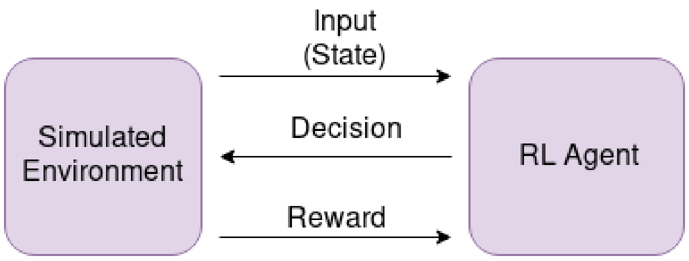


Figure A.4: Interaction between the Simulated Environment and the RL agent

### A.1.5 Methodologies

This scenario is going to be trained using two Reinforcement Learning algorithms: Deep Q-Learning and Asynchronous Actor Critic. The two algorithms will be compared in regards to their efficiency both in the quality of their results and the quickness of learning.

## A.2 Deep Reinforcement Learning Implementation and Results

Work developed was focused on fully understanding the algorithm and its state-of-the-art usages. DRL algorithms are very often tested using games, more precisely in the Atari 2600 domain using the *OpenAI Gym* library, which allows to use pre-built environments to test RL implementations. The DRL agent was implemented using *Python* and *Tensorflow* and *Google Colab* for training and testing.

The algorithm developed follows the implementation explained in (Mnih et al., 2015). Thus, below we are going to describe how the algorithm is employed, then showing the work developed by the intern and its results.

### A.2.1 Deep Q-Learning Method

As mentioned before, the DQL is very similar to the QL algorithm but, instead of making use of matrix in order so save the state-pair values, DQL makes use of a Neural Network (DQN) to do the same. This is useful in environments where the state space is too large and a matrix is not enough to deal with the problem.

Usually, when a human is playing a video-game, the main things he needs to know to make its in-game decision is the image that he sees on the screen, the current state of the game. Therefore, in the Atari 2600 environment, the agent's DQN receives the frames correspondent to the game's current state as input. Usually, to save memory and fasten the algorithm, each frame is pre-processed by cropping it, converting it to black and white and re-size it to 84x84.

As input however, the DQN does not receive only one frame representing the game's state. Instead, the state representation will be a group of subsequent frames, usually four. And why is that? Imagine we see this image of *Space Invaders*:

Figure A.5: Space Invaders Screenshot

From this image alone we can't tell in which direction the lasers are going. But if, on the other hand, wee see this two images:

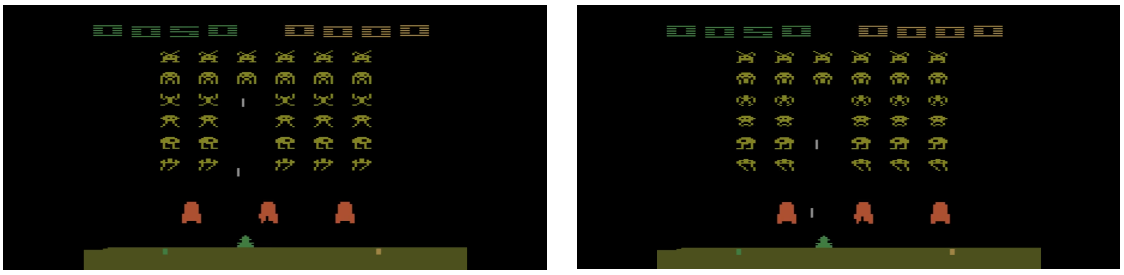


Figure A.6: Two Space Invaders Screen-shots

we can easily see the lasers are going in the player's direction. For this reason, the **state representation** will be 4 pre-processed subsequent frames.

After the input layer, the neural network is composed by three hidden convolutional layers, each of them having a Rectified Linear Unit (ReLU). The final hidden layer is fully-connected with 512 rectifier units. The last layer, the output, is also a fully-connected layer with a single output for each action possible (figure A.7). The algorithm also makes use of a **Target-Network** which will calculate the target values used in the update function. This second network is equal to the main NN (online-network).

The algorithm uses batch training, but with a modification. The samples selected for batch-training may not be consecutive to each other as they would be too related and could cause the network to converge to a local minimum. To solve this issue, a **Replay Buffer** is used, which stores a large number of previous experiences. Each experience object in the replay buffer contains the following information:

- State $S_t$
- Action $A_t$
- Reward $R_t$
- New state $S_{t+1}$
- Terminals: a Boolean which tell us if in the new state the agent lost the game or not.

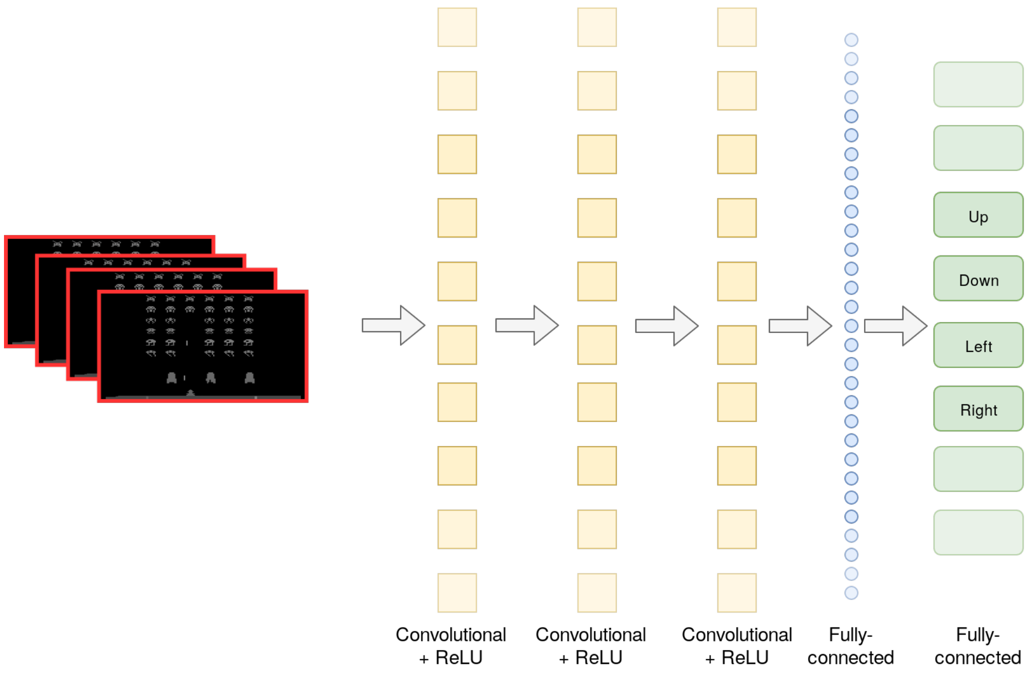


Figure A.7: Deep Q-Network Representation

During training, a sample of observations, $(s, a, r, s') \sim U(D)$, is selected randomly from the replay buffer, which is then used to update the online-network making use of the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)} = \left[\left(r + \gamma \max_a Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \tag{A.1}$$

where $\theta_i^-$ represent the target-network weights and $\theta_i$ the Q-Network parameters.

The DQL optimal value-function is defined by Mnih et al. (2015) as the maximum expected reward by taking an action $a$ in a state $s$ and then following any policy, and it is defined by:

$$Q * (s, a) = \max_\pi \mathbb{E}\Big[R_t | s_t = s, a_t = a, \pi\Big]. \tag{A.2}$$

In order to address the issue of the trade-off between exploration and exploitation, the algorithm makes use of the $\epsilon$-greedy algorithm, with $\epsilon$ decreasing linearly from 1 to 0.1.

Algorithm A.1 details the pseudo-code for the DQN algorithm (Mnih et al., 2015).

### A.2.2 Results

An implementation of the algorithm was made as the initial work for the intern. The Algorithm was tested with the *Atari* domain provided by *Gym*, more precisely for the *Seaquest* environment.

**Algorithm A.1** Pseudo-code for DQN

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target-action-value function $\overset{\wedge}{Q}$ with weights $\phi^- = \phi$
for episode=1, M do
    Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_t + 1$
        Set $s_t + 1 = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j, \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \overset{\wedge}{Q}(\phi_{j+1}), a'; \theta^-), \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(\gamma_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
network parameters $\theta$
        Every C steps reset $\overset{\wedge}{Q} = Q$
    end for
end for
```

The agent's interactions were saved in a replay buffer who held up to 5000 observations. Prior to the agent's training, that buffer was stored with 1000 random observations. The replay buffer acts as a queue so, whenever it is full, a new observation replaces the last one. The agent was trained for 1000000 frames and each 10000 frames the target-network would be updated by copying the values of the online-network. All the hyper-parameters used are detailed below (A.3).

| **Hyperparameter** | **Value** | **Description** |
|---|---|---|
| Batch size | 32 | Number of samples taken from the replay buffer to train our agent |
| Replay buffer size | 5000 | Size of the queue that saves the agent's observations |
| Pre-population size | 1000 | Number of random observations stored before training |
| Discount factor | 0.99 | Discount factor used in Q-learning |
| Frames stacked | 4 | Number of frames stacked to form a state $S$ |
| Learning rate | 0.00025 | Learning rate used by the network's optimizer |
| Max frames | 1000000 | Maximum number of frames the agent saw during training |
| Initial exploration | 1 | Initial value of $\epsilon$ |
| Final exploration | 0.1 | Final value of $\epsilon$ |
| Target Update | 10000 | Frequency with which the target-network is updated |

Table A.3: Hyperparameters used

Preliminary results showed a clear tendency for the agent to gradually better its results. In A.8 the maximum result was 760 and in figure A.9 it was 801 while, according to (Mnih et al., 2015), the average score for randomly choosing actions in *Seaquest* is 68.4. The main constraint to these implementations was the replay buffer. Mnih et al. use 1000000 frames while in here it was only possible to have

5000 in order to train the agent for a long number of frames. Because the role of the replay buffer is to avoid correlation between experiences, with 5000 that might not have been completely achieved. Still, the results were decent and there is a gradual grow in the agent's received rewards during training.

Several optimizers and initializers were tested in the neural network, having the best results been achieved with Adam optimizer (Kingma & Ba, 2014) and the RMSprop optimizer (Ruder, 2016), using the He initialization method (He, Zhang, Ren, & Sun, 2015).

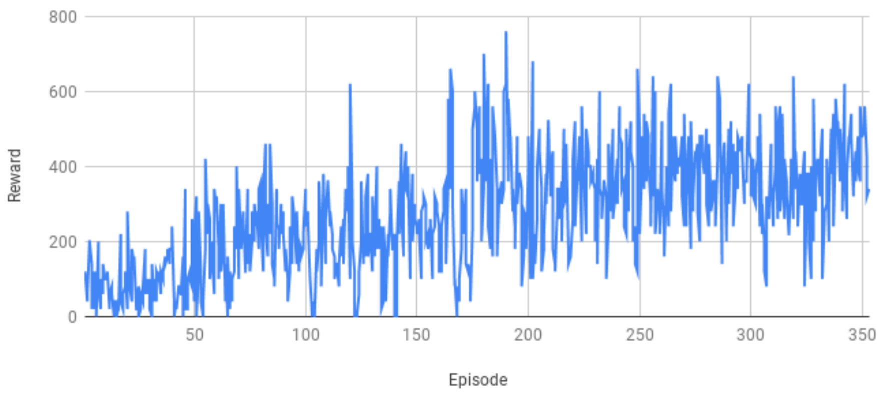


Figure A.8: Results with Adam optimizer

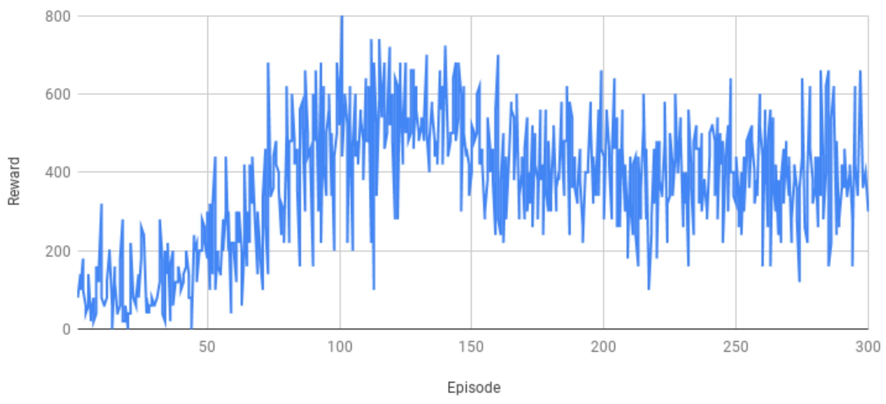


Figure A.9: Results with RMSprop optimizer

# Appendix B

# Genetic Algorithm

## B.1 Introduction

In computer science, a genetic algorithm (GA) is a search heuristic inspired by the process of natural selection. Typically, a genetic algorithm starts with a group of individuals (population), usually randomly created. After that, recombination operators such as mutation and crossover operations are performed on the population so the algorithm can explore the search space. Selection operators are also used to decide which more ”apt” individuals survive and go through the next generation as a way to “exploit” the good solutions. The decision of how good is an individual is made using a fitness function and is what defines the goal of the algorithm.

We can say that this method is very similar to reinforcement learning, in a lot of ways. We can think of mutation and crossover operations in GA as the actions in RL, the fitness function as the reward function and the individual as the state representation. This was one of the reasons that inspired the use of this method.

## B.2 Problem

Similar to the problem to be solved with RL, we also have a set $T$ of tasks created randomly, to be distributed for $D$ days making use of $H$ hangars. The difference here is that the number of hangars must obligatorily be 2. The goal, as before, is to schedule each task as close to the due date as possible without overlapping each other.

## B.3 Formulation

In a GA implementation three main components must be defined:

- Individuals

- Genotype to phenotype
- Fitness

## B.3.1 Individuals

Each individual illustrates a possible solution for the problem and it is a computational representation for that solution. An individual is usually composed of an array of numbers in which each number is said to be a gene. In our problem, each gene represents a task. We have to sort our tasks within a certain number of days, therefore each individual will be a permutation of $T+1$ genes. The extra gene is the number **0** which will signify the division between hangars, meaning that the tasks left to 0 will be scheduled to hangar 1 and the tasks right to the 0 will be scheduled to hangar 2. If our individual is, for example,

*3 2 4 0 1 5*,

the first hangar will be composed by tasks $t3$, $t2$ and $t4$ and hangar t2 by tasks $t1$ and $t5$, as shown in B.1.

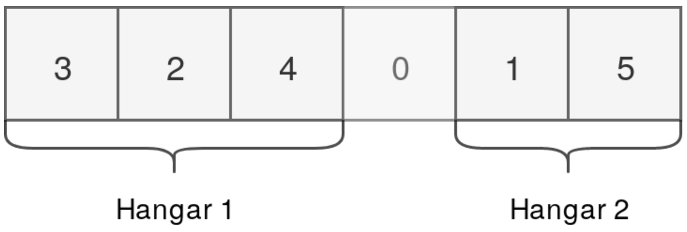


Figure B.1: Example of the hangar division in a individual

## B.3.2 Genotype to phenotype

Above, we said an individual was a computational representation of a possible solution. That computational representation is called genotype (similar to the state representation), while the phenotype is more "real" representation of the individual which allows us to evaluate it. In our example, the phenotype will be the calendar itself. What we need is a function that converts the genotype to a phenotype (state representation to calendar) in order to evaluate a certain solution.

What we will do is loop the genes (tasks) of each hangar, from right to left. We will have a variable that tells us what was the last scheduled day, meaning the last day of the calendar that was occupied. Tasks can not be scheduled after this value. We will then, for each task in an individual, verify if it is possible to schedule it on its due date. If it is we do it. If not, we schedule the task after the last day occupied. Algorithm B.1 shows what the process looks like for one hangar.

**Algorithm B.1** Algorithm to translate from genotype to phenotype

```
$d$ = day where the task will be scheduled
$L$ = last day where a task was scheduled
$dd$ = due date of the task
$l$ = length of a task
$h$ = genotype of one hangar
$C$ = phenotype of the hangar
for t in h do
    if $dd < L$ then
        $d = dd$
    else
        $d = L - 1$
    end if
    schedule task on $C$
    $L = d - l$
end for
```

The steps to translate a genotype to a phenotype might be a bit confusing, so let's go through an example, step by step, to better understand it.

Let's say we have to schedule the tasks detailed in figure B.2 along 10 days and our individual is the one represented in B.1. That being said we have $T = 5$, $D = 10$ and, of course, $H = 2$.

| Task ID | Length | Due Date |
|---|---|---|
| 1 | 3 | 6 |
| 2 | 2 | 4 |
| 3 | 1 | 3 |
| 4 | 3 | 6 |
| 5 | 1 | 5 |

Figure B.2: Table of tasks

We already know that the first hangar will have tasks 2, 3 and 4 and the second hangar will have tasks 1 and 5. Let's start with the first hangar. As was said before, the algorithm goes through the tasks from right to left, so the order will be: *4, 2, 3*. With $L$ being the last day that had a task scheduled in it, let's begin.

First, we have task 4. Because $L$ doesn't exist yet, $t4$ can be scheduled on due date (figure B.3).

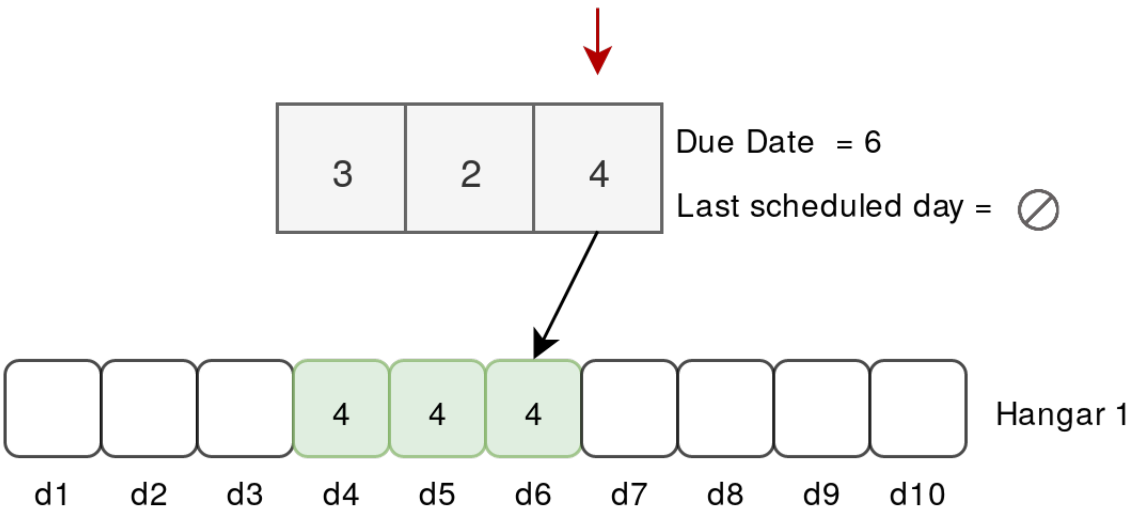


Figure B.3: Scheduling $t4$

Now we have $L = 4$ and $d4$ as the due date for $t2$. Because the due date equals $L$, $t2$ will be scheduled on $L - 1$. The new $L$ is now 2 (figure B.4).

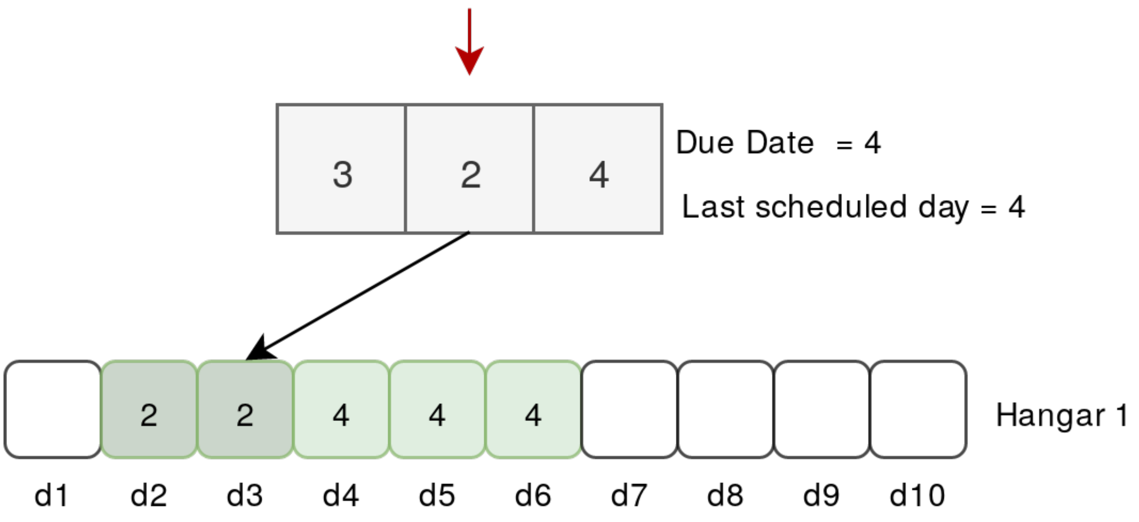


Figure B.4: Scheduling $t2$

For task 3 we have a similar situation as before but this time it is because its due date is bigger than $L$. $t3$ will then be scheduled on $L - 1$, which gives us $d1$ (figure B.5).

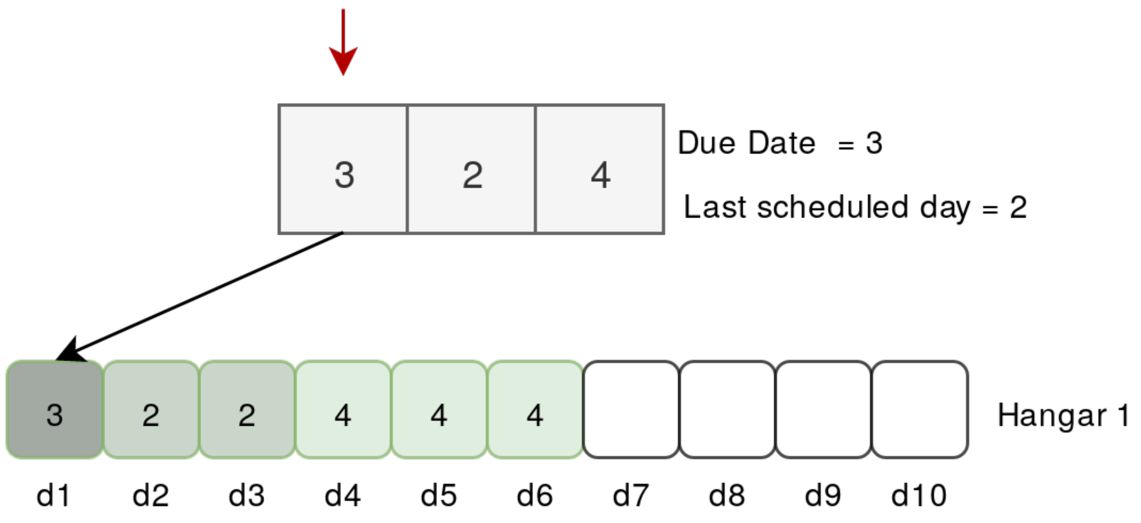


Figure B.5: Scheduling $t3$

Following the same procedure for the second hangar we get the calendar as shown in B.6

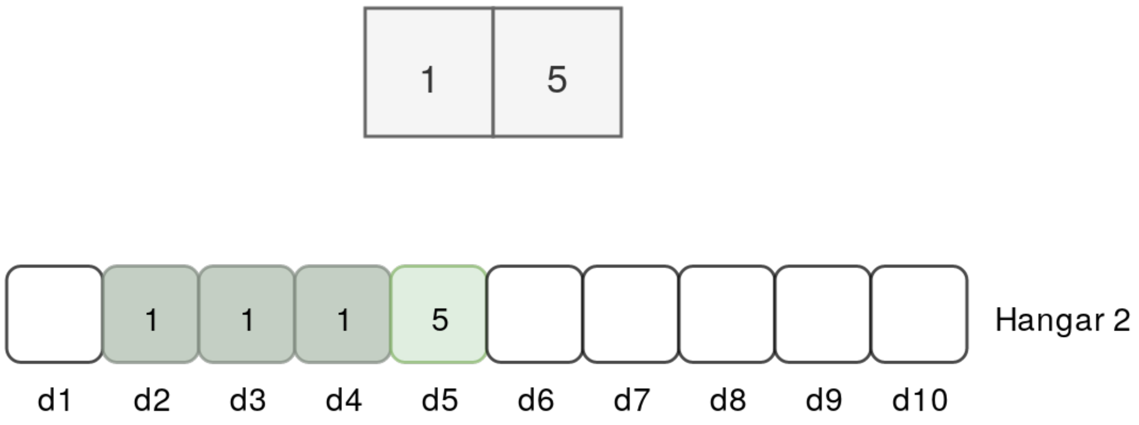


Figure B.6: Calendar for the second hangar

### B.3.3 Fitness

The fitness function used is similar to the one used in the RL formulation. The objective here is to schedule each task as close to the due date as possible while giving a bigger punishment for tasks scheduled after their due date. Thus, the fitness function is defined as follows:

```
$T$ = Set of tasks
$t_d$ = scheduled day of task $t$
$t_{dd}$ = due date of task $t$
$fitness = 0$
while $t < T$ do
    if $t_d < t_{dd}$ then
        $fitness$ -= $t_d - t_{dd}$
    else
        $fitness$ -= $5 * (t_{dd} - t_d)$
    end if
```

There is a situation, however, that was a significant issue while formulating this solution. If the last scheduled day gets to $d1$ and there are still tasks left to schedule, how can we deal with that situation, being that there are no more days left? We could ignore that situation and simply evaluate those individuals as less fitted. However, that would not encompass situations where tasks must be scheduled after the due date. To cover those situations, whenever $L$ is $d1$ and there are still tasks left to schedule, the algorithm schedules the following tasks after their due dates.

### B.3.4 Results

Just as in Chapter 6, the obtained results were compared with a greedy algorithm and an algorithm that chooses a random decision in each conflict. This random-action formulation was run 100 times in each exercise and the mean of the rewards were used as a comparison.

We are going to display the results for three different exercises. For each one of them, we show the mean and max fitness per generation plus a table that compares the results for the three algorithms mentioned above.

The number of generations was the only parameter that varied between exercises. Usually, a bigger number of generations was used for problems with a larger number of days and/or tasks. Below follows the list of parameters used for the GA algorithm:

- Population size: 100
- Number of generations: varies between exercises
- Crossover probability: 0.9
- Mutation probability 0.05
- Elitism percentage: 0.01
- Tournament size: 3

### 20 tasks, 40 days

- $|T| = 20$
- $|D| = 40$
- $1 \leq l \leq 3$

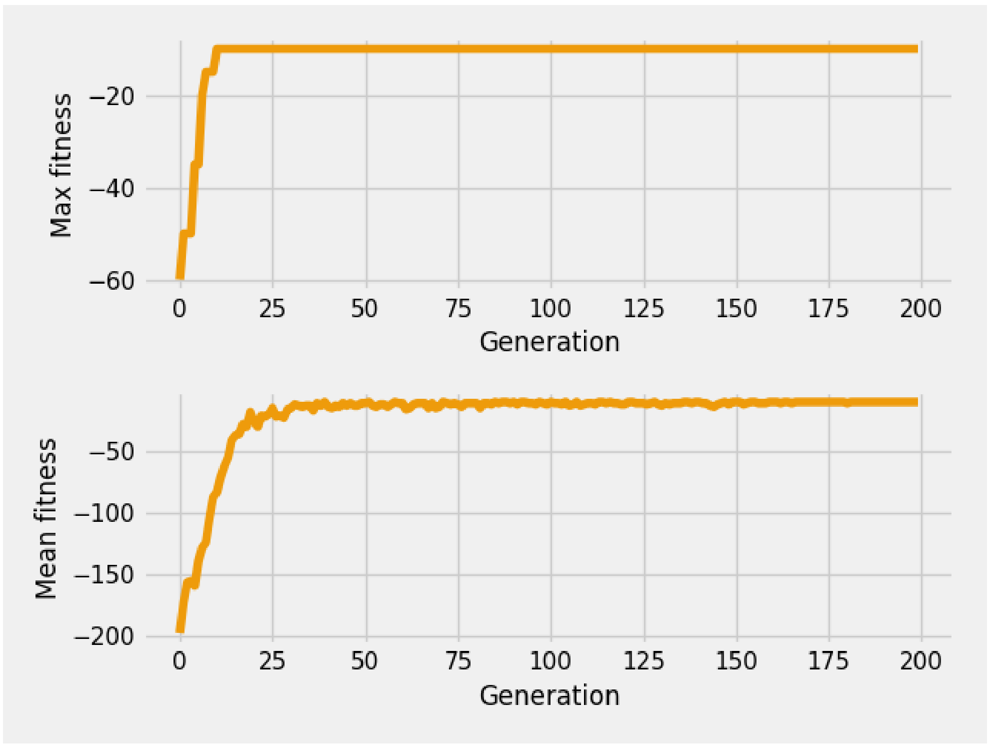


Figure B.7: Results with 20 tasks and 40 days

| | **GA** | **Greedy** | **Random** |
|---|---|---|---|
| **Result** | -10 | -12 | -14 |

### 20 tasks, 100 days

- $|T| = 20$
- $|D| = 100$
- $1 \leq l \leq 8$

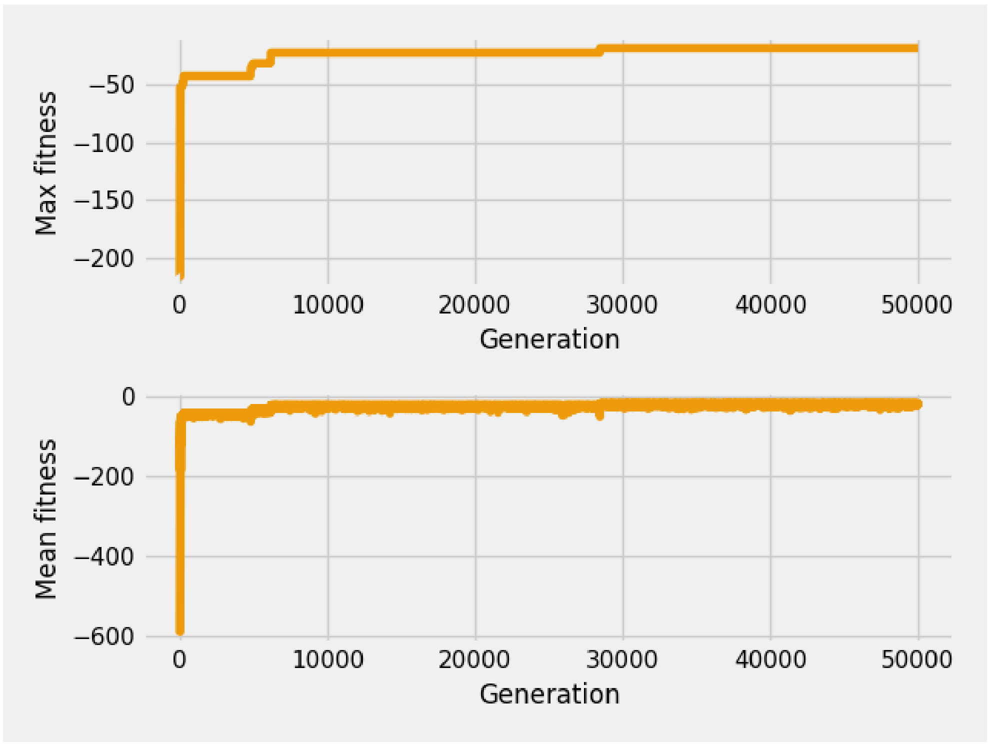


Figure B.8: Results with 20 tasks and 100 days

| | **GA** | **Greedy** | **Random** |
|---|---|---|---|
| **Result** | -19 | -24 | -56 |

### 50 tasks, 200 days

- $|T| = 50$
- $|D| = 200$
- $1 \leq l \leq 8$

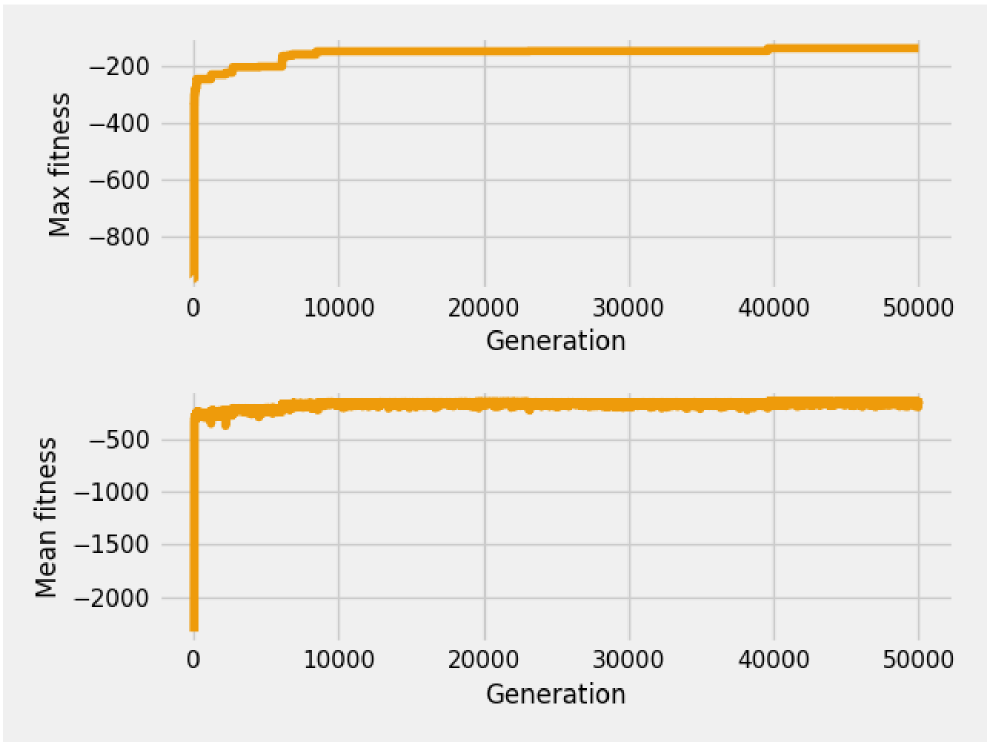


Figure B.9: Results with 50 tasks and 200 days

| | **GA** | **Greedy** | **Random** |
|---|---|---|---|
| **Result** | -139 | -83 | -586 |

## B.4 Conclusion

This genetic algorithm formulation was tested briefly and perhaps more time and tests would produce better results, also with a more thorough study made on the method.

Regardless, the algorithm developed was enough to achieve decent results with a lower number of tasks and days being that, in both exercises in which 20 tasks are used, the GA achieved better results than the greedy formulation.

With a bigger number of tasks and days, however, the results from the GA algorithm tend to be worse than the ones from the greedy formulation. Perhaps if the GA ran for more generations the results would be closer, although computationally expensive. Lastly and, in what came as no surprise, the GA formulation performed significantly better than the random-action algorithm.