

UNIVERSITATEA BABEȘ-BOLYAI  
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
SPECIALIZAREA INFORMATICĂ

## **LUCRARE DE LICENȚĂ**

# **Optimizarea transmiterii datelor în patrularea cu rețele multi-robot**

**Coordonator științific**  
**Prof. dr. Czibula Gabriela**

**Student**  
**Bădiță Marin-Georgian**

2020

BABEȘ-BOLYAI UNIVERSITY  
FACULTY OF MATHEMATICS AND COMPUTER  
SCIENCE

## **DIPLOMA THESIS**

# **Data Delivery Optimization in Multi-Robot Network Patrolling**

**Scientific supervisor**  
**Prof. dr. Czibula Gabriela**

**Student**  
**Bădiță Marin-Georgian**

2020

## Abstract

The subject of this thesis is the domain of Multi-Robot Patrolling (MRP). This thesis tackles a specific problem in the MRP called Data Delivery Optimization in Multi-Robot Network Patrolling. The MRP is a new research field with high potential for massive discoveries.

The thesis is structured in four chapters. In the first chapter some of the main concepts about multi-robot patrolling and reinforcement learning are introduced, while also reviewing the current literature about MRP. In the second chapter a novel approach to the Data Delivery Optimization problem, based on a Dynamic Network Flows algorithm, is discussed, called *DynFloR*. The following chapter aims to prove that a reinforcement learning approach to the problem is highly feasible, by introducing and testing a Q-learning model. The last chapter presents the software solution which solves the Data Delivery Optimization problem, providing details of the software development methodology used and implementation details.

The main elements of originality of this thesis consist in two new approaches to the problem, one based on Dynamic Network Flows (*DynFloR*) and another one based on Reinforcement Learning (*DronemRL*). A further original contribution consists in creating and designing the application showcased in the last chapter, as to the best of my knowledge there exists no application for creating, training and testing MRP environments publicly available.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Bădiță Marin-Georgian

Cluj-Napoca  
30-06-2020

# Contents

<b>Introduction</b>	<b>6</b>
<b>List of publications</b>	<b>7</b>
<b>1 Background</b>	<b>8</b>
1.1 Multi-robot patrolling . . . . .	8
1.2 Reinforcement learning (RL) . . . . .	10
1.2.1 Q-learning . . . . .	11
1.3 Literature review on <i>multi-robot</i> network patrolling . . . . .	12
<b>2 A Flow Approach for Data Delivery Optimization</b>	<b>13</b>
2.1 Methodology . . . . .	14
2.1.1 Problem definition. Theoretical model . . . . .	14
2.1.2 The proposed <i>DynFloR</i> approach . . . . .	16
2.1.3 Example . . . . .	19
2.2 Results and discussion . . . . .	21
2.2.1 Comparison to related work . . . . .	23
2.3 Conclusions and future work . . . . .	24
<b>3 A RL Approach for Data Delivery Optimization</b>	<b>25</b>
3.1 The proposed RL methodology . . . . .	26
3.1.1 The DronemRL model . . . . .	26
3.2 Results and discussion . . . . .	30
3.2.1 Comparison to related work . . . . .	31
3.3 Future work . . . . .	33
<b>4 A Software Solution for the Patrolling Problem</b>	<b>35</b>
4.1 Software development . . . . .	36
4.1.1 Analysis and design . . . . .	36
4.2 OpenAI . . . . .	41
4.2.1 The Dronem environment . . . . .	41
4.2.2 Advantages and Disadvantages . . . . .	42

4.3	Implementation . . . . .	43
4.3.1	Backend . . . . .	43
4.3.2	Django vs Flask . . . . .	43
4.3.3	Django REST Framework . . . . .	45
4.3.4	Database . . . . .	45
4.3.5	Frontend . . . . .	45
4.3.6	Design choices . . . . .	46
4.3.7	Testing . . . . .	48
4.3.8	Documentation . . . . .	49
4.4	User manual . . . . .	50
4.4.1	Creating environments . . . . .	50
4.4.2	Training environments and downloading Q-tables . . . . .	51
4.5	Discussion . . . . .	53
4.6	Future enhancements . . . . .	53
	<b>Conclusions</b>	<b>55</b>

# List of Figures

2.1	Synthetic example. . . . .	15
2.2	Simple example. . . . .	19
2.3	The dynamic network for the example from Figure 2.2. The decision making strategy for the robots is marked with red. . . . .	20
2.4	Results for 20 robots and $MaxCapacity = 1000$ . . . . .	21
2.5	Results for 20 robots and $MaxCapacity = 800$ . . . . .	21
2.6	Variation of lost data per time for the example from Figure 2.2. . . . .	23
3.1	Results for 3 robots and $InitialMemory = 4$ . . . . .	30
3.2	Results for 3 robots and $InitialMemory = 4$ . . . . .	30
3.3	Results for 3 robots and $MaxCapacity = 9, InitialMemory = 7$ . . . . .	31
3.4	Convergence graph with respect to $P$ . . . . .	31
3.5	Classic Q-learning vs Deep Q-learning . . . . .	32
3.6	Multi-Agent DDPG. . . . .	33
4.1	Use case diagram for Dronem web application . . . . .	37
4.2	Dronem Web conceptual model . . . . .	38
4.3	Dronem Web refined UML diagram . . . . .	40
4.4	Dronem environment life cycle . . . . .	42
4.5	Dronem environment Gym architecture . . . . .	42
4.6	Number of StackOverflow questions for Django and Flask . . . . .	44
4.7	Dronem Web database schema . . . . .	46
4.8	API documentation for <code>/users/signin/</code> path . . . . .	50
4.9	Drone Web add environment view . . . . .	51
4.10	Drone Web train view . . . . .	51
4.11	Drone Web environments view . . . . .	52
4.12	Drone Web statistics view . . . . .	52

# List of Tables

2.1	Quantity of data in memory, at the sink and lost, for different maximum capacities and time limits for the example from Figure 2.2. Initial data and new data received after every completed cycle is 15.	22
2.2	Progression of lost data as time increases for different <i>MaxCapacity</i> values	22

# List of Algorithms

1	Algorithm <i>DynFloR</i> . . . . .	17
2	Function <i>CreateGraph</i> . . . . .	18
3	Function <i>UpdateQTable</i> . . . . .	28
4	Algorithm <i>DronemRL</i> . . . . .	29



# Introduction

**Patrolling** represents the repeated visit of certain places, over an area. The *multi-robot patrolling* (also called *multi-agent patrolling*) problem (MRP) consists in minimizing the time between two consecutive visits of the same place (called *idleness*), using two or more robots. In the literature, this problem is generally considered to be NP-hard, while solutions often involve cyclic paths [Che04a, GSFA08]. Applications of the multi-agent patrolling problem include: surveillance tasks (e.g. anomaly detection, intruder detection), area coverage, data collection tasks (e.g. for the case of wireless sensor networks - WSNs), rescue operations (e.g. people or objects in dangerous situations) [Rob, DDLW09].

A major challenge when deploying fleets of mobile robots in real scenarios/environments is the ability of the robots to adapt to the complexity of their environment, that is its dynamics and uncertainty. In such dynamic environments the robots need to be able to provide robust solutions to complex tasks and to adapt to changes in their environment. The multi-agent patrolling task is intensively investigated within the multi-agent research community and various algorithms based on reactive and cognitive architectures have been introduced [OGEFSFPB17].

This thesis offers solutions for optimizing data delivery in multi-robot network patrolling in the scenario where the environment is deterministic, the network of robots is centralized and offline and the robots have periodic meetings and also aims to prove that in a more general scenario of the multi-robot network patrolling, where the patrolling environment is non-deterministic and uncertain a *reinforcement learning* approach is feasible. The problem was proposed in a bilateral collaboration between the MLYRE team from Cluj-Napoca coordinated by prof. Gabriela Czibula and the CHROMA team from Insa Lyon coordinated by prof. Olivier Simonin.

The rest of the thesis is structured as follows. Chapter 1 presents the background concepts used in this thesis, while our Dynamic Network Flows algorithm is outlined in Chapter 2. Chapter 3 is concerned with proving that a *reinforcement learning* approach is feasible and a detailed software solution is presented in Chapter 4. The conclusions of our thesis and directions to further continue and improve our research are given in Section 4.6.

# List of publications

- [ItMata<sup>+</sup>19] Vlad-Sebastian Ionescu, Zsuzsanna Onet-Marian, **Marin-Georgian Bădiță**, Gabriela Czibula, Mihai-Ioan Popescu, Jilles S. Dibangoye, Olivier Simonin. *DynFloR: A Flow Approach for Data Delivery Optimization in Multi-Robot Network Patrolling*. 23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2019), Procedia Computer Science Vol 159, (2019) pp. 97-106 (**ISI Proceedings**).

# Chapter 1

## Background

The MRP is a relatively new field in research, with high potential for discoveries, but yet at the same time with its own challenges, mostly concerning uncertain and dynamic environments.

The purpose of this chapter introduce some of the main concepts used in this thesis, by creating an underlying basis for the patrolling problem and its place in the research world. The chapter is structured as follows. The Multi-robot patrolling problem is explained and detailed in Section 1.1. A general introduction to Reinforcement Learning is presented in Section 1.2, while Section 1.3 contains a literature review on the *multi-robot* network patrolling.

### 1.1 Multi-robot patrolling

**Patrolling** represents the repeated visit of certain places, over an area. The *multi-robot patrolling* (also called *multi-agent patrolling*) problem (MRP) consists in minimizing the time between two consecutive visits of the same place (called *idleness*), using two or more robots. In the literature, this problem is generally considered to be NP-hard, while solutions often involve cyclic paths [Che04a, GSFA08]. Applications of the multi-agent patrolling problem include: surveillance tasks (e.g. anomaly detection, intruder detection), area coverage, data collection tasks (e.g. for the case of wireless sensor networks - WSNs), rescue operations (e.g. people or objects in dangerous situations) [Rob, DDLW09]. Figure 2.3 illustrates a solution for the case of data collection in WSNs (from [PRS16]).

We consider the general case of the patrolling of specific places with a fleet of robots. *Multi-robot patrolling* consists in organizing the continuous coverage of an area by several agents, e.g. drones, autonomous vehicles etc. The problem of *multi-robot patrolling* with minimum visiting frequency has been formalized in [PRS16] as a *clustering* problem. Considering that  $T = \{t_1, t_2, \dots, t_n\}$  is a set of targets that have to

be patrolled, a patrolling solution may be viewed as a partition  $\mathcal{K} = \{K_1, K_2, \dots, K_p\}$  of  $T$  ( $K_i \subseteq T, K_i \neq \emptyset, \forall 1 \leq i \leq p$  and  $T = \bigcup_{i=1}^p K_i$  and  $K_i \cap K_j = \emptyset, \forall 1 \leq i, j \leq p, i \neq j$ ) with minimum  $p$ . In addition,  $length(C_i) \leq B$ , where  $C_i$  is the cycle formed on cluster  $K_i, 1 \leq i \leq p$ , and  $B$  is a given bound imposed on the cycles' length, required due to the minimum visiting frequency that has to be achieved in the target patrolling.

A major challenge when deploying fleets of mobile robots in real scenarios/environments is the ability of the robots to adapt to the complexity of their environment, that is its dynamics and uncertainty. In such dynamic environments the robots need to be able to provide robust solutions to complex tasks and to adapt to changes in their environment. The multi-agent patrolling task is intensively investigated within the multi-agent research community and various algorithms based on reactive and cognitive architectures have been introduced [OGEFSFPB17]. Still, the existing patrolling approaches are specific ones and those regarding dynamic environments (*dynamic multi-robot patrolling* - DMRP) are in early phases [OGEFSFPB17]. An extended and more complex version of the patrolling is the *dynamic patrolling*, in which places (or targets) to patrol will be discovered progressively by agents/robots that spread out in the environment. A challenge in such a setting is to ensure that robots remain connected, i.e. are able to communicate, in order to cooperate and to pass collected data up to their base station (where they start).

Starting from the formalization introduced in [PRS16], the DMRP problem may be formalized as a *dynamic clustering* problem. In the DMRP, two types of information are discovered online by the robots, and they are required to adapt their current patrolling solution. First, the targets to patrol are not known a priori, they will be discovered while robots spread out in the environment. As the robots have to start to patrol as soon as they have detected some targets, they must modify their patrol when they discover new ones. The problem of adapting the clusters (cycles) when new targets are discovered has been approached by Popescu et al. [PSC<sup>+</sup>18], where a dynamic saturation-based auctioning algorithm (DSAT) was introduced for a continuous adaptation of the multi-robot target allocation process (MRTA) to new discovered targets.

While the robots patrol a group (or a cluster) of targets by following a cyclic path and collect data from the targets, they have to pass collected data to their neighboring robots/cycles in order to communicate information to the base station. In network patrolling, each robot can have the choice to transmit collected data to different neighbors, i.e. to other close paths (cycles).

## 1.2 Reinforcement learning (RL)

Since the beginning of computing, mathematicians and computer scientists were concerned about how to create programs or machines capable of learning in the same way as humans do, that is learning from interaction with our environment, which is a foundational idea underlying nearly all theories of learning and intelligence [SB98].

Reinforcement learning (RL) deals with the problem of how an autonomous agent which is situated in an environment, perceives and acts upon it and also can learn to select optimal actions to achieve its goals [Mit97]. Reinforcement learning is used in many practical problems, such as learning to control autonomous robots [KKGB12a], learning to find the solution of an optimization problem (such as operations in factories) or learning to play board games. In all these problems, the agent has to learn how to choose optimal actions in order to achieve its goals, through the reinforcements received after the interaction with its environment.

In a reinforcement learning task, the learner tries to perform actions in the environment and it receives *rewards* (or *reinforcements*) in the form of numerical values that represent an evaluation of how good were the selected actions [PUS99]. The learner (agent) simply has a given goal to achieve and it must learn how to achieve that goal by trial-and-error interactions with the environment. RL is learning how to map situations to actions in order to maximize the cumulative reward received when starting from some initial state and proceeding to a final state.

A general RL task is characterized by four components [SB98]. The *environment state space*  $\mathcal{S}$  represents all possible states of an agent in the *environment*, for example every cell on a word represented as a grid. The *action space*  $\mathcal{A}$  consists of all actions that the learning agent can perform in the environment. The *transition function*  $\delta$  specifies the non-deterministic behavior of the environment (i.e. the possibly stochastic outcomes of taking each action in any state). The last component of the RL task is the *reinforcement (reward) function* which defines the possible reward of taking an action in a particular state.

The agent's task in a reinforcement learning scenario is to learn an *optimal policy*,  $\pi : \mathcal{S} \rightarrow \mathcal{A}$ , that maximizes the expected sum of the delayed rewards for all states  $s$ :

$$V^\pi(s) = \sum_{i=0}^n r_i \cdot \gamma^i$$

The future rewards are discounted exponentially by their delay and  $\gamma$  ( $0 \leq \gamma < 1$ ) is the discount factor for the future rewards.

### 1.2.1 Q-learning

A popular and effective RL method is considered to be the *Q-learning* algorithm [SB98]. In this scenario, the agent learns an *action-value function* ( $Q$ ) giving the expected utility of taking a given action in a given state. In such a scenario, the agent does not need to have a model of its environment.

For training the RL agent, a *Q-learning* approach is usually used, in which the agent learns the  $Q$ -value function that gives the expected utility of performing an action in a given state [SB98]. The training process consist of the following. Through a number of training episodes, the agent will try (possible optimal) *candidate solution* paths from the initial to a final state. After performing an action in its environment, the agent will receive rewards and will update the  $Q$ -values estimations according to the *Bellman's equation* [DS94] where  $Q(s, a)$  denotes the estimation of the  $Q$ -value associated to the state  $s$  and action  $a$ ,  $\alpha$  represents the learning rate and  $\gamma$  is the discount factor for future rewards.

$$\text{Bellman's equation: } Q(s, a) = Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a' \in A} Q(s', a') - Q(s, a))$$

The general form of the *Q-Learning* algorithm is given in Algorithm 1:

---

```

Repeat
  Select the initial state of the agent in the environment as  $s_1$ .
  Select action  $a$  from  $s$  using an action selection mechanism.
  Repeat
    Perform action  $a$ , observe the reward  $r$  and the next state of the environment  $s'$ .
    Update the value  $Q(s, a)$  as follows

      
$$Q(s, a) = Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a' \in A} Q(s', a') - Q(s, a))$$


     $s \leftarrow s'$ 
  until  $s$  is terminal
Until the maximum number of episodes is reached or the  $Q$ -values do not change

```

---

**Algorithm 1.** The *Q-learning* algorithm [WD92].

### 1.3 Literature review on *multi-robot network patrolling*

The field of the multi-robot patrolling is relatively recent, with various contributions which are mainly based on operational research. The networks are usually represented as graphs, where the nodes are critical points (targets) that should be visited as often as possible and the edges represent paths between these targets.

Graph based algorithms were used for various patrolling tasks: Hamiltonian cycles for assuring that each point in the target area is covered at the same optimal frequency [EAK09], cyclic strategies which use heuristics to compute TSP cycles [Che04b], path-finding techniques for implementing the decision making strategy for the agents. Alternative approaches in multi-robot patrolling use classical techniques for computational agents coordination in certain environments [PR11], cooperative auction systems [HLH09] and algorithms based on swarm intelligence [CGS<sup>+</sup>07]. Santana et al. modelled the patrolling problem as a reinforcement learning problem [SRCR04], for allowing the agents to automatically adapt to their environment with the goal of optimizing (maximizing or minimizing) a certain performance criterion (e.g. minimizing the nodes idleness). The authors have shown that the adaptive solutions are superior to other solutions mainly due to the work is distributed (no centralized communication) and the adaptive behavior of the agents, which can be desirable in this domain [SRCR04].

Liemhetcharat et al. [LYTL15] approached the problem of foraging and item delivery with multi-robot networks. The authors proposed a formal model for the multi-robot item delivery problem and showed that the continuous foraging problem is a particular case of it. Distributed multi-robot algorithms were also proposed to solve the item delivery and foraging problems and experimented on simulated robots using a Java simulator. Chen et al. [CWS<sup>+</sup>15] investigated a multi-agent patrolling problem in uncertain environments. For dealing with uncertainty and possible threats, the environment was modelled as multi-state Markov chains, whose states are partially observable until the location is visited by an agent. The main goal of the approach from [CWS<sup>+</sup>15] was to maximize the amount of information gathered by the agents while reducing the damage incurred. Farinelli et al. [AFEZ17] formalized the problem of coordination (i.e. establishing collaborative interactions between robots to achieve individual and collective goals) as a Distributed Constrained Optimization problem, providing a solution based on the binary max-sum algorithm. The problem of *dynamic multi-robot patrolling* was recently approached by Othmani-Guibourg et al. [OGEFSFPB17]. The authors proposed a formal model for dynamic environment and on edge-markovian evolving graphs and they introduced and analyzed two strategies for the agents for patrolling in dynamic environments.

## Chapter 2

# *DynFloR*: A Flow Approach for Data Delivery Optimization in Multi-Robot Network Patrolling

Deploying fleets of mobile robots in real scenarios and environments raises several scientific challenges. One of them concerns the ability of the robots to adapt to the dynamics of their environment. We introduce *DynFloR* [ItMata<sup>+</sup>19], a dynamic network flow based approach for finding optimal policies for *data delivery* in *multi-robot network patrolling* where the robots can communicate instantly and free of charge one to another, there is a periodicity of the robot meetings and the distribution of the data collected during the patrol is regular. Experiments on randomly generated synthetic examples are performed for evaluating the performance of the *DynFloR* method. The performed experiments empirically show that independent of the problem setting (such as number of robots, memory of the robots) the amount of data transferred to a base station per unit of time converges to an equilibrium state. The case of lost data has been also examined through various experiments, but it requires further experimentation as well as in-depth analysis. This chapter represents the first step in a research which is being conducted for using *reinforcement learning* in order to optimize data delivery in a general multi-robot network patrolling case, when the patrolling environment is non-deterministic and uncertain.

The study performed in this chapter represents the starting point of a broader research which is being conducted for optimizing data delivery in DMRP using *reinforcement learning* (RL) [AHHS18], in the more general case when there is an uncertainty in the environment (i.e. there is no periodicity in the robots' meetings). In this general case, the robots will have to learn through *reinforcement* (RL) [KKGB12b] the data transfer policy, in order to maximize the quantity of transmitted data.



The contribution of this chapter consists in a new approach, *DynFloR*, based on dynamic network flows for determining the optimal policy for delivering data in multi-robot network patrolling under the following assumptions: the environment is deterministic, the network of robots is centralized and offline and the robots have periodic meetings. The proposed *DynFloR* method adapts the approach proposed by Kotnyek [Kot03] for the problem of MRP. It can be also extended for the DMRP under the previously mentioned assumptions. We assume the data transfer to be dynamic and we address the following research question: *How to optimize data-passing between each robot cycle in order to maximize the quantity of data transmitted to the sink?* Experiments and simulations performed on several synthetic and randomly generated examples emphasize that our proposal is able to determine a policy that maximizes the data transferred to the base station up to a given moment in time. The answer to the previously stated RQ through the *DynFloR* approach will provide us additional insights on the nondeterministic MRP/DMRP problem and its modelling as a reinforcement learning task. To the best of our knowledge, *DynFloR* approach is new in the MRP literature.

The rest of the chapter is structured as follows. Our *DynFloR* approach for data delivery optimization in a deterministic setting of the MRP is introduced in Section 2.1. Section 2.2 presents the experimental results obtained by evaluating *DynFloR* and provides a comparison to existing similar approaches. Furthermore, in Section 2.2 we have a discussion about the current solution as well as its advantages and limitations. Section 2.3 contains the conclusions of the chapter and directions to continue our research.

## 2.1 Methodology

This section introduces our centralized and offline *DynFloR* approach for data delivery optimization in a deterministic setting of the MRP. *DynFloR* is applicable in a deterministic setting of DMRP, as well.

### 2.1.1 Problem definition. Theoretical model

The problem we are approaching in this chapter is defined in the following. We assume the particular case of a deterministic environment, in which the network of robots is centralized and offline: *There are  $n$  robots, which can meet and exchange data. The robots are continuously collecting data on the cycles they patrol. There are initial meeting times for the robots and meeting periods. The goal is to maximize the quantity of data arriving to a sink (a base station) in a given time  $T$ .*

Let us consider the following theoretical model. We denote by  $\mathcal{R} = \{R_1, R_2, \dots, R_n\}$

the set of patrolling robots. We assume the sink is denoted by  $R_0$ . Each robot  $R_i \in \mathcal{R}$  patrols a certain cycle  $C_i$  consisting of  $k_i$  targets, i.e.  $C_i = \{t_i^1, t_i^2, \dots, t_i^{k_i}\}$ . At each visit of the target  $t_i^j$ , an amount  $d_i^j$  of data will be collected by robot  $R_i$ . Hence, after patrolling a complete cycle, robot  $R_i$  collects a quantity of data  $D_i = \sum_{j=1}^{k_i} d_i^j$ . This data is stored in robot  $R_i$ 's memory, of maximum capacity  $MaxCapacity_i$ . We consider that the sink has no memory limit, i.e.  $MaxCapacity_0 = \infty$ . For data exchange, two robots  $R_i$  and  $R_j$  have periodical meetings (*rendez-vous* points) starting at time  $T_{ij}$  when they first meet, with a period of  $\pi_{ij}$  time steps.

In our approach, we assume that the incoming data model is *regular* and *known* and that there is a single base station (called *sink*). The data is continuously collected by the robots after patrolling their cycle and has to be transmitted to the sink. We also assume that it takes one unit of time for a robot to go from one target to the next one and that transferring data does not take time.

Figure 2.1 depicts a synthetically generated example, a much simplified version of the MRP problem with 4 robots and 6 targets, in which data is static (i.e. collected only once by a robot, at the beginning of the simulation), the robots' meetings are periodical and the solution is the minimum time required for transferring all collected data to the *sink* (denoted by  $R_0$ ). In Figure 2.1  $RV$  represents the meetings between the robots,  $T_{ij}$  is the time stamp when robots  $i$  and  $j$  first meet and  $\pi_{ij}$  is the meeting period for robots  $i$  and  $j$ . Assuming that the initial data for every robot is 10 and the maximum memory for all robots is 15, the minimum time in which all data can be transferred to the sink is 8. The right side table from Figure 2.1 depicts the decision making strategy for the agents, at each time moment, obtained using a brute force approach. Each line from the table represent a time moment and the columns from the second to the fifth indicate the target visited by the robots (including the sink) at a given time. If two robots visit the same target at a given time, there is a meeting between them and data can be transferred. The last three columns on a row (time) present the data transfer strategy between the robots at that moment, i.e. robot **From** transfers to robot **To** the quantity **Quant**.

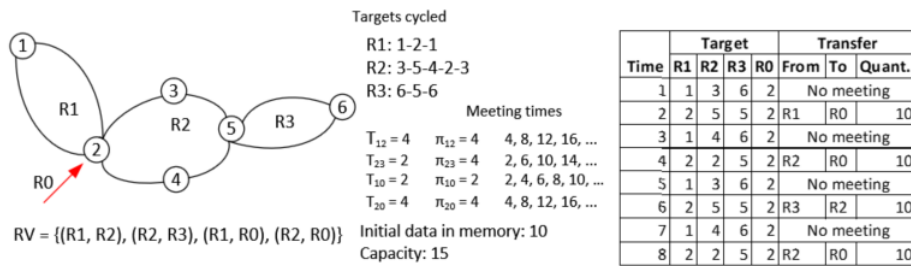


Figure 2.1: Synthetic example.

### 2.1.2 The proposed *DynFloR* approach

For modeling the dynamic data collection process from the multi-robot network previously defined in Section 2.1.1, we introduce a dynamic flow network based model which captures both the temporality of the patrolling process, as well as the dynamicity of the data collection and transfer.

#### The proposed dynamic network model

Our proposed model adapts the dynamic network flow approach introduced in [Kot03] by taking into account the specific requirements of multi-robot network patrolling. The flow network graph  $G$  is constructed from moment 0 up to a moment of time  $Time$ . For each time step, the graph captures the state of each robot and the sink. Each of the states is represented by a node in the graph, which includes information such as robot decisions (data currently collected, data currently transferred, robot met for data exchange) or the amount of data stored.

There are  $(n + 1) \cdot (Time + 1) + 1$  **nodes** in the dynamic flow graph. For each robot  $R_i$  including the sink ( $\forall i, 0 \leq i \leq n$ ) there are  $Time+1$  nodes, denoted by  $i_t$  ( $\forall t, 0 \leq t \leq Time$ ). The node  $i_t$  corresponds to robot  $R_i$  at time  $t$ . The *sink* is viewed, in the proposed model, as robot  $R_0$ . Besides the nodes corresponding to the robots, there is an additional *source* node  $s$  connected to all robots (except the sink) at time step 0.

The directed **edges** (connections) between the flow network's nodes and their capacities are set as follows. There is a connection between the source node  $s$  and node  $i_0$  ( $\forall i, 1 \leq i \leq n$ ) and its capacity is set to the initial data  $Q_i$  held by the robot  $R_i$  at the beginning of the simulation. For each time moment  $t$  when two robots  $R_i$  and  $R_j$  meet, two edges are added: one from  $i_t$  to  $j_t$  with capacity  $MaxCapacity_j$  (i.e., the maximum memory of  $R_j$ ) and one from  $j_t$  to  $i_t$  with capacity  $MaxCapacity_i$  (i.e., the maximum memory of  $R_i$ ). For each robot  $R_i$  (including the sink) an edge is created between  $i_t$  and  $i_{t+1}$ ,  $\forall t, 0 \leq t \leq Time-1$  (the edge has the capacity  $MaxCapacity_i$ ). For each time moment  $t$  when new data is collected by a robot  $R_i$ , an edge connecting the source node  $s$  and the node  $i_t$  is added and its capacity is set to  $D_i$ .

We mention that the currently proposed model contains nodes corresponding to an entire cycle patrolled by a robot (without detailing the targets patrolled during each cycle). We also assume that a robot collects the data after it patrolled an entire cycle, this is why we add one edge with capacity  $D_i$  after  $k_i$  time units, instead of adding one edge with capacity  $d_i$  to every node. The previous assumptions do not reduce the generality of *DynFloR*, as we can easily extend the model by adding new data at every time step to every robot and the *DynFloR* algorithm remains the same. In this case, new data will be collected by a robot in a node corresponding to a target

and not to an entire cycle (as in the current model).

### *DynFloR* algorithm

After the flow network graph  $G$  is built as described in Section 2.1.2, a maximum flow [Kot03] from the source  $s$  to the network sink (a node  $0_t, t \leq Time$ ) will give the total amount of data that can be transferred to the base station in  $t$  time steps. Our proposal *DynFloR* outputs the decision making strategy for each robot  $R_i$ , at each rendez-vous time, in order to increase the quantity of data received by the sink while respecting the memory constraint for each robot.

The pseudocode for constructing the flow graph  $G$  is presented in Algorithm 2 and the *DynFloR* algorithm is given in Algorithm 1. For determining the maximum flow in a graph  $G$ , the Preflow-Push algorithm [Net] denoted by  $getFlow(G)$  is used. We also denoted by  $addNode$  and  $addEdge$  the operations for creating a node and an edge with a certain capacity, respectively.

---

#### **Algorithm 1** Algorithm *DynFloR*

---

**Algorithm** *DynFloR* is:  
 // determines the optimal policy for the robots' data delivery  
**Require:**  $n, k$  - the number of robots and a list containing the robots' cycle lengths  
 $D, Q, MaxCapacity$  - lists containing the amount of data collected by each robot, its initial data and its maximum memory  
 $RV, Time$  - the list containing the pairs of robots that meet and the final moment of time considered  
 $T, \pi$  - the initial meeting time for each pair of robots from  $RV$  and their meeting period  
**Ensure:** returns the data delivery strategy for the robots from  $RV$  (the path that gives the maximum flow in  $G$ )  
 // read the input data  
 // construct the flow network graph  $G$   
 $G \leftarrow CreateGraph(n, k, D, Q, MaxCapacity, RV, Time, T, \pi)$   
 // the data delivery policy  $Policy$  is computed from the path that corresponds to the maximum flow  $MaxFlow$  in the graph  $G$   
 $(MaxFlow, Policy) \leftarrow getFlow(G)$   
**EndAlgorithm**

---

We note that the *objective function* which has to be maximized in our approach is

$$f(T) = \frac{\text{data to sink until time } T}{T}$$

Alternatively, we envisage the minimization of

$$g(T) = \frac{\text{input data} - \text{data to sink until time } T}{T}$$

**Algorithm 2** Function *CreateGraph*


---

**function** *CreateGraph*( $n, k, D, Q, \text{MaxCapacity}, RV, \text{Time}, T, \pi$ )

// Create the flow network graph corresponding to the MRP problem

**Require:**  $n, k, D, Q, \text{MaxCapacity}, RV, \text{Time}, T, \pi$  the input parameters for the *DynFloR* algorithm

**Ensure:** returns the graph  $G$  constructed as described in Section 2.1.2

// Create the nodes, edges and their capacities

 $\text{addNode}(G, s)$  // a node corresponding to the source

**for**  $i \leftarrow 0, n$  **do**
**for**  $t \leftarrow 0, \text{Time}$  **do**
 $\text{addNode}(G, i_t)$  // a node corresponding to robot  $R_i$  at time  $t$ 
**if**  $t \neq 0$  **then**
**if**  $i = 0$  **then**
 $\text{addEdge}(G, i_{t-1}, i_t, \infty)$  // add an edge between  $0_{t-1}$  and  $0_t$  with capacity  $\infty$ 
**else**
 $\text{addEdge}(G, i_{t-1}, i_t, \text{MaxCapacity}_i)$ 
**end if**
**end if**
**end for**
**if**  $i \neq 0$  **then**
 $\text{addEdge}(G, s, i_0, Q_i)$ 
**end if**
**end for**
**for each pair**  $(i, j) \in RV$  **do**
**for**  $t \leftarrow T_{ij}, \text{Time}, \pi_{ij}$  **do**
 $\text{addEdge}(G, i_t, j_t, \text{MaxCapacity}_i)$ 
**if**  $j \neq 0$  **then** //if  $j$  is not the sink

 $\text{addEdge}(G, j_t, i_t, \text{MaxCapacity}_j)$ 
**end if**
**end for**
**end for**

// Add additional edges to model the data dynamically collected by the robots

**for**  $i \leftarrow 1, n$  **do**
**for**  $t \leftarrow 1, \text{Time}$  **do**
**if**  $t \bmod k_i = 0$  // if an entire cycle was patrolled by robot  $R_i$  **then**

 // add an edge from the source to  $i_t$  for modelling that new amount of data  $D_i$  arrives in the node

 $\text{addEdge}(G, s, i_t, D_i)$ 
**end if**
**end for**
**end for**

 // Return the graph  $G$ 

 return  $G$ 
**end function**


---

A brief analysis of the time complexity of *DynFloR* is given below. The time complexity of the *CreateGraph* subalgorithm for constructing the dynamic network  $G$  is  $\mathcal{O}(|RV| \cdot \text{Time})$ . Therefore, the overall complexity of *DynFloR*, including computing the maximum flow, is  $\mathcal{O}(\mathcal{F}(n \cdot \text{Time}, |RV| \cdot \text{Time}))$ , where  $\mathcal{F}(V, E)$  is the time complexity of a flow algorithm applied on a graph with  $V$  nodes and  $E$  edges (in our graph we have  $n \cdot \text{Time}$  nodes and  $|RV| \cdot \text{Time}$  directed edges). For example, for the highest label preflow-push algorithm [CLRS09], we have  $\mathcal{F} = \mathcal{O}\left((n \cdot \text{Time})^2 \cdot \sqrt{|RV| \cdot \text{Time}}\right)$ .

### 2.1.3 Example

For exemplifying how *DynFloR* works, let us consider a very simple example of MRP illustrated in Figure 2.2. It consists of only two robots, denoted by R1 and R2. The sink is marked as R0. Figure 2.2 also illustrates the first meeting time  $T_{ij}$  between robots  $i$  and  $j$  and the meeting period  $\pi_{ij}$  for robots  $i$  and  $j$ . We are also assuming that the data collected by a robot is 15, and the maximum memory  $\text{MaxCapacity}_i$  for all robots is 55.

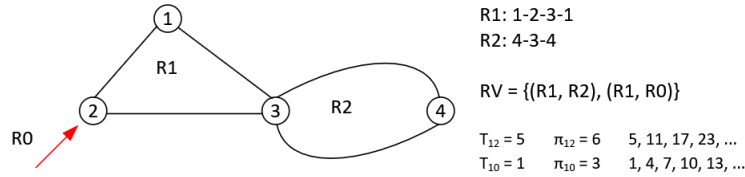


Figure 2.2: Simple example.

For the MRP example from Figure 2.2, the associated flow network is presented in Figure 2.3. In our example, the moment of time  $T$  until the network is presented is 20. The amount of data which is collected by the robots is marked with green. For a better readability of Figure 2.3, we did not connect the green arrows to the source (as described in Section 2.1.2), but in the implementation they are connected. The decision making strategy for the robots, which is the output of the *DynFloR* algorithm proposed in Section 2.1.2, is marked with red. If an arc between two nodes  $i_t$  and  $j_t$  is labelled with a value  $v$ , it means that at time  $t$  robot  $R_i$  transfers to robot  $R_j$  the amount  $v$  of data. If the arc between  $i_t$  and  $i_{t+1}$  is labeled with  $v$ , it suggests that robot  $R_i$  stores in its memory the amount of data  $v$  for one unit of time (from  $t$  to  $t + 1$ ).

From Figure 2.3 we can observe that 225 units of data were transferred to the sink (robot R0) by  $t = 20$ . We can also observe that there are 3 green arrows (robot R2 at times 14, 18 and 20) that do not have incoming data. This data is not lost, it is stored in the memory of the robots, but since until the time limit we have set for the

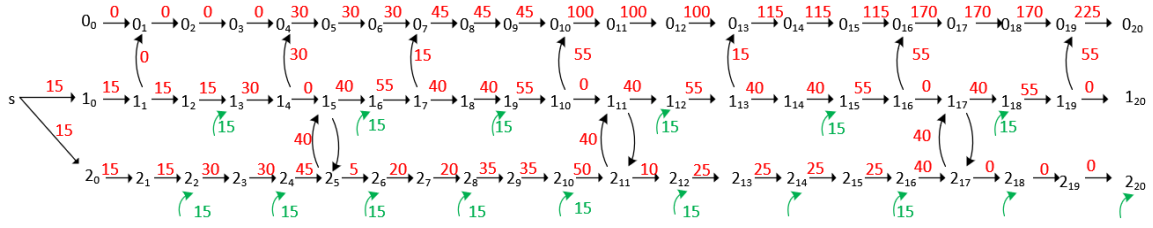


Figure 2.3: The dynamic network for the example from Figure 2.2. The decision making strategy for the robots is marked with red.

simulation ( $t = 20$ ) it cannot arrive to the sink, the flow algorithm does not consider it. But if we extended the time limit to include another meeting between robots R1 and R2, most of these data would arrive to the sink.

There is also the possibility to have lost data, a situation in which no matter how long we would extend the time limit, the data would never arrive to the sink. This is mainly due to the maximum memory capacity of the robots and the frequency of the collected data. For instance, if we considered the value 40 as the maximum capacity for robot R2 (and left the other settings used for Figure 2.3 the same), by time 5, when R2 first meets R1 and has the chance to transfer data, it should have accumulated 45 units of data, which is impossible. In our model, if the robot's memory is full, the new data will not be collected (instead of overwriting existing data with the new one).

In order to determine which data is lost and which data is in the memory of a robot (and consequently will arrive to the sink in the future), we used the following method: after measuring the flow for time  $Time$ , as presented in Algorithm 1, we build a new, extended graph for time  $2 \cdot Time$ . In this extended graph we added the extra edges denoting new data coming from the source to the robots only until time step  $Time$ , so second half of the graph has no extra data coming. We measure the flow in this extended graph and the difference between the flows denotes the data that is in the memory of the robots at time  $Time$  and will arrive to the sink later. We also know the total data that should enter the graph (the sum of the capacities of the outgoing arcs of the source node) and the difference between this data and the flow of the extended graph is the quantity of the data that is lost. The robot that loses data is determined by the flow algorithm, and in the current implementation we only try to optimize the total quantity of data that arrives to the sink, and do not try to balance the lost quantities between robots.



## 2.2 Results and discussion

In this section we present our current results obtained using the algorithm presented previously. The network flow part of our DynFloR algorithm was implemented using the Python NetworkX [HSS08] graph library.

We performed incipient simulations on randomly generated inputs of 20 robots with the same *MaxCapacity* for each robot. Figures 2.4 and 2.5 show the evolution of the  $f(T)$  and  $g(T)$  objective functions for two different values of *MaxCapacity*. We observe that both functions converge to an equilibrium state and this convergence is independent of the memory capacity. Multiple random experiments confirm this.

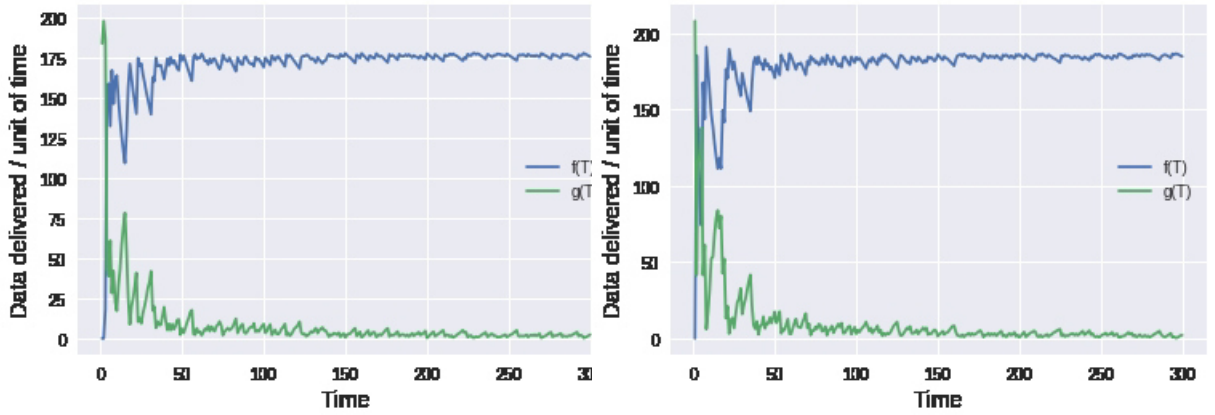


Figure 2.4: Results for 20 robots and Figure 2.5: Results for 20 robots and *MaxCapacity* = 1000. *MaxCapacity* = 800.

In order to better visualize the relation between data that enters the system, data that is currently in the memory of robots and data that is lost, we have performed some experiments using the simple example from Figure 2.2, setting the quantity of initial data and new data that is collected after every cycle to 15. We have considered different values for *Time* and *MaxCapacity* (but considering the same capacity for both robots). The results are presented in Table 2.1.

The value in parentheses after *Time* represents the total quantity of data that enters the system until that time moment and which is divided into data that arrived to the sink, data that is currently in the memory of the robots, and data that is lost. We can see that as the *MaxCapacity* of the robots increases, so does the amount that arrives to the sink and consequently the quantity of lost data decreases. From Table 2.1 it looks like setting the *MaxCapacity* to 60 is enough to make sure that no data is lost, but in order to be certain of this, more experiments and maybe mathematical proof is necessary, because at time 20, there was no lost data for *MaxCapacity* 55 either, but later this changed. Data that is currently in the memory of the robots is always bounded by the total memory the robots have, in our case  $2 \cdot \text{MaxCapacity}$ .



Time	Maximum capacity	Data arrived at sink	Data in memory	Lost data	Time	Maximum capacity	Data arrived at sink	Data in memory	Lost data
20 (270)	40	180	40	50	40 (420)	40	360	40	125
	45	195	45	30		45	390	45	90
	50	210	45	15		50	420	50	55
	55	225	45	0		55	450	55	20
	60	240	30	0		60	480	45	0
60 (705)	40	525	70	185	80 (1020)	40	730	40	250
	45	570	75	135		45	795	45	180
	50	615	80	85		50	860	45	115
	55	600	85	35		55	925	45	50
	60	705	75	0		60	990	30	0

Table 2.1: Quantity of data in memory, at the sink and lost, for different maximum capacities and time limits for the example from Figure 2.2. Initial data and new data received after every completed cycle is 15.

From Table 2.1 we can also observe that for a given *MaxCapacity* the quantity of lost data is not double if we double *Time*. For example, for *MaxCapacity* 50, we have 15 units of lost data after *Time* 20, but if we double *Time*, the quantity of lost data becomes 55. We performed some experiments in order to better understand how the quantity of lost data changes if time increases, for a fixed *MaxCapacity*. In Table 2.2 we included two scenarios: the left part of the table was computed for *MaxCapacity* 50, while the right part was computed for *MaxCapacity* 40. For both scenarios, we computed also the rate of Lost data / Time (in the 4th and 8th column).

Maximum capacity	Time	Lost Data	Lost data / Time	Maximum capacity	Time	Lost Data	Lost data / Time
50	10	5	0.5	40	10	25	2.5
	20	15	0.75		20	50	2.5
	50	65	1.3		50	150	3
	100	155	1.55		100	325	3.25
	200	315	1.575		200	650	3.25
	500	815	1.63		500	1650	3.3
	1000	1655	1.655		1000	3325	3.325
	5000	8315	1.663		5000	16650	3.33
	10000	16655	1.6655		10000	33325	3.3325
	50000	83315	1.6663		50000	166650	3.333
	100000	166655	1.66655		100000	333325	3.33325

Table 2.2: Progression of lost data as time increases for different *MaxCapacity* values

As expected, Table 2.2 reveals that the quantity of lost data per time significantly decreases as the maximum memory capacity of the robots increases. Additionally, as illustrated in Figure 2.6, it seems that the lost data per time converges to an equilibrium state, independent of the maximum capacity of the robots. This can be observed more clearly by running the proposed flow algorithm for more time steps,

thus allowing all the data to arrive at the sink. As we have previously discussed, the data which does not arrive to the sink until a certain time limit is not necessarily lost, as it may arrive to the destination if we extend the time limit. Still, there is also the possibility to have lost data, a situation in which the data would never arrive to the sink no matter how long we would extend the time limit. Further experiments and theoretical analyses will be performed in this direction.

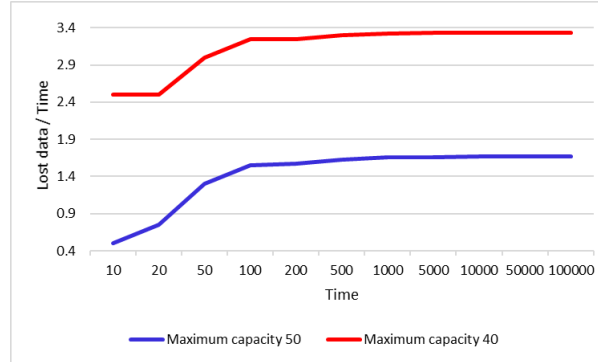


Figure 2.6: Variation of lost data per time for the example from Figure 2.2.

### 2.2.1 Comparison to related work

The literature contains various approaches related to multi-robot patrolling described in Section 1.3, but we have not found approaches for optimizing data delivery in MRP.

Our work differs from the approaches described in Section 1.3, as it has a different target than the previously described related work. It approaches the problem of data delivery optimization in a deterministic MRP setting using a dynamic flow based approach. As far as we know, approaches similar to *DynFloR* do not exist in the MRP/DMRP literature. The most similar approach to ours is the one of Liemhetcharat et al. [LYTL15]. The problem approached in the current chapter differs from the one of Liemhetcharat et al. [LYTL15], even if both chapters are dealing with the data delivery problem. The perspectives are different, since the chapter [LYTL15] considers data delivery from a central location to several local targets, while in our chapter the perspective is the opposite one.

A major advantage of *DynFloR* is its polynomial time complexity, compared to the classical brute-force exponential solutions [RBBA18]. In addition, *DynFloR* works well for dynamic data, i.e. data which continuously appear on each robot cycle. Another advantage of *DynFloR* is that it finds the global optima, unlike the metaheuristic approaches (such as *genetic algorithms* - GAs) [PR11]) which may provide a local optima. Besides, another difficulty when modelling the MRP using GAs relies on the chromosomes' modelling and on incorporating the continuous data col-

lection into the model. An additional benefit of *DynFloR* is that it can be adapted for the dynamic case of the MRP, under the same assumptions currently considered. When new targets are discovered during the dynamic patrolling problem and they are allocated to new patrolling robots, new nodes are simply added to the flow network graph  $G$  and *DynFloR* is executed again. An adaptive version of *DynFloR* may be further investigated, with the goal of adapting the solution provided by *DynFloR* instead of running it from scratch when new cycles to be patrolled are discovered.

One of the current limitations of *DynFloR* is given by the assumption regarding the periodicity of the robots' meetings. In the general case, in real DMRP scenarios, instead of having initial meeting times and meeting periods, there is an uncertainty of the meetings as the robots' speed is not constant. For offering a solution to the data delivery optimization for the general case, *reinforcement learning* would be further investigated for adapting the robots' decision making policy to the uncertainty of the world.

## 2.3 Conclusions and future work

In this chapter we have investigated the data delivery optimization in a simplified setting for the *multi-robot patrolling* problem in which the environment is deterministic. We introduced accordingly a centralized and offline approach *DynFloR* based on flows in dynamic networks. The goal of *DynFloR* is to determine the robots' policy for maximizing the quantity of data delivered to a base station in a given time, assuming that the robots are continuously collecting data during the patrolling. The more general aim of the research initiated in this chapter is to explore the general case of the problem of optimizing the data delivery in DMRP when the environment is non-deterministic and uncertain (i.e. there are communication failures).

Further work will extend the experimental evaluation and theoretical analysis of *DynFloR* in order to better assess its performance. We also aim to decentralize the decision making process and to incorporate uncertainty in the communication between robots. An extension of *DynFloR* to an online approach and the incoming data model to an irregular one will be subjects to future developments. In order to achieve these goals, a *reinforcement learning* perspective will be considered.

## Chapter 3

# A Reinforcement Learning Approach for Data Delivery Optimization in Multi-Robot Network Patrolling

This chapter aims to illustrate the feasibility of a *reinforcement learning* approach to optimizing the data delivery in MRP. The *DynFloR* method showcased in *Chapter 2* lacks one key asset in order to be fully reliable, that being *learning* how to react in uncertain conditions and dynamic environments.

When thinking about the nature of learning, perhaps several examples of instances come up in our minds when we learn something simply by interacting with an environment, and receiving a positive or negative reward when making a decision. One issue encountered in *RL* modelling of complex environments is the exponential pool of states in which the environment can be found, and the large number of actions that the robots can perform at any given time.

The study conducted in this chapter consists in a new approach, based on *Q-learning*, under the assumptions defined in *Chapter 2* (i.e. the environment is deterministic, the network of robots is centralized and offline, and the robots have periodic meetings). Experiments and simulations carried out on several synthetic and randomly generated examples prove that the *RL* approach for the data delivery optimization in MRP is feasible.

The rest of the chapter is structured as follows. The *reinforcement learning* approach for data delivery optimization in a deterministic setting of the MRP is introduced in Section 3.1. Section 3.2 is concerned with the advantages and limitations of the proposed model, while Section 3.3 contains the conclusions of the chapter and possible future developments of this work.

### 3.1 The proposed RL methodology

This section introduces a centralized and offline *RL* approach for data delivery optimization in a deterministic setting of the MRP, being applicable in a deterministic setting of DMRP, with minimal improvements.

#### 3.1.1 The DronemRL model

The problem approached in this chapter is the same problem defined in *Chapter 2*, so we will focus solely on the *reinforcement learning model*.

##### The action model

For this specific problem an action was defined to be a list  $X = [x_1, x_2, \dots, x_k]$ , where  $k = \frac{n(n-1)}{2}$ . There are  $\frac{n(n-1)}{2}$  elements in our list, because we have  $\binom{n}{2}$  ordered pairs of robots  $(R_i, R_j)$  with  $i < j$ . That being said an element at any given position  $p, 1 \leq p \leq k$  in our action list describes the quantity of data exchanged by robots  $(R_i, R_j), i < j$ , this quantity is bounded by robot's *MaxCapacity* meaning that there cannot be transfers between  $(R_j, R_i)$  if  $j > i$ , so in order to address this problem the domain of values for  $x_i \in X$  is  $\{0, 1, \dots, 2 \cdot \text{MaxCapacity}\}$ . Therefore  $x_i \in X$  has the following meaning:

$$x_i = \begin{cases} \text{transfer } x_i \text{ data from } R_i \rightarrow R_j, & 0 \leq x \leq \text{MaxCap} \\ \text{transfer } x_i - \text{MaxCap} \text{ data from } R_j \rightarrow R_i, & \text{MaxCap} < x \leq 2 \cdot \text{MaxCap} \end{cases}$$

Every action of the environment can be thought to be an element of the following cartesian product:

$$A = \bigtimes_{i=1}^{\frac{n(n-1)}{2}} A_i$$

where  $A_i = \{0, 1, 2, \dots, 2 \cdot \text{MaxCapacity}\}$  meaning that there are  $(2 \cdot \text{Maxcapacity})^{\frac{n(n-1)}{2}}$  actions in total. Thus it can already be seen that the action space is extremely large. For example if we have 10 robots and  $\text{MaxCapacity} = 10$ , we would have  $20^{45}$  actions. This problem is addressed in [DAEvH<sup>+</sup>15], by having an actor-critic model, where the actor applies actions and the critic learns to generate actions in some proximity of the current action using some defined mathematical norm. Even though this approach solves the problem of very large action spaces, there is still the problem that at any given time there is a small number of valid actions to choose from and most of the time the critic will generate invalid values, hence determining the model

to apply only invalid actions.

In order to address the issue of large state and action spaces the classic Q-learning algorithm, from Section 1.2.1 was modified, to initialize and update the Q-table in a dynamic way (i.e. *states* are added in the Q-table only when they are encountered, not at the beginning of the algorithm) and variation of the Q-learning algorithm will be called *DronemRL*.

### The state model

For the state model of the environment, it is enough to know the quantity of data each robot holds, the current time step and the position of each robot. Therefore, a state is represented as a list  $S = [s_1, s_2, \dots, s_{2n+1}]$ , where elements  $[s_1, \dots, s_n]$  represent the quantity of data of each robot,  $s_{n+1}$  is the current time step and  $[s_{n+2}, \dots, s_{2n+1}]$  represent the current position of each robot.

### The reward function

After trying several types of reward functions, the one which proved to be the best in practice is defined below:

- If the action is invalid (i.e.  $R_j$  cannot receive the data transferred by  $R_i$  or  $R_i$  does not have enough data in its memory), or there is an invalid meeting return a penalty reward of  $invalidActionReward = -10^7$ .
- If the chosen action is to do nothing when it is not the case (i.e. the robots can exchange data) return  $\frac{2}{3} \cdot invalidActionReward$ .
- Else add  $dataTrasferred$  to the reward value for each transfer to  $R_0$  (the sink) and 1 for any other valid transfer.

The reward function is defined in such a way that if our model chooses an invalid action, it receives a large negative reward in order to adjust the  $Q(s, a)$  value for that action in that particular state, thus next time when our model is in state  $s$  this invalid action won't be chosen. Also the reward function encourages the model to select actions which facilitate transfer of large amount of data to our *sink*. In *RL* the reward function is the core element of *learning* because it must be defined in a way that it emphasizes the environment characteristics. The best known reward function [SB98] is the one which gives  $-1$  reward for every action and  $0$  for terminal actions, it is guaranteed that the model will learn to choose the best actions in every state, but the drawback is that the learning time required for such a reward function is tremendously large, therefor impractical for many complex environments such as ours.

**DronemRL algorithm**

After constructing the environment having the properties stated in Section 3.1.1 we train the reinforcement learning model for a number of *episodes*. For every episode of the training process we apply the update rule defined in Section 1.2.1, based on *Bellman's equation*. One important aspect to take into consideration about this algorithm is that, we do not initialize the entire Q-table (i.e. a matrix storing every possible (*state, action*) pairs), instead we initialize the Q-table to be empty and only add new states and actions dynamically when we encounter them.

The pseudocode for the dynamic Q-table update is presented in Algorithm 3 while the DronemRL algorithm is given in Algorithm 4.

---

**Algorithm 3** Function *UpdateQTable*

---

```

function UpdateQTable(Q, state, action, reward, newState,  $\alpha, \gamma$ )
Ensure: Dynamically updates the Q-table based on Bellman's equation
    if newState  $\notin$  Q then
        //if the new state is not in the Q-table add it
        addStateInQTable(newState)
    end if
    //get the maximum Q-value of the new state
    maxQFuture  $\leftarrow \max(Q(newState))$ 
    //get the current Q-value
    currentQValue  $\leftarrow Q(state, action)$ 
    //update the Q-table
     $Q(state, action) \leftarrow Q(state, action) + \alpha \cdot (reward + \gamma \cdot maxQFuture)$ 
end function
```

---

---

**Algorithm 4** Algorithm *DronemRL*

---

**Algorithm** DronemRL is:

//determines the optimal policy for the robots data delivery

**Require:**  $env, nEpisodes, \epsilon, decay, maxNumOfSteps, \alpha, \gamma$  - the input parameters for the *DronemRL* algorithm

**Ensure:** returns the Q-table containing  $Q(s,a)$  values as described in Section 1.2.1

//initialize empty Q-table

$Q \leftarrow emptyQTable()$

//applies the Q-learning algorithm

**for**  $episode \leftarrow 0, nEpisodes$  **do**

    //sets the reward for current episode to 0

$episodeReward \leftarrow 0$

    //the current episode is not done

$done \leftarrow false$

    //sets the environment to its initial state

$state \leftarrow reset(env)$

**repeat**

$action \leftarrow selectAction(\epsilon, state)$

        //applies action to the environment

$actionResult \leftarrow step(action)$

        //gets the new state, the reward of applying the action and if the new state

is terminal

$newState \leftarrow getState(actionResult)$

$reward \leftarrow getReward(actionResult)$

$done \leftarrow getDone(actionResult)$

$episodeReward \leftarrow episodeReward + reward$

        //if the episode is not done (i.e. we are not in a terminal state, and we have

not reached the maximum number of steps per episode

**if**  $done \neq true$  **then**

            //updates the Q-table

$UpdateQTable(Q, state, action, reward, newState, \alpha, \gamma)$

**else**

$time \leftarrow getTime(newState)$

**if**  $time = maxNumOfSteps$  **then**

                //if we reached the end of the episode because we've used the max-

imum number of steps, penalize the action

$Q(state, action) \leftarrow -1$

**else**

$Q(state, action) \leftarrow reward$

**end if**

**end if**

$state \leftarrow newState$

**until**  $done = true$

**end for**

---



## 3.2 Results and discussion

In this section the results obtained using the proposed algorithm are showcased. All operations involving the manipulation of Q-tables were performed using the Python Numpy [vCV11] numerical computation library.

The algorithm was tested against several randomly generated environments having 3 robots with the same  $MaxCapacity = 10$  for each robot. Figure 3.1 and Figure 3.2 show the evolution of the minimum and maximum reward per 50 episodes for a total number of 50000 episodes.

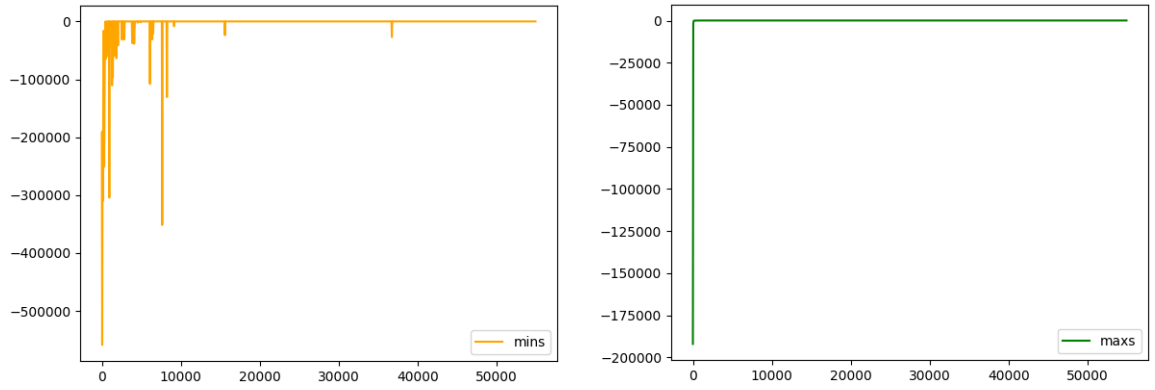


Figure 3.1: Results for 3 robots and  $InitialMemory = 4$       Figure 3.2: Results for 3 robots and  $InitialMemory = 4$

It can be seen from the figures above that the algorithm reaches convergence well before 50000 episodes. The Q-learning model quickly identifies the optimal data transfer policy between robots. Given the fact that the initial memory of the robots is 4 and the  $MaxCapacity$  for all robots is 10, one robot  $R_i$  can accommodate all the memory initially contained in the memory of  $R_j$ , explaining the rapid convergence of our algorithm on this particular case.

However, tests were ran on more complex environments, Figure 3.3 depicts the minimum, maximum and the average reward per 50 episodes on an environment having 3 robots, with  $MaxCapacity = 9$ , but having  $initialMemory = 7$ . In this case the transfer policy is more complex because when two robots meet for the first time they cannot exchange all their data, as they need to figure out that they can only send 2 units of data between them.

It can be observed that the algorithm converged slowly on this more complex environment and the main factor which contributes to the convergence is the ratio between  $MaxCapacity$  and  $InitialMemory$ . Therefore, the following question arises: Given  $P = \frac{InitialMemory}{MaxCapacity}$  will the algorithm eventually converge? In order to provide an answer, several tests were made with different ratios and results can be seen in Figure 3.4

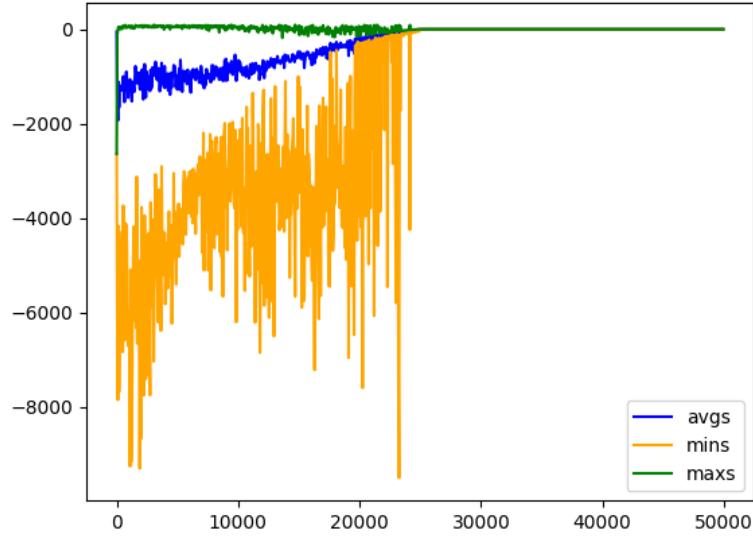


Figure 3.3: Results for 3 robots and  $MaxCapacity = 9$ ,  $InitialMemory = 7$

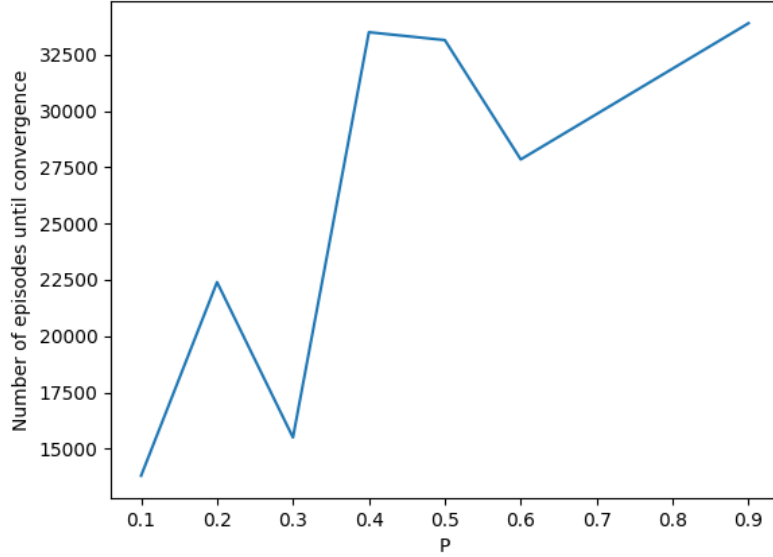


Figure 3.4: Convergence graph with respect to  $P$

As expected, Figure 3.4 reveals that eventually our algorithm convergence independent of the ratio  $P = \frac{InitialMemory}{MaxCapacity}$  with linear dependence on on some intervals.

### 3.2.1 Comparison to related work

As it was stated in Section 2.2.1, no approaches for optimizing data delivery in MRP were found, but techniques for similar problems will be discussed.

Because classical RL algorithms are limited due to the exponential number of states and actions, we will focus on approaches oriented on *DeepQ-learning*. In Deep Q-learning instead of having a Q-table in memory for storing the Q-values for every pair  $(s, a)$  where  $s \in S$  - the set of total States and  $a \in A$  - set of total actions, we have a neural network which will *predict* our QValues as seen in Figure 3.1.

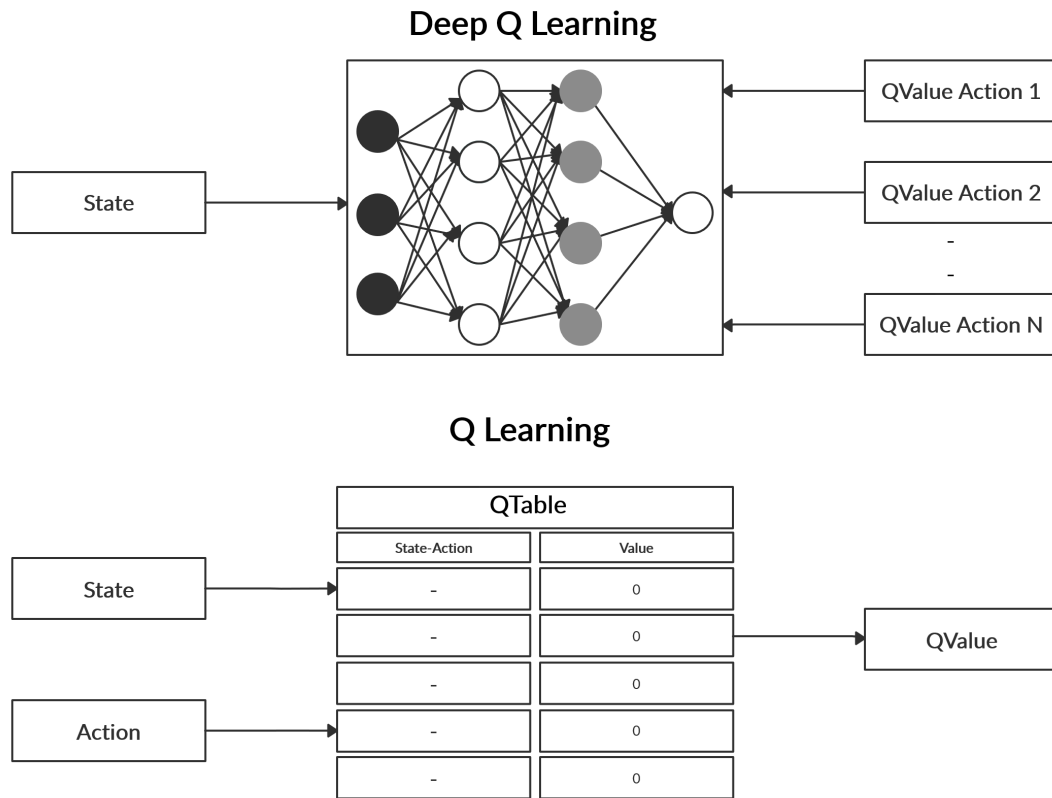


Figure 3.5: Classic Q-learning vs Deep Q-learning

One widely used Deep Q-learning (DQN) architecture is the one presented in [DAEvH<sup>+</sup>15], also called Deep Deterministic Policy Gradient (DDPG), based on the actor critic model discussed earlier. This architecture is based on two Neural Networks, one is the actor and the other one is the critic. These networks compute action predictions for the current state and generate a temporal-difference error signal at each step. The input of the actor network is the current state, while the output is a single real value representing an action chosen from a *continuous* or large discrete action space. The output of the critic is simply the Q-value of the current state and action supplied by the actor. The deterministic policy gradient theorem [SLH<sup>+</sup>14] provides the update rule for the weights of the actor network while the critic network is updated from the gradients obtained from the temporal difference error signal. While this approach gives good results it is often hard to find a suitable modelling for the MRP problem because of the high complexity of the environment.

Another way of tackling the training in environments with large number of actions and states is using Multi Agent approaches (every agent develops its own policy and learns how to cooperate with others) even though algorithms using this approach are notoriously unstable to train. However the OpenAI research team managed to come up with a feasible and robust approach [LWT<sup>+</sup>17] by extending

the actor-critic model. In the Multi-Agent DDPG (MaDDPG) OpenAI adapted the above DDPG algorithm for multi agent environments. It uses a decentralised actor, centralised critic training but during execution agents select actions relying only on their own state. This helps in easing the training as the environment becomes more stationary for each agent and the number of states and actions for each agent decreases exponentially. A diagram of the algorithm can be found in Figure 3.6 [LWT<sup>+</sup>17, Figure-1]

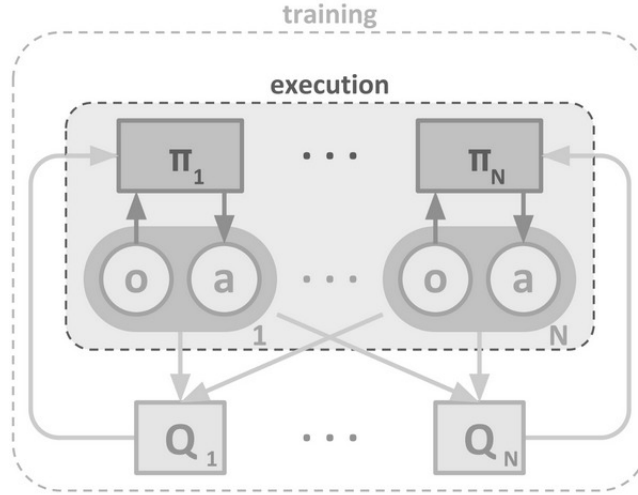


Figure 3.6: Multi-Agent DDPG.

### 3.3 Future work

In this chapter we have investigated the *reinforcement learning* approach for the data delivery optimization problem in a simplified setting for the *multi-robot patrolling* problem in which the environment is deterministic. Also we have introduced our centralized and offline approach whose goal is to maximize the quantity of data which arrives to our station (*sink*). The aim of the above study was to prove that a *reinforcement learning approach* is feasible.

Future work will include more rigorous testing and increase in performance. We also aim to transform the centralized system into a multi-agent system by using the approach defined in [ZYB19], this would reduce the state and action space exponentially, while increasing the performance of the system. Another enhancement which can be added to our approach is integrating the DDPG algorithm discussed earlier, which would certainly increase the performance, but our ultimate goal is to use the MaDDPG approach as it is considered to be state of the art (SOTA) when it comes to multi agent reinforcement learning. Using the MaDDPG would be suitable for the Data Delivery Optimization problem in the MRP, having each robot learn its own

policy while learning to cooperate with others in training, with minimal state and action spaces, further improving the work done by *DynFloR*.

## Chapter 4

# A Software Solution for the Patrolling Problem

This chapter proposes a web application called *Dronem Web*, based on the *DronemRL* algorithm. The purpose of this platform is to facilitate a way to create, train and share results on multi-robot patrolling problems in the research community.

This application represents an element of originality, as to the best of our knowledge there is no system currently available for manipulating and analyzing MRP environments. One of the strongest points of the proposed software solution is the possibility to fetch already trained cloud *reinforcement learning* Q-tables for different environments, in order to further use them in real applications.

Several popular and highly optimized frameworks were used for building this application on both *backend* and *frontend*, such as OpenAI for creating and managing the MRP environments, Django and Django Rest Framework for managing user and environment data, and ReactJS for creating the application layout and handling user interactions with the application. All this frameworks and libraries will be further detailed in this chapter.

The remainder of chapter is structured as follows. In Section 4.1 we make an analysis of the proposed application from a software development point of view. Section 4.2 shows how the MRP environments are created using the *OpenAI Gym* [BCP<sup>+</sup>16] framework, while in Sections 4.3.1 and 4.3.5 the main *backend* and *frontend* technologies are discussed. Implementation details and design choices are highlighted in Section 4.3 and a user manual is proposed in Section 4.4. Towards the end of the chapter, some conclusions are drawn in the form of a brief discussion in Section 4.5, and finally, possible future enhancements are presented in Section 4.6.

## 4.1 Software development

Software development [IBM14] implies all the necessary activities regarding creation, testing and deployment of applications or other types of software. When developing an application a series of activities need to be taken into consideration in order to minimize the possibility of developing applications containing bugs, this series of activities is called a *software development process* [HT00].

The most important activities in Software development are the following:

- Identify the need for creating the application.
- Planning - discovering the requirements of the application and its functionalities.
- Designing - creating a high-level design of the application (main modules, packages, entities, what kind of data will be managed by the application)
- Implementation - writing the code of the application.
- Testing - ensuring that the application behaves as expected according to the requirements.
- Documenting - writing specification for modules and packages easing maintainability of the code and extending the application.
- Deployment and maintenance - releasing the application and correcting any error which may appear in the future.

The need for Software development comes as a natural consequence of the fact that the code is written by humans, making it unreliable, hence a methodology needs to be followed in order to ensure that major bugs do not appear, and minor bugs are identified quickly and fixed. One good example of a disaster which occurred due to a major bug is The Patriot Missile Failure [Dou00], where on 25<sup>th</sup> of February 1991, 28 American soldiers died because of a floating point arithmetic error.

### 4.1.1 Analysis and design

This subsection is focusing solely on the methodology defined in Section 4.1

#### The necessity of this application

While reading papers about different types of MRP problems, it was noticed that there is no application where, as a user, you can create your own environments

and try to apply different operations on them, therefore, an application capable of managing, analyzing MRP environments and sharing results is needed. Also due to the fact that MRP problems have high applicability in real life scenarios such as smart surveillance systems, this application can also serve as a tool to design such useful systems and test their capabilities.

## Planning

In the planning phase, also known as *collection of requirements*, we have gathered the main functional requirements of the application with the help of a use case diagram which can be observed in Figure 4.1

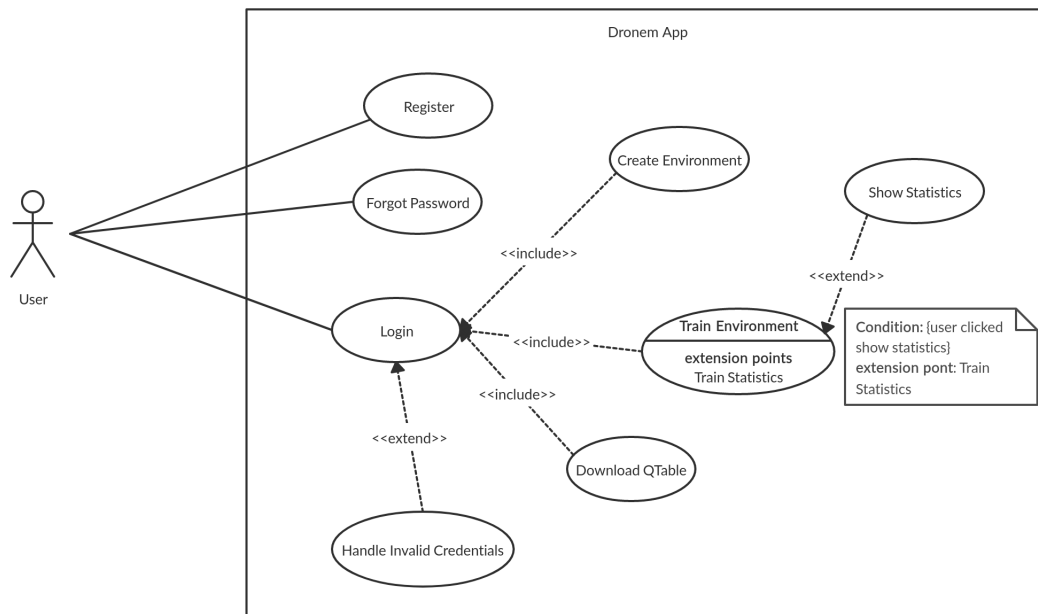


Figure 4.1: Use case diagram for Dronem web application

From the diagram above we can already identify the user as being the only actor of our application and the following use cases:

- Register - complete registration flow for becoming a user in the Dronem Web application.
- Login - Authentication functionality which provides access to the core features of the application.
- Handle Invalid Credentials - verification of user input at login.
- Create Environment - use case for creating and managing MRP environments.
- Download QTable - possibility to download trained Q-ables for different environments.





It is known that one of the biggest problems of software development is the defective communication between clients and developers [Kat17], as usually the actual people who write code are not directly interacting with the client. The conceptual model diagram is an important tool which tries to solve this problems, since it eases the communication between the developers and the client by highlighting the main entities of the application and the interactions between them. Of course, because diagram presented in Figure 4.2 is just an intermediate step in designing the application, it is not detailed enough in order to start writing code, thus a refinement process was needed in order to obtain the final class diagram which can be viewed in Figure 4.3



## 4.2 OpenAI

*OpenAI Gym* [BCP<sup>+</sup>16] is a *python* toolkit for reinforcement learning environments which exposes a common interface and an website where researchers can share and compare their results. Reinforcement learning assumes that there is an agent which is situated in an environment. Each step, the agent takes an action, and it receives an observation and reward from the environment. A RL algorithm seeks to maximize some measure of the agents total reward, as the agent interacts with the environment. In the RL literature, the environment is formalized as a partially observable Markov decision process [SB98]. OpenAI Gym focuses on the episodic setting of reinforcement learning, where the agents experience is broken down into a series of episodes. In each episode, the initial state of the agent is randomly sampled from a distribution, and the interaction proceeds until the environment reaches a terminal state. The goal in episodic reinforcement learning is to maximize the expectation of total reward per episode, and to achieve a high level of performance in as few episodes as possible.

### 4.2.1 The Dronem environment

The OpenAI Gym environment for our reinforcement learning model (*Dronem environment*) is constructed upon the Gym architecture [BCP<sup>+</sup>16], which is based on the following interface:

- *env.reset()* which resets the environment to its initial state;
- *env.step(action)* which applies an action to the environment and returns back the new *state* of the environment as an *observation*, a variable called *done* which tells if the episode should end and the *reward* for applying the given action;
- *env.render()* which renders the environment;

As we can see in Figure 4.4 the life cycle of a Gym Environment is highlighted, by first starting the environment when calling *env.reset()*, then the environment expects an action in order to call *env.step()*, after that we analyze the new state and reward to draw conclusions about our actions. After calling the *env.step()* function, if the *done* variable is true, we then stop the Dronem environment.

Following the interface defined in Section 4.2.1, it was easy to design and implement the architecture from Figure 4.5 in order to obtain a reinforcement learning environment following the specification of the problem defined in Chapter 2. We can observe that we are not limited in the number or type of our parameters and functions used in the environment, as long as we implement the functions in the *Gym* interface.

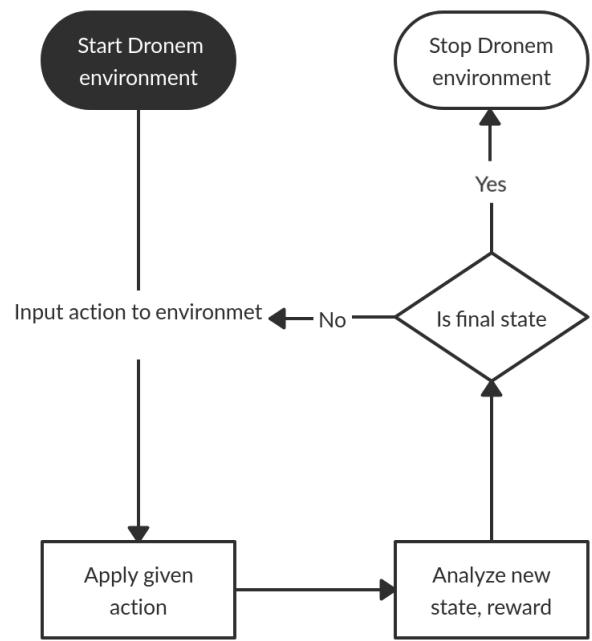


Figure 4.4: Dronem environment life cycle

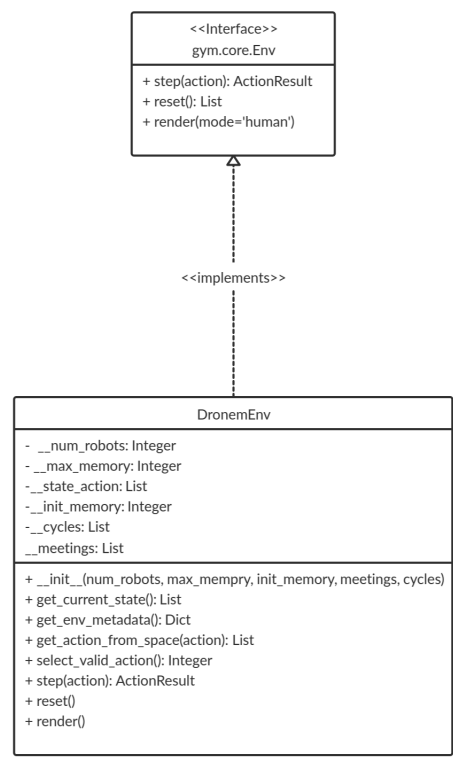


Figure 4.5: Dronem environment Gym architecture

### 4.2.2 Advantages and Disadvantages

In the past, one of the problems in the reinforcement learning community was the lack of benchmarking tools, often the researchers were forced to code their own

environments, and because no common interface was used, it was very hard for a research team to apply a reinforcement learning algorithm on environments created by others.

This problem was solved by OpenAI Gym by not only exposing a common interface between all reinforcement learning environments, but also by offering more interesting environments as benchmarks such as the Atari Games. Even more, Gym offers a standardized test bed for algorithms in academic publishing.

The only inconvenience exposed by OpenAI Gym is the fact that only episodic reinforcement learning environments can be used, therefore for some environments a series of constraints have to be applied, while in the same time some RL classes of algorithms may not be suitable for episodic environments.

## 4.3 Implementation

The implementation phase plays a major role in the life cycle of a software application, because big decisions are made in important aspects such as testing methodologies and development technologies. We will focus more on implementation aspects along this section.

### 4.3.1 Backend

Backend refers to any part of a software which is not accessible by the user and it is usually composed of a server which, together with an application acts as an operation manager and a database for storing application data. Backend technologies consist of a programming language, in our case *Python*, and usually the programming language which is going to be used for development is enhanced by various frameworks such as *Django*.

### 4.3.2 Django vs Flask

In recent years many web development frameworks for Python have been released, but none of them are nearly as popular as Django [Dja20] and Flask [Fla20]. Django is an well-established framework having a bigger community of active users as it can be seen in Figure 4.6[Mar19], while Flask is a lightweight web framework.

Unlike Flask, Django has a 'battery included' philosophy, meaning that all you need in order to fully develop a web application is already included in the framework. Django makes it easier to handle common project administration tasks, by providing a fully functional admin interface, in order for the same functionality to be achieved using Flask, more packages needs to be installed and configured.

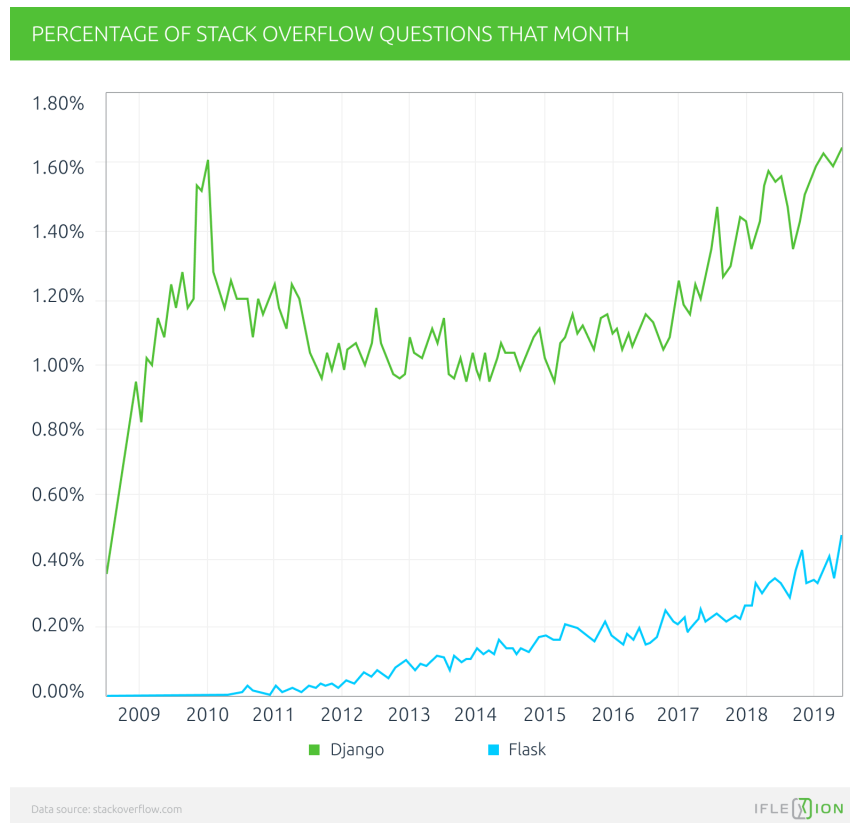


Figure 4.6: Number of StackOverflow questions for Django and Flask

Another advantage of Django over Flask is allowing the developers to take advantage of a robust object-relational mapping (ORM) mechanism. The developers can use the ORM provided by Django to work with many popular databases such as MySQL, Oracle, SQLite, PostgreSQL whereas in Flask there is no ORM system built in. Flask requires developers to work with databases and perform database operations through SQLAlchemy [Bay12].

Another argument which favors Django over Flask is the fact that developers can divide an application into multiple subapplications, hence it becomes easier for the developers to integrate new features into the application by splitting work and even reuse subapplications in other projects. Flask requires developers to create each project as a single application, however there is the option to add multiple models and views to the same application. When it comes to testing, both Django and Flask can use the Python integrated unittest framework, and both can be configured to use a secondary solution as necessary. However, the Django's integrated automated testing feature is a strong argument in its favor, especially when there is a need to create tests using dummy databases, as in Flask such a configuration is not trivial to accomplish.

As a conclusion, in order to build the Dronem Web application Django is the chosen framework because it offers many ways of creating scalable web applications

through its ease of use, robustness, testing versatility and very active community.

### 4.3.3 Django REST Framework

While Django alone is sufficient in order to develop a scalable web application, in order to enhance it with the latest capabilities of manipulating and serializing data, Django REST Framework (DRF) will be used.

Django REST Framework [Dja19] is a very powerful toolkit for building Web Application Programming Interfaces (APIs). One of the biggest reasons to use DRF is that it makes it very easy to serialize data. The Django ORM is already a powerful tool for handling database migrations and queries, but it is not that easy to convert entities stored in the database to more used formats such as JavaScript Object Notation (JSON). This is the problem that DRF solves, by easing the conversion of objects stored in the database in formats such as JSON with just a few lines of code.

### 4.3.4 Database

These days, there is a big debate regarding the database type you should use for your project. Depending on what kind of data your application manages, there are SQL databases and NoSQL databases. For choosing what type of database is suitable for an application, the question "Is the data relational or not?" must be answered. Because our data is highly relational, we chose to use MySQL [Ora20] as database, the following database schema is proposed for the Dronem Web application in Figure 4.7.

### 4.3.5 Frontend

As opposed to backend, the frontend side of a web application is concerned with the parts of the application which the user can see. The main tools used in frontend development are:

- Hyper Text Markup Language (HTML) which is the backbone of any website. Being a markup language it means that it indicates that text can be turned into images, tables or other representations.
- Cascading Style Sheets (CSS) which controls the presentation aspect of the site.
- JavaScript which is an event-based imperative programming language used to add dynamic interactions to a website, while also maintaining communication with the backend side of the application.



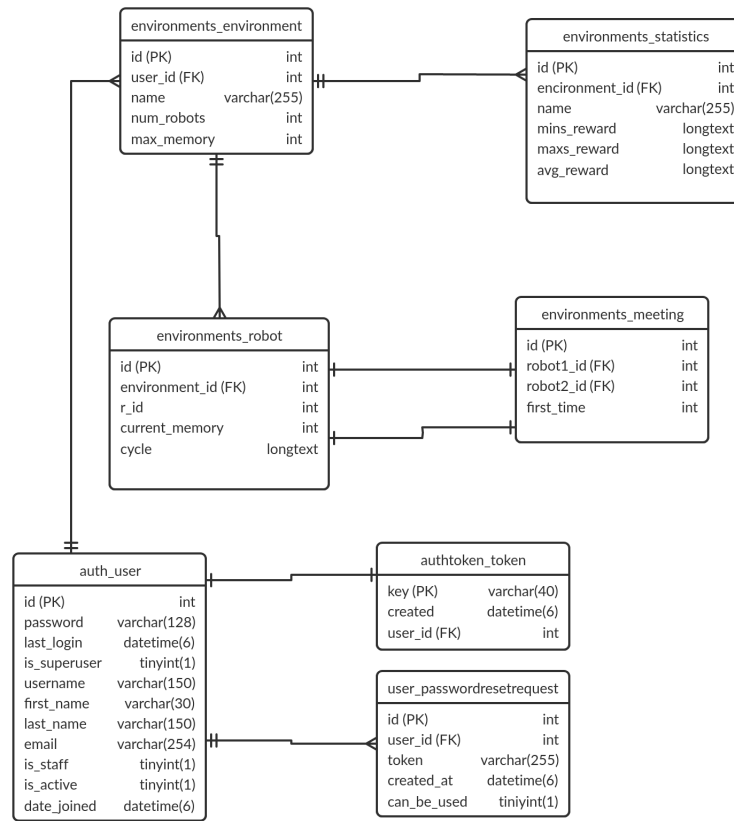


Figure 4.7: Dronem Web database schema

The JavaScript language which is the most used programming language for frontend side applications, is usually enhanced using various frameworks and libraries such as the one used for the *Dronem Web* frontend ReactJS [Rea20]. We chose ReactJS as our frontend library for JavaScript because it is proven to be highly scalable and extensible by maintaining several big web applications such as Facebook, Dropbox or Tesla.

### 4.3.6 Design choices

#### REST vs GraphQL

REpresentational State Transfer (REST) [Fie00] is the most popular web architectural style providing a standard way of communication between computers on the web. Systems which use REST communication are often called RESTful. In order for an interface to be called RESTful the following requirements need to be met:

- Separation between client and server - the interface must separate the concerns of the client (frontend) and server (backend), by doing this the same interface may be used across multiple platforms and the server is more scalable by simplifying its components.

- Stateless - requests coming from the client must contain all the information needed in order to fulfill that request, and cannot be directly based on previous requests.
- Cacheable - every response needs to be labeled as being cacheable or not as the client can use that response for future equivalent requests.
- Uniform interface - Applying software engineering guidelines, having an uniform interface, the overall complexity of the system is minimized.
- Layered system - The architecture of the system must be composed as an hierarchical structure.

GraphQL[The20] on the other hand is a query language for an API and a server-side runtime for executing queries by using a type system defined for the application data. It was created by Facebook and it is supposed to fundamentally change the way applications communicate across the web. GraphQL promises to solve the main problems which arise from using REST, such as over and under data fetching. Over-fetching means downloading unnecessary data, which is a common problem encountered when using REST, because a REST endpoint can only return fixed data structures. Returning fixed data structures also causes under-fetching, for example think about making a call to retrieve a list of users, then for every user we have to make another request to retrieve their followers. These problems are solved by GraphQL with an ingenious solution, declaring specific queries, which give no more, no less, but the necessary data.

Because GraphQL is still an emerging technology, we chose to use REST for the Dronem Web application, but as more and more documentation and applications using GraphQL will start to appear, we will consider enhancing our application by replacing the RESTful interface with GraphQL.

## Redis

Redis (REmote DIctionary Server) [Sal20] is an open source in-memory data-structure, used as a database, message broker and cache. At a high level Redis can be viewed as key-value database, each value is mapped by a key. It supports various types of data structures such as strings, hashes, sets, lists, sorted sets etc. Values can only be retrieved if its key is known. Once installed in a server Redis is very easy to operate using Redis Command Line Interface (CLI), having the following main features:

- Speed - Redis is capable of loading a whole dataset in memory, approximately 81.000 GETs/second.

- Many Supported Languages - There are many programming languages which support Redis bindings by default.
- Master/Slave Replication - Redis supports a very simple and fast Master/Slave replication, being so simple that it only needs one line of configuration.

The main reason Redis has been chosen to be the message broker for the Dronem Web application is that it has a different evolution path in the key-value databases where values can store more complex data types such as our training configurations for environments, with atomic operations on those complex data types.

## Celery

Celery [[Ask18](#)] is a task queue (mechanism to distribute work across machines) used in Django. Its main use cases are:

- Tasks that need to run asynchronously.
- Heavy background computations.
- Periodic tasks.
- Interaction with external API's.

Celery communicates via messages, using a broker such as Redis or RabbitMQ, whose goal is to mediate between clients and workers. In order for the client to initiate a task, the client adds a message into the queue, then the broker assigns the message to a worker. After the worker finishes its job the effects of the task are stored in a task store result, in our case the database.

Because training an environment is not an instantaneous task we want to exclude the process from the request-response HTTP cycle. In order to achieve this, whenever a user requests an environment training a Celery task is started so that a user can train multiple environments in the same time while also being able to use other features of the application because it is not limited by the request-response HTTP cycle (i.e. the user does not need to wait until the training process is complete in order to use the application).

### 4.3.7 Testing

Following the software development methodology stated in Section 4.1, one of the most important activities is testing, as most of the time software testing determines the quality of software after a programmer develops it. The testing process involves evaluating information that is related to a product. Also testing ensures

that the application satisfies the security requirements in order to protect confidential data.

While there are many software testing techniques and procedures we choose to use testing based on specification (Black Box Testing) and testing based on implementation (White Box Testing). Combining the two mentioned techniques in practice yields the best results, by using the Equivalence Class Partitioning (ECP) and Boundary Value Analysis (BVA) from Black Box Testing and Cyclomatic Complexity (CC) and the coverage criteria from White Box Testing.

In order to create flexible and easy to write tests the Pytest[Pyt20] framework was used. Pytest is a framework which allows to write Python tests for different application levels including database, API or User Interface (UI). One of the most important advantages of Pytest over other testing frameworks is the ease of use and test readability, which are key features when it comes to testing. What differentiate Pytest from other testing frameworks is the usage of a system based on fixtures (functions which initialize test functions), instead of using the classic *setUp*, *tearDown* testing system. Another important feature of Pytest is the ease of creating *mock* objects using a special fixture, called *monkeypatch*, which allows changing the behaviour of any Python object with only one line of code.

For the Dronem Web application, several tests were designed and implemented in order to ensure that the application works according to the system requirements defined in Section 4.1.1. Extensive testing was done around the *QLearning* algorithm and the *DronemGymEnv* architecture, while numerous tests were conducted for the API of the application in order to ensure optimal functionality.

### 4.3.8 Documentation

Documenting application is a mandatory activity in software development because good documentation makes it easier for the developers to maintain and add new features to the application. The main focuses of a documentation are development, maintenance and knowledge transfer to other developers. Usually the principal components of a documentation concern with server environments, databases, troubleshooting, application installation and code deployment.

The documentation for the *Dronem Web* application is composed of two main parts. The first part consists in detailed comments for every relevant function, including input parameters, requirements and effects. Also detailed comment blocks were written for the *DronemGymEnv* environment highlighting the main characteristics of the environment. The second part of the documentation is represented by the API documentation which was automatically generated using Django REST Swagger. An example of the API documentation generated for one REST */users/sign-*

in/ path can be saw in Figure 4.8

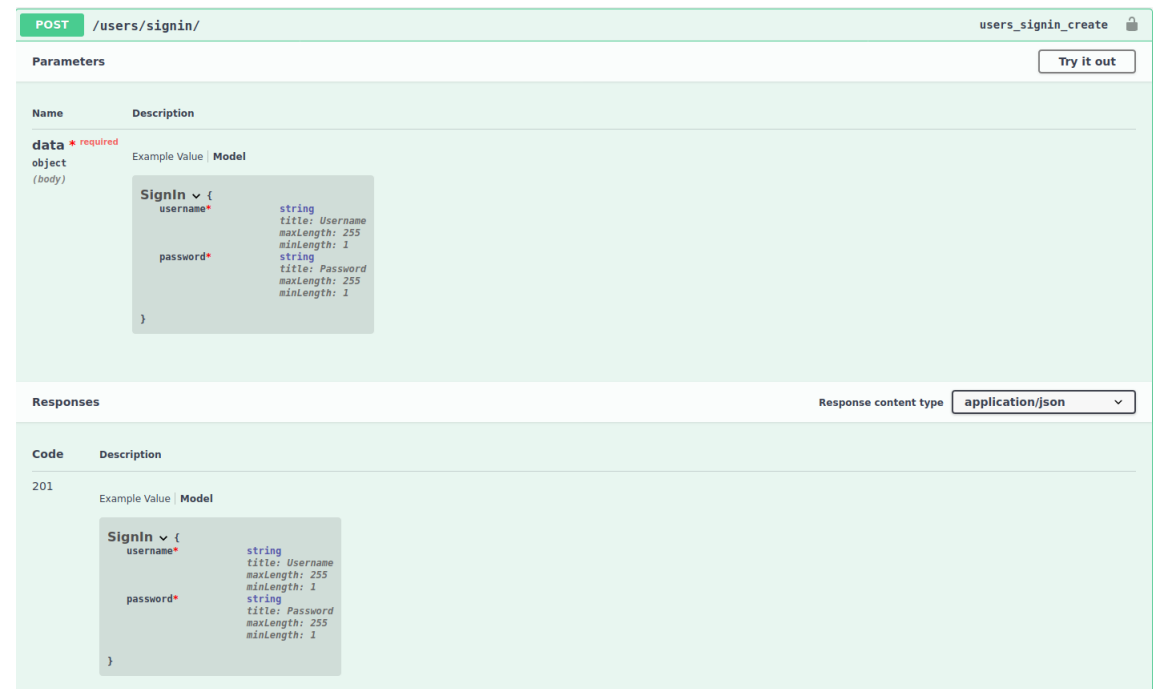



Figure 4.8: API documentation for `/users/signin/` path

## 4.4 User manual

It is very important to write a good User Manual of the application in order to ease the first interactions of the user with the main functionalities of the platform, hence this section is focused on familiarizing the user with the application.

### 4.4.1 Creating environments

To achieve creation of an environment the user must click the Add Environment tab in order to to have access to the environment add form shown in Figure 4.9. After that, the user can choose between adding the environment through a JSON file or by completing the multi-step form, of course the preferred way is via a JSON file, as adding a big environment by hand using a form is a tedious work. After completing the multi step form, or uploading the JSON file, an *Add Environment* button will appear, clicking it will result in creating a new environment.



[Home](#) [Add Environment](#) [Train Environment](#) [Environments](#) [Sign out](#)

### General information

NEXT STEP

ADD FROM JSON

Figure 4.9: Drone Web add environment view

### 4.4.2 Training environments and downloading Q-tables

#### Training an environment

In order to start a training session, the user must select the *Training Environment* tab. After clicking the training tab, the screen shown in Figure 4.10 will appear, the user must select an environment and input the training parameters for the algorithm. Finally clicking the Train button will start a training session, and live statistics of the training will appear on the canvas situated on the right side of the screen.

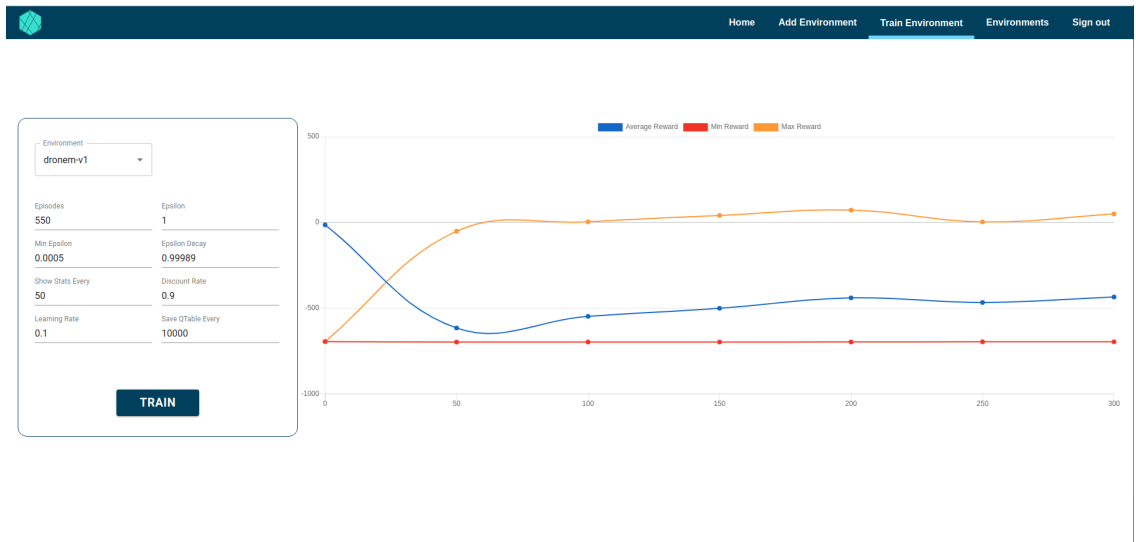


Figure 4.10: Drone Web train view

### Downloading Q-tables

The environments tab, presents numerous functionalities as can be observed in Figure 4.11. The user can download a JSON file containing the best Q-table for an environment by clicking the *Download Best QTable* button.

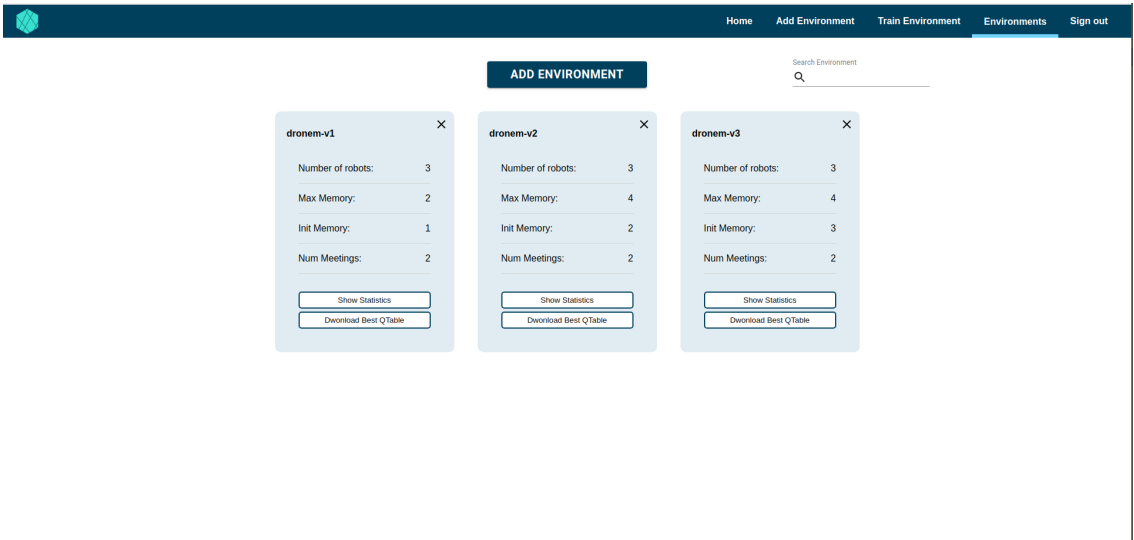


Figure 4.11: Drone Web environments view

### Showing training statistics

Also the user can retrieve statistics about past training of an environment, by clicking the *Show Statistics* button, leading to the view depicted in Figure 4.12

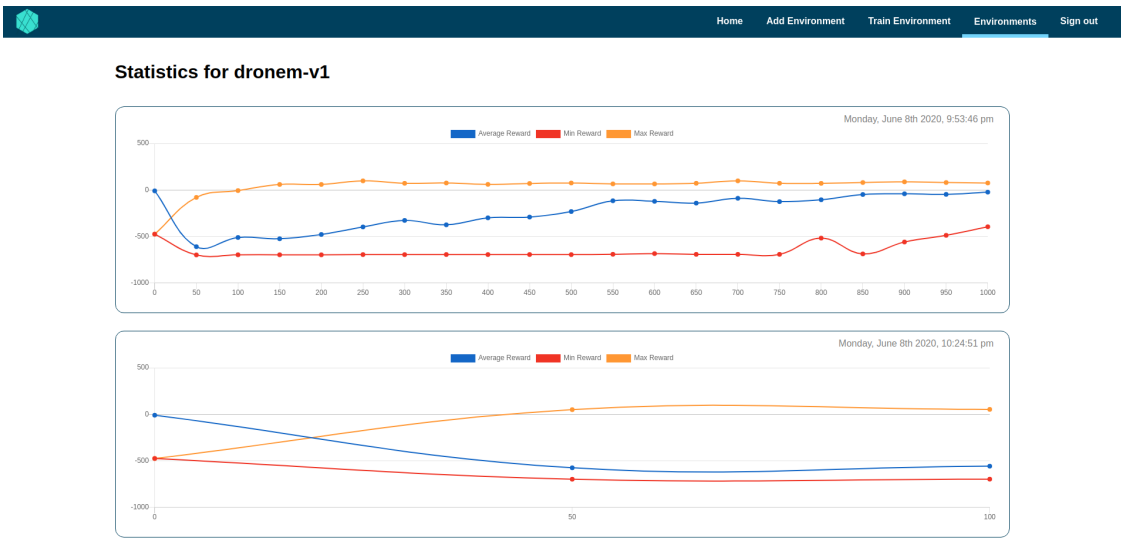


Figure 4.12: Drone Web statistics view

## 4.5 Discussion

The purpose of this chapter was to present the software development process followed in order to develop the *Dronem Web* application. Several development activities were documented along this chapter, starting with analysis and design, implementation, testing and documentation of the application while outlining the importance of following a software development methodology when constructing an application and what catastrophic consequences can arise when no strategy is used. This section presents the original contributions and benefits brought to the Reinforcement Learning community by the creation of this application.

First of all the *Dronem Web* is, to the best of our knowledge, the only application available for creating, manipulating and training MRP environments. We presumably think that more and more users will start using our application for different tasks ranging from university level projects to real world problems such as smart surveillance systems, as long as there is a suitable MRP modelling for those problems. Another original contribution of our application consists in the possibility to retrieve the Q-table for the best training session of an environment, hence the training results can be further used in real applications that are modelled as MRP problems. Also we encourage the users to further extend our *Dronem Gym Environment* which will be publicly available at: <https://github.com/GeorgianBadita/Dronem-gym-envirnoment>. We hope to grow a flourishing community of RL enthusiasts around our application who will further develop the foundation presented by this thesis.

Being constructed with the latest technologies and with the best software development practices, we believe the *Dronem Web* application to be very scalable, being able to handle many users and training sessions in parallel if deployed on capable enough hardware.

## 4.6 Future enhancements

This application is laid as a foundation for an efficient way to create, manipulate and train MRP environments, but it opens up several paths for future enhancements.

As every application strives to be better and better, we think that our application may benefit from some enhancements too. First of all, the application is limited in the types of environments it can manage, users are able to only create and train environments corresponding to the specification defined in Chapter 2. In the future we are going to add the possibility to manipulate different types of environments or even create environments for different reinforcement learning problems.



When it comes to training times, currently for some environments waiting times are very high, due to the fact that the application runs on a computer with low system specifications. A possible solution to overcome this issue would be to deploy the backend of the application on a much more powerful machine, reducing training times significantly, for example an EC2 Amazon Web Services (AWS) instance can be used. A task framework solution has already been implemented in order to reduce training times, but as python is not a 'real' multithreaded programming language, because of its Global Interpreter Lock (GIL), other solutions need to be taken into consideration.

Another important area where the Dronem Web application can be improved is the crash recovery. If the machine where the backend of the application is stored crashes, or even if the Redis or Celery services stop working all environments which are trained in that moment will stop and every interaction with the application will fail. Hence, a mechanism for crash recovery is also needed to be considered for the application, so that after any service crash all training environments should return to their state before the crash. One solution for this problem would be to save the entire application state in the database whenever a training step is taken, although this would increase training times, it will ensure that the application can recover whenever a something bad happens. Nevertheless more research in this direction is needed and more enhancements will be taken into consideration in the future.

# Conclusions

This thesis investigates the data delivery optimization problem in a simplified setting of the *multi-robot patrolling* problem, in which the environment is deterministic. Accordingly a centralized and offline approach *DynFloR* based on flows in dynamic networks was introduced. The goal of *DynFloR* is to determine the robots' policy for maximizing the quantity of data delivered to a base station in a given time, assuming that the robots are continuously collecting data during their patrolling. The more general aim of the research initiated in this thesis is to prove that a *reinforcement learning* approach of the MRP problem is highly feasible, and that further research conducted in this direction could lead to impressive results. Therefore, we introduced a reinforcement learning approach *DronemRL*, which is an improvement of the foundation laid by *DynFloR*.

More contributions have been made in this thesis by designing, implementing and documenting the *Dronem Web* application, which to the best of my knowledge is the only application publicly available capable of modeling and training multi-robot patrolling environments. In order to create the application, a client-server architecture was designed; the main libraries and frameworks used were: OpenAI Gym, Django, Django Rest Framework, and ReactJS - ensuring a scalable architecture. I am confident that in the future, many reinforcement learning enthusiasts will start using this application as a learning tool, because it is easy to use, intuitive and efficient.

Further work will extend the experimental evaluation and theoretical analysis of *DronemRL* in order to better assess its performance. I also aim to decentralize the decision making process and to incorporate uncertainty in the communication between robots. In order to achieve these goals, a very refined Deep Q-learning perspective might be considered.

# Bibliography

- [AFEZ17] Nicolo Boscolo’ Alessandro Farinelli and Enrico Pagello Elena Zantotto. Advanced approaches for multi-robot coordination in logistic scenarios. *Robotics and Autonomous Systems*, 90(11):34–44, 2017.
- [AHHS18] Kristopher De Asis, J. Fernando Hernandez-Garcia, G. Zacharias Holland, and Richard S. Sutton. Multi-step reinforcement learning: A unifying algorithm. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*, pages 2902–2909, 2018.
- [Ask18] Ask Solem & Contributors. Celery - Distributed Task Queue. <https://docs.celeryproject.org/en/master/index.html>, 2018. Accessed: 2020-09-06.
- [Bay12] Michael Bayer. Sqlalchemy. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. aosabook.org, 2012.
- [BCP<sup>+</sup>16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.
- [CGS<sup>+</sup>07] H. Chu, A. Glad, O. Simonin, F. Semp, A. Drogoul, and F. Charpillet. Swarm approaches for the patrolling problem, information propagation vs. pheromone evaporation. In *Int. Conf. on Tools with Art. Intelligence*, pages 442–449, France, 2007.
- [Che04a] Yann Chevaleyre. Theoretical analysis of the multi-agent patrolling problem. *Intelligent Agent Technology, IEEE / WIC / ACM International Conference on*, 00:302–308, 2004.
- [Che04b] Yann Chevaleyre. Theoretical analysis of the multi-agent patrolling problem. *Proceedings. IEEE/WIC/ACM International Conference on*

- Intelligent Agent Technology, 2004. (IAT 2004).*, pages 302–308, 2004.
- [CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [CWS<sup>+</sup>15] S. Chen, F. Wu, L. Shen, J. Chen, and S.D. Ramchurn. Multi-agent patrolling under uncertainty and threats. *PLoS ONE*, 10(6):e0130154, 2015.
- [DAEvH<sup>+</sup>15] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces, 2015.
- [DDLW09] Kai Daniel, Bjoern Dusza, Andreas Lewandowski, and Christian Wietfeld. AirShield: A system-of-systems MUAV remote sensing architecture for disaster response. In *2009 3rd Annual IEEE Systems Conference*, pages 196–200. IEEE, March 2009.
- [Dja19] Django Rest Framework Team. Django REST framework. <https://www.django-rest-framework.org/>, 2019. Accessed: 2020-06-06.
- [Dja20] Django core team. Django: A Web framework for the Python programming language. Django Software Foundation. <http://www.djangoproject.com>, 2020. Accessed: 2020-04-06.
- [Dou00] Douglas N. Arnold. The Patriot Missile Failure. <http://www-users.math.umn.edu/~arnold//disasters/patriot.html>, 2000. Accessed: 2020-04-06.
- [DS94] P. Dayan and T.J. Sejnowski. Td(Lambda) converges with probability 1. *Mach. Learn.*, 14:295–301, 1994.
- [EAK09] Yehuda Elmaliach, Noa Agmon, and Gal A. Kaminka. Multi-robot area patrol under frequency constraints. *Annals of Mathematics and Artificial Intelligence*, 57(3-4), December 2009.
- [Fie00] Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

- [Fla20] Flask Team. Flask Framework User Guide. <https://flask.palletsprojects.com/en/1.1.x/>, 2020. Accessed: 2020-04-06.
- [GSFA08] Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors. *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2008.
- [HLH09] K. Hwang, J. Lin, and H Huang. Cooperative patrol planning of multi-robot systems by a competitive auction system. In *Proceedings of ICCAS-SICE*, Fukuoka, Japan, 2009.
- [HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In Gel Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, 2008.
- [HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., USA, 2000.
- [IBM14] IBM Research. Software Development. [https://researcher.watson.ibm.com/researcher/view\\_group.php?id=5227](https://researcher.watson.ibm.com/researcher/view_group.php?id=5227), 2014. Accessed: 2020-02-06.
- [ItMata<sup>+</sup>19] Vlad-Sebastian Ionescu, Zsuzsanna Oneţ Marian, Marin-Georgian Bădiţă, Gabriela Czibula, Mihai-Ioan Popescu, Jilles S. Dibangoye, and Olivier Simonin. *DynFloR: A flow approach for data delivery optimization in multi-robot network patrolling*. In *23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2019)*, volume 159, pages 97–106. Elsevier, 2019.
- [Kat17] Kat Boogaard. 5 Communication Challenges Web Developers Face With Clients (and How to Fix Them). <https://toggl.com/blog/web-development-clients-communication-challenges>, 2017. Accessed: 2020-04-06.
- [KKGB12a] G.D. Konidaris, S.R. Kuindersma, R.A. Grupen, and A.G. Barto. Robot learning from demonstration by constructing skill trees. *International Journal of Robotics Research*, 31(3):360–375, March 2012.

- [KKGB12b] George Konidaris, Scott Kuindersma, Roderic A. Grupen, and Andrew G. Barto. Robot learning from demonstration by constructing skill trees. *I. J. Robotics Res.*, 31(3):360–375, 2012.
- [Kot03] Balzs Kotnyek. An annotated overview of dynamic network flows. Technical report, INRIA, 01 2003.
- [LWT<sup>+</sup>17] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments, 2017.
- [LYTL15] Somchaya Liemhetcharat, Rui Yan, Keng Peng Tee, and Matthew Lee. Multi-robot item delivery and foraging: Two sides of a coin. *Robotics*, 4:365–397, 2015.
- [Mar19] Martin Anderson. Flask vs Django: A Comparative Guide to Python Frameworks. <https://www.iflexion.com/blog/python-flask-vs-django>, 2019. Accessed: 2020-06-06.
- [Mit97] Thomas M. Mitchell. *Machine learning*. McGraw-Hill, Inc. New York, USA, 1997.
- [Net] NetworkX. Prepush-Flow maximum flow algorithm. [https://networkx.github.io/documentation/stable/\\_\\$modules/networkx/algorithms/flow/maxflow.html](https://networkx.github.io/documentation/stable/_$modules/networkx/algorithms/flow/maxflow.html).
- [Obj17] Object Management Group. Unified Modeling Language. <https://toggl.com/blog/web-development-clients-communication-challenges>, 2017. Accessed: 2020-04-06.
- [OGEFSFPB17] Mehdi Othmani-Guibourg, Amal El Fallah Seghrouchni, Jean-Loup Farges, and Maria Potop-Butucaru. Multi-agent patrolling in dynamic environments. In *International Conference on Agents*, Beijing, China, July 2017. IEEE digital library .
- [Ora20] Oracle Corporation. MySQL 8.0 Reference Manual. <https://dev.mysql.com/doc/refman/8.0/en/>, 2020. Accessed: 2020-10-06.
- [PR11] David Portugal and Rui Rocha. A survey on multi-robot patrolling algorithms. In *Technological Innovation for Sustainability*, pages 139–146, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

- [PRS16] M. I. Popescu, H. Rivano, and O. Simonin. Multi-robot patrolling in wireless sensor networks using bounded cycle coverage. In *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 169–176, Nov 2016.
- [PSC<sup>+</sup>18] M. I. Popescu, O. Simonin, G. Czibula, A. Spalanzani, and F. Valois. DMRR: Dynamic multi-robot routing for evolving missions. In *Proceedings of the 6th Workshop on Distributed and Multi-Agent Planning (DMAP) Workshop at ICAPS*, pages 33–41, 2018.
- [PUS99] Andrs Prez-Urbe and Eduardo Sanchez. A comparison of reinforcement learning with eligibility traces and integrated learning, planning and reacting. In *Concurrent Systems Engineering Series*. IOS Press, 1999.
- [Pyt20] Pytest-Dev Team. pytest: helps you write better programs. <https://docs.pytest.org/en/stable/contents.html#toc>, 2020. Accessed: 2020-08-06.
- [RBBA18] Marta Romeo, Jacopo Banfi, Nicola Basilico, and Francesco Amigoni. *Multirobot Persistent Patrolling in Communication-Restricted Environments*, pages 59–71. Springer International Publishing, 2018.
- [Rea20] ReactJS Development Team. Getting Started. <https://reactjs.org/docs/getting-started.html>, 2020. Accessed: 2020-08-06.
- [Rob] RoboCup - Rescue. Building Rescue Systems of the Future . [www.robocuprescue.org](http://www.robocuprescue.org).
- [Sal20] Salvatore Sanfilippo & Contributors. Redis Documentation. <https://redis.io/documentation>, 2020. Accessed: 2020-09-06.
- [SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [SLH<sup>+</sup>14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic Policy Gradient Algorithms. In *ICML*, Beijing, China, June 2014.
- [SRCR04] Hugo Santana, Geber Ramalho, Vincent Corruble, and Bohdana Ratitch. Multi-agent patrolling with reinforcement learning. In *Proceedings of the Third International Joint Conference on Autonomous*

- Agents and Multiagent Systems - Volume 3*, AAMAS '04, pages 1122–1129, Washington, DC, USA, 2004. IEEE Computer Society.
- [The20] The GraphQL Foundation. Introduction to GraphQL. <https://graphql.org/learn/>, 2020. Accessed: 2020-10-06.
- [vCV11] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011.
- [WD92] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.
- [ZYB19] Kaiqing Zhang, Zhuoran Yang, and Tamer Baar. Multi-agent reinforcement learning: A selective overview of theories and algorithms, 2019.