UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2023-027
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 31/03/2023 in Esch-sur-Alzette

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Gabriel DUFLO
Born on 16 July 1994 in Vernon, (France)

LEARNING OPTIMISATION ALGORITHMS OVER GRAPHS

Dissertation defence committee

Dr Grégoire DANOY, dissertation supervisor
Université du Luxembourg

Dr Pascal BOUVRY, Chairman
Professor, Université du Luxembourg

Dr El-Ghazali TALBI, Vice Chairman
Professor, Université de Lille

Dr Roland BOUFFANAIS
Professor, University of Ottawa

Dr Ann Nowé
Professor, Vrije Universiteit Brussel

Abstract

The paradigm of *learning to optimise* relies on the following principle: instead of designing an algorithm to solve a problem, we design an algorithm which will automate the design of such a solver. The initial idea was to alleviate the limitations stated by the *No Free Lunch Theorem* by producing an algorithm which efficiency is less dependent upon known instances of the problem to tackle. Hyper-heuristics constitute the main *learning-to-optimise* techniques. These rely on a high-level algorithm performing a search process into a space of low-level heuristics to tackle a given problem. Because the latter search space is problem-dependent, the vast majority of hyper-heuristics are designed to tackle a specific problem. Due to this lack of generality, existing works fully redesign hyper-heuristics when tackling a new problem, despite the fact that they may share a similar structure.

In this dissertation, we tackle this challenge by proposing a generic way for *learning to optimise* any problem. To this end, this thesis introduces three main contributions: (i) an analysis of the formal functioning of *learning-to-optimise* techniques; (ii) a model of generic hyper-heuristic, named Algorithm Learner for Graph Optimisation problems (ALGO), constituting the central point of this work; (iii) a real-world use case where we use our generic hyper-heuristic to automate the design of behaviours within a swarm of drones.

In the first part, we provide a formalism for optimisation and learning concepts, which we use to describe the large body of knowledge that combines two layers of optimisation and/or learning. We then put an emphasis on approaches using learning to improve an optimisation process, *i.e.*, aiming at *learning to optimise*. In the second part, we present ALGO, our model of generic hyper-heuristic. We explain how we abstract from a given problem with a graph structure so that it can be used to tackle any optimisation problem. We also detail the steps to follow in order to use ALGO to tackle a given problem. We finally present the modularity of ALGO with inner components that a user can implement. The second part ends with a validation of our model, *i.e.*, using ALGO to tackle a classical optimisation problem. In the third part, we use ALGO to tackle the problem of area surveillance with a swarm of drones. We demonstrate that ALGO constitutes a novel and efficient way to automate the design of such a distributed and multi-objective problem.

Acknowledgements

First and foremost, I would like to thank Dr. Grégoire DANOY for his immeasurable support. There is no word to express how grateful I am for his patience, his knowledge and the quality of his guidance, which helped me for my research and for writing this thesis. I could not have imagined having a better supervisor.

I would like to express my sincere gratitude to Prof. Pascal BOUVRY who gave me the opportunity to pursue my PhD by welcoming me into his dynamic team. I felt lucky to work in a pleasant atmosphere throughout my whole PhD studies. I would also like to thank Prof. El-Ghazali TALBI for his insights and advice which allowed me to deepen my research.

Besides my supervision committee, I cannot name all my colleagues from the Parallel Computing & Optimisation Group but I give a special thank you to Pierre-Yves HOUITTE for his support. He clearly contributed to the good atmosphere within the team, despite his taste for Stade Rennais.

Contents

List of Figures

List of Tables

List of Algorithms

Symbols

NFLT	No Free Lunch Theorem
SL	Supervised Learning
UL	Unsupervised Learning
RL	Reinforcement Learning
QL	Q-Learning
TSP	Travelling Salesman Problem
VRP	Vehicle Routing Problem
MVCP	Minimum Vertex Cover Problem
OJSP	Optimal Job Scheduling Problem
MO	Multi-Objective
MA	Multi-Agent
GA	Genetic Algorithm
GP	Genetic Programming
GNN	Graph Neural Network
NN	Neural Network
HH	Hyper-Heuristic
HLA	High-Level Algorithm
LLH	Low-Level Heuristic
ALGO	Algorithm Learner for Graph Optimisation problems
AFOP	ALGO-Friendly Optimisation Problem
UAV	Unmanned Aerial Vehicle
CCUS	Coverage of a Connected-UAV Swarm
CCUS3O	CCUS with 3 Objectives
$G_{(N,E)}$	Graph G where N is the set of nodes and E is the set of edges

Chapter 1

Introduction

Contents

1.1 Context

In this PhD work, we explore an area where machine learning is used to improve optimisation, hence the paradigm of *learning to optimise*. Given a problem P, we do not design here an algorithm to tackle it, *i.e.*, to find a solution to an instance of P, we aim at extracting information which makes an algorithm efficient or not to find a good solution. This information is then used to output an algorithm that we can use to tackle P. The idea behind *learning to optimise* is to alleviate the limitations stated by the *No Free Lunch Theorem*. To tackle P, if we manually design an algorithm based on known instances of P, it will be at the expense of worse performances on unknown instances. Automating the design of an algorithm to tackle P thus aims at obtaining an algorithm with an overall behaviour less dependent on known instances. This process of automating the design of a heuristic is referred to as a Hyper-Heuristics (HHs).

1.2 Motivation & Contributions

Along with the paradigm of *learning to optimise*, we show in Chapter 2 that other techniques consist in combining two layers of optimisation/learning. For instance, optimisation can be used to improve the performance of a learning algorithm or another optimisation process. We referred to these approaches as hybrid techniques. Within them, both involved processes are entangled, resulting to a blurred border between optimisation and learning. The literature is indeed missing a formal definition of optimisation and learning which can be used to describe these hybrid techniques.

As we describe in Chapter 3, hyper-heuristics have been used in numerous contexts and applied on a wide range of optimisation problems. Nevertheless, they have demonstrated that they are well suited for tackling problems based on graphs [Burke et al., 2013, Mazyavkina et al., 2021]. In the literature, many HHs using the graph structure of instances can be found, even though their functioning is very similar. Some HHs designed to tackle several optimisation problems can already be found in the literature [Khalil et al., 2017, Hao et al., 2021, Zhao et al., 2021, Zhang et al., 2022, Lu et al., 2022]. Their implementation is however not modular. It means that, despite a theoretical generic model, a significant amount of design and rewriting is required per problem.

In addition, one limitation of hyper-heuristics is the simplicity of the algorithm generated to tackle a given problem. It is an advantage in terms of running time, control or explainability, but it may also induce limited performance. This is even more true when the tackled problems are classical ones which have been deeply studied in the literature. We believe that such HHs cannot generate an algorithm which is competitive with existing techniques. On the other hand, we see a promising usage of HHs in a real-world context, where problems may have multiple objectives, consider a dynamic environment, multiple agents, and/or distributed heuristics. Using Reinforcement Learning (RL) in that context is promising thanks to its ability to approximate a future reward from local interactions. In addition, RL has already shown a good potential for being used in a multi-objective context, referred to as Multi-Objective Reinforcement Learning (MORL) [Van Moffaert et al., 2013, Moffaert and Nowé, 2014], and in a multi-agent context, referred to as Multi-Agent Reinforcement Learning (MARL) [Zhang et al., 2021]. As an example, we illustrate in Chapter 6 that there is a gap to fill when it comes to designing the behaviour of a swarm of Unmanned Aerial Vehicles (UAVs). The dynamic aspect comes from the fact that a UAV moves according to the position of others moving at the same time. One specificity of a swarm is the limited control from human operators [Arnold et al., 2019]. It means that UAVs are flying considering their local information, hence in a distributed way.

1.2.1 Research questions

Based on the aforementioned facts, this PhD manuscript aims at addressing the following three main research questions. These are ordered from a general context to an application context.

Q1 How to depict a structured view of hybrid techniques?

Q2 How to learn to optimise any problem with reinforcement learning?

Q3 How to automate the design of a swarm of UAVs without predefining specific actions?

1.2.2 Research objectives

In order to answer the research questions, we first analyse the wide area of techniques combining both learning and optimisation, which constitute the first objective O1. The purpose is to describe these various works by using a single paradigm, and therefore to address a lack of analysis of approaches combining optimisation and learning. It also aims at drawing a general context, to which belongs the area of *learning to optimise*, *i.e.*, using learning to improve optimisation. The state of the art analysis of this specific field corresponds to objective O2. Validating both objectives O1 and O2 therefore answers P1.

With objective O3, we want to design a generic hyper-heuristic that can be used to tackle different problems. The aim is to go beyond a generic theoretical model by providing a complete framework which permits to generate heuristics for both classical and real-world problems, without the need to implement a new model for

each problem. This represents the objective O4. Successfully tackling a classical benchmark with our generic hyper-heuristic is used as a validation for our model, and thus answers P2.

Finally, we aim at applying our generic hyper-heuristic to automate the design of a swarm of UAVs. For that purpose, as stated by objective O5, we model the task of surveillance with a swarm of drones as an optimisation problem. We then use our generic hyper-heuristic to generate distributed heuristics for that problem to answer P3.

The five research objectives of this PhD work are summarised hereinafter.

O1 Categorise techniques combining both learning and optimisation processes

O2 Survey the field of *learning to optimise*

O3 Design a generic hyper-heuristic based on reinforcement learning to tackle different optimisation problems

O4 Demonstrate the capacity of our hyper-heuristic to generate efficient and stable heuristics to tackle both a classical benchmark and a real-world problem, *i.e.*, surveillance with a swarm of drones

O5 Model the surveillance with a swarm of drones as a distributed optimisation problem

1.2.3 Contributions

As a first step towards achieving the objectives mentioned above, we introduce a formalism to describe both optimisation and learning concepts. The idea is to extract components proper to both processes regardless of the specific technique used. This description is then extended to hybrid techniques, *i.e.*, combining optimisation and/or learning at different levels. We indeed provide an analysis of their functioning by determining the components involved in these hybridisations.

We then provide a state of the art analysis of the area to which this PhD work belongs to, *i.e.*, hybrid techniques using learning to improve optimisation aiming at *learning to optimise*.

As our main contributions, we design a generic hyper-heuristic, named Algorithm Learner for Graph Optimisation problems (ALGO), and its abstract optimisation problem counterpart, referred to as ALGO-Friendly Optimisation Problem (AFOP). The idea of ALGO is to use a single algorithm to generate a heuristic for any compatible optimisation problem. A problem is compatible when it can be represented as an AFOP which is based on a graph structure. We then define the latter abstract problem and we provide information on how to override it for a given optimisation problem.

In this PhD thesis, we use ALGO in different contexts. For each of them, a description of the implementation process is provided. We first apply our generic hyper-heuristic on a classical optimisation problem, the Travelling Salesman Problem (TSP). As mentioned earlier, the purpose is not to generate a competitive heuristic to tackle the TSP, given the wide range of competitive solvers for this problem. Nonetheless, we demonstrate the efficiency and stability of the algorithm generated by ALGO. Secondly, we want to use ALGO in a dynamic and distributed context, *i.e.*, to automate the design of swarming behaviours. The quality of the behaviour generated by ALGO illustrates its ability to tackle a wide range of problems with different natures: centralised or distributed; single or multiple objectives; single or multiple agents.

In order to use ALGO to automate the design of swarming behaviours, we defined an optimisation problem to describe the coverage of an area with a swarm of UAVs by considering the connectivity within the swarm. We

name that problem the Coverage of a Connected-UAV Swarm (CCUS). A heuristic for that problem, generated by ALGO, is then equivalent to the distributed behaviour of a UAV within the swarm.

1.3 Overview

This thesis is divided into three main parts, plus the conclusion, as depicted in Figure 1.1.

In Part I, background information is provided in order to introduce *learning-to-optimise* techniques in the existing literature. Chapter 2 starts by a presentation of a wide range of techniques combining learning and optimisation. After determining in which subarea this PhD work takes place, we analyse the state of the art of advances in hyper-heuristics in Chapter 3.

In Part II , we introduce our novel model of hyper-heuristics, ALGO. We propose a formal definition of ALGO in Chapter 4 and validate its usage and the performance of the algorithm it generates on a well-known benchmark in Chapter 5.

Part III focuses on the usage of ALGO to tackle a real-world use case that is both multi-objective and distributed, i.e. the coverage of an area by a swarm of UAVs. Chapter 6 first provides a survey of the state of the art on robot/UAV swarm design, ending with a formal description of CCUS. The latter serves as a basis to motivate the novelty of our hyper-heuristic approach in that domain. Chapter 7 contains the experimental details and results of ALGO on the UAV swarm problem. The obtained swarming behaviour's performance is analysed and compared with state-of-the-art techniques.

The last part concludes this dissertation by summarising all the contributions and outcomes that have been obtained during this PhD. A discussion on the perspectives and future works concludes the manuscript.

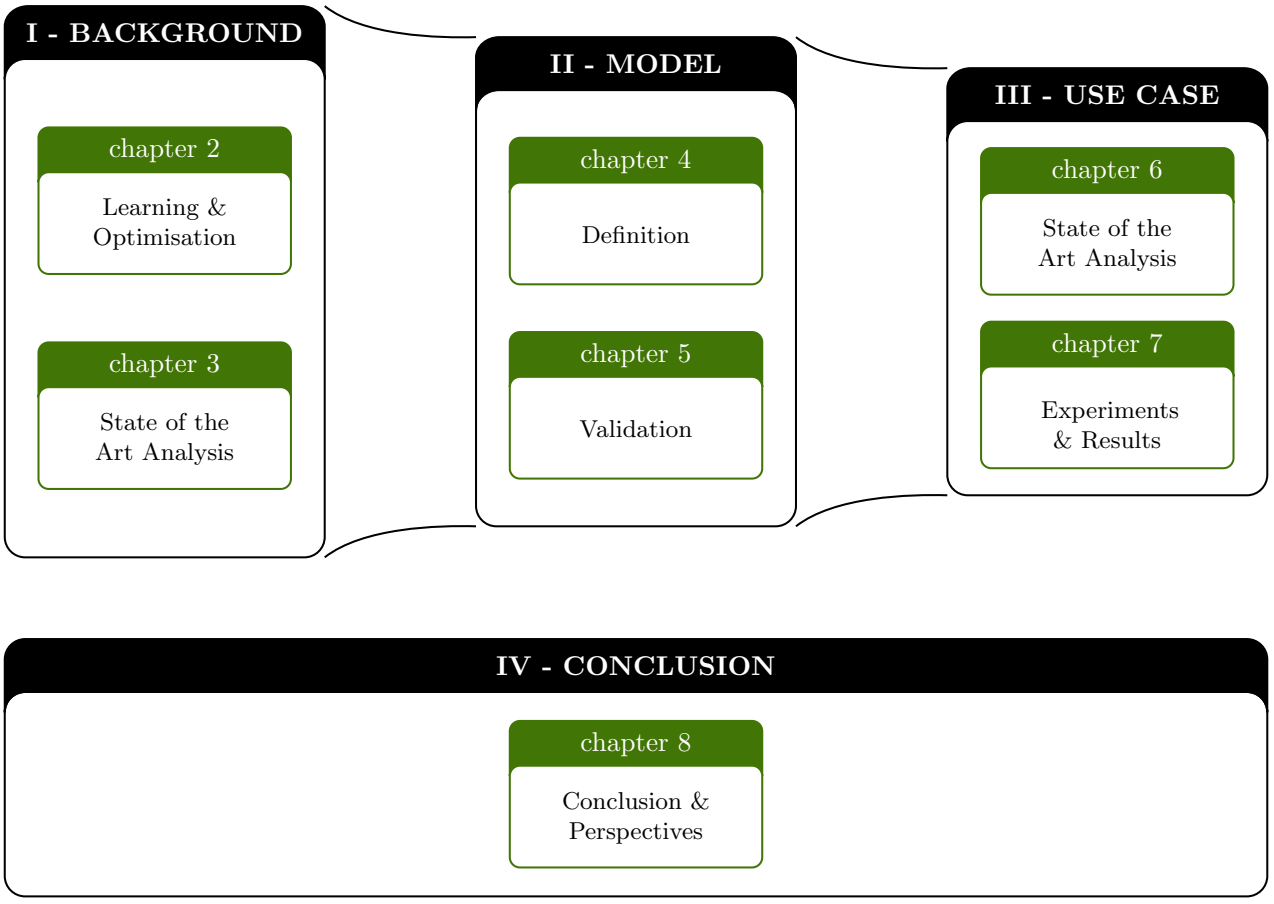


FIGURE 1.1: Organisation of the dissertation.

Part I

Background & State of the Art

Chapter 2

Learning vs Optimisation

Contents

2.1 Introduction

This PhD work is using learning techniques in order to help optimisation. It thus lies at the edge of both the optimisation and the learning domains. At lot of work has proposed to combine these two areas in different ways. The purpose of this chapter is to formally describe the large body of works combining both optimisation and learning and to define the specific area of this PhD work within the state-of-the-art.

Optimisation and learning are two powerful problem-solving tools. Both processes are different and require different inputs, even if they both result in finding a solution to a problem. In order to illustrate how both approaches can be applied to different types of problems, we introduce three problem examples, P1, P2 and P3, that will be used throughout this chapter:

P1: Given a road network and two locations A and B, find the shortest way to go from A to B;

P2: Given a picture, detect whether it represents a dog or a cat;

P3: Given a person from a group, find the other similar individuals from that group.

The remainder of this chapter is organised as follows. Both the optimisation and the learning concepts are first categorised in Section 2.2. We introduce there a formalism aiming at identifying optimisation and learning.

Such a formal distinction between both processes is an important first step towards the description of the possible hybridisation techniques, *i.e.*, involving optimisation and learning. We present these techniques in Section 2.3.

2.2 Categorisation

In this section, we focus on the categorisation of both optimisation and learning concepts. The objective is to provide a formal description of optimisation and learning, to define inputs and outputs of both processes and to extract inner components proper to them. The formalism introduced here is independent of specific techniques. For instance, the description of an optimisation process encompasses any heuristic or metaheuristics algorithm. Similarly, our description of learning applies for supervised learning, unsupervised learning and reinforcement learning.

2.2.1 Optimisation

We first describe the concept of optimisation with a high-level overview. We want here to define an optimisation process by what is required as an input and what is returned as an output. This overview is schematised in Figure 2.1 for optimising over a space $\mathcal{X}$, where $\mathcal{X}$ is the space of solutions of the problem to solve.

- **Inputs.** Two requirements are needed for the optimisation process: defining the constraints and defining the objective function. Given a problem, if $\mathcal{X}$ is the space of solutions, let $\mathcal{C}_\mathcal{X} \subseteq \mathcal{X}$ be the space of feasible solutions, *i.e.*, solutions respecting the constraints and $F_\mathcal{X} : \mathcal{X} \to \mathbb{R}$ be the objective function assigning a value to each solution.
- **Output.** An optimisation process returns an element of $\mathcal{C}_\mathcal{X}$, *i.e.*, a feasible solution. This elements is wanted to minimise or maximise $F_\mathcal{X}$. For the sake of simplicity, without loss of generality, it is considered that an objective function is to be minimised in this chapter.

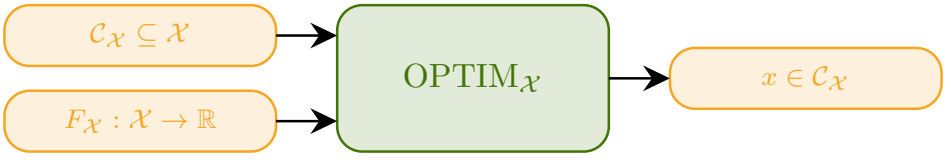


FIGURE 2.1: Inputs and output of the process of optimising over a space $\mathcal{X}$.

Example 2.1. *The following presents possible models of the optimisation process to tackle the three aforementioned illustrative problems (see Section 2.1). For each problem these must be defined:*

- $\mathcal{X}$*, the set of solutions,*
- $\mathcal{C}_\mathcal{X}$*, the set of feasible solutions,*
- $F_\mathcal{X}$*, the objective function.*

An example of optimisation problem model is provided below for each of the three illustrative problems. It is worth noting that other models are however possible.

P1:
- *$\mathcal{X}$ is the set of all possible paths in the road map.*
- *$\mathcal{C}_\mathcal{X}$ is the of all paths going from A to B.*
- *Given a path $x \in \mathcal{X}$, $F_\mathcal{X}(x)$ is its length.*

P2:
- *The solution must be a choice between "dog" and "cat", so $\mathcal{X} = \{dog, cat\}$.*
- *From the definition of $\mathcal{X}$, all solutions are feasible, so $\mathcal{C}_\mathcal{X} = \mathcal{X} = \{dog, cat\}$.*
- *The function $F_\mathcal{X}$ must assign a score for each possible solution so that $F_\mathcal{X}(dog) \leq F_\mathcal{X}(cat)$ if a dog is in the given picture, and vice versa. That objective function is supposed to analyse the given picture, on a per-pixel basis for instance, and compute a score according to the given choice. Such a function is very tough to manually design, which shows that simply optimising the detection for that problem may not be appropriate.*

P3: Let P be the set of persons in the group and p the person to whom it is asked to find similar people.
- *$\mathcal{X}$ is the power set of P,* i.e., *elements of $\mathcal{X}$ are subsets of P.*
- *$\mathcal{C}_\mathcal{X}$ is the set of all solutions $x \in \mathcal{X}$,* i.e., *all subgroups of P, containing p.*
- *Finally, to define a proper objective function $F_\mathcal{X}$, more information about the problem is required,* e.g., *attributes of persons making a comparison possible or a threshold to detect whether another person is "similar enough". From this information, given a subgroup of persons $x \in \mathcal{X}$, $F_\mathcal{X}(x)$ should return a score corresponding to a global similarity of people in x.*

The concept of optimisation illustrated in Figure 2.1 can be extended by integrating inner components common to each optimisation process, regardless of the specific technique used (heuristic or metaheuristic). The process of optimisation is based on searching the optimal solution among feasible ones. We show that the searching process relies on two inner components: an initial solution $x_0 \in \mathcal{X}$ and an optimiser which can be assimilated to a function $O : \mathcal{X} \to \mathcal{X}$. They are both represented in Figure 2.2.

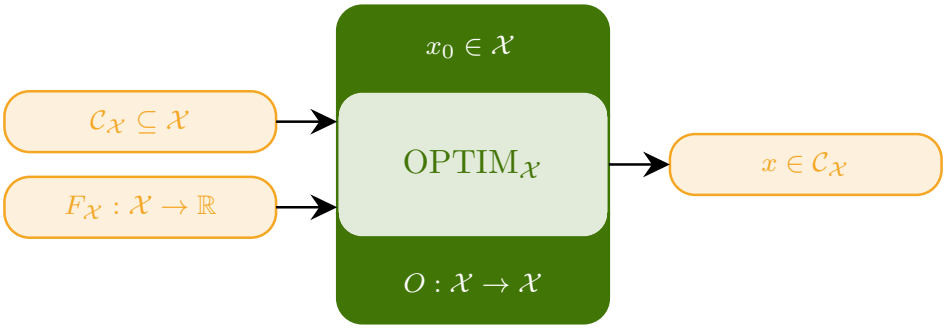


FIGURE 2.2: Optimisation process (green rectangle) with two inner components: the initial solution (at the top) and the optimiser (at the bottom).

Two main categories of techniques can be distinguished: exact algorithms where the optimal solution is guaranteed to be returned; and heuristic algorithms which can run in worst-case polynomial time for NP-hard problems. The focus is made on heuristic algorithms in this chapter due to the nature of this PhD work. In the following, we then show the two inner components are used in both heuristic and metaheuristic algorithms.

2.2.1.1 Heuristics

A heuristic algorithm can be described by the following: from an initial solution, the optimiser is called to update the current solution until a termination condition is achieved. The last current solution is finally returned. Any

heuristic algorithm follows this generic template, whether it be constructive or perturbative heuristics. Starting from a non-feasible solution, a constructive heuristic consists in updating the current solution with the optimiser until it becomes feasible (see Algorithm 2.1). The optimiser is here seen as a step to construct the solution. The purpose of a perturbative heuristic is meanwhile to update a feasible solution with the optimiser until it reaches a local optimum (see Algorithm 2.2). Perturbative heuristics work with the definition of a neighbourhood in the space of solution $\mathcal{X}$. The purpose of the optimiser is then to return the best solution from the neighbourhood of the current solution.

Algorithm 2.1 Constructive heuristic

Input: $\mathcal{C}_{\mathcal{X}} \subset \mathcal{X}$
$F_{\mathcal{X}} : \mathcal{X} \to \mathbb{R}$
Output: $x \in \mathcal{C}_{\mathcal{X}}$

1: $x \leftarrow x_0 \in \mathcal{X}$
2: **while** $x \notin \mathcal{C}_{\mathcal{X}}$ **do**
3: $\quad x \leftarrow O(x)$
4: **end while**
5: **return** x

Algorithm 2.2 Perturbative heuristic

Input: $\mathcal{C}_{\mathcal{X}} \subset \mathcal{X}$
$F_{\mathcal{X}} : \mathcal{X} \to \mathbb{R}$
Output: $x \in \mathcal{C}_{\mathcal{X}}$

1: $x \leftarrow x_0 \in \mathcal{C}_{\mathcal{X}}$
2: **while** $F_{\mathcal{X}}(O(x)) < F_{\mathcal{X}}(x)$ **do**
3: $\quad x \leftarrow O(x)$
4: **end while**
5: **return** x

Example 2.2. *In problem P1, a solution,* i.e.*, an element of $\mathcal{X}$, is a path in the road network, which can be represented as a sequence of adjacent locations. A feasible solution,* i.e.*, an element of $\mathcal{C}_{\mathcal{X}}$, is a path going from A to B.*

- ***Constructive heuristic.*** *The initial solution is an empty path starting from A,* i.e.*, a sequence containing only the location A. The optimiser then consists in adding a location to the current solution. The process ends when the last location added is B.*
- ***Perturbative heuristic.*** *The initial solution is a path going from A to B. The action of the optimiser can then be to replace a sub-path of the solution by another pass with the same starting and ending locations. This is done until a local optimum is reached,* i.e.*, there is no possibility to replace a sub-path by improving the solution.*

2.2.1.2 Metaheuristics

The process of metaheuristics is based on perturbative heuristics using a stochastic optimiser. The purpose is to introduce a balance between exploitation and exploration. The exploration consists in not following the optimiser of a perturbative heuristic, in order to avoid being stuck in a local optimum. Because of the exploration, the termination condition must be different. Usually, a convergence condition is used.

There are different ways to classify metaheuristics. One of them is to distinguish the ones dealing with a single solution at each iteration and ones working with a population of solutions. In case of population-based metaheuristics, the optimiser has to update a set of solutions instead of a single one. More generally, the optimiser can be represented by $O : \mathcal{X}^a \to \mathcal{X}^b$, where $a = b = 1$ in the metaheuristic is single-solution based. For instance, in a Genetic Algorithm (GA), the optimiser comprises all of operations to update a population of individuals from a generation to another one.

2.2.2 Learning

As with optimisation, we start by describing the concept of learning by the format of its inputs and output. An high-level overview of a learning process if schematised in Figure 2.3. Given a problem to solve, as opposed to optimisation, using learning does not consist in returning the solution to the problem. The purpose is to learn a task which can be described by a space of features $\mathcal{F}$ and a space of solutions $\mathcal{X}$. The task therefore consists in finding a solution according to some features. As for optimisation, a high-level overview of the concept of learning is schematised in Figure 2.3 where we define a learning process by the format of its inputs and output.

- **Inputs.** A learning process is based on a training data, *i.e.*, a sample of features $V_{\mathcal{F}}$ and a sample of corresponding solutions $V_{\mathcal{X}}$. This training data provides an initial mapping from $\mathcal{F}$ to $\mathcal{X}$.
- **Output.** From the initial mapping provided by the training data, the learning process aims at building a complete mapping $L : \mathcal{F} \to \mathcal{X}$ in order to find a solution from any element of $\mathcal{F}$.

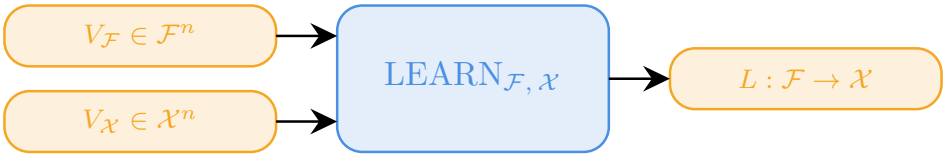


FIGURE 2.3: Input and output of the process of learning to map an element of $\mathcal{F}$ to an element of $\mathcal{X}$.

Example 2.3. *The following introduces possible models for the learning process for tackling the three illustrative problems (see section 2.1). For each of them, the goal is to identify both spaces $\mathcal{F}$ and $\mathcal{X}$,* i.e., *to define the wanted mapping $L : \mathcal{F} \to \mathcal{X}$.*

P1:
- *$\mathcal{F}$ represents the set of all possible couples of locations in the road map.*
- *$\mathcal{X}$ is the set of all possible paths.*

P2:
- *$\mathcal{F}$ is a set of possible pictures.*
- *A solution is a choice between "cat" and "dog",* i.e., *$\mathcal{X} = \{cat, dog\}$.*

P3: Let P be the set of people in the group.
- *Any element of P can be given as a feature,* i.e., *the person to whom it is asked to find similar people, so $\mathcal{F} = P$.*
- *The task is to map a person to a group of persons from P, so $\mathcal{X} = \mathcal{P}(P)$,* i.e., *the power set of P.*

Different ways exist to learn the solution to a problem. Learning techniques can indeed be divided into three categories: supervised learning, unsupervised learning, reinforcement learning. While these differ in their input required for training, we show in the following that they always follow the scheme depicted in Figure 2.3.

2.2.2.1 Supervised Learning

In Supervised Learning (SL), the training data is directly composed of elements of $\mathcal{F}$ and $\mathcal{X}$. To each feature given as an input is assigned a label. The purpose is thus to assign a label to unknown features. SL techniques can still be divided into two categories according to the nature of $\mathcal{X}$. Both are depicted by Figure 2.3. If $\mathcal{Y}$ is discrete, the process is called classification. To each element of $\mathcal{X}$ must be assigned a label. Solving P2 as in Example 2.3 is an example of classification with two labels: "dog" and "cat". If $\mathcal{Y}$ is continuous, the process is a regression.

2.2.2.2 Unsupervised Learning

Unlike SL, Unsupervised Learning (UL) does not directly take as an input elements of $\mathcal{X}$. The input data is said unlabelled. So $X \in \mathcal{X}^n$ is not directly given as an input. The main purpose of UL is not to label data but to gather data into clusters. The latter can represent a high correlation or similarity between elements of $\mathcal{F}$. If the process takes $V_{\mathcal{F}} \in \mathcal{F}^n$ as an input, to each feature $f \in \mathcal{F}$ must be assigned a subset of $V_{\mathcal{F}}$ corresponding to the elements in the cluster of f. Hence, $\mathcal{X} \equiv \mathcal{P}(V_{\mathcal{F}})$ in Figure 2.3. Since an UL algorithm only needs $V_{\mathcal{F}} \in \mathcal{F}^n$ as an input, a vector of solutions V must be generated in an initialisation phase as shown in Figure 2.4. There are two main approaches for clustering algorithms. Let $V_{\mathcal{F}} = \{f_1, \cdots, f_n\}$ be the input data.

- $V = \{\{f_1\}, \{f_2\}, \cdots, \{f_n\}\}$
  At initialisation, each element is in its own single-element cluster. The purpose of the algorithm is then to merge clusters until a certain termination condition is met.
- $V = \{V_{\mathcal{F}}, V_{\mathcal{F}}, \cdots, V_{\mathcal{F}}\}$
  At initialisation, there is only one single cluster containing every element. The purpose of the algorithm is then to split in different clusters until a certain termination condition is met.

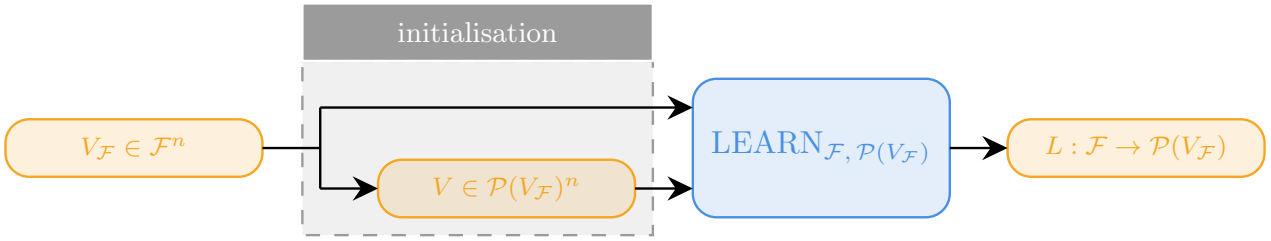


FIGURE 2.4: In unsupervised learning, an initial vector of solutions V is obtained from the input data during an initialisation phase.

2.2.2.3 Reinforcement Learning

Reinforcement Learning (RL) involves an agent evolving in a certain environment. The environment is described by a set of states $\mathcal{S}$, and the agent has at its disposal a set of actions $\mathcal{A}$. At each iteration, the agent chooses an action from a certain state according to a policy. For each action chosen, the state of the environment is updated and a reward is produced. The agent then processes that reward to improve its policy, *i.e.*, next actions chosen will provide better rewards (see Figure 2.5).

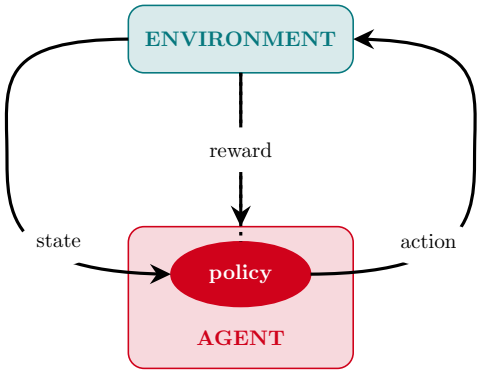


FIGURE 2.5: Reinforcement learning makes an agent evolve in an environment by following the policy to learn.

The purpose of RL is to learn a policy which can be represented by a function $\Pi : \mathcal{S} \to \mathcal{A}$, which is the output of a learning algorithm according to Figure 2.3 with $\mathcal{F} \equiv \mathcal{S}$ and $\mathcal{X} \equiv \mathcal{A}$. For the input, a RL algorithm does not require a sample of states and a sample of actions, but the definition of an environment and an agent. The interaction of the agent with the environment however produces a vector of actions chosen by the agent and a vector of states from which actions have been chosen (see Figure 2.6). This set of states and actions are used in the RL process in order to return a policy.

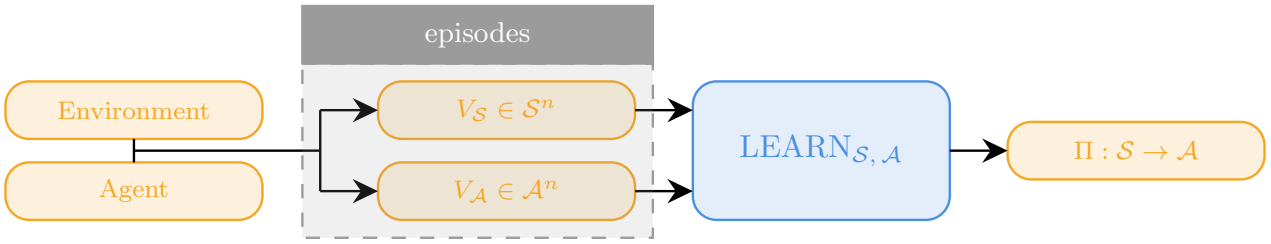


FIGURE 2.6: The elements of $\mathcal{S}$ and $\mathcal{A}$, respectively the states and actions, are obtained during episodes through the interaction of an RL agent with an environment. They are used as an input of the reinforcement learning process.

2.2.2.4 Optimisation component

Let $\mathcal{F}$ and $\mathcal{X}$ represent a task to learn, *i.e.*, the purpose is to return a mapping from $\mathcal{F}$ to $\mathcal{X}$. Given samples of $\mathcal{F}$ and $\mathcal{X}$, the learning process to return a function $L : \mathcal{F} \to \mathcal{X}$ is driven by an optimisation process, as shown in Figure 2.7. Let $\mathcal{L}$ be the set of all possible function going from $\mathcal{F}$ to $\mathcal{X}$. Therefore, the output of the learning process, *i.e.*, $L : \mathcal{F} \to \mathcal{X} \in \mathcal{L}$, can be considered as the output of an optimisation process over $\mathcal{L}$.

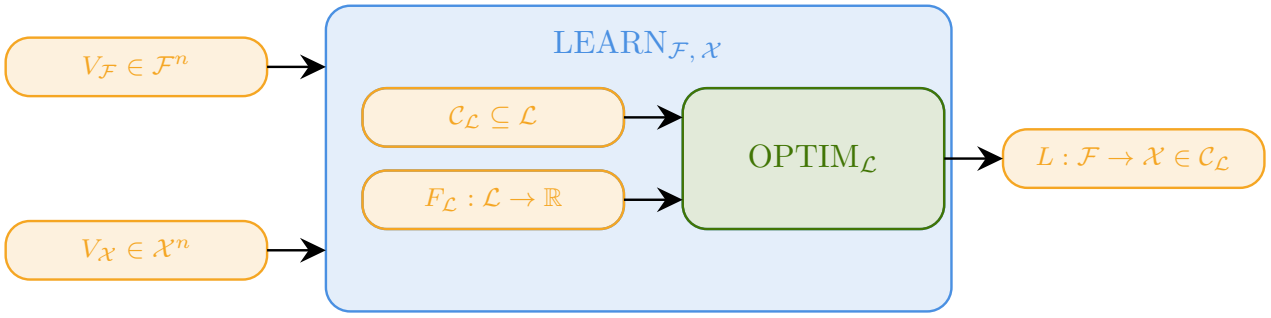


FIGURE 2.7: Optimisation component within the process of learning.

It is necessary to define a feasible solutions space $\mathcal{C}_{\mathcal{L}}$ and an objective function $F_{\mathcal{L}}$ for the inner optimisation process. The space of feasible solutions corresponds to all the functions that can be obtained according to its structure. This depends on the algorithm used. For instance, in the case of a neural network, its architecture defines the space of feasible functions. The objective function depends on the technique used, *i.e.*, supervised, unsupervised or reinforcement learning.

- **SL:** The purpose is to find a function $L : \mathcal{F} \to \mathcal{X}$ so that the distance between $V_{\mathcal{X}}$ and $L(V_{\mathcal{F}}) \equiv \{L(f)\}_{f \in V_{\mathcal{F}}}$ is minimised, *i.e.*, the function well predicts the label of elements of $\mathcal{F}$. Given a distance $d : \mathcal{X}^n \times \mathcal{X}^n \to \mathbb{R}$,

$$L = \underset{L' \in \mathcal{C}_{\mathcal{L}}}{\arg\min}\, F_{\mathcal{L}}(L') = \underset{L' \in \mathcal{C}_{\mathcal{L}}}{\arg\min}\, d(L'(V_{\mathcal{F}}), V_{\mathcal{X}}) \tag{2.1}$$

- **UL:** The quality of the returned function $L : \mathcal{F} \to \mathcal{P}(V_{\mathcal{F}})$ is evaluated by the similarity of elements within each cluster $L(f)$, with $f \in V_{\mathcal{F}}$. Given a similarity score $s : \mathcal{P}(\mathcal{F}) \to \mathbb{R}$,

$$L = \underset{L' \in \mathcal{C}_{\mathcal{L}}}{\arg\min}\, F_{\mathcal{L}}(L') = \underset{L' \in \mathcal{C}_{\mathcal{L}}}{\arg\min} \sum_{f \in V_{\mathcal{F}}} s(L'(f)) \tag{2.2}$$

- **RL:** For each action $a \in \mathcal{A}$ taken from state $s \in \mathcal{S}$, a reward is computed. The purpose is to find a policy $L : \mathcal{S} \to \mathcal{A}$ so that the total reward is maximised. Given a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$,

$$L = \underset{L' \in \mathcal{C}_{\mathcal{L}}}{\arg\max}\, F_{\mathcal{L}}(L') = \underset{L' \in \mathcal{C}_{\mathcal{L}}}{\arg\max} \sum_{s \in V_{\mathcal{S}}} r(s, L'(s)) \tag{2.3}$$

2.3 Hybridisation

The design of an optimisation or learning process relies on empirical choices, *e.g.*, a parameterisation, the definition of an optimiser, the architecture of a neural network. Some works then aim at improving the efficiency of an optimisation/learning algorithm by removing this empirical aspect with an automation process. The idea is to use another level, referred to as the high level, of optimisation/learning from which results an element used by the main optimisation/learning process, referred to as the low level. We refer to this methodology as hybridisation. The four possible hybridisations are shown in Table 2.1. The column and row headers give respectively the high-level and low-level processes. For instance, a learning-to-optimise technique uses optimisation as at the low level and learning at the high level.

low level \ *high level*	**Optimisation**	**Learning**
Optimisation	Optimise Optimisation (2.3.1)	Learning to Optimise (2.3.2)
Learning	Optimise Learning (2.3.3)	Learning to Learn (2.3.4)

TABLE 2.1: Different possible hybridisations with learning and optimisation.

When optimisation is used at the low level (first row in Table 2.1), the high-level algorithm can be used for the definition of the optimiser, the generation of initial solutions or the definition of the objective function (all components are displayed in Figure 2.2).

Since a learning algorithm relies on an inner optimisation process (see Section 2.2.2.4), the high-level algorithm usually occurs there when learning is used at the low level (second row in Table 2.1). All components from optimisation can be targetted by the high-level algorithm.

When optimisation is used at the high level (first column in Table 2.1), the solution returned is directly used as an input of the low-level algorithm.

When learning is the high-level algorithm (second column in Table 2.1), the returned function can be directly used as a component of the low-level process. That function can also be called to return an element used within the low-level algorithm.

The remainder of this chapter consists in presenting the four possible hybridisations (presented in Table 2.1)using the formalism presented in the previous chapter. The objective is to provide a better understanding of how both layers of optimisation/learning interact.

2.3.1 Optimise Optimisation

Two main techniques aim at optimising another optimisation process. The high-level algorithm can be used to generate good initial solutions for the low-level one (see Section 2.3.1.1), or to optimise the parameterisation of the optimiser at the low level (see Section 2.3.1.2).

2.3.1.1 Generation of initial solutions

The first way to improve an optimisation algorithm is to start with better initial solutions (see Figure 2.8). This choice is decisive for the local optimum to which the algorithm will converge. For that purpose, a high-level optimisation process (green box on the left) can produce one or several solutions which are used as initial solution(s) for the low-level optimisation process (green box on the right). In that case, both optimisation algorithm work over the same space $\mathcal{X}$. They also share the space of feasible solutions $\mathcal{C}_\mathcal{X}$ and objective function $F_\mathcal{X}$.

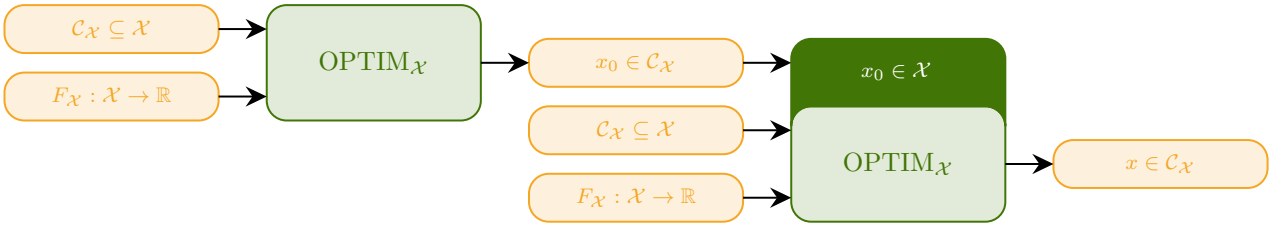


FIGURE 2.8: Using optimisation to obtain the initial solutions of another optimisation process.

When a perturbative heuristic is used at the low level, the initial solution must be feasible, *i.e.*, $x_0 \in \mathcal{C}_\mathcal{X}$. It can therefore be obtained by a first constructive heuristic. Likewise, in case of a population-based metaheuristic, the first individuals can be populated by solutions obtained with heuristics beforehand.

2.3.1.2 Hyper-parameterisation

Another way to improve an optimisation process is to find an optimal parameterisation for the optimiser (see Figure 2.9). This technique is referred to as hyper-parameterisation. Let $\mathcal{P}$ be the set of all possible parameterisations for the optimiser of the low-level optimisation (green box on the right). The latter can then be seen as a function $O : \mathcal{X} \times \mathcal{P} \to \mathcal{X}$. Consequently, the high-level algorithm (green box on the left) aims at optimising over $\mathcal{P}$, so that the element returned is the parameterisation used by the optimiser at the low level. The optimisation over $\mathcal{P}$ then requires a definition of constraints $\mathcal{C}_\mathcal{P}$ and of an objective function $F_\mathcal{P}$. The constraints could be simply bounds to the values of parameters. Given a parameterisation $p \in \mathcal{P}$, the objective value $F_\mathcal{P}(p)$ is computed by applying the low-level optimisation process on different instances, *i.e.*, over different spaces $\mathcal{X}$, using the parameterisation p.

$$F_\mathcal{P}(p) = \sum_\mathcal{X} F_\mathcal{X}\left(\text{OPTIM}_\mathcal{X}\left(\mathcal{C}_\mathcal{X}, F_\mathcal{X}, p\right)\right) \tag{2.4}$$

where $\text{OPTIM}_\mathcal{X}\left(\mathcal{C}_\mathcal{X}, F_\mathcal{X}, p\right)$ is the solution provided by the low-level optimisation process over $\mathcal{X}$ by using p.

For instance, Stolfi et al. [2020] designed a coevolutionary algorithm to optimise the coverage of an area with a swarm of Unmanned Aerial Vehicles (UAVs), which can be assimilated to the low-level optimisation process. That algorithm is based on repulsive pheromone that UAVs drop on their way, and relies on three parameters: the amount of pheromone left by each vehicle ($\tau_a \in [0.0, 1.0]$); the radius of the area around the vehicle on which

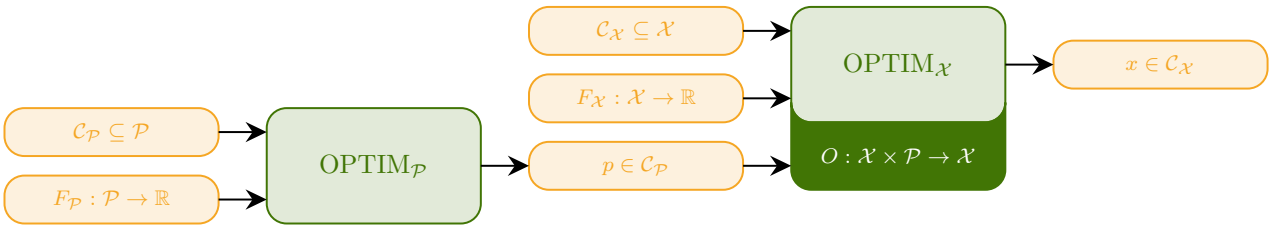


FIGURE 2.9: Using optimisation to obtain a parameterisation for the optimiser of another optimisation process.

the pheromone are dropped ($\tau_r \in [0.5, 2.5]$); the distance from which UAVs can detect pheromone ($\tau_d \in [1, 10]$). The authors use a Genetic Algorithm (GA) to find the optimal value for τ_a, τ_r and τ_d, which is the high-level optimisation algorithm. In that case, a tuple (τ_a, τ_r, τ_d) is an element of $\mathcal{P} \equiv \mathbb{R}^2 \times \mathbb{Z}$, and the space of feasible solutions is depicted by the bounds for each parameter, hence $\mathcal{C}_\mathcal{P} \equiv [0.0, 1.0] \times [0.5, 2.5] \times [1, 10]$.

2.3.2 Learning to Optimise

The vast majority of techniques using a high-level learning process for a low-level optimisation algorithm can be split into two categories. Learning can first be used to approximate the objective function of the low-level optimisation process (see Section 2.3.2.1). On the other hand, learning can also be applied to the optimiser at the low level (see Section 2.3.2.2). A state of the art of works using "machine learning at the service of metaheuristics" is provided by Karimi-Mamaghan et al. [2022]. Most of the works presented there can thus be assimilated to the formalism presented in this section.

2.3.2.1 Approximation of the objective function

One way to use learning to improve an optimisation process is to approximate the objective function (see Figure 2.10). In some case, the latter can indeed be time-consuming to evaluate on a solution. Where appropriate, the objective function can be replaced by the function returned by a learning process. Since that function must map a solution, *i.e.*, an element from $\mathcal{X}$, to a real value, a regression algorithm must be used (see Section 2.2.2.1). It needs as an input a sample of solutions $V_\mathcal{X}$ and a real-value vector $V_\mathbb{R}$.

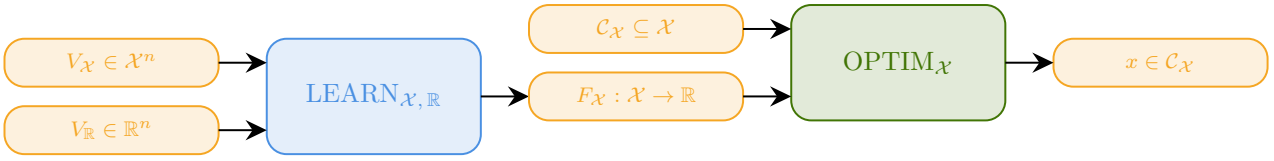


FIGURE 2.10: Using learning to approximate the objective function of an optimisation process.

Consider an optimisation process over a space $\mathcal{X}$ with $F_\mathcal{X}$ as an objective function, and $F_\mathcal{X}(x)$ costly to evaluate for a solution $x \in \mathcal{X}$. The idea is to provide a relatively small amount of evaluations as an input of a regression algorithm. The latter high-level process then requires a sample of solutions, noted $V_\mathcal{X}$, on which the evaluation has been done, and $V_\mathbb{R} \equiv \{F_\mathcal{X}(x)\}_{x \in V_\mathcal{X}}$. That regression algorithm returns a mapping $L : \mathcal{X} \to \mathbb{R}$ which can be used as the objective function of the low-level optimisation process. As an example, Zheng et al. [2019] use high-level random forests as surrogates to approximate multi-objective evaluation functions.

2.3.2.2 Improvement of the optimiser

Learning can also be aplied to improve the optimiser of an optimisation process (see Figure 2.11). The term "hyper-heuristic" is used to describe these techniques. In that case, the optimiser requires an additional element to obtain the next solution. This element is obtained from the current solution by calling the function returned by the high-level learning process. Let $\mathcal{Y}$ be the set of additional elements for the optimiser. The latter is then defined by $O : \mathcal{X} \times \mathcal{Y} \to \mathcal{X}$. Given a current solution $x \in \mathcal{X}$, the next solution in the optimisation process is given by $O(x, L(x))$ where $L : \mathcal{X} \to \mathcal{Y}$ is the function returned by the learning process.

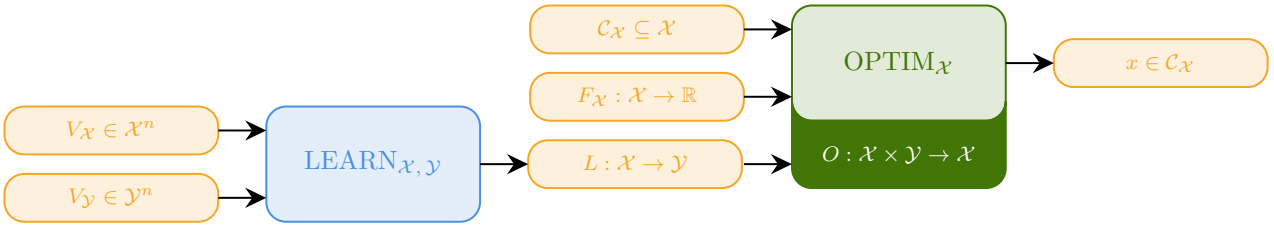


FIGURE 2.11: Using learning to improve the optimiser of an optimisation process.

Let a constructive heuristic be the optimisation process to tackle problem P1. A solution, *i.e.*, an element of $\mathcal{X}$, is a path which is feasible if it goes from A to B. The constructive heuristic, starting from the initial solution $x_0 = [\text{A}]$ (a sequence containing only location A), then consists in adding a location to the current solution until B is added. A learning algorithm can be used to help the optimiser to choose the location to add into the current solution. For that purpose, the learning process is designed to return a mapping $L : \mathcal{X} \to \mathcal{Y}$ where Y is the set of all locations. Given a current solution $x = [\text{A}, \cdots] \in \mathcal{X}$, calling $L(x)$ then returns the new location to add into x. The new solution $x' = O(x, L(x)) = [\text{A}, \cdots, L(x)] \in \mathcal{X}$ is thus obtained. In a previous work, we use this principle to generate heuristics for the Travelling Salesman Problem (TSP) [Duflo et al., 2019a]. The idea, common to other such works, is to assimilate a heuristic to its optimiser and therefore the function L which is returned by the high-level learning process. The search space of the high-level algorithm then becomes a space of heuristics.

2.3.3 Optimise Learning

Techniques aiming at optimising a learning algorithm mainly consists in applying optimisation to the inner optimisation component of the low-level learning process. Similarly to the optimise-optimisation technique described in Section 2.3.1.2, a hyper-parameterisation can occur within the inner optimisation component (see Section 2.3.3.1). On the other hand, the high-level optimisation process can be used to define the constraint of the search space within the low-level learning algorithm (see Section 2.3.3.2).

2.3.3.1 Hyper-parameterisation

One way to apply optimisation on a learning process is to search for a parameterisation for the optimiser of the inner optimisation process (see Figure 2.12). Since $\mathcal{L}$ is the space of all function mapping an element of $\mathcal{F}$ to an element of $\mathcal{X}$, the inner optimiser can be regarded as a function $O : \mathcal{L} \to \mathcal{L}$. Similarly to what we presented in Section 2.3.1.2, with $\mathcal{P}$ the set of all parameterisations for that optimiser, the latter can be seen as $O : \mathcal{L} \times \mathcal{P} \to \mathcal{L}$. The parameterisation $p \in \mathcal{P}$ chosen then results from a high-level optimisation process over $\mathcal{P}$.

For instance, if a learning algorithm is based on a neural network representing the function mapping an element of $\mathcal{F}$ to an element of $\mathcal{X}$, the purpose of the inner optimisation process is to find the optimal weights within

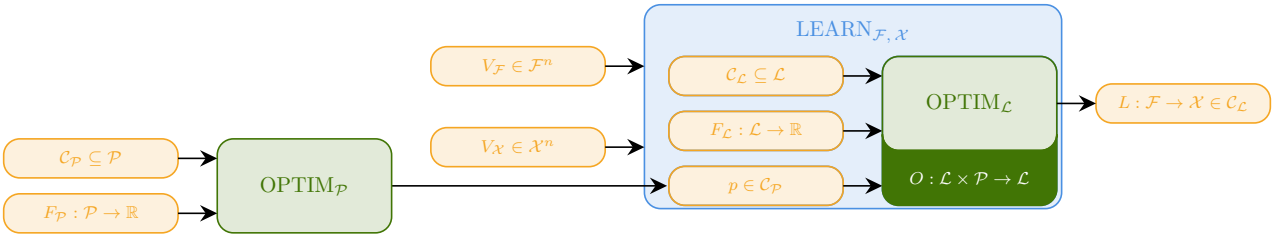


FIGURE 2.12: Using optimisation to obtain a parameterisation for the optimiser of the inner optimisation component of a learning process.

the neural network. In that case, the optimiser is the technique used to update those weights, *e.g.*, a Stochastic Gradient Descent (SGD). In that case, the value of the learning rate in the SGD can be obtained with a high-level optimisation process over $\mathcal{P}$ where $\mathcal{P}$ is the domain of values for the learning rate.

2.3.3.2 Constraint of the search space

Optimisation can also be used to define the constraints of the inner optimisation process (see Figure 2.13). In that case, the high-level optimisation process must return a subset of $\mathcal{L}$, hence its definition over $\mathcal{P}(\mathcal{L})$, the power set of $\mathcal{L}$. The objective function at the high level then evaluates the quality of the function returned by the low-level learning process.

$$F_{\mathcal{P}(\mathcal{L})}(c) = F_{\mathcal{L}}\left(\text{OPTIM}_{\mathcal{L}}(c, F_{\mathcal{L}})\right) \tag{2.5}$$

where OPTIM$_{\mathcal{L}}$ $(c, F_{\mathcal{L}})$ is the function obtained by the low-level learning process where c describes the space of functions obtainable.

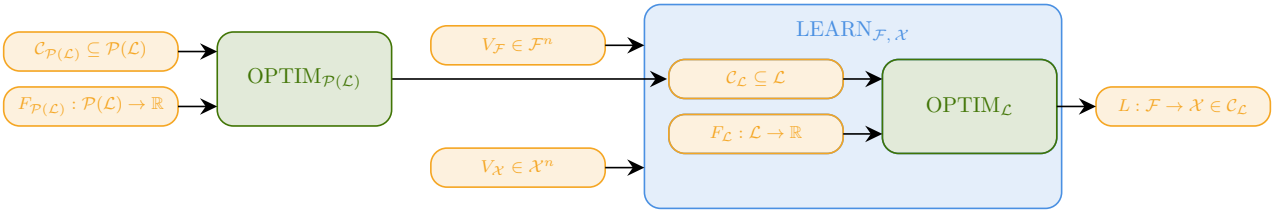


FIGURE 2.13: Using optimisation for constraining the inner optimisation component of a learning process.

For instance, if a learning process is based on a Neural Network (NN), an element of $\mathcal{L}$ is defined by the topology of the NN. Constraining the search space $\mathcal{L}$ usually happens with an hyper-parameterisation where hyper-parameters define the topology of the NN, for instance the number of hidden layers and neurons per layer. This approach is proposed by Qolomany et al. [2017] where they used Particle Swarm Optimisation (PSO) at the high level. In that case, the space of feasible solutions at the high level ($\mathcal{C}_{\mathcal{P}(\mathcal{L})}$ in Figure 2.13) can be depicted by bounds to both variables. Moreover, given a number of layers and neurons per layer, the objective function ($F_{\mathcal{P}(\mathcal{L})}$ in Figure 2.13) evaluates the accuracy of the low-level learning process by using the NN with the specified number of layers and neurons per layer.

2.3.4 Learning to Learn

Using learning to improve another learning process is referred to as meta-learning. Existing techniques are surveyed by Vanschoren [2019] and can then be described by the formalism presented in this section. It mainly consists in initialising a model by another one according to the similarity between both tasks (see Figure 2.14).

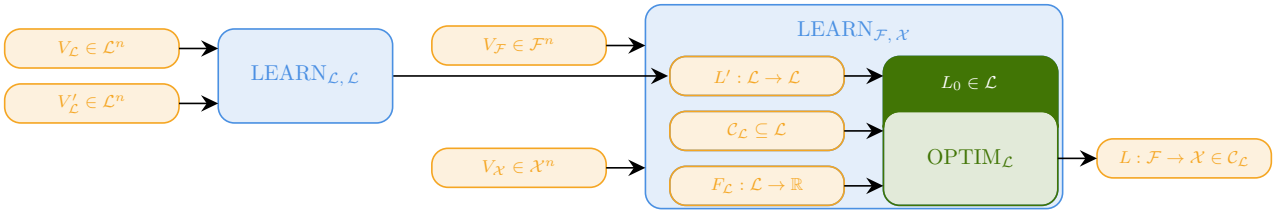


FIGURE 2.14: Using a learning process for initialising another learning process.

The high-level learning algorithm (blue box on the left) is designed to map a learning model to another one. The returned function is then called to initialised the low-level learning process (blue box on the right).

Learning processes operating for different tasks can rely on a Neural Network (NN) with a same architecture. An element of $\mathcal{L}$ then corresponds to an assignment to the weights of that NN. Given a learning algorithm based on a similar NN, the idea is to relate the corresponding task to previous ones. A task is therefore equivalent to an element of $\mathcal{L}$, as a result of the corresponding learning process. A clustering algorithm (see Section 2.2.2.2) can be used for that purpose. It will return a function $L' : \mathcal{L} \to \mathcal{P}(\mathcal{L})$, which can be called on the new task and therefore returns a set of tasks. Among them, one can be chosen to initialise the weights of the NN, or an average of weights can be done, which produces $L_0 \in \mathcal{L}$.

2.4 Conclusion

In this chapter, a formal representation of the optimisation and learning processes has been introduced. It has been shown that this formalisation is independent of the specific optimisation or learning technique used. We first defined these two concepts with the format of their inputs and output. Given a problem to solve and a space of solutions, using an optimisation process will directly return a feasible solution according to a space of feasible solutions and an objective function. On the other hand, a learning process will return a function which aims at mapping a space of features to the space of solutions. To solve the problem, the function must then be called on the feature which can be assimilated to the instance of the problem to get a solution. The process of learning is based on a training data, *i.e.*, a sample of features and a sample of corresponding solutions. We then identified inner components for both optimisation and learning processes. We show that any optimisation algorithm is based on the definition of one or several initial solutions and an optimiser aiming at mapping one or several solutions to another one(s). A learning process is meanwhile driven by an entire optimisation process.

In the second part of this chapter, we explored different hybridisation techniques permitting to combine optimisation and learning at different levels. The purpose is to use a process of optimisation/learning (at the high level) to improve the efficiency of another process of optimisation/learning (at the low level). The identification of inner components made in the first part of this chapter becomes useful to determine which components of the low-level process are involved according to the hybrid technique.

- When the low-level process is optimisation, the helper process mainly happens within its optimiser component. It comes to hyper-parameterisation or hyper-heuristics when the high-level process is optimisation or learning respectively.
- Since a learning process incorporates an optimisation component, the latter is usually involved when learning is used as low-level process. Another level of learning can be used to initialise the model according to similarities with previous models, while an optimisation process can help defining the searching space of functions.

This work consists in using reinforcement learning for generating heuristics for an optimisation problem. It thus falls within the area of learning to optimise (Figure 2.11), and more particularly within hyper-heuristics. The next chapter then provides a state of the art of hyper-heuristics, with a special focus on techniques based on reinforcement learning.

Chapter 3

Hyper-Heuristics and Reinforcement Learning

Contents

3.1 Introduction

This PhD work takes place in the context of "learning to optimise", and specifically hyper-heuristics which can be equated to the category of applying a learning process to improve the optimiser of an optimisation process (see Section 2.3.2.2 in Chapter 2). A first presentation of hyper-heuristics is provided in Section 3.2 along with a state of the art. In our case, we use Reinforcement Learning (RL) as a learning process. A particular focus is given to techniques relying on RL. A detailed state of the art is then provided in Section 3.3.

3.2 Hyper-heuristics

A hyper-heuristic consists in an algorithm performing a searching process in a space of heuristics. Initially described by Cowling et al. [2001] as a "heuristic to choose a heuristic", the purpose was to select the right heuristic to execute on a given instance. The idea is to overcome the limitation of the No Free Lunch Theorem (NFLT) stating that "if an algorithm performs well on a certain class of problems then it necessarily pays for that with degraded performance on the set of all remaining problems" [Wolpert and Macready, 1997]. If one heuristic cannot perform well on every instance, one can choose which one to apply among a set of predefined

heuristics. Such hyper-heuristics correspond to selective approaches (detailed in Section 3.2.1). They are to be put in perspective with generative approaches (detailed in Section 3.2.2). Heuristic generation was more recently introduced as a technique that does not rely on a pre-existing set of heuristics to be designed. Instead of providing predefined heuristics, the user provides components of the problem that the hyper-heuristic uses to build a new heuristic. These techniques have the advantage of covering a wider space of heuristics by reducing the limitation of human imagination. Heuristic selection and heuristic generation constitute the main two categories of hyper-heuristics. However, a more detailed classification has been proposed by Burke et al. [2013] and is depicted in Figure 3.1.

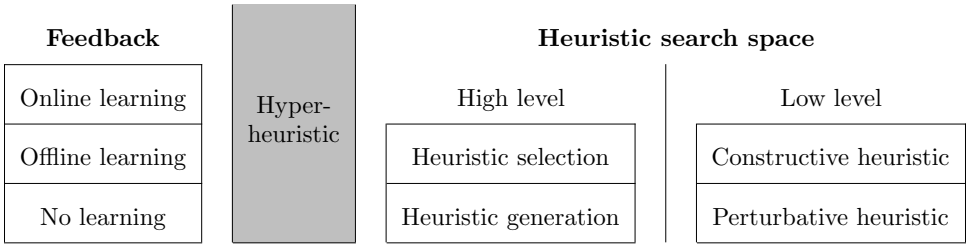


FIGURE 3.1: Classification of hyper-heuristics proposed by Burke et al. [2013]

The heuristic search space is divided into two levels: the high-level algorithm performing the search process, and the nature of the low-level heuristics.

High level The high-level algorithm performing the search process is categorised as a heuristic selection or a heuristic generation. That category determines the format of the search space. In selective approaches, the space is composed of pre-existing heuristics. In generative approaches on the other hand, a template of heuristics is defined and comprises a dynamic part defining the search space.

Low level Hyper-heuristics are also distinguished whether the search space contains constructive or perturbative heuristics. Both types of heuristic are defined and categorised in Section 2.2.1.1 in Chapter 2. There we provide a generic description of constructive and perturbative heuristics based on an optimiser, respectively in Algorithms 2.1 and 2.2. That optimiser constitutes the dynamic part within the template of heuristics in generative hyper-heuristics.

Feedback Independently of the heuristic search space, hyper-heuristics are also divided according to the nature of their feedback. The feedback is the information that the high-level algorithm gets from the space of low-level heuristics and which is used for the searching process. An online learning means that it takes place while the hyper-heuristic is executed on instances of a problem. RL is mainly used for that purpose. A hyper-heuristic using an offline learning performs the searching process beforehand. A set of training instances must be provided in order to select or generate a heuristic which can then be executed on future instances. In that case, a Genetic Algorithm (GA) or Genetic Programming (GP) is often used at the high-level. The hyper-heuristic can also use no feedback if the high-level algorithm is simply a heuristic (instead of a metaheuristic or a machine learning algorithm).

In the remainder of this section, a state of the art of hyper-heuristics is presented. A distinction is made between selective and generative approaches. Techniques based on RL are omitted in this section, since there is an emphasis on hyper-heuristics based on RL presented in Section 3.3.

3.2.1 Selective approaches

Most of hyper-heuristics in the literature are based on selective approaches. It can be explained by the fact that they were the first introduced [Cowling et al., 2001] and they are simpler to design than generative approaches. Considering an optimisation problem with several (meta)heuristics already existing to tackle it, there are high chances that a selective hyper-heuristic using that pool of low-level heuristics will have better performance than executing one of them only.

Recent works using selective approaches do not simply select one low-level heuristic among the provided pool, but try to combine them. In that case, an Evolutionary Algorithm (EA) is used where each individual's chromosome represents a sequence of heuristics to apply. For each individual, the fitness is computed by applying the sequence of heuristics to a set of training instances and summing the objective values obtained. Lin et al. [2020] use that principle to tackle the Multi-Skill Resource Constrained Project Scheduling Problem (MS-RCPSP). They provided ten low-level perturbative heuristics that each GP individual combine within a tree representation. In [Hao et al., 2021], authors propose a framework of hyper-heuristics for tackling multiple optimisation problems by using a high-level Evolutionary Multitasking Algorithm (EMA). Each individual's chromosome is a sequence of heuristics which is executed on different tasks, *i.e.*, to tackle different graph-based problems. Elaziz et al. [2020] focus on the problem of finding thresholds in an image segmentation. The authors utilise several existing metaheuristics which are combined by individuals within a EA population.

3.2.2 Generative approaches

In generative hyper-heuristics, defining the space of low-level heuristics is a harder task than in selective hyper-heuristics. Given an optimisation problem to tackle, the main way to define the space of low-level heuristics is to extract the information common to several heuristics and to make it a template of heuristics. The dynamic part of such a template represents the searching space. For instance, the most common way of addressing it is to make a template of greedy heuristics (constructive heuristics then) where the dynamic part is a scoring function. A Genetic Programming (GP) algorithm is then a straightforward approach to use at the high-level since the tree structure of individuals then represents a function [Burke et al., 2009].

In some previous work, we used that principle to design a hyper-heuristic based on GP to generate constructive heuristics for the Travelling Salesman Problem (TSP) [Duflo et al., 2019a]. The low-level heuristics consist in greedy ones where nodes are added into the solution at each iteration. The choice of nodes is based on the ranking induced by the scoring function to learn. In [Kieffer et al., 2020], the authors deal with the Bi-level Cloud Pricing Optimisation Problem (BCPOP). They use a GP hyper-heuristic to generate constructive heuristics for low-level instances of BCPOP. The low-level greedy heuristics start by selecting all bundles of the instance, and remove one item at each iteration according to the scoring function. Guizzo et al. [2020] designed a hyper-heuristic based on Grammatical Evolution (GE) to generate a mutant reduction strategy. Each individual has a genotype which is a sequence of rules from the grammar. It is then translated into a tree-structure phenotype which represent the mutant reduction strategy.

Even though most generative hyper-heuristics relying on an offline learning use GP as a high-level algorithm, some works considered learning the scoring function with a neural network. Kieffer et al. [2022] designed a hyper-heuristics to tackle the Multi-dimensional 0-1 Knapsack Problem (MKP). The authors use the tree structure of GP individuals to represent a scoring function. In that case, the latter function evaluates the relevance of putting a certain object into a sack. They however do not use a GP algorithm to generate the tree, as a classical hyper-heuristic would do [Drake et al., 2014]. They indeed designed a Recurrent Neural Network

(RNN) to build the tree structure depicting the scoring function. Starting from the root of the tree, each step of the RNN returns a node from a set of operators and terminals.

3.3 Hyper-Heuristics based on Reinforcement Learning

As mentioned earlier, RL is the main technique used as a high level for hyper-heuristic relying on online learning. RL has indeed become the most widely used heuristic selection method in recent literature. This is due to the convenience of representing low-level heuristics as RL actions. More detail is provided in Section 3.3.1 where we present RL-based hyper-heuristics relying on heuristic selection. It is also interesting to note that recent works from RL community propose techniques comparable to hyper-heuristic (without mentioning it) [Mazyavkina et al., 2021]. These works constitute most of existing usage of RL in generative approaches. We present them in Section 3.3.2.

3.3.1 Selective approaches

The purpose of RL is to learn a policy comparable to a function $\Pi : \mathcal{S} \rightarrow \mathcal{A}$ with $\mathcal{S}$ the set of states and $\mathcal{A}$ the set of actions (see Section 2.2.2.3). Given an optimisation problem to tackle, the basic approach when using RL is to map the pool of existing heuristics to $\mathcal{A}$. The goal of choosing an action at each RL iteration then becomes choosing a heuristic in a selective hyper-heuristic. The RL states represent instances of the problem in case of constructive heuristics, so that choosing an action will produce a solution for those instances, and feasible solutions for perturbative heuristics, so that choosing an action will modify those solutions. Figure 3.2 shows the typical workflow followed by selective hyper-heuristics with perturbative low-level heuristics.

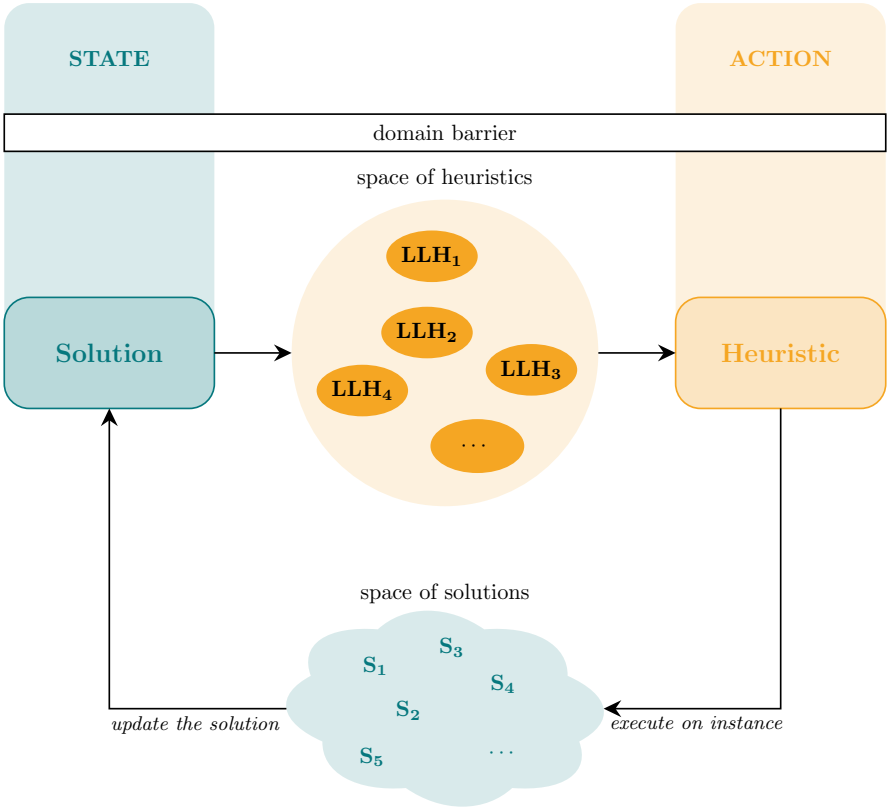


FIGURE 3.2: Overview of selective hyper-heuristics based on RL with perturbative low-level heuristics.

Pylyavskyy et al. [2020] provides four perturbative heuristics to tackle the "Kiwi.com" problem. The latter problem "seeks to find the best possible flight routes between specifically given areas in order to minimise the travelling cost". Their hyper-heuristic consists in giving a score to each low-level heuristic. At each iteration, the heuristic with the best score is selected and applied on an instance so that its score is updated according to the improvement of the result obtained with the newly obtained solution. Lassouaoui et al. [2020] designed a hyper-heuristic with the same principle. They use a Thompson Sampling algorithm to select among six perturbative low-level heuristics to tackle the problem of feature selection. In [Zhao et al., 2021], the authors diversify the way to select actions, *i.e.*, low-level heuristics. They alternate between single-point and a multi-point search. With the single-point search, heuristics are selected one or two at a time. On the other hand, the multi-point search employs a GA where individuals are encoded with a set of low-level heuristics. The individual retained defines the space of low-level heuristics for the single-point search. The multi-point search is called at initialisation and when a single-point search does not change the state of the solution in continuous time.

The aforementioned selective approaches using RL have the drawback of not using the information of the current solution when choosing which low-level heuristic to apply. In contrast, Zhang et al. [2022] propose a framework of selective hyper-heuristics based on deep RL. They work with a Deep Q-Network (DQN), so the last layer contains one neuron per action, *i.e.*, per low-level perturbative heuristic provided. The first layer describing the state is provided by transforming a solution into a vector. For each problem to tackle, expert knowledge is thus required to select the feature to use to encode the state. In [Qin et al., 2021], the authors designed a hyper-heuristic which selects a population-based metaheuristic to evolve a population of solutions for the Heterogeneous Vehicle Routing Problem (HVRP). They use a Convolutional Neural Network (CNN) for that purpose. The state, *i.e.*, the input of the CNN, is a vector containing objective values of all individuals from the current population and a matrix indicating the difference between each individuals.

3.3.2 Generative approaches

Generative approaches using RL are more recent and more scarce in the literature. There is however a classical way to design such hyper-heuristics, especially for generating constructive heuristics (depicted in Figure 3.3). Given an optimisation problem to tackle, a solution is constituted of components from the instance (x_i in Figure 3.3). In general, instances are represented as graphs, and a solution as a sequence/set of nodes/edges from that graph. With such a representation, RL states can be assimilated to solutions and the set of elements x_i represents RL actions.

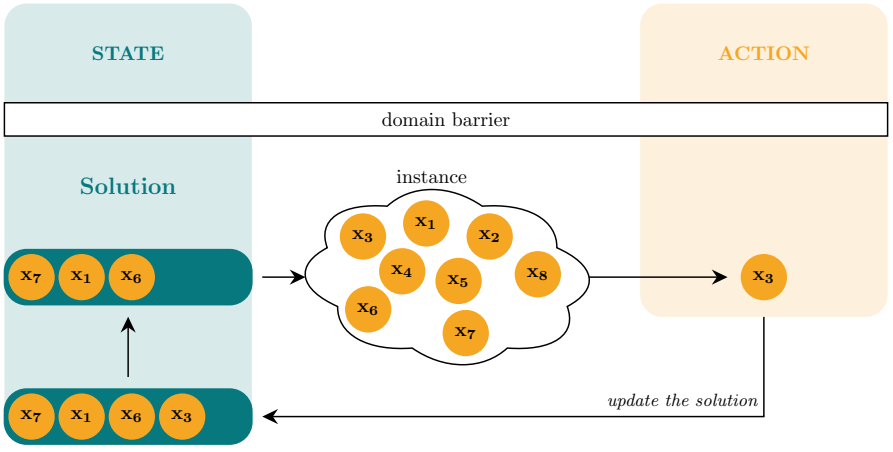


FIGURE 3.3: Overview of generative hyper-heuristics based on RL with constructive low-level heuristics.

Applying the above principle on graph-based problems has recently shown a growing interest. For that purpose, a Graph Neural Network (GNN) is used to encode the information of a graph, and consequently RL states and actions, with state variables assigned to nodes and edges. Khalil et al. [2017] were the first developing this idea by using their GNN named structure2vec (S2V) [Dai et al., 2016]. They thereby generate greedy heuristic for three graph-based optimisation problems: Minimum Vertex Cover (MVC), Maximum Cut (MAXCUT) and Travelling Salesman Problem (TSP). This work paved the way to RL generating heuristics for graph-based problems. Kool et al. [2022] use the same idea to generate heuristics for the TSP and multiple routing problems. They use a Graph Attention Network (GAT) [Veličković et al., 2018] as a GNN. In [Manchanda et al., 2020], authors also tackle different graph-based problem: Maximum Coverage Problem (MCP), MVC and Influence Maximisation (IM). Their hyper-heuristic is based on a Graph Convolutional Network (GCN) [Hamilton et al., 2017].

Our work is also in the continuity of the research made by Khalil et al. [2017]. We extracted information from all works made in this area and notice that there is much similarity within the structure of proposed hyper-heuristics. We thus design a general model which can be applied on a wide range of optimisation problems based on a graph structure. A strong point of our model is its modularity. For instance, it is not limited to the usage of one GNN. The hyper-heuristic is designed to take any GNN in consideration. Moreover, the hyper-heuristic also works for problems with a multi-objective aspect. We also think that a greedy heuristic generated by such an algorithm cannot be competitive for classical combinatorial problems which have already been fully reviewed. We thus designed our model to be applied on dynamic instances of a problem. If a problem involves multiple agents, it also considers generating distributed heuristics. We have already applied our model in the context of the coverage of an area by a swarm of Unmanned Aerial Vehicles (UAVs) [Duflo et al., 2020a,b, 2021, 2022a,b]. In these work, a multi-objective graph-based optimisation problem has been designed for the coverage of a swarm of UAVs remaining connected.

3.4 Conclusion

A hyper-heuristics consists in a High-Level Algorithm (HLA) performing a searching process in a space of Low-Level Heuristics (LLHs). The purpose is to automate the design of an algorithm to tackle a specific optimisation problem. This automation aims at leverage the limitations stated by the No Free Lunch Theorem (NFLT). When designing an algorithm to tackle a problem, we are indeed led by a set of known instances to solve. The efficiency of the produced algorithm is paid by worse performance when applied on unknown instances. The idea of hyper-heuristics is thus to extract the complex information which makes an algorithm efficient or not on an instance.

Hyper-heuristics have been deeply surveyed by Burke et al. [2013]. The proposed classification has thereafter been taken over by all works contributing in this area. Hyper-heuristics are classified according to the nature of the HLA and the LLHs (see Figure 3.1). Since this PhD work is based on hyper-heuristics using Reinforcement Learning (RL) as a HLA, we provide a special focus on the usage of RL in that area.

We show that our work goes beyond related works by filling a lack of generality. Even though most of hyper-heuristics based on RL have a similar behaviour, they are indeed redesigned from scratch. We thus propose a model of generic hyper-heuristic based on RL, presented in next chapter, aiming at tackling any compatible optimisation problem with the same algorithm generation.

Part II

Learning Optimisation Algorithms over Graphs

Chapter 4

Algorithm Learner for Graph Optimisation problems (ALGO)

Contents

4.1 Introduction

Hyper-heuristics have been used in order to overcome the challenge for heuristics to tackle unknown instances. That difficulty lies in the diversity of instances and the large number of parameters to set for a heuristic. Given a problem and a set of known instances, designing an efficient algorithm to tackle known instances does not guarantee an efficient behaviour for unknown ones. The purpose of a hyper-heuristic is therefore to extract useful information from known instances to automate the design of a heuristic.

The design of a hyper-heuristic is based on two components: the High-Level Algorithm (HLA) and the space of Low-Level Heuristics (LLHs), both interacting as depicted in Figure 4.1. The HLA (in blue) performs a search process by manipulating representatives of heuristics from the space of LLHs (in green). In selective approaches, a representative is simply a reference to a LLH. In generative approaches, they are usually scoring functions (more details are provided in the state of the art in Section 3.2.2). If GP is used as a HLA, representatives are

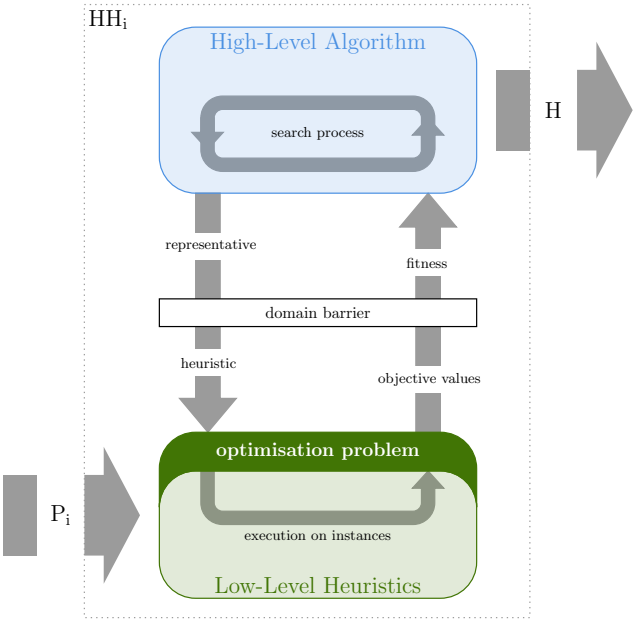


FIGURE 4.1: Overview of the process of hyper-heuristics. An hyper-heuristic HH_i returns a heuristic H given an optimisation problem P_i. The space of low-level heuristics of HH_i must be defined according to P_i.

trees. If RL is the HLA, a representative can be a Q-table or a neural network. Each representative is given a fitness by applying the corresponding heuristic on instances from the problem to tackle. The HLA finally returns a representative, *i.e.*, a heuristic.

Let P_i be an optimisation problem and HH_i a hyper-heuristic to tackle it. It is not possible to use HH_i on a new problem P_j. Indeed while the HLA of HH_i can remain unchanged as it is problem (over the domain barrier in Figure 4.1), the space of LLHs must be redefined. This thus leads to a new hyper-heuristic HH_j. In order to alleviate this limitation, we propose Algorithm Learner for Graph Optimisation problems (ALGO), a novel framework that aims at standardising the design of hyper-heuristics so that the same algorithm can be used to generate a heuristic for any given problem (see Figure 4.2).

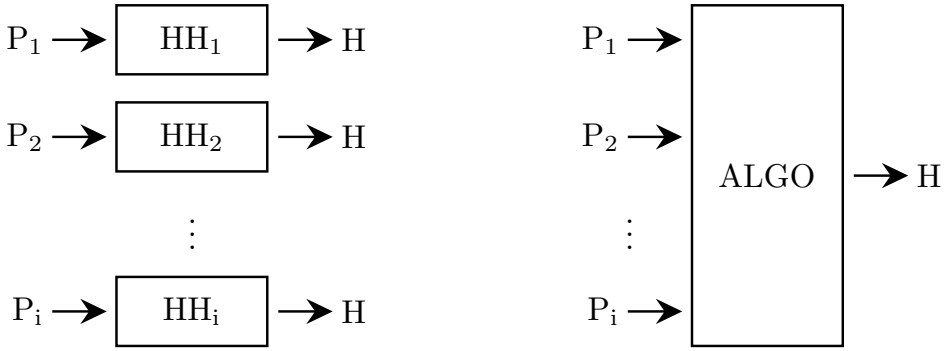


FIGURE 4.2: On the left side, for each problem P_i, a hyper-heuristic HH_i must be defined to generate a heuristic H. On the right side, ALGO does not need to be redefined to tackle different problems.

In ALGO, each problem is initially expressed as a generic optimisation problem, so called a ALGO-Friendly Optimisation Problem (AFOP), via an overriding process (see Figure 4.3). A single hyper-heuristic can then be defined for tackling an AFOP. The overriding process consists in implementing any compatible problem as an AFOP. That process is wanted to be convenient an intuitive. A strength of ALGO relies on the fact that expressing a new problem as an AFOP is easier than redefining a new space of LLHs.

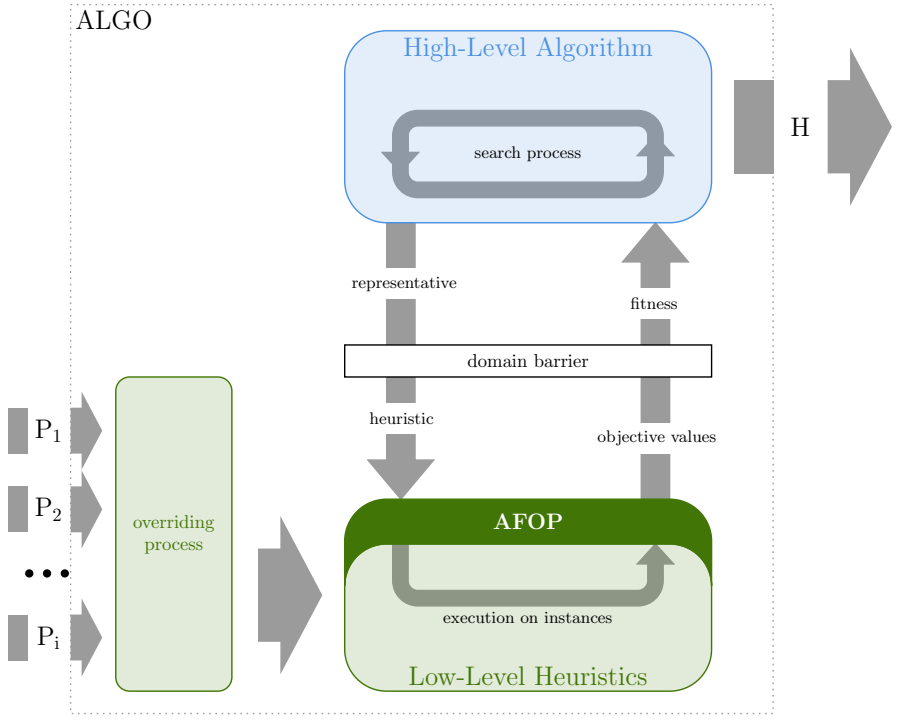


FIGURE 4.3: Overview of the process of ALGO. The space of low-level heuristics is defined for a generic AFOP. An overriding process is used in order to implement any compatible problem as an AFOP.

The hyper-heuristic structure in ALGO (right part of Figure 4.3) is overviewed in Figure 4.4. ALGO uses RL as a HLA. Since it is independent of the problem, the top part of Figure 4.4 is identical to Figure 2.5, *i.e.*, it represents the workflow of a classical RL algorithm. On the other side of the domain barrier, LLHs are defined for an abstract AFOP. The latter problem is based on a graph structure in order to make this abstraction possible (more details are provided in Section 4.2.1 where an AFOP is defined). The purpose of RL being to learn a policy determining which action to choose from a given state, it is transformed into choosing which node to add into a current solution. Given a graph representing an AFOP instance, the greedy heuristics (bottom part of Figure 4.4) consist in iteratively adding nodes into a solution. For each solution (red nodes), a node is chosen according to a scoring function (green bars).

The remainder of this chapter details the functioning of ALGO. The low-level aspect is first detailed in Section 4.2. We define there an AFOP, how to model a problem as an AFOP, and how to design a heuristic for such an abstract problem. The high-level part is explained in Section 4.3.

4.2 Low-Level Heuristics

This section describes the search space of low-level heuristics in ALGO. In a traditional hyper-heuristic, the space of low-level heuristics is defined for a specific optimisation problem. The hyper-heuristic design process then requires to define a template of low-level heuristics tackling the wanted problem. The learning process thus occurs on the dynamic part of this template. In ALGO the input problem can be any AFOP, the latter class of problems must thus first be defined, as detailed in Section 4.2.1. Some classical optimisation problems are modelled as AFOPs in Section 4.2.2. Finally, a generic template of low-level heuristics for tackling a generic AFOP is described in Section 4.2.3.

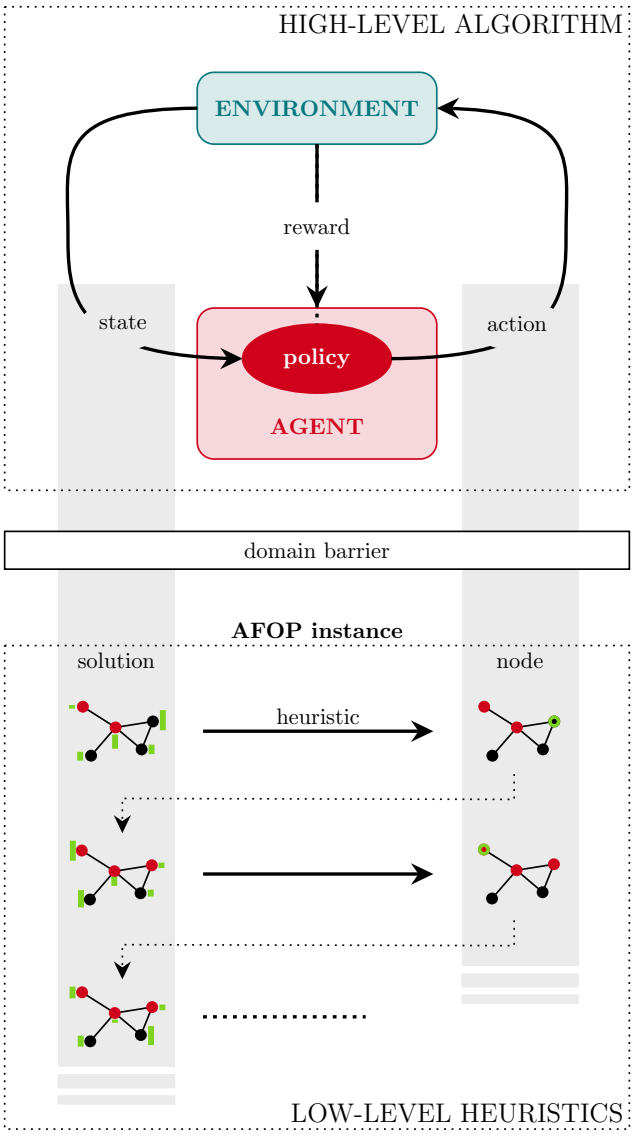


FIGURE 4.4: Workflow of ALGO for the high-level algorithm and the low-level heuristics.

4.2.1 Description of an ALGO-Friendly Optimisation Problem (AFOP)

An AFOP is an optimisation problem with one or more minimisation objectives $\mathcal{O}$. It is subject to a set of constraints $\mathcal{C}$. An AFOP can then be described by the following mathematical program:

$$(\text{AFOP}) \begin{cases} \text{Minimise} & \mathtt{f}_o(S) & \forall o \in \mathcal{O} \\ \text{subject to} & \mathtt{f}_c(S) \leq \mathtt{b}_c & \forall c \in \mathcal{C} \\ & S \in \mathcal{X}_I \end{cases} \tag{4.1}$$

where $\mathcal{X}_I$ is the domain of solutions S for an instance I.

Definition 4.1. Let $G_{(N,E)}$ be a graph (where N is the set of nodes and E the set of edges) and A a set of agents. An AFOP is an optimisation problem where any solution S (feasible or not) can be expressed as:

$$S = \left\{ (a_{i_1}, n_{j_1})_{t_1}, \cdots, (a_{i_k}, n_{j_k})_{t_k}, \cdots \right\} \tag{4.2}$$

where $a_{i_k} \in A$, $n_{j_k} \in N$ and $t_k \in \mathbb{R}_+$, $\forall k \geq 1$.

In other terms, a solution must be describable as a sequence of nodes per agent. Each element of such a solution is a tuple $(a, n)_t \in A \times N \times \mathbb{R}_+$ meaning that the agent a added the node n into the solution at the time t. Since a solution is a set, there is no restriction about different elements sharing the same agent, node or time, but two identical tuples (same agent, node and time) cannot be found in an AFOP solution.

An instance of an AFOP is thus defined by $G_{(N,E)}$ and A. Let $I = \big(G_{(N,E)}, A\big)$ be an AFOP instance, then $\mathcal{S}_I$ refers to the set of all possible solutions obtainable from I. The domain of solutions $\mathcal{X}_I$ mentioned in (4.1) is a subset of $\mathcal{S}_I$ and defines the search space of solutions. Consequently, any tuple $(a, n)_t \in A \times N \times \mathbb{R}_+$ cannot be added into a solution. For each specific AFOP the function $\mathtt{f} : \mathcal{X}_I \to \mathbb{R}^{|\mathcal{O}|+|\mathcal{C}|}$ must also be defined to provide a value for every objective and constraint, along with the vector $\mathtt{b} \in \mathbb{R}^{|\mathcal{C}|}$ giving the upper bounds in constraints.

Implementation We present here a object-oriented-like syntax included in the implementation of ALGO. It provides attributes and methods to elements involved in an AFOP, *i.e.*, nodes, edges, agents and solutions. It must be noted that when using ALGO it is possible to add problem-specific attributes. Those presented below are available with the creation of an AFOP. These attributes and methods can be used for any operation involving an AFOP, *e.g.*, computing the objective value of a solution, determining which nodes an agent can add into a solution, and so on.

An AFOP instance is created from an adjacency matrix and the initial positions of agents (if necessary). Let $I = \big(G_{(N,E)}, A\big)$ be an AFOP instance. Then, $\forall n \in N$, $\forall e \in E$, $\forall a \in A$, the following methods and attributes allow to describe I.

$$\left|\begin{array}{ll} n.neighbours() & := \{n' \mid (n; n') \in E\} \subseteq N \\ e.n1 \in N & \text{first node of edge } e \text{ (base of the arrow if the graph is directed)} \\ e.n2 \in N & \text{second node of edge } e \text{ (head of the arrow if the graph is directed)} \\ e.w \in \mathbb{R} & \text{weight on edge } e \text{ (1 if the graph is non-weighted)} \\ a.first \in N & \text{initial position of agent } a \text{ (}\mathtt{NULL}\text{ if none is given)} \end{array}\right. \tag{4.3}$$

Thanks to its representation given in (4.2), a solution $S \in \mathcal{S}_I$ can be filtered according to agents, nodes and times. It can be useful for operations involving a solution, *e.g.*, computing its objective value or verifying whether it is feasible.

$$\left|\begin{array}{lll} S.get\_agents() & := \{a \mid \exists n, \exists t, (a, n)_t \in S\} \subseteq A & \\ S.get\_nodes() & := \{n \mid \exists a, \exists t, (a, n)_t \in S\} \subseteq N & \\ S.get\_times() & := \{t \mid \exists a, \exists n, (a, n)_t \in S\} \subset \mathbb{R}_+ & \\ S.filter\_agents(A') & := \{(a, n)_t \mid (a, n)_t \in S, a \in A'\} \in \mathcal{S}_I & \text{with } A' \subset A \\ S.filter\_nodes(N') & := \{(a, n)_t \mid (a, n)_t \in S, n \in N'\} \in \mathcal{S}_I & \text{with } N' \subset N \\ S.filter\_times(t') & := \{(a, n)_t \mid (a, n)_t \in S, t \leq t'\} \in \mathcal{S}_I & \text{with } t' \in \mathbb{R}_+ \end{array}\right. \tag{4.4}$$

The first three filters, *i.e.*, $S.get\_agents()$, $S.get\_nodes()$, $S.get\_times()$, consist in extracting respectively agents involved in S, nodes which have been added into S, times at which node have been added. The last three methods are used to filter elements from S according to the given set of agents, node or the given time. Since they return an element of $\mathcal{S}_I$, they can be chained in order to combine filters.

- $S.filter\_agents(A')$ returns elements from S where an agent from A' is involved. It can be useful to know every node added by a specific agent.

- $S.filter\_nodes(N')$ returns elements from S corresponding to a node from N'. It can be used to know when a specific node has been added.
- $S.filter\_times(t')$ returns elements from S which have been added before the time t'. It can be called to know the state of a solution at a certain time.

4.2.2 Example of AFOPs

This section presents four standard optimisation problems modelled as AFOPs: the Travelling Salesman Problem (TSP); the Vehicle Routing Problem (VRP); the Minimum Vertex Cover Problem (MVCP); the Optimal Job Scheduling Problem (OJSP). To model a problem as an AFOP, an instance I of that problem must be defined as a graph $G_{(N,E)}$ and a set of agents A. A solution S must moreover be represented as in Equation 4.2, *i.e.*, $S \in \mathcal{S}_I$ (see Definition 4.1). In the remainder of this section, we provide such a definition of instances and representation of solutions for each of the four aforementioned problems.

4.2.2.1 Travelling Salesman Problem

Given a set of locations, the TSP consists in finding the shortest way to visit all of them and come back to the initial location. It can be assimilated to finding the shortest Hamiltonian cycle in a complete weighted graph. For the Symmetric TSP (STSP), the graph is undirected.

Since a TSP instance is already a graph, it can be directly assimilated to $G_{(N,E)}$ from an AFOP instance, with $N = \{n_i\}_{1 \leq i \leq |N|}$ and $E = \{(n, n') = (n', n) \mid n \in N, n' \in N, n \neq n'\}$. A TSP solution is a path which can be represented as a sequence of nodes. In addition, we consider only one agent for that problem, *i.e.*, $A = \{a_1\}$. Accordingly, an AFOP solution $S \in \mathcal{S}_I$ is equivalent to a sequence of nodes, ordered according to the time they have been added into S. That sequence of nodes then represents the path of the TSP solution (the path is a Hamiltonian cycle in case of a feasible solution).

Figure 4.5 shows a feasible TSP solution for a graph with seven nodes. The solution is represented as a selection of edges, but is equivalent to a sequence of nodes, *e.g.*, $[n_3, n_4, n_7, n_6, n_5, n_2, n_1, n_3]$ if we consider n_3 as the first node. Such a solution can be assimilated to a solution $S \in \mathcal{S}_I$ with the following definition.

$$S = \Big\{(a_1, n_3)_0, (a_1, n_4)_{104}, (a_1, n_7)_{187}, (a_1, n_6)_{246}, (a_1, n_5)_{327}, (a_1, n_2)_{401}, (a_1, n_1)_{524}, (a_1, n_3)_{662}\Big\} \tag{4.5}$$

As mentioned above, only one agent a_1 adds nodes into S. It adds them starting from n_3 and following the edges belonging to the cycle. The time at which a node n is added corresponds to the sum of weights of edges traversed from n_3 to n. By doing so, the length of the cycle can be obtained from S by taking the time of the element added last, *i.e.*, by calling $\max(S.get\_time())$. In Equation 4.5, since $(a_1, n_3)_{662}$ is the last element, the length of the cycle is 662.

4.2.2.2 Vehicle Routing Problem

Given a number of vehicles and a set of locations including a depot, vehicles must visit all locations departing from, and returning to, the depot. The VRP consists in finding the paths taken by vehicles so that the total distance is minimised and all locations, excluding the depot, are visited exactly one time.

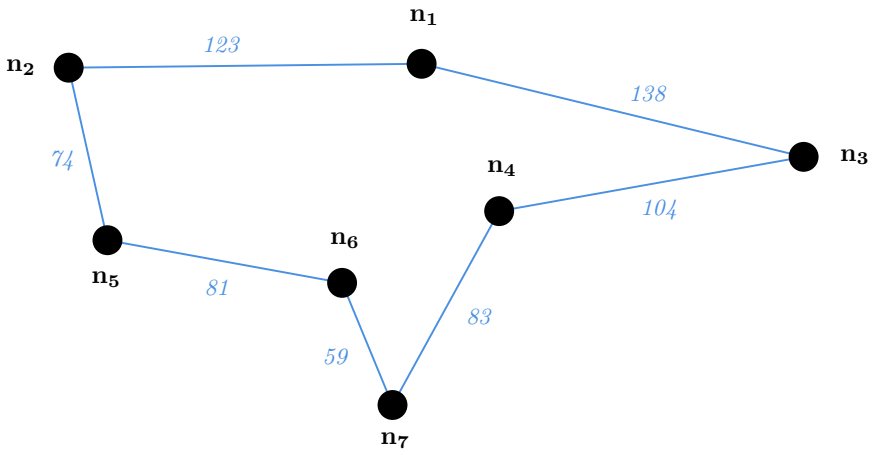


FIGURE 4.5: A solution for the Travelling Salesman Problem (TSP).

As for TSP, $G_{(N,E)}$ is a complete weighted graph, with $N = \{d\} \cup \{n_i\}_{1 \leq i < |N|}$ where n_i are the locations to visit and d the depot. Each vehicle moreover represents an agent, then $A = \{a_i\}_{1 \leq i \leq |A|}$. Given an AFOP solution $S \in \mathcal{S}_I$, the path made by an agent a can be extracted by selecting elements of S involving a, *i.e.*, by calling $S.filter\_agents(\{a\})$, and ordering nodes as a sequence according to the time they have been added into S.

Figure 4.6 shows an example of solution for the VRP with eight locations to visit and three vehicles, each colour representing the path taken by each vehicle. That VRP solution is then described by three sequences of nodes (one per vehicle): $[d, n_2, n_1, d]$; $[d, n_5, n_6, n_3, d]$; $[d, n_8, n_7, n_4, d]$. It can be written as the following solution $S \in \mathcal{S}_I$.

$$S = \Big\{(a_1, d)_0, (a_2, d)_0, (a_3, d)_0, (a_2, n_5)_{45}, (a_1, n_2)_{59}, (a_3, n_8)_{61}, (a_2, n_6)_{88}, (a_3, n_7)_{113}, (a_1, n_1)_{122}, \\ (a_2, n_3)_{147}, (a_1, d)_{179}, (a_3, n_4)_{197}, (a_2, d)_{270}, (a_3, d)_{304}\Big\} \tag{4.6}$$

Each agent adds nodes into S by following its corresponding cycle. For example, agent a_1 adds nodes d, n_2, n_1 and d again (orange cycle in Figure 4.6). Similarly to the TSP, the time at which a node n is added into S by an agent a is the sum of the weight of edges traversed by a from d to n. The time at which an agent adds its last node then correspond to the length of its path. By summing, that time for each agent, *i.e.*, by calling $\sum_{a \in A} \max\left(S.filter\_agents(\{a\}).get\_times()\right)$, we obtain the total distance travelled. In Equation 4.6, the total distance is then $179 + 270 + 304 = 753$.

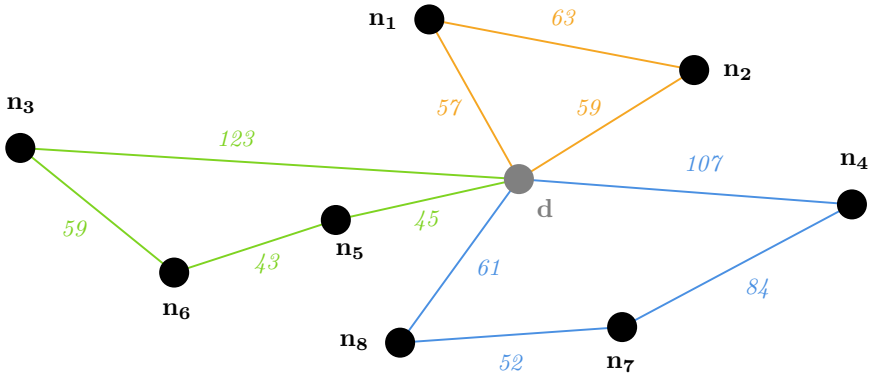


FIGURE 4.6: A solution for the Vehicle Routing Problem (VRP).

4.2.2.3 Minimum Vertex Cover Problem

Given an undirected graph, the MVCP consists in selecting the smallest subset of nodes covering every edge, *i.e.*, at least one of each edge's node is selected.

Since a MVCP instance is a graph, it can be directly assimilated to $G_{(N,E)}$, with $N = \{n_i\}_{1 \leq i \leq |N|}$ and $E = \{e_i\}_{1 \leq i \leq |E|}$. With only one agent considered, *i.e.*, $A = \{a_1\}$, an AFOP solution $S \in \mathcal{S}_I$ can simply represent the nodes which are selected. In that case, the time at which they are added does not matter.

Figure 4.7 shows an example of a MVCP instance. The depicted solution (blue nodes) corresponds to the set $\{n_1, n_6, n_7, n_8, n_9, n_{10}, n_{13}\}$. Since only agent a_1 is involved and there is no temporal aspect, the following solution $S \in \mathcal{S}_I$ is also equivalent.

$$S = \Big\{(a_1, n_7)_0, (a_1, n_{13})_0, (a_1, n_6)_0, (a_1, n_9)_0, (a_1, n_8)_0, (a_1, n_{10})_0, (a_1, n_1)_0\Big\} \tag{4.7}$$

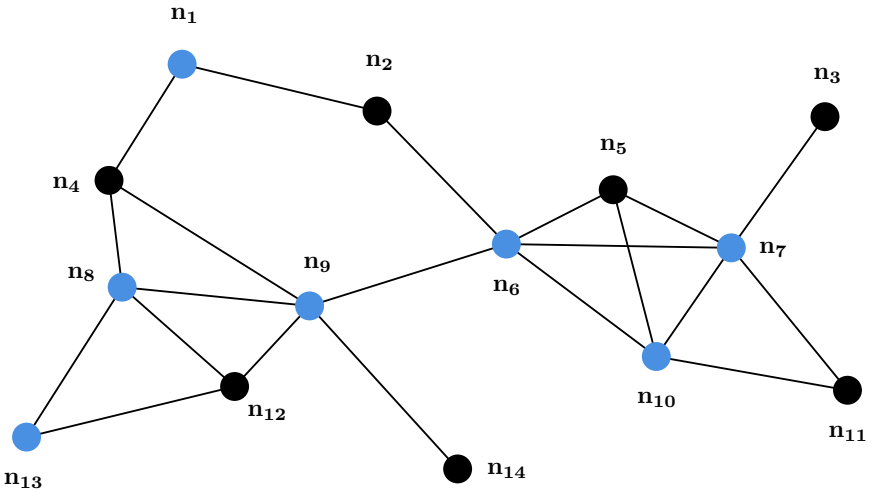


FIGURE 4.7: A solution for the Minimum Vertex Cover Problem (MVCP).

4.2.2.4 Optimal Job Scheduling Problem

Given a set of jobs with a certain running time and a set of parallel machines executing jobs, the OJSP consists in assigning jobs to machines by minimising the total running time.

To translate an OJSP instance into an AFOP instance, we can map jobs to the nodes of the graph and machines to the agents. An assignment of jobs to machines is therefore represented by agents building a path within the graph. The nodes belonging to the path of an agent equates to jobs executed by a machine. We can then define $A = \{a_i\}_{1 \leq i \leq |A|}$ and a directed graph $G_{(N,E)}$ with $N = \{s\} \cup \{n_i\}_{1 \leq i < |N|}$ and $E = \{(n, n_i) \mid n \in N, 1 \leq i < |N|\}$. Nodes n_i correspond to jobs that agents can add into the solution. The node s is a "null" job assigned to each machine, and can be seen as the starting point of each agent. Arrows in the graph have a weight symbolising the running time of the job at their head. That is why there is no arrow pointing to node s.

Figure 4.8 depicts a solution for an OJSP instance with eight jobs and three machines. The solution can be described as follow: machine a_1 (in orange) executes jobs n_2, n_1 and n_3; machine a_2 (in green) executes jobs n_6, n_4 and n_5; machine a_3 (in blue) executes jobs n_8 and $n7$. As with the VRP, a solution $S \in \mathcal{S}_I$ can represent

a set of paths taken by all agents, and can be written as follow.

$$S = \Big\{(a_1, s)_0, (a_2, s)_0, (a_3, s)_0, (a_3, n_8)_5, (a_2, n_6)_{14}, (a_1, n_2)_{60}, (a_2, n_4)_{63}, (a_3, n_7)_{84}, (a_2, n_5)_{92}, (a_1, n_1)_{93}, (a_1, n_3)_{114}\Big\} \tag{4.8}$$

Similarly to the TSP and VRP, the time at which an agent a adds a node n into S is the sum of the weight of arrows traversed by a from s to n. The total running time can then been obtained by calling $\max_{a \in A} \max\left(S.\mathit{filter\_agents}(\{a\}).\mathit{get\_times}()\right)$.

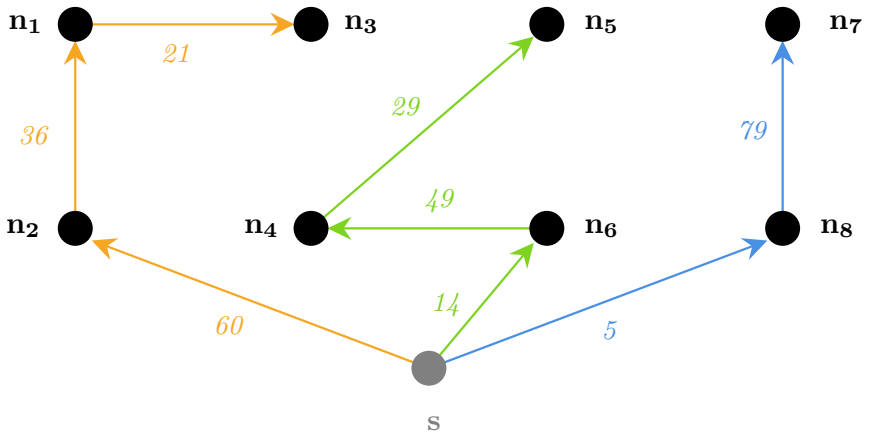


FIGURE 4.8: A solution for the Optimal Job Scheduling Problem (OJSP).

4.2.3 Template of heuristics for an AFOP

Unlike traditional hyper-heuristics, ALGO is designed to tackle an abstract problem, referred to as an AFOP. Given a new problem, there is therefore no need to redefine a space of low-level heuristics by using ALGO.

According to Definition 4.1, the format of its solution is the only specificity of an AFOP. Low-level heuristics in ALGO must thus return solutions as in Equation 4.2. In this work, a greedy heuristic is used for that purpose where, at each iteration, an agent a adds a node n at the time t into the solution, *i.e.*, the element $(a, n)_t$ is added into the solution. The choice of the node to add is done according to a scoring function which is the only dynamic part among such heuristics. A low-level heuristic is thus depicted by its scoring function. The purpose of the high-level algorithm thus consists in finding the best definition of that scoring function, which will result in finding the best low-level heuristic. The generic process of such low-level heuristics is described in Algorithm 4.1 as a template of heuristics.

In Algorithm 4.1, each agent's process is done asynchronously (lines 2–17). The template of heuristics considers distributed heuristics where each agent deals with local information denoted by S_a which is the current solution according to the local knowledge of agent a. An agent a thus adds nodes into S_a until the solution is terminal (lines 5–16). The condition for a solution to be terminal for an agent is determined by $\texttt{terminal} : \mathcal{X}_I \times A \to \{0, 1\}$. For each iteration, agents first select the nodes which are allowed to be added into the solution with $\texttt{get\_nodes} : \mathcal{X}_I \times A \to \mathcal{P}(N)$ (line 6). The definition of the latter function is directly linked to $\mathcal{X}_I$, *i.e.*, the domain of solutions for the problem. It means that $S \cup \{(a, n)_t\} \in \mathcal{X}_I$ if and only if $n \in \texttt{get\_nodes}(S, a)$. Among all possible nodes to add into the solution, the agent selects the one maximising a certain scoring function $\texttt{SCORE} : \mathcal{X}_I \times A \times N \to \mathbb{R}$. In other words, $\texttt{SCORE}(S, a, n)$ evaluates the choice of adding a couple (a, n) into S. It still remains to set the time at which it must be added. To the duration of a's process so far is added "the time for agent a to reach node n according to the current solution S_a", depicted by $\texttt{time} : \mathcal{X}_I \times A \times N \to \mathbb{R}_+$

Algorithm 4.1 Template of low-level heuristics

Input: Instance $I = \left(G_{(N,E)}, A\right)$
Output: Solution $S \in \mathcal{S}_I$

1: $S \leftarrow \emptyset$
2: **for all** agent $a \in A$ **do** {asynchronously}
3: $S_a \leftarrow \emptyset$
4: $t \leftarrow 0$
5: **while** $\neg$`terminal`(S_a, a) **do**
6: $N_{cand} \leftarrow$ `get_nodes`$(S_a, a) \subseteq N$
7: $n \leftarrow \arg\max_{n' \in N_{cand}}$ `SCORE`(S_a, a, n')
8: $t \leftarrow t +$ `time`(S_a, a, n)
9: $S_a \leftarrow S_a \cup \{(a, n)_t\}$
10: $S \leftarrow S \cup \{(a, n)_t\}$
11: **for all** agent $a' \in A \backslash \{a\}$ **do**
12: **if** `can_communicate`(S, a, a') **then**
13: $S_a \leftarrow S_a \cup S_{a'}$
14: **end if**
15: **end for**
16: **end while**
17: **end for**
18: **return** S

(line 8). The element $(a, n)_t$ is thus added into the solution known by agent a, *i.e.*, S_a, and into the centralised solution S (lines 9–10). Since this template of heuristics considers distributed heuristics, agents must be able to communicate and share their local knowledge. For that purpose, every agent searches for other agents with whom it can communicate (lines 11–15). The condition for two agents to communicate is depicted by `can_communicate` : $\mathcal{X}_I \times A^2 \rightarrow \{0, 1\}$. Sharing the local knowledge of two agents a and a' simply consists in processing the union of S_a and $S_{a'}$ (line 13), as shown below.

- local information of a_1: $S_{a_1} = \{(a_1, n_{15})_4, (a_2, n_{96})_{40}, (a_2, n_{41})_{52}, (a_4, n_3)_{59}, (a_1, n_{17})_{98}\}$
- local information of a_2: $S_{a_2} = \{(a_1, n_{15})_4, (a_2, n_{96})_{40}, (a_2, n_{41})_{52}, (a_3, n_{30})_{77}\}$

$\Rightarrow$ shared information: $S_{a_1} \cup S_{a_2} = \{(a_1, n_{15})_4, (a_2, n_{96})_{40}, (a_2, n_{41})_{52}, (a_4, n_3)_{59}, (a_3, n_{30})_{77}, (a_1, n_{17})_{98}\}$

In Algorithm 4.1, four methods must be defined for a specific AFOP: `terminal` for the terminal condition of a solution; `get_nodes` for getting nodes that can be added into the solution; `time` for determining the time at which a node is added into the solution; `can_communicate` for the communication condition between two agents. They thus belong to the dynamic part of ALGO.

Example 4.1. *We present a possible implementation of these four methods for the TSP.*

- `terminal`$(S, a) = \begin{cases} 1 & \text{if } |S| = |N| + 1 \\ 0 & \text{otherwise} \end{cases}$
  The solution is terminal when the agent adds its initial node after adding every node one time. In that case, the solution hence contains $|N| + 1$ *elements.*
- `get_nodes`$(S, a) = \begin{cases} N \backslash S.get\_nodes() & \text{if } |S| < |N| \\ \{a.first\} & \text{otherwise} \end{cases}$
  Agent a can add only nodes not present in the solution unless they have been all added. In that case, the agent is forced to add its initial node.
- `time`$(S, a, n) = (p, n).w$
  where $p \in N$ *is the "position" of agent a according to* S, i.e., *the last node added by a into* S. *It can be*

obtained with

$$p = \underset{n' \in S.filter\_agents(\{a\}).get\_nodes()}{\arg\max} \max\left(S.filter\_agents(\{a\}).filter\_nodes(\{n'\}).get\_times()\right)$$

The time for a to reach n according to S is then the weight of the edge linking the position of a and n.

- `can_communicate`$(S, a_1, a_2) = 0$
  Since only one agent is considered here, no communication is needed and this function is never called.

Given a problem to tackle, these four methods become static for the template of low-level heuristics. The scoring function `SCORE` remains the only dynamic part and must be learnt by the high-level algorithm. The next section then details how the high-level RL algorithm is used to generate a definition for `SCORE`.

4.3 High-Level Algorithm

RL is used as the high-level algorithm in ALGO. Its purpose is thus to find the best definition of `SCORE` (see Algorithm 4.1). Since RL aims at learning a policy $\Pi : \mathcal{S} \to \mathcal{A}$, with $\mathcal{S}$ the set of states and $\mathcal{A}$ the set of actions, a mapping between RL elements and AFOP elements must be done in order to define a policy depending on `SCORE`. This mapping is illustrated in Figure 4.4 where RL states are AFOP solutions and RL actions are nodes from an AFOP instance.

Let $I = \left(G_{(N,E)}, A\right)$ be an AFOP instance. At each step, an RL agent chooses an action from a certain state by following its policy. With the proposed mapping, it is equivalent to an agent $a \in A$ choosing a node $n \in N$ from a certain solution $S \in \mathcal{S}_I$ according to a scoring function. Since AFOP may consider several agents, the RL context is here a Multi-Agent Reinforcement Learning (MARL). While different MARL approaches exist in the literature, in this work the most standard one has been chosen , *i.e.*, every agent shares the same policy [Zhang et al., 2021]. The RL state must hence be defined according to the agent choosing the action (otherwise, every agent would choose the same action at the same time), *i.e.*, $\mathcal{S} \equiv \mathcal{S}_I \times A$. By considering $\mathcal{A} \equiv N$, line 7 of Algorithm 4.1 can be seen as a step for an RL agent, *i.e.*, $action \leftarrow \Pi(state)$.

$$n \leftarrow \underset{n' \in N_{cand}}{\arg\max}\, \texttt{SCORE}(S_a, a, n') \equiv n \leftarrow \Pi\left(S_a, a\right) \tag{4.9}$$

The method `SCORE` then acts like a state-action value, *i.e.*, $\texttt{SCORE} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Among existing RL techniques, we thus decided to choose one aiming at learning such a function, and more particularly Q-Learning (QL). The state-action value in QL is $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The purpose is to learn $Q(state, action)$ in order to represent the maximal future reward that the agent can get by choosing *action* from *state*. In that case, the policy is thus defined as $\Pi(state) = \arg\max_{action \in \mathcal{A}} Q(state, action)$. By using QL as the high-level algorithm, the method `SCORE` can therefore be depicted by Q, since line 7 can be replaced by the step of a QL agent.

$$n \leftarrow \Pi(S_a, a) = \underset{n' \in N_{cand}}{\arg\max}\, Q(S_a, a, n') \tag{4.10}$$

where $Q : \mathcal{S}_I \times A \times N \to \mathbb{R}$ must be learnt so that at each iteration, agents detect which node will maximise the future reward by adding it into the current solution. To learn Q, ALGO uses a parameterised policy, *i.e.*, the line 7 of Algorithm 4.1 can be rewritten as

$$n \leftarrow \Pi_\Theta\left(S_a, a\right) = \underset{n' \in N_{cand}}{\arg\max}\, Q_\Theta\left(S_a, a, n'\right) \tag{4.11}$$

where Θ is a set of learnable parameters. The purpose of the high-level QL is thus to find the optimal value of Θ so that $Q_\Theta(S, a, n)$ represents the maximal future reward obtainable if agent a adds node n into solution S.

An overview of the whole high-level process is shown in Figure 4.9. The interaction between the RL agent and the environment seen in Figure 2.5 is detailed here in the context of ALGO. The whole process can be divided into four steps, including three detailed in the following sections (illustrated in green).

(1) **Choice**. At each iteration, an agent $a \in A$ chooses an action, *i.e.*, a node $n \in N$, from a certain state, *i.e.*, the current solution $S \in \mathcal{S}_I$ (each agent's process is similar and executed asynchronously). This operation, assimilated to the policy, is based on the state-action value $Q_\Theta(S, a, n)$ and therefore depends on the current value of Θ.

Transition. When an agent processes an action, the state of the environment is changed. In an AFOP context, when a node n has been chosen by agent a, it is added into solution S to make a new solution $S' \in \mathcal{S}_I$. This new state will be used for the next iteration of step (1).

(2) **Reward**. A reward is processed according to the action chosen by the agent. The purpose is to evaluate how adding a node into the current solution improves or worsens it.

(3) **Update**. The reward computed is used to update the value of Θ. The latter which corresponds to a new heuristic, will be used for the next iteration of step (1).

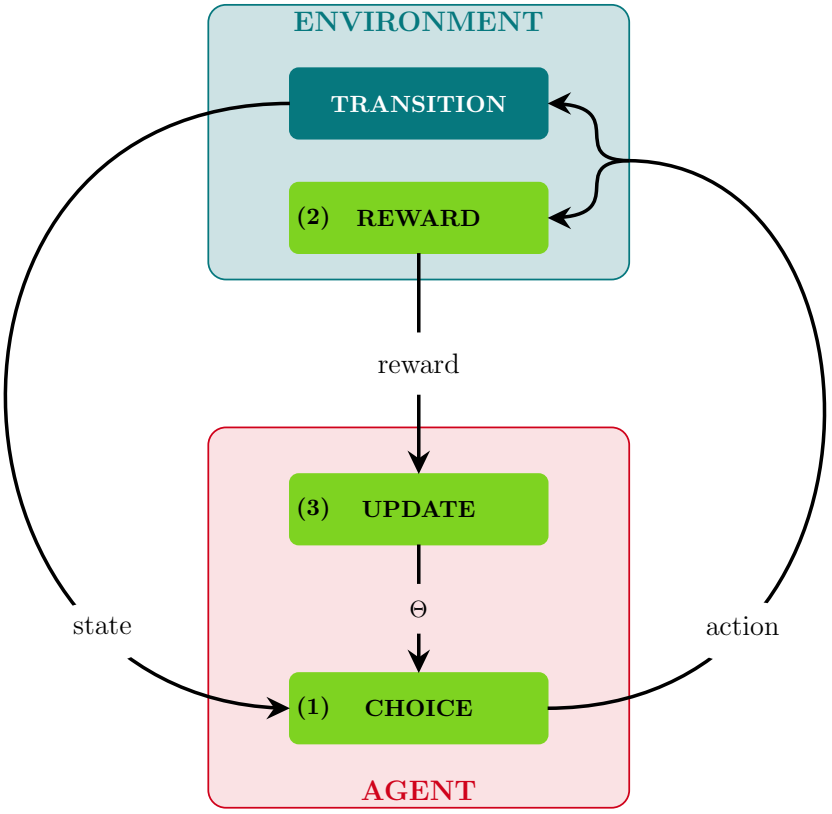


FIGURE 4.9: Overview of the high-level process in ALGO.

The remainder of this section consists in detailing the three steps shown in orange in Figure 4.9, *i.e.*, the evaluation of an action for an agent (see Section 4.3.1), the computation of the reward for such an action (see Section 4.3.2) and the update of the policy (see Section 4.3.3).

4.3.1 Choosing actions

This section describes the step (1) shown in Figure 4.9. The evaluation of an action, *i.e.*, a node $n \in N$, for an agent $a \in A$ is made by calling $Q_\Theta(S, a, n)$ where S is the current solution (see Equation 4.11) in line 7 of Algorithm 4.1.

4.3.1.1 Computation of the state-action value

The process for computing $Q_\Theta(S, a, n)$ is depicted in Figure 4.10. A set of state variable $\{\mathtt{x}(S, a, n')\}_{n' \in N}$ for nodes and $\{\mathtt{y}(S, a, e')\}_{e' \in E}$ for edges are given as an input to a Graph Neural Network (GNN).

- $\mathtt{x} : \mathcal{S}_I \times A \times N \to \mathbb{R}^{k_x}$ returns the state variables for a node from the perspective of an agent from the current solution.
- $\mathtt{y} : \mathcal{S}_I \times A \times E \to \mathbb{R}^{k_y}$ returns the state variables for a edge from the perspective of an agent from the current solution.

These state variables are defined for the specific AFOP. Each node thus has k_x state variables and each edge has k_y state variables. The output of the GNN after L layers is an embedding structure assigning to each node a p-dimensional vector where p and L are parameters of ALGO. For agent a, the embedded representation of a node n' is then given by $\mu^{(L)}_{a,n'} \in \mathbb{R}^p$. From this embedding structure, two p-dimensional vectors are extracted depicting respectively the embedded state and the embedded action: $\sum_{n' \in N} \mu^{(L)}_{a,n'}$ and $\mu^{(L)}_{a,n}$ where n is the node to evaluate. These two vectors are then given as an input of a Neural Network (NN) which outputs a scalar value, *i.e.*, the wanted evaluation $Q_\Theta(S, a, n)$. The set of learnable parameters Θ corresponds to the weights of both the GNN and the NN.

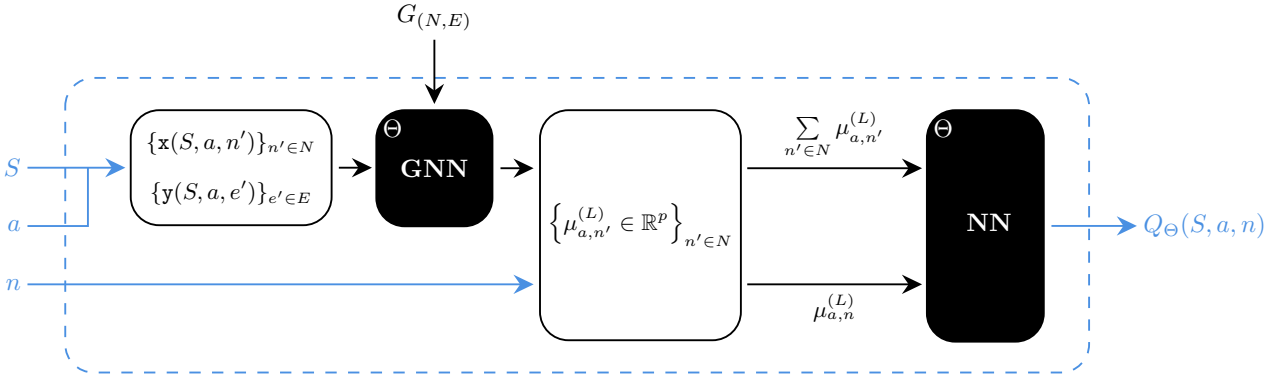


FIGURE 4.10: Process for computing the state-action value $Q_\Theta(S, a, n)$.

The GNN and NN are thought modular. It means that any existing technique could be used. Sections 4.3.1.2 and 4.3.1.3 respectively describe GNNs and NNs approaches that can be used.

4.3.1.2 Graph Neural Networks

As mentioned earlier, one strength of ALGO is its design which does not depend on a single GNN. The only condition for a GNN to be compatible is to embed nodes of the given graph into p-dimensional vectors, from state variables assigned to nodes and edges. We present structure2vec (S2V) [Dai et al., 2016] as an example in this section. Other GNNs can however be used, like GraphSAGE [Hamilton et al., 2017] or graph attention networks (GATs) [Veličković et al., 2018].

Equation 4.12 details how to compute the embedding vector $\mu_{a,n}^{(L)}$ for a node n and an agent a after L layers by using S2V. Our work extends the one of Khalil et al. [2017] to the multi-agent context. The embedding vector in the GNN that they used [Dai et al., 2016] thus depends on node n only.

$$\forall l \geq 1 \begin{cases} \mu_{a,n}^{(l)} = relu\left(\theta_1 \cdot \mathtt{x}(S,a,n) + \theta_2 \cdot \sum_{(n,n')\in E} relu\left(\theta_3 \cdot \mathtt{y}(S,a,(n,n'))\right) + \theta_4 \cdot \sum_{(n,n')\in E} \mu_{a,n'}^{(l-1)}\right) \in \mathbb{R}^p \\ \mu_{a,n}^{(0)} = 0 \in \mathbb{R}^p \end{cases} \tag{4.12}$$

with $\theta_1 \in \mathbb{R}^{p\times k_x}$, $\theta_3 \in \mathbb{R}^{p\times k_y}$, $\theta_2, \theta_4 \in \mathbb{R}^{p\times p}$ and *relu* is the REctified Linear Unit. An embedded information is computed for each node and edge according to their state variable, respectively $\theta_1 \cdot \mathtt{x}(S,a,n) \in \mathbb{R}^p$ for node n and $\theta_3 \cdot \mathtt{y}(S,a,e) \in \mathbb{R}^p$ for edge e. The embedding vector $\mu_{a,n}^{(l)}$ then depends on the embedded information of n, the embedded information of all adjacent edges, and the embedding vector of adjacent nodes at the layer $l-1$. The embedding vector $\mu_{a,n}^{(l+1)}$ thus contains information about the l-hop neighbourhood of node n.

4.3.1.3 Neural Networks

Similarly to the GNN, any NN could be used at this stage. The only condition is to have $2p$ neurons on the input layer, p neurons for the embedded state and p neurons for the embedded action, and one neuron on the output layer, representing the final state-action value. We present in Equation 4.13 an example which is used by Khalil et al. [2017].

$$Q_\Theta(S,a,n) = \theta_1^\top \cdot relu\left(\left[\theta_2 \cdot \sum_{n'\in N} \mu_{a,n'}^{(L)}, \theta_3 \cdot \mu_{a,n}^{(L)}\right]\right) \in \mathbb{R} \tag{4.13}$$

with $\theta_1 \in \mathbb{R}^{2p}$, $\theta_2, \theta_3 \in \mathbb{R}^{p\times p}$ and $[\cdot,\cdot]$ is the concatenation operator.

4.3.2 Computing rewards

This section describes Step (2) in Figure 4.9. In RL the reward $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is given for the choice of an action from a certain state, *i.e.*, $r : \mathcal{S}_I \times A \times N \to \mathbb{R}$ in the context of an AFOP.

4.3.2.1 Vectorial reward

In the definition of an AFOP, several objectives may be defined, *i.e.*, $|\mathcal{O}| > 1$, or constraints may be considered, *i.e.*, $|\mathcal{C}| > 0$. For every action that an agent chooses, there must thus be a reward per objective and constraint. The vectorial reward for an agent a choosing the node n from the current solution S is then given by $\vec{r}(S,a,n) \in \mathbb{R}^{|\mathcal{O}|+|\mathcal{C}|}$.

$$\vec{r}(S,a,n) = \begin{pmatrix} \vec{r}_{o_1}(S,a,n) \\ \vec{r}_{o_2}(S,a,n) \\ \vdots \\ \vec{r}_{c_1}(S,a,n) \\ \vec{r}_{c_2}(S,a,n) \\ \vdots \end{pmatrix} = \begin{pmatrix} \mathtt{f}_{o_1}(S) - \mathtt{f}_{o_1}(S') \\ \mathtt{f}_{o_2}(S) - \mathtt{f}_{o_2}(S') \\ \vdots \\ \mathtt{f}_{c_1}(S) - \mathtt{f}_{c_1}(S') \\ \mathtt{f}_{c_2}(S) - \mathtt{f}_{c_2}(S') \\ \vdots \end{pmatrix} = \mathtt{f}(S) - \mathtt{f}(S') \in \mathbb{R}^{|\mathcal{O}|+|\mathcal{C}|} \tag{4.14}$$

where $\mathcal{O} = \{o_1, o_2, \cdots\}$, $\mathcal{C} = \{c_1, c_2, \cdots\}$, and $S' = S \cup \{(a,n)_t\}$ with $t = t_a + \mathtt{time}(S,a,n)$ and $t_a = \max\left(S.filter\_agents(\{a\}).get\_times()\right)$ is the time at which a added the last node into S. For each objective

and constraint, the reward is thus the difference of objective value of solutions before and after the action of the agent. Since objectives must be minimised in an AFOP, the reward must be positive if the objective value after the action is smaller, and vice versa. The reward is therefore $\mathtt{f}(S) - \mathtt{f}(S')$ and not $\mathtt{f}(S') - \mathtt{f}(S)$.

For each action of type "agent a adds node n into the current solution S", a vectorial reward $\vec{r}(S, a, n)$ is thus computed as defined in Equation 4.14. In ALGO, the reward may not be provided immediately after an action but after τ consecutive actions from the same agent a (since the value of τ is set arbitrarily, it is a parameter of ALGO). In that case, a cumulative reward $\vec{r}^{\Sigma}(S, a, n)$ is provided, *i.e.*, the sum of consecutive rewards, for that action. If $\tau = 1$, then $\vec{r}^{\Sigma}(S, a, n) = \vec{r}(S, a, n)$ and the reward is given after each action made.

Let $I = \left(G_{(N,E)}, A\right)$ be an AFOP instance with only one agent, *i.e.*, $A = \{a\}$, and $S \in \mathcal{S}_I$. Then $S^{(i)}$ is the solution obtained after i action made by agent a from the current solution S, and $n^{(i)}$ is the node added by agent a into $S^{(i)}$.

$$\begin{aligned} S' &= S \cup \{(a, n)_t\} \\ S'' &= S' \cup \{(a, n')_{t'}\} \\ &\vdots \\ S^{(i+1)} &= S^{(i)} \cup \left\{\left(a, n^{(i)}\right)_{t^{(i)}}\right\} \end{aligned} \tag{4.15}$$

On the assumption that $|A| = 1$ and with Equation 4.14, the cumulative reward $\vec{r}^{\Sigma}(S, a, n)$ can be reduced to the difference of objective values of solutions S and $S^{(\tau)}$, *i.e.*, the solutions after and before the τ consecutive actions.

$$\vec{r}^{\Sigma}(S, a, n) = \sum_{i=0}^{\tau-1} \vec{r}\left(S^{(i)}, a, n^{(i)}\right) = \mathtt{f}\left(S\right) - \mathtt{f}\left(S^{(\tau)}\right) \in \mathbb{R}^{|\mathcal{O}|+|\mathcal{C}|} \tag{4.16}$$

We now consider anAFOP instance $I = \left(G_{(N,E)}, A\right)$ with several agents, *i.e.*, $|A| \geq 2$. Since solutions in Equation 4.15 consider actions made by only one agent, the equality in Equation 4.16 is not true. More than τ actions may have been made, all agents included, from S to reach $S^{(\tau)}$ indeed. Even if $\vec{r}^{\Sigma}(S, a, n)$ only considers actions made by a, $\mathtt{f}\left(S\right) - \mathtt{f}\left(S^{(\tau)}\right)$ remains a good estimator of the cumulative reward since it still represents the sum of consecutive actions from S to reach $S^{(\tau)}$, all agents' actions included and all agents sharing the same policy. Equality in Equation 4.16 is thus considered true even with an instance considering several agents.

Eventually, the vectorial reward must be reduced to a scalar reward. For that, rewards received for objectives are treated differently than rewards for constraints. Let $\vec{r}^{\Sigma}(S, a, n) \in \mathbb{R}^{|\mathcal{O}|+|\mathcal{C}|}$ be a vectorial reward computed with Equation 4.16. Then $\vec{r}^{\Sigma}_{\mathcal{O}}(S, a, n) \in \mathbb{R}^{|\mathcal{O}|}$ and $\vec{r}^{\Sigma}_{\mathcal{C}}(S, a, n) \in \mathbb{R}^{|\mathcal{C}|}$ are subvectors of $\vec{r}^{\Sigma}(S, a, n)$ corresponding to rewards for objectives and constraints respectively.

$$r(S, a, n) = \underbrace{scal\left(\vec{r}^{\Sigma}_{\mathcal{O}}(S, a, n)\right)}_{\text{scalarisation of objectives}} + \underbrace{\lambda^{\top}\left(\vec{r}^{\Sigma}_{\mathcal{C}}(S, a, n) + \mathtt{b}\right)}_{\text{penalisation of constraints}} \in \mathbb{R} \tag{4.17}$$

where $\lambda \in \mathbb{R}_{+}^{|\mathcal{C}|}$ is a learnable parameter and $scal : \mathbb{R}^{|\mathcal{O}|} \to \mathbb{R}$ is the scalarisation of the rewards for objectives. Different scalarisation techniques exist and can be used here. They are presented below in Section 4.3.2.2. On the other side, if a constraint $c \in \mathcal{C}$ is violated, *i.e.*, $\vec{r}^{\Sigma}_{c}(S, a, n) + \mathtt{b}_c < 0$ as explained in (4.18), then a penalty

$\lambda_c \geq 0$ is added to the scalar reward $r(S, a, n)$.

$$\begin{aligned} c \in \mathcal{C} \text{ is respected} &\Leftrightarrow \mathtt{f}_c(S) \leq \mathtt{b}_c && \forall S \in \mathcal{S}_I \\ &\Leftrightarrow \mathtt{f}_c\left(S\right) - \mathtt{f}_c\left(S^{(\tau)}\right) \geq -\mathtt{b}_c && \forall S \in \mathcal{S}_I, \forall \tau \geq 1 \\ &\Leftrightarrow \vec{r}_c^{\Sigma}(S, a, n) \geq -\mathtt{b}_c && \forall S \in \mathcal{S}_I, \forall (a, n) \in A \times N, \forall \tau \geq 1 \\ &\Leftrightarrow \vec{r}_c^{\Sigma}(S, a, n) + \mathtt{b}_c \geq 0 && \forall S \in \mathcal{S}_I, \forall (a, n) \in A \times N, \forall \tau \geq 1 \end{aligned} \tag{4.18}$$

4.3.2.2 Scalarisation

This part of ALGO is modular. It means that our framework uses an abstract way to transform a vectorial reward into a scalar one. The user has then the possibility to define the scalarisation function according to its needs. Most scalarisations functions rely on weights, one associated to each objective. It makes it possible to control the nature of the heuristic generated by ALGO, if for instance the user wants to put an emphasis on one or more objectives. The trade-off between objectives is given by $\{w_o\}_{o \in \mathcal{O}}$ and

$$\sum_{o \in \mathcal{O}} w_o = 1 \tag{4.19}$$

where $w_o \in [0, 1]$ (by default, weights are set to $w_o = 1/|\mathcal{O}|)$ $\forall o \in \mathcal{O}$. We present in this section two examples of scalarisation which can be used in ALGO. Different scalarisation functions are however conceivable.

Linear scalarisation The linear scalarisation is the most commonly used function, and is equivalent to a weighted sum. By default, *i.e.*, all weights are equal, it then consists in doing an average of the reward obtained from objectives.

$$scal\left(\vec{r}_{\mathcal{O}}^{\Sigma}(S, a, n)\right) = \sum_{o \in \mathcal{O}} \left(w_o \cdot \vec{r}_o^{\Sigma}(S, a, n)\right) \tag{4.20}$$

Chebyshev scalarisation The Chebyshev scalarisation is a non-linear function introduced by Van Moffaert et al. [2013]. It consists in taking the distance in the multi-objective space with a target point z^* according to the weighted L_∞ metric. We remind that the infinite norm of a vector x is given by $\|x\|_\infty = \max_i |x_i|$. Given a vector $x \in \mathbb{R}^{|\mathcal{O}|}$ a vector in the multi-objective space,

$$scal(x) = \max_{o \in \mathcal{O}} \left(w_o \cdot |x_o - z_o^*|\right) \tag{4.21}$$

where $z^* \in \mathbb{R}^{|\mathcal{O}|}$ the target point.In their work, Van Moffaert et al. [2013] scalarise the state-action value $Q(state, action)$. In that case, the target point corresponds to $Q^* = \{Q_o^*\}_{o \in \mathcal{O}}$ where $Q_o^*(state, action)$ is the optimal state-action value for objective o, *e.g.*, the maximal future reward obtainable after choosing *action* from *state* in Q-learning. Among multiple state-action values, the Chebyshev scalarisation then consists in returning the one with the farthest value from the target point.

In ALGO, the scalarisation occurs on the reward, which means that there is no finite target point since the reward must be maximised without upper bound. Among multiple rewards, returning the one with the farthest value from the target reward is therefore equivalent to returning the one with the lowest value.

$$scal\left(\vec{r}_{\mathcal{O}}^{\Sigma}(S, a, n)\right) = \min_{o \in \mathcal{O}} \left(w_o \cdot \vec{r}_o^{\Sigma}(S, a, n)\right) \tag{4.22}$$

4.3.3 Updating the policy

This section describes Step (3) in Figure 4.9. The update of the policy is done in two steps since there are two learnable parameters in ALGO: Θ and λ. The update of Θ aims at improving the state-action value (see Section 4.3.1). Updating λ is intended to correct the computation of the scalar reward (see Section 4.3.2).

4.3.3.1 Updating Θ

For each action made, the following tuple is stored in a memory $\mathcal{M}$: the state in which the action is made, *i.e.*, the current solution and the agent performing the action; the action considered, *i.e.*, the node chosen by the agent; the cumulative reward for the τ following actions made by the same agent; the state after these τ actions. In the context of AFOP, the tuple $\left(S, a, n, \vec{r}^{\Sigma}(S, a, n), S^{(\tau)}\right)$ is thus added into $\mathcal{M}$ for the following action: "agent a adds node n into solution S". Every such item $i = \left(S_i, a_i, n_i, \vec{r}^{\Sigma}(S_i, a_i, n_i), S_i^{(\tau)}\right) \in \mathcal{M}$ makes it possible to compute the following two values.

$$\begin{cases} pred_i(\Theta) = Q_\Theta\left(S_i, a_i, n_i\right) \\ \quad targ_i = r(S_i, a_i, n_i) + \gamma \cdot \max\limits_{n^{(\tau)} \in \texttt{get\_nodes}\left(S_i^{(\tau)}, a_i\right)} Q_\Theta\left(S_i^{(\tau)}, a_i, n^{(\tau)}\right) \end{cases} \tag{4.23}$$

where $\gamma \in [0, 1]$ is the discount factor and represents the importance given to the future reward depicted by $\max_{n^{(\tau)}} Q_\Theta\left(S_i^{(\tau)}, a_i, n_i^{(\tau)}\right)$. It must be noted that in the computation of $targ_i$, the scalar reward $r(S_i, a_i, n_i)$ is used and computed with (4.17) from the vectorial reward $\vec{r}^{\Sigma}(S_i, a_i, n_i)$ stored in memory. For an element $i \in \mathcal{M}$, $pred_i(\Theta)$ is the predicted value of the reward by doing the corresponding action, while $targ_i$ is the actual reward, *i.e.*, the target value. The purpose is thus to modify Θ in order to minimise the gap between $pred_i(\Theta)$ and $targ_i$. For that, a mini-batch $\mathcal{B} \subset \mathcal{M}$ is randomly selected from the memory, and a Stochastic Gradient Descent (SGD) is processed over $\mathcal{B}$ to minimise the loss, formally defined below as the squared mean error.

$$loss = \sum_{i \in \mathcal{B}} \left(pred_i(\Theta) - targ_i\right)^2 \tag{4.24}$$

It should be noted that the target value is considered as a constant. It means that during its calculation, there is no trace of Θ for the SGD, unlike the predicted value which depends on Θ. Moreover, the learning rate $\alpha \in [0, 1]$ of the SGD is a parameter of ALGO.

4.3.3.2 Updating λ

In an AFOP, constraints do not restrict the choice of a node by an agent. It means that the current policy, *i.e.*, the current heuristic, may lead to a non-feasible solution for some instances. If a constraint $c \in \mathcal{C}$ turns out to be violated, it means that the penalty for violating that constraint was not high enough (see Section 4.3.2) and the value of λ_c must then be increased. On the opposite, the penalty is high enough but care must be taken not to have it too high. It would indeed increase the reward and may neglect the reward obtained with objective values. If a constraint $c \in \mathcal{C}$ is respected, then λ_c must decreases.

A Gradient Descent (GD) is processed to update λ, where the value of the gradient depends on whether actions from the mini-batch $\mathcal{B}$ have violated constraints.

$$\lambda \leftarrow \max\left(0, \lambda - \eta \cdot \sum_{i \in \mathcal{B}} \left(\vec{r}_{\mathcal{C}}^{\Sigma}(S_i, a_i, n_i) + \mathtt{b}\right)\right) \tag{4.25}$$

where $\eta \in [0,1]$ is the learning rate of the GD. For each element $i \in \mathcal{B}$ selected from the memory, $\bar{r}_c^{\Sigma}(S_i, a_i, n_i) + \mathtt{b_c} \geq 0$ means that the constraint $c \in \mathcal{C}$ is respected as shown in (4.18) and thus makes λ_c decrease. Finally, any negative value of λ must be set to 0.

4.4 How to use ALGO?

In this section, we describe how a user can use ALGO for a specific optimisation problem. We start by doing a formal description of steps to follow in Section 4.4.1. There is summarised all of dynamic parts in the generic model ALGO. We then provide in Section 4.4.2 a more code-oriented description. We briefly describe the structure of the implementation of ALGO so that one can use it to generate heuristics for a specific problem.

4.4.1 Formal description

Let P be an optimisation problem for which you want to generate a heuristic. This section describes the steps to follow if you want to use ALGO to generate a heuristic for P. The first part consists in representing an instance of P as $I = \big(G_{(N,E)}, A\big)$ with $G_{(N,E)}$ a graph and A a set of agents. According to Definition 4.1, a solution of I must then be represented as in Equation 4.2. Some examples are shown in Section 4.2.2. After defining objectives, constraints and the domain of solutions as in Equation 4.1, you must implement all the dynamic parts involved in ALGO. They are listed in Table 4.1 and are divided into three parts depending on whether they refer to the definition of the optimisation model (see Section 4.2.1), the design of low-level heuristics (see Section 4.2.3) or the state variables given as an input to the GNN (see Section 4.3.1).

optimisation model	
$\mathtt{f} : \mathcal{X}_I \to \mathbb{R}^{\lvert\mathcal{O}\rvert+\lvert\mathcal{C}\rvert}$	objective values (including constraints)
$\mathtt{b} \in \mathbb{R}^{\lvert\mathcal{C}\rvert}$	upper bounds of constraints
low-level heuristic	
$\mathtt{terminal} : \mathcal{X}_I \times A \to \{0,1\}$	terminal condition for an agent
$\mathtt{get\_nodes} : \mathcal{X}_I \times A \to \mathcal{P}(N)$	nodes that an agent can add into the current solution
$\mathtt{time} : \mathcal{X}_I \times A \times N \to \mathbb{R}_+$	time for an agent to add a node into the current solution
$\mathtt{can\_communicate} : \mathcal{X}_I \times A^2 \to \{0,1\}$	condition of communication between two agents
state variables	
$\mathtt{x} : \mathcal{X}_I \times A \times N \to \mathbb{R}^{k_x}$	state variables of nodes
$\mathtt{y} : \mathcal{X}_I \times A \times E \to \mathbb{R}^{k_y}$	state variables of edges

TABLE 4.1: Elements to define for a problem in order to use ALGO.

Let $S \in \mathcal{X}_I$ be a solution of instance I. For each objective $o \in \mathcal{O}$, $\mathtt{f}_o(S)$ is the objective value of solution S. Each constraint $c \in \mathcal{C}$ also provides an objective value $\mathtt{f}_c(S)$. An upper bound b_c must however be defined for $f_c(S)$. These objective values and upper bounds are used for the computation of the reward.

In the heuristic generated by ALGO, each agent adds a new node into the current solution asynchronously. Let S be that current solution. An agent $a \in A$ thus runs until $\mathtt{terminal}(S, a) = 1$. At each iteration, agent a cannot add any node into S, only nodes contained in $\mathtt{get\_nodes}(S, a)$. Once agent a knows which nodes it can adds into S, it evaluates each node $n \in N$ with the state-action value $Q_\Theta(S, a, n)$ and choose the one maximising it.

The computation of the state-action value uses all of information contained in $G_{(N,E)}$. You must hence define state variables for nodes and edges which can be used in the computation of $Q_\Theta(S, a, n)$. One can add as many state variables as he needs, so that $\texttt{x}(S, a, n)$ gives the k_x state variables of node $n \in N$ according to agent a and the current solution S, and $\texttt{y}(S, a, e)$ gives the k_y state variables of edge $e \in E$. In case the heuristic considers multiple agents, and you want them to run in a distributed way, you must define the condition for two agents to share their local knowledge. Two agents a_1 and a_2 can thus communicate if $\texttt{can\_communicate}(S, a_1, a_2) = 1$.

4.4.2 Implementation

Figure 4.11 depicts a part of the class diagram of ALGO. It shows the part to override in order to apply ALGO on a specific optimisation problem. More detail about the implementation, including a complete class diagram, is available in Appendix C. Among all classes involved in Figure 4.11, `Graph` represent an AFOP instance $I = \big(G_{(N,E)}, A\big)$. It has three lists as attributes, containing objects of type `Node`, `Edge` or `Agent`. Moreover, an object of type `Solution` is equivalent to a solution $S \in \mathcal{S}_I$. Such an object is seen as a set of items, where each of them represents a tuple $(a, n)_t \in A \times N \times \mathbb{R}_+$, *i.e.*, it has three attributes of type `Node`, `Agent` and `float`. Since a solution is relevant for a certain instance, an object of type `Solution` also have an instance of `Graph` as an attribute.

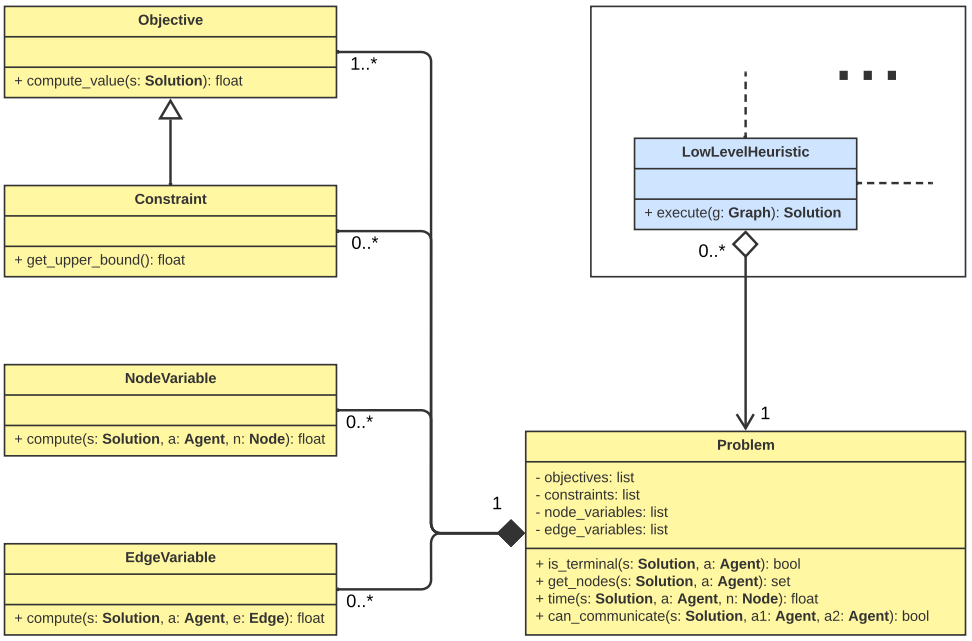


FIGURE 4.11: Partial class diagram of ALGO, showing what one must override in order to apply ALGO on a specific problem.

In our implementation, a heuristic is represented as an instance of `LowLevelHeuristic`. The heuristic being dependent of the problem to tackle, it has an attribute of type `Problem` which is an abstract class (in yellow in Figure 4.11). That class has four abstract methods corresponding to the elements to define in the context of the low-level heuristic (see Section 4.4.1). These methods are called by the heuristic when executed on an instance, *i.e.*, within the method `execute(g)` where `g` is an object of type `Graph`. In order to apply ALGO on a specific problem, a user must then create a class inheriting from `Problem` and override these four methods.

A problem also has a list of objectives and constraints to be written as in Equation 4.1. For each objective, the user must then create a class inheriting from `Objective` and override the abstract method `compute_value(s)`

which returns the corresponding objective value of the given solution `s`. Similarly, the user must create a class inheriting from `Constraint` for each wanted constraint. Additionally to `compute_value(s)`, he must override `get_upper_bound()`.

Finally, the user must define the state variables to be given as an input to the GNN. For each variable assigned to a node, one must create a class inheriting from `NodeVariable` and overriding the abstract method `compute(s, a, n)`. An equivalent process must be done with edge variables and class `EdgeVariable`.

4.5 Conclusion

In this chapter, Algorithm Learner for Graph Optimisation problems (ALGO), a model of generic hyper-heuristic based on Reinforcement Learning (RL) has been introduced. ALGO permits to generate a heuristic for any compatible optimisation problem, without the need to redesign a space of low-level heuristics for each problem. The strengths of ALGO are threefold:

- **Flexibility.** ALGO can be used to tackle a wide range of optimisation problems, from classical ones, *e.g.*, routing or scheduling problems, to problems considering multiple agents or a dynamic environment. ALGO is besides capable of generating distributed heuristics in a multi-agent context. Furthermore, ALGO is not limited to single-objective optimisation problems.
- **Modularity.** Several parts of ALGO are interchangeable, *e.g.*, the architecture of the Graph Neural Network (GNN) within the RL algorithm or the scalarisation technique. A user then has the possibility to use ALGO without being limited to a certain model.
- **Ease of use.** We designed the task needed to use ALGO on a new problem to be easy and straightforward. We wanted to avoid the situation where a user, with a generic model of hyper-heuristic, is obliged to implement everything so that the theoretical model suits its optimisation problem.

The next chapter presents the application of ALGO on a classical optimisation problem: the Travelling Salesman Problem (TSP). It will aim at validating the ability of ALGO to generate efficient heuristic in a classical context, before using it in a real-world context (detailed in next chapters).

Chapter 5

Validation on the Travelling Salesman Problem

Contents

5.1 Introduction

In this chapter, we validate ALGO, our proposed generic model of hyper-heuristic. Before using ALGO in a real-world use case (presented in the next part of this thesis), we demonstrate the ability of ALGO to generate efficient greedy heuristics for a classical optimisation problem, the Travelling Salesman Problem (TSP). Given a set of points and the distance between all of them, the TSP consists in finding the shortest way to visit all points exactly one time by ending at the initial point. In other terms, it is equivalent to finding the shortest Hamiltonian cycle within a complete graph. The symmetric TSP is tackled in this chapter. Given two points A and B, the distance to go from A to B is the same as going from B to A. We first present our modelling of the TSP as an AFOP in Section 5.2. The implementation within ALGO is then detailed in Section 5.3. These two sections can be used as a concrete example for a user who wants to apply ALGO on a specific problem. ALGO was then trained to generate a greedy heuristic for the TSP. The training process is described and the obtained results are presented in Section 5.4. The performance of the generated heuristic is compared to state-of-the-art TSP heuristics.

5.2 Modelling as an AFOP

This section presents how we modelled the TSP as an AFOP so that ALGO can generate TSP heuristics. The first step is to represent a TSP instance as an AFOP instance, *i.e.*, $I = \left(G_{(N,E)}, A\right)$ where G is a graph and A a set of agents. A solution $S \in \mathcal{S}_I$ must moreover be mappable to a TSP solution.

$$S = \left\{ (a_{i_1}, n_{j_1})_{t_1}, \cdots, (a_{i_k}, n_{j_k})_{t_k}, \cdots \right\} \in \mathcal{S}_I \tag{5.1}$$

where $a_{i_k} \in A$, $n_{j_k} \in N$ and $t_k \in \mathbb{R}_+$, $\forall k \geq 1$.

A TSP instance can be described as a complete graph $G_{(N,E)}$ which means that $N = \{n_i\}_{1 \leq i \leq |N|}$ and $E = \{(n, n') = (n', n) \mid n \in N, n' \in N, n \neq n'\}$. An equivalent AFOP instance can then be $I = \left(G_{(N,E)}, A\right)$ where $A = \{a_1\}$ meaning that there is only one agent.

A TSP solution is a Hamiltonian cycle which can be represented as a sequence of nodes from N or a set of edges from E. Since only one agent is considered, a solution written as in Equation 5.1 can be seen as a sequence of nodes, where nodes are ordered by the time they were added into the solution. In the proposed modelling, the latter sequence does not directly represent the TSP solution, but the order in which nodes are added to a subtour. A node is added at the position where it minimises the growth of the subtour.

The mapping to get a TSP solution from a solution $S \in \mathcal{S}_I$ is illustrated in Figure 5.1 with $|N| = 5$ and $S = \{(a_1, n_3)_{t_1}, (a_1, n_1)_{t_2}, (a_1, n_4)_{t_3}, (a_1, n_5)_{t_4}, (a_1, n_2)_{t_5}\}$. Supposing that t_i increases with i, nodes are added to the subtour by following the order n_3, n_1, n_4, n_5, n_2. The subtour is thus initialised with node n_3. Nodes n_1 and n_4 are then added. Until that point, there has been only one way to insert a node into the subtour. For next iteration, node n_5 can be inserted into three different positions. The position minimising the growth of the subtour is between nodes n_3 and n_4. The same process is repeated to insert n_2, and to obtain the TSP solution corresponding to S.

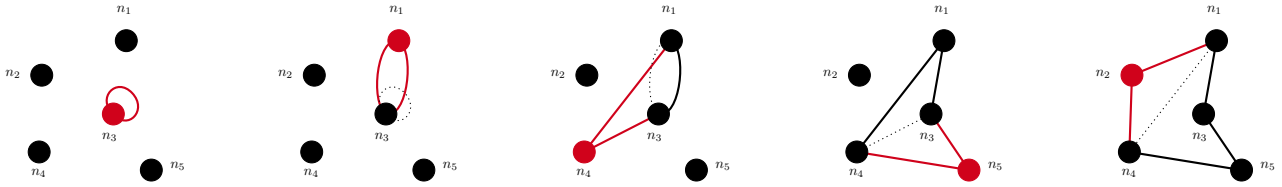


FIGURE 5.1: TSP solution obtained from $S = \{(a_1, n_3)_{t_1}, (a_1, n_1)_{t_2}, (a_1, n_4)_{t_3}, (a_1, n_5)_{t_4}, (a_1, n_2)_{t_5}\}$. Nodes are added into the subtour by following the order given by the time of items in S, *i.e.*, $n_3 \to n_1 \to n_4 \to n_5 \to n_2$. A node is added at the position where it minimises the growth of the subtour.

Finally, the domain of solutions $\mathcal{X}_I \subseteq \mathcal{S}_I$, the set of objectives $\mathcal{O}$ and the set of constraints $\mathcal{C}$ must be defined. The TSP must then be represented as the following mathematical program.

$$\begin{cases} \text{Minimise} & \mathtt{f}_o(S) & \forall o \in \mathcal{O} \\ \text{subject to} & \mathtt{f}_c(S) \leq \mathtt{b}_c & \forall c \in \mathcal{C} \\ & S \in \mathcal{X}_I \end{cases} \tag{5.2}$$

In this modelling, we do not want ALGO to produce non-feasible solutions during its exploration phase. The domain of solutions $\mathcal{X}_I$ then represents any Hamiltonian cycle within the graph $G_{(N,E)}$. With our modelling, a solution $S \in \mathcal{S}_I$ then belongs to $\mathcal{X}_I$ if all nodes from N appear exactly one time in S. Since $\mathcal{X}_I$ represents the set of feasible solutions, there is no constraint defined here, *i.e.*, $\mathcal{C} = \emptyset$. Finally, only one objective is required

for a TSP solution, *i.e.*, $\mathcal{O} = \{o_1\}$, which is to minimise the length of the obtained Hamiltonian cycle. The objective value $\mathtt{f}_{o_1}(S)$ then returns the length of the cycle obtained from $S \in \mathcal{X}_I$.

5.3 Implementation for ALGO

Now that we demonstrated that the TSP can be modelled as an AFOP, this section presents how the TSP was implemented within ALGO's implementation. The followed process is described in Chapter 4, Section 4.4.

An overview of the implementation is provided in Figure 5.2. The class `TSP` inherits from the abstract class `Problem`. The four abstract methods used by the low-level heuristics must then be overridden (presented in Section 5.3.2). An instance of `TSP` must declare a list of objectives and constraints so that the TSP is defined by the mathematical program of an AFOP (Equation 5.2, see more detail in Section 5.3.1). In this implementation, no constraint is given and one objective is defined, hence the class `TourLength` inheriting from `Objective` and overriding the abstract method `compute_value`. Finally, a list of state variables for nodes and edges must be declared to be used as an input of the GNN (described in Section 5.3.3). In the proposed implementation, we declare one state variable for nodes, identified by the class `VisitedNode` inheriting from the class `NodeVariable` and overriding the abstract method `compute`. We also declare two state variables for edges, *i.e.*, we write two classes `VisitedEdge` and `DistanceEdge` inheriting from `EdgeVariable` and overriding the abstract method `compute`. In addition, the class `TSP` defines two methods `get_node_tour` and `get_edge_tour` which return a TSP solution, respectively as a sequence of nodes (see Algorithm 5.1) and a set of edges (see Algorithm 5.2), from an object of type `Solution`.

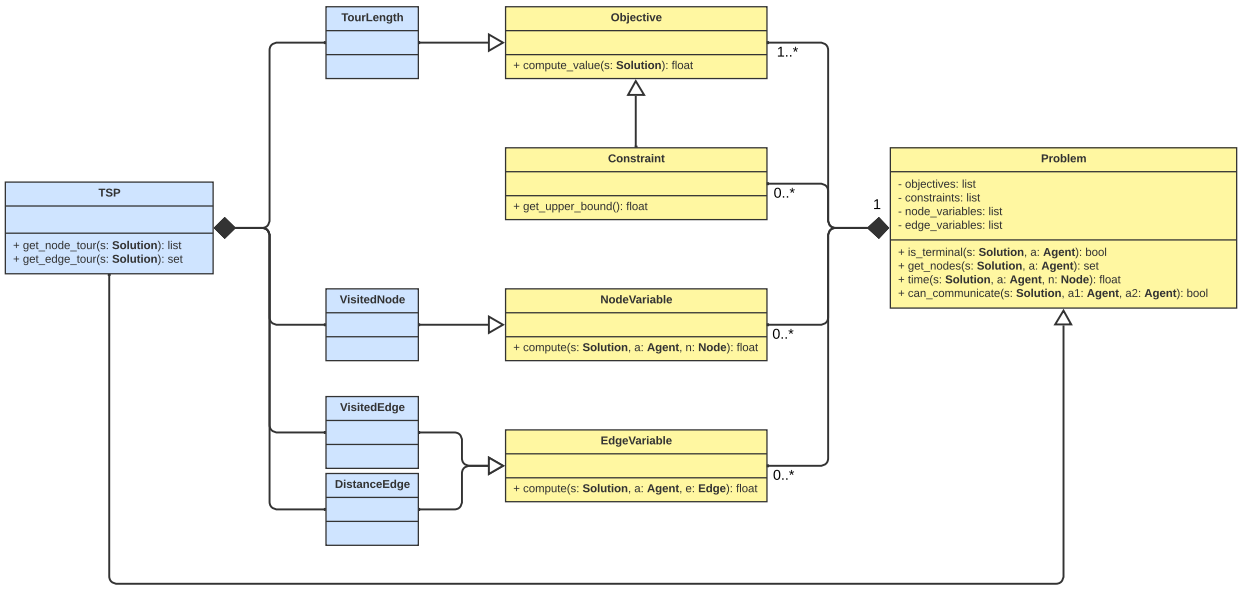


FIGURE 5.2: UML diagram showing the implementation of the TSP.

We remind that the classes `Graph` and `Solution` represent respectively an AFOP instance, *i.e.*, $I = \big(G_{(N,E)}, A\big)$, and an AFOP solution, *i.e.*, $S \in \mathcal{S}_I$. The class `Graph` has three attributes: a list of `Node` objects, a list of `Edge` objects and a list of `Agent` objects. The class `Solution` is seen as a set of `Item` objects, where `Item`, corresponding to a tuple $(a, n)_t \in A \times N \times \mathbb{R}_+$, has three attributes of type `Agent`, `Node` and `float`.

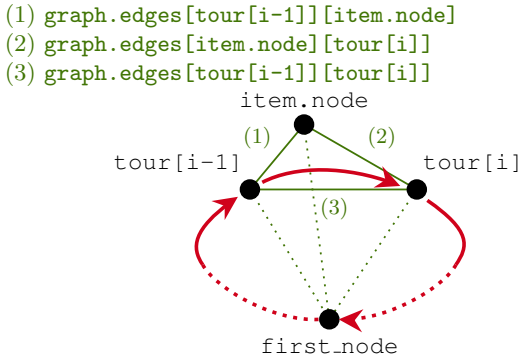
Algorithm 5.1 describes how to map a solution `s` to a TSP solution written as a sequence of nodes. Since `s` is a set of items, it must be transformed into a list `s_sorted` where items are ordered according to their time

(line 2). The tour is initialised with the first node in `s_sorted` (line 4). At each iteration, the index `i_min` at which `item.node` must be added in order to minimise the growth of the tour is calculated (lines 6–14 and Figure 5.3(a)). The node `item.node` is then added into the tour by concatenating the first part of the tour, *i.e.*, `tour[1 : i_min-1]`, a list containing only `item.node`, and the second part of the tour, *i.e.*, `tour[i_min : count+1]` (line 15 and Figure 5.3(b)).

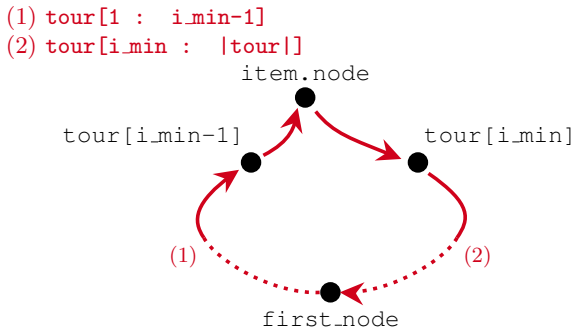
Algorithm 5.1 `TSP::get_node_tour(s)`

Input: `s: Solution`
Output: `list`

```
1:  graph ← s.get_graph()
2:  s_sorted ← sort(s) {according to times}
3:  first_node ← s_sorted[1].node
4:  tour ← [ first_node, first_node ]
5:  for item ∈ s_sorted[2 : |s|] do
6:    i_min ← 2
7:    v_min ← graph.edges[tour[1]][item.node].w + graph.edges[item.node][tour[2]].w
                                                    - graph.edges[tour[1]][tour[2]].w
8:    for i ← 3 to |tour|-1 do
9:      v_tmp ← graph.edges[tour[i-1]][item.node].w + graph.edges[item.node][tour[i]].w
                                                    - graph.edges[tour[i-1]][tour[i]].w
10:     if v_tmp < v_min then
11:       i_min ← i
12:       v_min ← v_tmp
13:     end if
14:   end for
15:   tour ← tour[1 : i_min-1] + [ item.node ] + tour[i_min : |tour|]
16: end for
17: return tour
```



(A) To compute the growth of the tour if `item.node` is added at the i^{th} position, the weight of edges linking `item.node` to the tour at that position (green lines (1) and (2)) are summed. The weight of the edge which would be remove from the tour (green line (3)) is substracted. The index `i_min` is the one minimising that value.



(B) The new tour is obtained by concatenating the first part of the tour (from the beginning to $(i\_min-1)^{th}$ position), `item.node`, and the second part of the tour (from i_min^{th} position to the end).

FIGURE 5.3: Insertion of a node at the position where it minimises the growth of the current tour. The growth of the tour if `item.node` is inserted at the i^{th} position is shown in (a). The insertion of `item.node` into the current tour is shown in (b).

Similarly to the above algorithm, Algorithm 5.2 describes how to map `s` to a TSP solution written as a set of edges. As previously, the solution `s` is ordered according to the time of its items (line 2). The tour is then initialised by the edge linking the first two nodes in `s_sorted` (line 4). A node is added into the tour by removing an edge from it and adding edges linking both adjacent nodes to the node to add. The edge to remove is calculated so that the node `item.node` is added into the tour by minimising its growth (lines 6–14).

The tour is then updated by removing the calculated edges, *i.e.*, `edge_to_remove`, and by adding the two edges to reach `item.node`, *i.e.*, `edges_to_add` (line 15).

Algorithm 5.2 `TSP::get_edge_tour(s)`

```
Input: s: Solution
Output: set
 1: graph ← s.get_graph()
 2: s_sorted ← sort(s) {according to times}
 3: first_edge ← graph.edges[s_sorted[1].node][s_sorted[2].node]
 4: tour ← { first_edge, first_edge }
 5: for item ∈ s_sorted[3 : |s|] do
 6:   v_min ← ∞
 7:   for e ∈ tour do
 8:     v_tmp ← graph.edges[e.n1][item.node].w + graph.edges[item.node][e.n2].w - e.w
 9:     if v_tmp < v_min then
10:       edges_to_add ← { graph.edges[e.n1][item.node], graph.edges[item.node][e.n2] }
11:       edge_to_remove ← e
12:       v_min ← v_tmp
13:     end if
14:   end for
15:   tour ← tour \ { edge_to_remove } ∪ edges_to_add
16: end for
17: return tour
```

5.3.1 Optimisation model

We created a class `TourLength` inheriting from `Objective` and therefore overriding the abstract method `compute_value(s)`, calculating the objective value of solution `s` and described in Algorithm 5.3. It consists in summing the weight of every edge belonging to the tour. The latter is obtained by calling `get_edge_tour(s)` on `tsp`, an object of type `TSP`.

Algorithm 5.3 `TourLength::compute_value(s)`

```
Input: s: Solution
Output: float
1: tour ← tsp.get_edge_tour(s)
2: length ← 0
3: for e ∈ tour do
4:   length ← length + e.w
5: end for
6: return length
```

5.3.2 Low-level heuristics

The implementation of the low-level heuristics corresponds to overriding the abstract methods from the class `Problem` which are used by the low-level heuristic.

A solution is considered terminal when all nodes from the graph have been added into it. In Algorithm 5.4, the method `terminal(s, a)` returns `TRUE` whether the number of items in `s` is the same as the number of nodes in the graph on which it is defined, and `FALSE` otherwise.

Since a solution cannot contain a node more than one time, the method `get_nodes(s, a)` in Algorithm 5.5 returns the set of all nodes which are not contained in the given solution `s`.

Algorithm 5.4 `TSP::terminal(s, a)`

Input: `s: Solution, a: Agent`
Output: `bool`

```
1: graph ← s.get_graph()
2: return |s| = |graph.nodes|
```

Algorithm 5.5 `TSP::get_nodes(s, a)`

Input: `s: Solution, a: Agent`
Output: `set`

```
1: graph ← s.get_graph()
2: candidates ← ∅
3: for n ∈ graph.nodes do
4:   if n ∉ s.get_nodes() then
5:     candidates ← candidates ∪ { n }
6:   end if
7: end for
8: return candidates
```

Given a solution `s`, the time to add the given node `n` is the difference of length between the tour corresponding to `s` and the tour after adding `n`. Algorithm 5.6 describes that process. The tour corresponding to solution `s` is obtained by calling `get_node_tour(s)` on an instance of the problem (line 2). The difference of length is calculated for each possible index at which `n` can be added (line 5). Finally the lowest difference is returned.

Algorithm 5.6 `TSP::time(s, a, n)`

Input: `s: Solution, a: Agent, n: Node`
Output: `float`

```
1: graph ← s.get_graph()
2: tour ← tsp.get_node_tour(s)
3: delta ← ∞
4: for i ← 1 to |s| do
5:   delta_tmp ← graph.edges[tour[i]][n].w + graph.edges[tour[i+1]][n].w
                                          - graph.edges[tour[i]][tour[i+1]].w
6:   delta ← min(delta, delta_tmp)
7: end for
8: return delta
```

In this implementation, only one agent is considered. The method `can_communicate` described in Algorithm 5.7 is thus never been called.

Algorithm 5.7 `TSP::can_communicate(s, a1, a2)`

Input: `s: Solution, a1: Agent, a2: Agent`
Output: `bool`

```
1: return FALSE
```

5.3.3 State variables

The implementation of state variables is used as an input to the GNN. We should then provide any information that we want the GNN to consider in order to return a relevant embedded solution.

In this implementation, only one state variable is given for nodes. It states if a node has already been added into the current solution and is represented by an instance of `VisitedNode`. Algorithm 5.8 shows the state variable of a given node `n` is 0.0 if `n` is in the given solution `s`, and 1.0 otherwise.

Algorithm 5.8 `VisitedNode::compute(s, a, n)`

```
Input: s: Solution, a: Agent, n: Node
Output: float
1: if n ∈ s.get_nodes() then
2:     return 0.0
3: else
4:     return 1.0
5: end if
```

Two state variables are defined for edges: its distance (depicted by an instance of `DistanceEdge`) and whether it belongs to the tour corresponding to the given solution `s` (represented by an instance of `VisitedEdge`). In Algorithm 5.9, the state variable of the given edge `e` equals to its weight if `e` is "involved" for the choice of a new node, and 0.0 otherwise. An edge `e` is considered "involved" if it may belong to the tour of the solution obtained after the next iteration, *i.e.*, if it belongs to the tour of `s`, or if at least one of its adjacent node is not contained in `s`.

Algorithm 5.9 `DistanceEdge::compute(s, a, e)`

```
Input: s: Solution, a: Agent, e: Edge
Output: float
1: tour ← get_edge_tour(s)
2: if e ∈ tour or e.n1 ∉ s.get_nodes() or e.n2 ∉ s.get_nodes() then
3:     return e.w
4: else
5:     return 0.0
6: end if
```

Algorithm 5.10 describes the state variable determining if an edge belongs to the given solution. It returns 0.0 if it is the case, and 1.0 otherwise.

Algorithm 5.10 `VisitedEdge::compute(s, a, e)`

```
Input: s: Solution, a: Agent, e: Edge
Output: float
1: tour ← get_edge_tour(s)
2: if e ∈ tour then
3:     return 0.0
4: else
5:     return 1.0
6: end if
```

5.4 Experiments

This section presents the experiments performed to assess the performance of ALGO on the TSP. These have been conducted on the High Performance Computing (HPC) platform of the University of Luxembourg [Varrette et al., 2022]. We first describe the training process in Section 5.4.1. We then present all heuristics which have been used as a comparison basis for the heuristic generated by ALGO. We finally present our results in Section 5.4.3.

5.4.1 Training process

Training instances are complete graphs which have been randomly generated by placing nodes uniformly on a 1000×1000 grid. The number of nodes is also uniformly chosen from 50 and 100. ALGO has been trained on 30 000 episodes with the parameterisation summarised in Table 5.1. The definition and notation of ALGO parameters are provided in the previous chapter, Section 4.3, where the high-level part of ALGO, *i.e.*, related to RL, is described.

Parameter	**Notation**	**Value**
ALGO		
learning rate	α	0.01
exploration rate	ϵ	0.05
discount factor	γ	0.9
size of movement frame	τ	30
embedding dimension	p	32
number of layers	L	3
memory size	$\lvert\mathcal{M}\rvert$	1000
mini-batch size	$\lvert\mathcal{B}\rvert$	32
training instances		
number of episodes		30 000
minimal number of nodes		50
maximal number of nodes		100
array size		1000×1000

TABLE 5.1: Experimental parameters used for training ALGO for the TSP

5.4.2 Comparison heuristics

In order to assess the performance of the heuristic generated by ALGO, we have executed it on testing instances along with six classical existing heuristics for the TSP. We briefly describe each of them in the remainder of this section with a pseudo-code considering a complete weighted graph as an instance and a subset of edges from that graph as a solution. A summary is provided in Table 5.2 which indicates the type of heuristic and their time complexity.

Name	**Type**	**Time complexity**
Nearest Neighbour	Constructive	$\mathcal{O}\left(n^2\right)$
Greedy Edge	Constructive	$\mathcal{O}\left(n^2 \log(n)\right)$
Nearest Insertion	Constructive	$\mathcal{O}\left(n^2\right)$
Farthest Insertion	Constructive	$\mathcal{O}\left(n^2\right)$
Christofides	Constructive	$\mathcal{O}\left(n^3\right)$
2-opt	Perturbative	-

TABLE 5.2: Summary of TSP heuristics used as a comparison basis.

Nearest Neighbour (Algorithm 5.11) It consists in building a tour as a sequence of nodes. At each iteration, an unvisited node is added at the end of the solution. The chosen node is the nearest one from the last node added into the solution. The first node is chosen randomly.

Algorithm 5.11 *Nearest Neighbour* heuristic for the TSP

Input: complete weighted graph $G_{(N,E)}$
Output: $tour \subset E$

$tour \leftarrow \emptyset$
$n_{curr} \leftarrow select(N)$
$N_{rem} \leftarrow N \backslash \{n_{curr}\}$
while $|N_{rem}| \neq \emptyset$ **do**
  $n_{next} \leftarrow \arg\min_{n \in N_{rem}} weight((n_{curr}, n))$
  $tour \leftarrow tour \cup \{(n_{curr}, n_{next})\}$
  $N_{rem} \leftarrow N_{rem} \backslash \{n_{next}\}$
  $n_{curr} \leftarrow n_{next}$
end while
return *tour*

Greedy Edge (Algorithm 5.12) Edges are first sorted according to their weight. Starting from the smallest one, they are iteratively considered to be added into the solution. At each iteration, if adding the considered edge can still lead to a feasible solution, it is added into the solution.

Algorithm 5.12 *Greedy Edge* heuristic for the TSP

Input: complete weighted graph $G_{(N,E)}$
Output: $tour \subset E$

$tour \leftarrow \emptyset$
$E_{ord} \leftarrow sort(E)$ {according to weights}
for all $e \in E_{ord}$ **do**
  if $tour \cup \{e\}$ can lead to a feasible solution **then**
    $tour \leftarrow tour \cup \{e\}$
  end if
end for
return *tour*

Nearest Insertion (Algorithm 5.13) Starting from a subtour obtained with the two nearest nodes, it iteratively grows this subtour by adding a node at the position where its length increases least. At each iteration, the node chosen is the nearest one from a node already in the subtour.

Algorithm 5.13 *Nearest Insertion* heuristic for the TSP

Input: complete weighted graph $G_{(N,E)}$
Output: $tour \subset E$

$(n_1, n_2) \leftarrow \arg\min_{e \in E} weight(e)$
$tour \leftarrow \{(n_1, n_2)\}$
$N_{rem} \leftarrow N \backslash \{n_1, n_2\}$
while $N_{rem} \neq \emptyset$ **do**
  $n \leftarrow \arg\min_{n' \in N_{rem}} \min_{n'' \in N \backslash N_{rem}} weight((n', n''))$
  if $|tour| = 1$ **then**
    $tour \leftarrow tour \cup \{(n_1, n), (n, n_2)\}$
  else
    $(n_x, n_y) \leftarrow \arg\min_{(n', n'') \in tour} weight((n', n)) + weight((n, n'')) - weight((n', n''))$
    $tour \leftarrow tour \cup \{(n_x, n), (n, n_y)\} \backslash \{(n_x, n_y)\}$
  end if
  $N_{rem} \leftarrow N_{rem} \backslash \{n\}$
end while
return *tour*

Farthest Insertion (Algorithm 5.14) Similarly to the *Nearest Insertion* heuristic, it iteratively grows a subtour by adding a node at the position where its length increases least. However, the first subtour is made

with the two farthest nodes, and at each iteration, the chosen node is the farthest one from a node already in the subtour.

Algorithm 5.14 *Farthest Insertion* heuristic for the TSP

Input: complete weighted graph $G_{(N,E)}$
Output: $tour \subset E$

$(n_1, n_2) \leftarrow \arg\min_{e \in E} weight(e)$
$tour \leftarrow \{(n_1, n_2)\}$
$N_{rem} \leftarrow N \backslash \{n_1, n_2\}$
while $N_{rem} \neq \emptyset$ **do**
  $n \leftarrow \arg\max_{n' \in N_{rem}} \min_{n'' \in N \backslash N_{rem}} weight((n', n''))$
  if $|tour| = 1$ **then**
    $tour \leftarrow tour \cup \{(n_1, n), (n, n_2)\}$
  else
    $(n_x, n_y) \leftarrow \arg\min_{(n', n'') \in tour} weight((n', n)) + weight((n, n'')) - weight((n', n''))$
    $tour \leftarrow tour \cup \{(n_x, n), (n, n_y)\} \backslash \{(n_x, n_y)\}$
  end if
  $N_{rem} \leftarrow N_{rem} \backslash \{n\}$
end while
return *tour*

Christofides (Algorithm 5.15) The minimum spanning tree is first obtained from the given graph. In this tree, nodes with an odd degree are selected to form a subgraph on which is run a minimum-weight perfect matching. Both the spanning tree and the matching are united which gives a graph where all nodes have an even degree. An Euler tour can then be calculated, from which the solution is obtained by removing repeated nodes.

Algorithm 5.15 *Christofides* heuristic for the TSP

Input: complete weighted graph $G_{(N,E)}$
Output: $tour \subset E$

$E_{mst} \leftarrow minimum\_spanning\_tree\left(G_{(N,E)}\right)$
$N_{odd} \leftarrow \{n \mid |\{n' | (n, n') \in E_{mst}\} \cup \{n' | (n', n) \in E_{mst}\}| \text{ is odd}\}$
$E_{odd} \leftarrow \{(n, n') \mid n \in N_{odd}, n' \in N_{odd}\}$
$E_{mwpm} \leftarrow minimum\_weight\_perfect\_matching\left(G_{(N_{odd}, E_{odd})}\right)$
$N_{tour} \leftarrow eulerian\_path\left(G_{(N, E_{mst} \cup E_{mwpm})}\right)$ {ordered}
$n_{curr} \leftarrow N_{tour}[1]$
$N_{rem} \leftarrow N \backslash \{n_{curr}\}$
for all $n \in N_{tour}$ **do**
  if $n \in N_{rem}$ **then**
    $tour \leftarrow tour \cup \{(n_{curr}, n)\}$
    $N_{rem} \leftarrow N_{rem} \backslash \{n\}$
    $n_{curr} \leftarrow n$
  end if
end for
return *tour*

2-opt (Algorithm 5.16) It iteratively updates a solution by swapping two edges. At each iteration, if swapping two edges from the current solution decreases the length of the tour, they are swapped.

5.4.3 Results

The heuristic generated by ALGO and the heuristics presented in Section 5.4.2 have been executed on 1000 instances from different classes, each class being defined by the range of number of nodes. Instances have been

Algorithm 5.16 *2-opt* heuristic for the (symmetric) TSP

Input: complete weighted graph $G_{(N,E)}$ **and** $tour \subset E$
Output: $tour \subset E$

```
improvement ← TRUE
while improvement do
    improvement ← FALSE
    for all (n1, n2) ∈ tour do {assuming that n1 → n2}
        for all (n3, n4) ∈ tour\{(n1, n2)} do {assuming that n3 → n4}
            if weight((n1, n3)) + weight((n2, n4)) < weight((n1, n2)) + weight((n3, n4)) then
                tour ← tour ∪ {(n1, n3), (n2, n4)}\{(n1, n2), (n3, n4)}
                improvement ← TRUE
            end if
        end for
    end for
end while
return tour
```

randomly generated in the same way as for the training process (see Section 5.4.1). For example, instances from the class `100-200` have a number of nodes uniformly generated between 100 and 200. After executing all heuristics on one instance, the score given to heuristics is the approximation ratio, *i.e.*, the ratio of the length of the obtained tour to the length of the optimal tour. For each instance, the shortest tour obtained among all heuristics is considered to be the optimal one for the computation of the approximation ratio. The average over the 1000 testing instances per class is retained and shown in Figure 5.4.

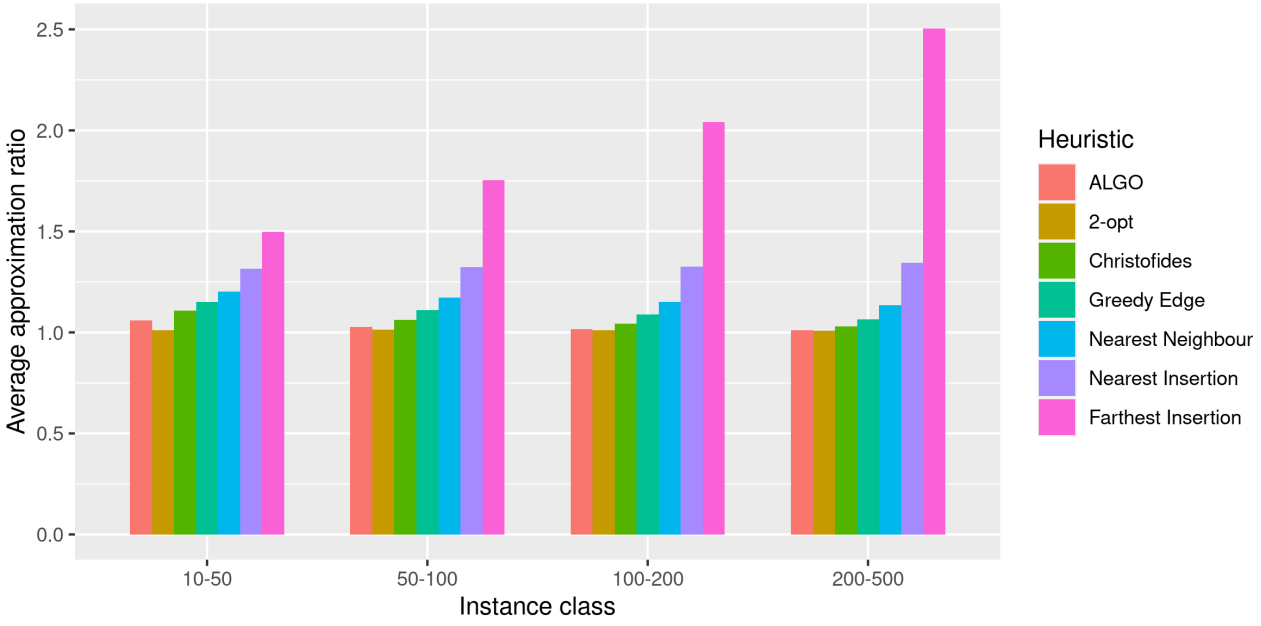


FIGURE 5.4: Results obtained by executing different TSP heuristics on 1000 instances from four classes.

Figure 5.4 shows that the heuristic generated by ALGO provides better solutions than other constructive heuristics. This outperformance is confirmed with a Pairwise Wilcoxon Signed-Rank (PWSR) test (see Table 5.3). The generated heuristic, with a quadratic running time, thus outperforms *Greedy Edge* heuristic, running in $\mathcal{O}\left(n^2 \log(n)\right)$, and *Christofides* heuristic, running in $\mathcal{O}\left(n^3\right)$. It is also interesting to notice that the generated heuristic drastically improves the performance of other insertion heuristics, which shows the impact of the learning process. For all instance classes, the *Nearest Insertion* heuristic indeed provides solutions approximately 1.3 times worse than the generated heuristic. The *Farthest Insertion* heuristic's quality decreases with the higher

number of nodes. The returned solutions are approximately 1.4 times worse than the ones provided by the generated heuristic for instances from `10-50`, and it goes up to 2.5 times worse for instances from `200-500`.

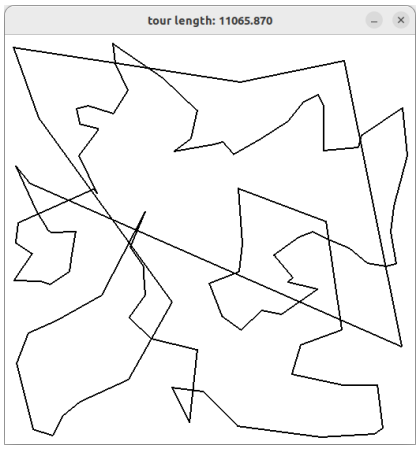
Another important property of ALGO is its good stability. It has indeed been trained on instances from `50-100`. The performance of the generated heuristic is however similar for any instance class. The approximation ratio even slightly decreases with the higher number of nodes. Another interesting aspect of the heuristic generated by ALGO is its lower standard deviation. Only *Christofides* heuristic for `100-200` instances shows a lower deviation (1016 for 1015). In general, that standard deviation is close to the one provided by *Christofides* and *2-opt* heuristics. This ability also demonstrates the good stability of the heuristic.

Despite the good performance of the generated heuristic, the provided solutions are in average worse than solutions obtained with the *2-opt* heuristic, the only perturbative heuristic. This counter performance is however alleviated by the better running time. In addition, solutions returned by the generated heuristic are often 2-opt local optima. It can be viewed on Figure 5.5 where all heuristics have been executed on an instance with 100 nodes (randomly generated as for the training process). The solution obtained by the generated heuristic, in Figure 5.5(g), does not have crossing edges which means it is a 2-opt local optimum. With the decrease in the approximation ratio of the generated heuristic with the high number of nodes, the performance becomes even similar to the *2-opt* heuristic. This is determined by the PWSR test, whose results are depicted in Table 5.3, stating that there is no statistical confidence in distinguishing the performance of both heuristics on instances from `200-500`.

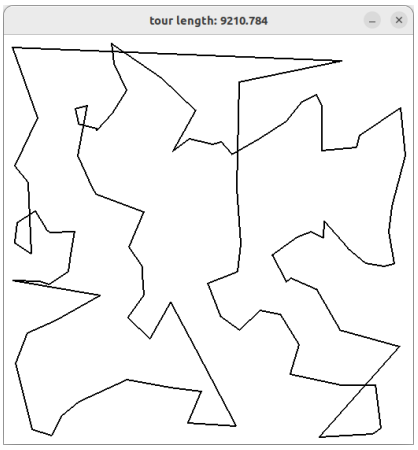
After observing the good performance of the heuristic generated by ALGO on instances randomly generated, we wanted the execute it on more diverse instances. We thus applied the heuristic learnt by ALGO from random instances to instances from TSPLIB [Reinelt, 1991]. All heuristics have then been executed on 104 TSPLIB instances, with a size going from 17 to 2152 nodes. The comparison between heuristics is made according to a PWSR test and is shown in Table 5.3. In that table, the mean and standard deviation is less relevant than for random instances since there is much more diversity among instances. When two heuristics provide distinct results with a 95% confidence, the mean and standard deviation are however useful to determine which heuristic outperforms the other one.

From Table 5.3, we can see that the heuristic generated by ALGO outperforms other heuristics with a quadratic running time, hence other insertion heuristics. It shows that ALGO could learn knowledge from random instances that the generated heuristic uses for more diverse instances. The difference with the *Nearest Insertion* is similar to the difference for random instances, *i.e.*, the solution is approximately 1.3 times worse. The difference with the *Farthest Insertion* is even bigger than for big random instances since the solutions are more than 3.5 times worse than solutions provided by the generated heuristic.

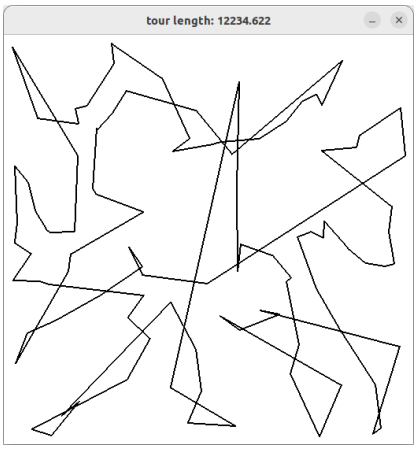
However, the heuristic generated by ALGO shows a lack of performance compared to heuristics with a higher running time. Even if there is no statistical distinction of performance with the *Greedy Edge* heuristic, the generated heuristic is outperformed by *Christofides* and *2-opt* heuristics. According to the good performance of the heuristic generated by ALGO on random instances and its stability, we believe that this lack of performance on TSPLIB could be reduced by diversifying training instances. For example, instances with a non-uniform distribution of nodes in the 1000×1000 square could be used along with instances that we have used for these experiments.



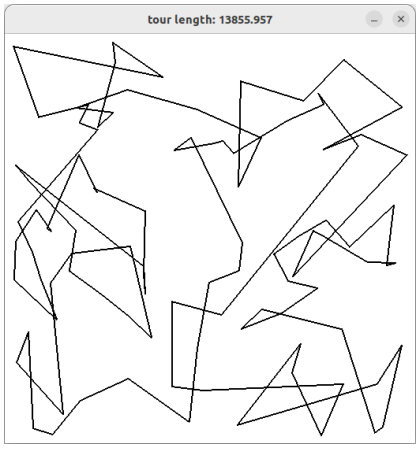
(A) Nearest Neighbour



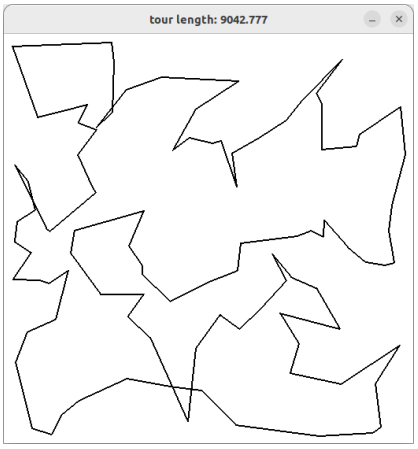
(B) Greedy Edge



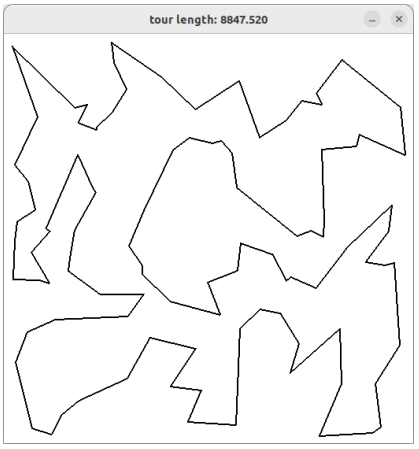
(C) Nearest Insertion



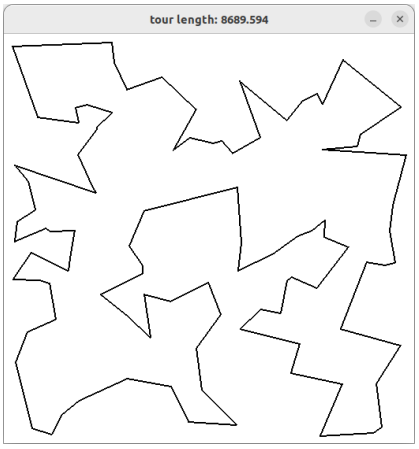
(D) Farthest Insertion



(E) Christofides



(F) 2-opt



(G) ALGO

FIGURE 5.5: Tour obtained from different heuristics on a same instance where 100 nodes have been uniformly spread in a 1000×1000 square.

	RANDOM				TSPLIB
	10–50	**50–100**	**100–200**	**200–500**	
ALGO	4.702e+3 ± 9.966e+02	7.378e+03 ± 7.540e+02	1.031e+04 ± 1.016e+03	1.544e+04 ± 1.938e+03	2.591e+05 ± 2.127e+06
Nearest Neighbour	5.350e+03 ± 1.225e+03	8.424e+03 ± 9.568e+02	1.168e+04 ± 1.206e+03	1.735e+04 ± 2.182e+03	2.855e+05 ± 2.389e+06
p-value	8.452e-153	3.165e-164	4.606e-165	3.331e-165	1.027e-03
Greedy Edge	5.117e+03 ± 1.135e+03	7.976e+03 ± 8.499e+02	1.104e+04 ± 1.118e+03	1.627e+04 ± 2.006e+03	2.559e+05 ± 2.105e+06
p-value	1.234e-130	1.981e-148	1.754e-156	5.870e-160	4.680e-01
Nearest Insertion	5.876e+03 ± 1.425e+03	9.508e+03 ± 1.123e+03	1.346e+04 ± 1.518e+03	2.059e+04 ± 2.803e+03	3.487e+05 ± 2.936e+06
p-value	8.662e-165	3.331e-165	3.331e-165	3.331e-165	9.354e-17
Farthest Insertion	6.764e+03 ± 1.996e+03	1.266e+04 ± 2.255e+03	2.080e+04 ± 3.593e+03	3.860e+04 ± 8.260e+03	1.002e+06 ± 8.679e+06
p-value	5.336e-165	3.331e-165	3.331e-165	3.331e-165	2.647e-17
Christofides	4.928e+03 ± 1.044e+03	7.643e+03 ± 7.669e+02	1.059e+04 ± 1.015e+03	1.572e+04 ± 1.941e+03	2.494e+05 ± 2.057e+06
p-value	4.114e-116	1.153e-118	7.834e-120	1.252e-122	3.869e-02
2-opt	4.523e+03 ± 1.064e+03	7.288e+03 ± 7.683e+02	1.026e+04 ± 1.036e+03	1.543e+04 ± 1.944e+03	2.434e+05 ± 2.016e+06
p-value	2.057e-72	6.262e-18	6.727e-06	2.293e-01	1.180e-10

TABLE 5.3: Comparison between the heuristic generated by ALGO and other heuristics according to a pairwise Wilcoxon signed-rank test. A blue cell indicates that the heuristic generated by ALGO outperforms the given heuristic (row name) for the given instance class (column name) with a 95% statistical confidence. A gray cell indicates that there is no statistical confidence to differentiate both heuristics.

5.5 Conclusion

In this chapter, we intended to validate ALGO. For that purpose, we used it to generate heuristics for a classical optimisation problem, the Travelling Salesman Problem (TSP). We first provide our implementation of the TSP, *i.e.*, we describe the steps to follow in order to achieve our objective to generate TSP heuristics. That section can be used as a concrete example for using our generic model of hyper-heuristic. We want to provide an exhaustive description at the coding level, *i.e.*, what classes to create, what methods to override, so that the implementation can be used by ALGO.

In the second part of this chapter, we describe our experiments. We explain how we trained ALGO to generate a heuristic to tackle the TSP. For the training process, instances have been randomly generated by placing nodes uniformly on a square. The number of nodes has also been uniformly generated between 50 and 100, *i.e.*, instances belong to the class `50-100`. We also briefly describe six existing classical TSP heuristics. We used them as a comparison basis to assess the performance of the heuristic generated by ALGO. All heuristics have then been executed on random instances from different classes. Our results show that the heuristic generated by ALGO outperforms all other constructive heuristics for any instance class, which demonstrates a good performance and a good stability. Only the *2-opt* heuristic provides better performance on random instances, but with a difference decreasing with the greater size of instances. There is even no statistical distinction between both heuristics for instances from the class `200-500`.

We also executed heuristics on instances from TSPLIB. We could notice a slight decline in performance from the generated heuristic. It still outperforms heuristics with a quadratic running time, including other insertion

heuristics which shows the impact of the learning process. There is however no statistical distinction with the *Greedy Edge* heuristic and it is outperformed by *Christofides* and *2-opt* heuristics. This lack of performance is due to a lack of diversity in training instances. We believe that including instances generated non-uniformly in the training set would improve the quality of the heuristic generated by ALGO. This task could be part of future work, but our purpose with ALGO was to tackle dynamic problems. We have thus shown in this chapter the stability of ALGO and its ability to generate efficient heuristics for a classical optimisation problem. In next part, we use ALGO in a real-world context, which is to automate the design of swarming behaviours.

Part III

Use Case: Covering an Area with a Swarm of UAVs

Chapter 6

Design of Robot/UAV Swarms

Contents

6.1 Introduction

The usage of Unmanned Aerial Vehicles (UAVs) finds its roots where human intervention might be difficult, risky or costly. Initially thought for military purposes, UAVs have demonstrated their tremendous potential in civilian applications such as parcel delivery, rescue mission or environment monitoring. Current applications nonetheless rely on the usage of a single UAV (remotely piloted or autonomous), which faces multiple limitations, such as its range of action, payload capacity and system resilience.

Using several autonomous UAVs simultaneously as a swarm is one promising solution to address these limitations. Inspired by natural phenomena, *e.g.*, birds flocks or ant colonies, swarm intelligence allows one to achieve complex tasks while solely relying on local decisions and interactions. Such a distributed and self-organised approach allows multi-UAV systems to be more scalable, resilient and flexible.

While robot swarming has received an increasing interest over the past years, no single definition yet exists. In this work, the definition of Arnold et al. [2019] will be considered. The authors define a robot swarm as "a group of three or more robots that perform tasks cooperatively while receiving limited or no control from human operators". This definition implies three requirements. First, a swarm should first contain at least three robots/UAVs, otherwise the internal interaction would be absent or too low. Second, robots in a swarm should cooperate to perform tasks which implies that the behaviour of one will influence the behaviour of the others. Third, a swarm finally receives few or no control from human operators which makes a swarm differ from a

fleet in which there is a centralised behaviour, with a central unit shared by every robot/UAV. The behaviour of a swarm should thus be distributed. The few control from human operators implies that the behaviour to achieve a task must emerge from the local interactions between robots/UAVs.

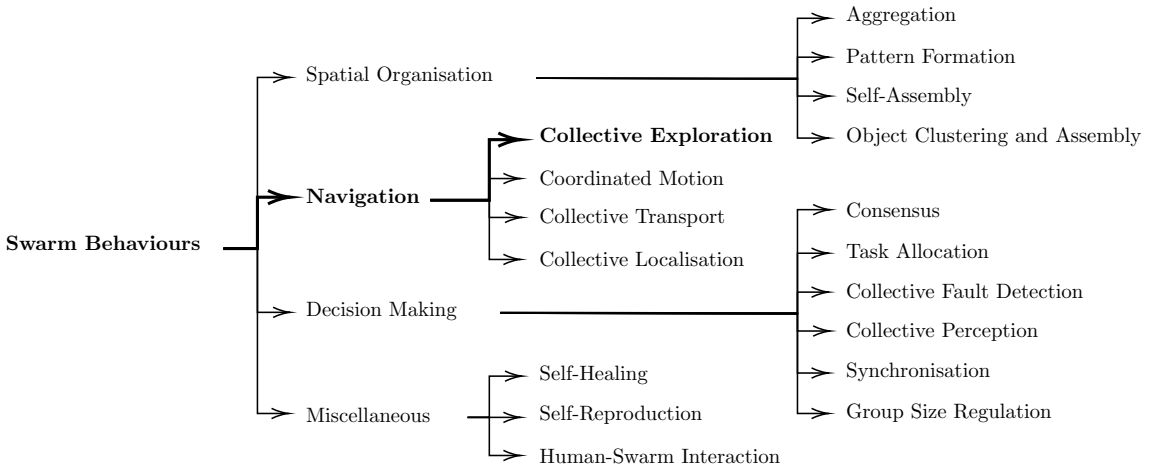


FIGURE 6.1: Classification of swarm behaviours proposed by Brambilla et al. [2013] and extended by Schranz et al. [2020]

Such behaviours have been classified by Brambilla et al. [2013], Schranz et al. [2020] in four categories as depicted in Figure 6.1: spatial organisation (robots have to organise themselves in order to form patterns or simply aggregate), navigation (robots have to coordinate their movement), decision making (robots have to influence themselves in order to take a common decision) and miscellaneous (other significant behaviours which could not belong to any of the three previous categories). Since this work focuses on the area coverage by a swarm of UAVs, it falls in the "navigation" category containing four subcategories: collective exploration, coordinated motion, collective transport and collective localisation. The remainder of this section focuses on collective exploration since it is the target of this work.

In the remainder of this chapter, we present existing works about the design of robot/UAV swarms. We start this state of the art by introducing techniques based on a manual design in Section 6.2. We then present approaches aiming at automating the design of swarming behaviours in Section 6.3. Finally, we describe in Section 6.4 an optimisation problem that we defined for the coverage of an area by a swarm of UAVs.

6.2 Manual design of robot/UAV swarms

The literature contains a variety of studies based on path planning to manually design the behaviour of robots or UAVs, including in the context of coverage as surveyed in Cabreira et al. [2019]. The idea is to compute the optimal path of robots/UAVs offline, *i.e.*, before starting the mission. In some works, these pre-computed paths may be updated or recomputed. For instance Siemiatkowska and Stecz [2021] recalculate a Vehicle Routing Planning (VRP) when UAVs detect a threat on their way. Besides, the problem of path planning is often formulated as a variant of VRP. It is also the case of Semiz and Polat [2020] who solve a problem of area coverage (different from the problem tackled in this work). These path planning techniques are not comparable the work proposed in this thesis. Indeed, most of these studies do not suit the above definition of a swarm. Nevertheless, with path planning techniques, UAVs are following the path computed beforehand, which implies a strong control from a human operator. Moreover, these techniques are usually exact methods and/or metaheuristics which are too expensive for an online usage.

Another way to manually design robot/UAV swarms considers online techniques, based on the direct or indirect communication between robots/UAVs. Some of those are based on line-of-sight communication. The work of Nouyan et al. [2008] consists in chaining a swarm of robots between two points where robots have different colours according to the direction of the chain. Ducatelle et al. [2011] propose a communication system for making a swarm of robots reach an unknown target point. However, when dealing with collective exploration, the vast majority relies on a stigmergy process such as pheromone-based systems. Some works focus on the design of the swarm system Sun et al. [2019], Na et al. [2020] which abstracts from the application, *i.e.*, the collective task. The idea is to design a pheromone system as close as possible to how pheromone would evolve in nature. These pheromone systems are used to investigate the impact of using multiple pheromones in the context of a collective task Liu et al. [2020, 2021].

Most of the literature, meanwhile, deals with task-specific systems, *i.e.*, how to use the pheromone for the wanted collective task. Kuiper and Nadjm-Tehrani [2006] first introduced a pheromone-based system to optimise the coverage of an area by a swarm of UAVs. In that work, UAVs have to choose a direction according to the amount of repulsive pheromones left by other UAVs. Rosalie et al. [2018] proposed an extension of the latter work where the random aspect is replaced by a chaotic system. The objective is to obtain a movement which seems unpredictable from the outside, while being deterministic. Danoy et al. [2015] combined the work of Kuiper and Nadjm-Tehrani [2006] with a multi-hop clustering approach in order to consider the connectivity of the swarm during the coverage task. The added connectivity objective aims to maintain a good exchange of information inside the swarm and thus improve its performance. Brust et al. [2017] extended the previous work with a dual-pheromone model in order to tackle three objectives of the swarm: area coverage, swarm connectivity and target tracking.

Although such pheromone-based approaches have demonstrated good results, Hunt et al. [2019] show that they face limitations particularly when the density of the swarm is very high (the number of robots/UAVs relative to the size of the area). Added to this limitation, such a pheromone-based behaviour has to be designed manually, which can be very time-consuming and is specific to an application. For instance, the pheromone evaporation rate or the balance between different pheromones (in case of using multiple types) may differ according to the emergent task of the swarm.

6.3 Automated design of robot/UAV swarms

One approach that recently raised some interest consists in automating the design of swarming behaviours, as mentioned by Birattari et al. [2019]. This process can be assimilated to a hyper-heuristic which has been deeply surveyed by Burke et al. [2013], Epitropakis and Burke [2018], Burke et al. [2019]. Hyper-heuristics are used more generally in optimisation for automating the design of heuristics for a given problem. Li and Malik [2017] first assimilate that process to *learning to optimise* where they view an optimisation algorithm as a policy in a Markov Decision Process (MDP). In the context of robot/UAV swarms, a heuristic corresponds to the local behaviour of each UAV used to tackle global problems like optimising the coverage of an area.

Another way to automate the design of a swarm of robots/UAVs is proposed by Kouzehgar et al. [2020]. They use Reinforcement Learning (RL) to generate the behaviour of a swarm of autonomous buoys to cover an area. The RL algorithm makes it possible to adapt to any shape change of the area to cover, even while the coverage occurs. Such approaches are however assimilated to metaheuristics since the behaviour learnt is specific to the current shape of the area. We will hence not detail similar approaches in the remainder of this section.

6.3.1 Selective approaches

The process of hyper-heuristics has only been recently applied in the context of UAV/robot swarm and is still an open research area as outlined by Birattari et al. [2019]. The latter manifesto presents the current challenges of "automatically designing a swarm for any mission with a given class". Most existing techniques consist in providing UAVs/robots a set of predefined actions, so that they can learn at each step which action to use according to the given task and the state of the environment. For instance, Birattari et al. [2021] developed AutoMoDe which is a framework for automating the design of a robot swarm with different specialisations (corresponding each to a set of behaviours). Ligot et al. [2022] extend the latter work with a way to automatically generate different missions along with a performance indicator. Reinforcement Learning (RL) is mainly used as a high-level algorithm. The idea is to represent the actions of RL by the different behaviours of the swarm. Yu et al. [2018] used it in the context of a self-assembling swarm and Yu et al. [2019] in the more specific case of surface cleaning. It is also used by Nagavalli et al. [2017] for choosing a sequence of behaviours for a given task, like navigating in an environment with obstacles.

These selective approaches are prominently used in the context of robot/UAV swarming, since it is convenient to provide specific behaviours to a robot or UAV. The searching space of behaviours is however limited to the actions specified by the user. Similarly to classical hyper-heuristics, considering generative approaches considerably increase the size of the searching space. In the context of robot/UAV swarm, it permits to obtain more complex behaviours with a greater adaptability to dynamic environment.

6.3.2 Generative approaches

A second and more recent branch of hyper-heuristics relying on a generative approach exists in the literature but it has not received much attention yet in the robot/UAV community. Generative hyper-heuristics do not need a set of predefined low-level heuristics, but a set of "building blocks" from which the high-level algorithm will construct possibly unseen low-level heuristics. We were first to explore this research direction [Duflo et al., 2020a,b, 2021, 2022a,b]. We represented the task of the swarm as a multi-objective optimisation problem (presented in Section 6.4). We then used ALGO to generate distributed heuristics for that problem.

6.4 Coverage of a Connected-UAV Swarm (CCUS)

The following UAV swarm surveillance scenario is considered in this work: several UAVs equipped with ad hoc communication capabilities take off from different bases to cover a common area (see Figure 6.2(a) with blue squares as bases of UAVs). These UAVs evolve as a swarm which needs to cover as much area as possible, as fast as possible while remaining as connected as possible. That connectivity aspect is crucial in such a distributed system since it will enhance the communication within the swarm. The local information of each UAV will therefore spread faster, which in turn will result in an improved performance of the global system (i.e., swarm). In this context, the information exchanged by UAVs is the current solution according to their distributed knowledge. A formalisation of a solution is described in Section 6.4.1.4. Once the UAVs have finished covering the area, they must return to their base (i.e., initial starting point). Such a surveillance scenario can find numerous civil applications that require a fast and efficient coverage of a large area, such as search and rescue missions, forest fire or pollution detection.

We designed in [Duflo et al., 2020b] the Coverage of a Connected-UAV Swarm (CCUS) optimisation problem for that purpose. It optimises the coverage of an area by a swarm of UAVs with two objectives: the coverage

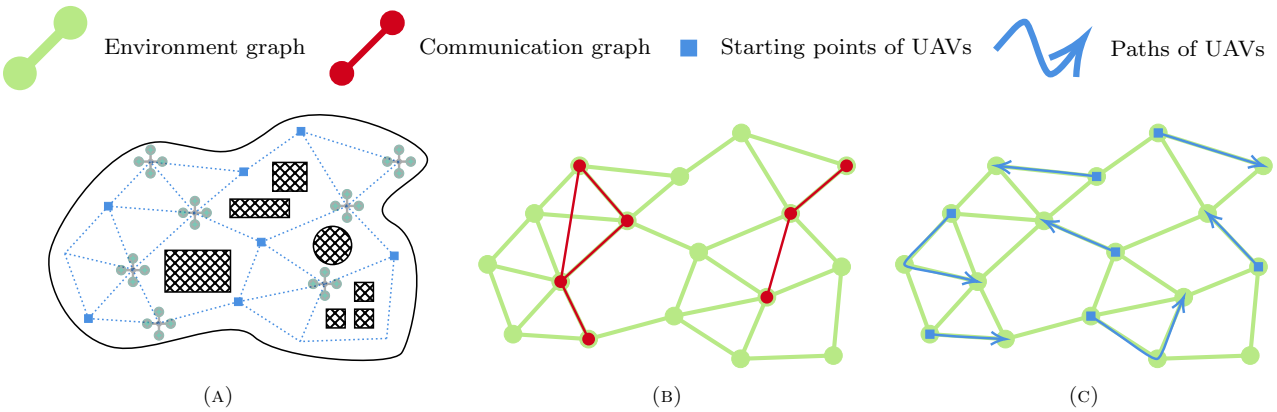


FIGURE 6.2: Swarm of UAVs covering an area with obstacles. The UAVs are flying from different bases (blue squares) following a discretisation of the map (dashed blue lines), as shown in 6.2(a). At that time, the environment graph (in green) and the communication graph (in red) are represented in 6.2(b). The current solution as a set of paths (in blue) is depicted in 6.2(c).

speed and the swarm connectivity. We later extended CCUS in [Duflo et al., 2022b] to a three-objective model, referred to as CCUS3O, which targets coverage time, coverage rate and connectivity. What motivates the introduction of a third objective is that the coverage objective in CCUS includes both the coverage time and the coverage rate in one single value. Such a scalarisation introduces a bias that not only can negatively impact the performance but also reduces the explainability of the objective value. Using three objectives in CCUS3O thus permits to have more atomic objectives. The remainder of this section first presents a formal definition of instances and solutions (see Section 6.4.1), followed by the evaluation of solutions according to the three objectives (see Section 6.4.2).

6.4.1 Formal expression

The CCUS3O model considers two entities: an environment graph which is a discretisation of the area and the communication graph which depicts the communication network of the swarm. Both are represented in Figure 6.2(b) and detailed hereinafter.

6.4.1.1 Environment graph

The environment graph is a discretisation of the environment which defines where UAVs can move and which ways they can take (see Figure 6.2(b)). It is represented as $G^e = (V, E^e)$ with V the set of vertices and E^e the set of edges. This graph is here considered static, *i.e.*, the set of edges remains the same along with their length, and connected.

As a notation, $dist : V^2 \rightarrow \mathbb{R}$ returns the length of the shortest path between two given vertices. The neighbourhood of a vertex v is besides represented by $\mathcal{N}^e(v)$ which is the set of every vertex linked to v in the environment graph.

6.4.1.2 Communication graph

The communication graph indicates the position of UAVs (i.e., vertices) and connects them (i.e., edges) if they are close enough for communicating , *i.e.*, below a predefined communication range threshold D^{com} from each other (see Figure 6.2(b)). The value of D^{com} is directly related to the ad hoc communication setup. The

communication graph is noted $G^c = (U, E^c)$, with U the set of UAVs. Unlike the environment graph, it is dynamic and not always connected since the position of UAVs changes during the coverage task.

As a notation, $pos : U \to V$ returns the position of a given UAV, *i.e.*, the vertex on which the UAV is currently located. Moreover, the neighbourhood of a UAV u is depicted by $\mathcal{N}^c(u)$ which is the set of every UAV in the communication range of u.

6.4.1.3 Definition of instances

A CCUS3O instance is defined by an environment graph and an initial communication graph, and can be written as $I = (G^e, G^c)$. Since G^c belongs to the instance, the initial position of UAVs is therefore specific which means that two different initial positions result in two different instances. The set of CCUS3O instances is depicted as $\mathbb{I}$. Given two graphs $G^e = (V, E^e)$ and $G^c = (U, E^c)$, then $I = (G^e, G^c) \in \mathbb{I}$ if and only if $\forall u \in U$

$$\mathcal{N}^c(u) = \{u' \in U \backslash \{u\} \mid dist(pos(u), pos(u') \leq D_{com}\}$$

From that definition, an instance class can be defined by an environment graph and the size of the communication graph. Given a connected graph G and an integer $k > 0$, the class of instances $\mathcal{C}(G, k)$ thus represents the instances with k UAVs in an environment graph G. Instances from a same class thus only differ in terms of the initial position of UAVs.

$$\mathcal{C}(G, k) = \{(G, (U, E^c)) \in \mathbb{I} \mid |U| = k\}$$

6.4.1.4 Definition of solutions

A solution for a CCUS3O instance $I \in \mathbb{I}$ is a set of paths in G^e. It can be defined as $S = \{P_u\}_{u \in U}$, with each path P_u starting at the origin vertex of the corresponding UAV u, as represented in Figure 6.2(c). A solution is moreover considered feasible if and only if the paths P_u are cycles, *i.e.*, $P_u[1] = P_u[|P_u|]$, $\forall u \in U$. In the CCUS3O scenario, it means that every UAV has returned to its starting point. As a notation, $\bar{S}$ refers to the vertices of G^e not appearing in S, *i.e.*, not visited by any UAV.

$$\bar{S} = \{v \in V \mid \nexists u \in U, v \in P_u\}$$

6.4.2 Multi-objective aspect

At any moment during the execution of the coverage mission, the current solution can be evaluated according to the three CCUS3O objectives: coverage time, coverage rate and connectivity. Any solution S is thus evaluated by $O(S) \in \mathbb{R}^3$:

$$O(S) = \begin{pmatrix} O^{(rate)}(S) \\ O^{(time)}(S) \\ O^{(conn)}(S) \end{pmatrix}$$

where $O^{(rate)}(S)$, $O^{(time)}(S)$ and $O^{(conn)}(S)$ refer to the objective values of S according to the coverage time, the coverage rate and the connectivity. Let $\mathcal{O} = \{rate, time, conn\}$ denote the set of CCUS3O objectives.

6.4.2.1 Coverage rate

During a coverage mission, UAVs are expected to cover as much area as possible. In CCUS3O it corresponds to maximising the number of vertices visited in the environment graph. For any solution S, its objective value for the coverage rate $O^{(rate)}(S)$ is then defined as the difference between the number of non-visited vertices, *i.e.*, $\left|\bar{S}\right|$, and the number of vertices in the environment graph, *i.e.*, $|V|$.

$$O^{(rate)}(S) = \left|\bar{S}\right| - |V|$$

The result is always a non-positive number to obtain a minimisation objective. For missions where the whole area covered is considered, this objective is then equal to the opposite of the number of vertices in the environment graph.

6.4.2.2 Coverage time

Besides covering as much area as possible, the time for UAVs to come back to their base must be minimised. In the CCUS3O model, UAVs are considered to fly at a constant speed. The fact that the speed is constant states that the time spent and the distance travelled are proportional, regardless of the value of the speed. The coverage time can therefore be calculated from the distance travelled by UAVs. During a coverage mission, the UAVs do not return to their base at the same moment. The total coverage time thus corresponds to the time needed by the UAV which finished last, *i.e.*, with the longest trip. For any solution S (including non-feasible ones), $O^{(time)}(S)$ is defined as:

$$O^{(time)}(S) = \max_{u \in U} l_u$$

where l_u is the length of the path made by UAV u at the current time and the path from the current position to the starting vertex (i.e., base station). It thus depict the length of the cycle made by UAV u if the latter comes back to its initial vertex from the current position.

$$l_u = \underbrace{dist\left(pos(u), P_u[1]\right)}_{\text{distance from the initial vertex}} + \underbrace{\sum_{i=1}^{|P_u|-1} dist\left(P_u[i], P_u[i+1]\right)}_{\text{distance travelled so far}}$$

Since the objective is to cover the area as fast as possible, the coverage time objective should be minimised. Furthermore, minimising the longest cycle (compared to minimising the average for instance) prevents the situation where some UAVs finish their tour much earlier than other ones.

6.4.2.3 Connectivity

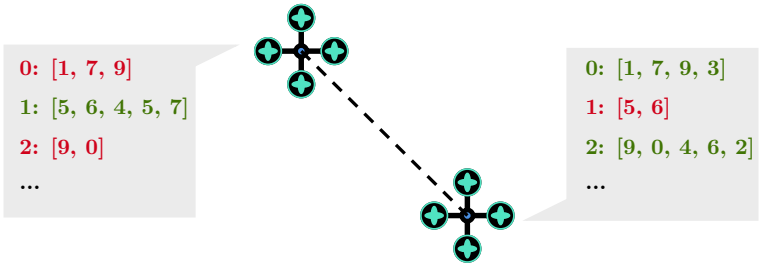
Efficient information sharing is of prime importance for UAV swarms which are highly mobile ad hoc networks relying on distributed decision making. In CCUS3O, it is considered that every UAV u asynchronously shares its

local information with every UAV in its communication neighbourhood depicted by $\mathcal{N}^c(u)$. As a consequence, every UAV in the same connected component of the communication graph has similar local information about the area that has been covered. This connectivity objective thus aims at minimising the average number of connected components in G^c. For that purpose a discretisation of the time $T = \{t_1, t_2, \cdots\} \subset \mathbb{R}$ is considered and for a solution S, $T_S \subset T$ contains every time lower than or equal to the current time of S. The objective value for the connectivity $O^{(conn)}(S)$ is then obtained as:

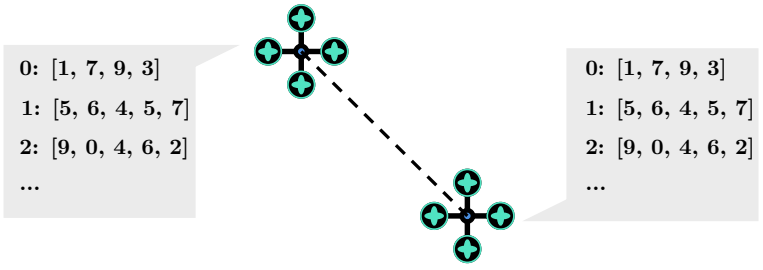
$$O^{(conn)}(S) = \frac{1}{|T_S|} \sum_{t \in T_S} c_t$$

where c_t is the number of connected components in G^c at time t.

Each UAV keeps in memory the representation of the solution (according to its distributed knowledge), *i.e.*, the path made by every UAV so far. When two UAVs communicate, they share that information and keep for each path the most recent one, as shown in Figure 6.3.



(A) Each UAV's known path is compared to the one from the coming information. It is red if the length is smaller than the one from the coming information, and green otherwise.



(B) Each UAV's known path is updated according to the path with the biggest length among both information.

FIGURE 6.3: Every UAV stores the known path of each UAV. When two UAVs can communicate, they compare their known paths (6.3(a)) and update them according to their length (6.3(b)).

6.5 Conclusion

When designing a swarm of robots/UAVs, the difficulty lies in the fact that the wanted behaviour is only emergent from the controlled local interactions. Manually defining local rules to robots or UAVs within the swarm, by aiming a emerging behaviour, can therefore be a challenging and time-consuming task. It explains the rise of techniques with the purpose of automating that process. These techniques can be assimilated to hyper-heuristics where the behaviour to return is a low-level heuristic that a high-level algorithm aims at searching. However, the vast majority of such existing techniques rely on selective approaches, *i.e.*, UAVs are given a set of predefined tasks/rules/actions, and the purpose is thus to choose when to apply an action according to the situation.

In this work, we go beyond the state-of-the-art by proposing a generative approach to automate the design of a swarm of UAVs. For that purpose, we first describe the task of covering an area with a swarm of UAVs as a multi-objective optimisation problem. We then apply our designed algorithm ALGO on that problem to return a distributed heuristic for it.

Chapter 7

Learning to Optimise a Swarm of UAVs

Contents

7.1 Introduction

Defining a priori the behaviour of each individual swarm member to obtain a desired collective behaviour is difficult and time-consuming due to the high uncertainty of a swarm operation. As a consequence, classical multi-robot design techniques are not applicable, as they require the global specifications of the systems to define the behaviour of individual robots Birattari et al. [2019]. Other methods, mainly belonging to the field of evolutionary swarm robotics, Silva et al. [2016] have been proposed to automate the design of more complex systems like robot swarms. Here, the design problem is modelled as an optimisation problem which consists in finding an optimal parameterisation and architecture of the neural network used to control each swarm member. However, as outlined in Francesca and Birattari [2016], these have limitations, for instance, they are typically applied to a single use case (i.e., no generalisation study).

Automating the design of flexible/reusable swarming behaviours would, thus, overcome these challenges. However, as outlined by Birattari et al. [2019], automatic robot swarm design still remains an open research problem.

In this chapter, we propose to automate the design phase of UAV swarms. This is done by using ALGO to generate distributed heuristics for the problem of CCUS3O described in Chapter 6, Section 6.4. More precisely, it seeks to improve the state-of-the-art in automated algorithm design to generate efficient swarming behaviours in the context of area coverage. The problem of covering an area consists of moving UAVs so that any location of the area is visited by at least one UAV at some point in time.

The remainder of this chapter is organised as follows. It first presents in Section 7.3 how to implement CCUS3O as an AFOP so that it can be used in ALGO. The following sections are related to the experiments. In Section 7.4, the experimental setup and process are described, while the results are shown in Section 7.5.

7.2 Modelling as an AFOP

This section presents how we modelled CCUS3O as an AFOP so that ALGO can generate CCUS3O heuristics. The first step is to represent CCUS3O as an AFOP. An instance must therefore be mapped to an AFOP instance, *i.e.*, $I = \left(G_{(N,E)}, A\right)$ where G is a graph and A a set of agents. A solution $S \in \mathcal{S}_I$ must moreover be mappable to a CCUS3O solution.

$$S = \left\{ (a_{i_1}, n_{j_1})_{t_1}, \cdots, (a_{i_k}, n_{j_k})_{t_k}, \cdots \right\} \in \mathcal{S}_I \tag{7.1}$$

where $a_{i_k} \in A$, $n_{j_k} \in N$ and $t_k \in \mathbb{R}_+$, $\forall k \geq 1$.

A CCUS3O instance is composed of the environment and the communication graphs. The environment graph is assimilated to $G_{(N,E)}$ and the vertices of the communication graph, *i.e.*, UAVs, represent A. A CCUS3O solution being a set of paths in the environment graph, one path per UAV, can hence be written as in Equation 7.1. Given a solution $S \in \mathcal{S}_I$, the path of each agent a can be extracted by selecting elements added by a, *i.e.*, by calling $S.\mathit{filter\_agents}(\{a\})$. The elements are then ordered by the time they have been added into S. It gives a sequence of nodes per agent, which is the definition of a CCUS3O solution.

Finally, the domain of solutions $\mathcal{X}_I \subseteq \mathcal{S}_I$, the set of objectives $\mathcal{O}$ and the set of constraints $\mathcal{C}$ must be defined. CCUS3O must then be represented as the following mathematical program.

$$\text{(AFOP)} \begin{cases} \text{Minimise} & \mathtt{f}_o(S) & \forall o \in \mathcal{O} \\ \text{subject to} & \mathtt{f}_c(S) \leq \mathtt{b}_c & \forall c \in \mathcal{C} \\ & S \in \mathcal{X}_I \end{cases} \tag{7.2}$$

In this modelling, we do not want ALGO to produce non-feasible solutions during its exploration phase. The domain of solutions $\mathcal{X}_I$ then represents any feasible solution, *i.e.*, paths of agents are cycles which union covers all nodes from N. With our modelling, if an agent consecutively adds two non-adjacent nodes, the path is filled by the shortest path between those two nodes. An agent cannot however consecutively adds the same node. Such solution then belongs to $\mathcal{X}_I$. Since all solutions belonging to $\mathcal{X}_I$ are feasible, there is no constraint defined here, *i.e.*, $\mathcal{C} = \emptyset$. Finally, the three objectives of CCUS3O are considered, so $\mathcal{O} = \{\mathit{rate}, \mathit{time}, \mathit{conn}\}$.

7.3 Implementation for ALGO

Now that we demonstrated that CCUS3O can be modelled as an AFOP, this section presents the proposed implementation to be included within ALGO's implementation. The process that we follow is described in Chapter 4, Section 4.4.

An overview is provided in Figure 7.1. The class `CCUS3O` inherits from the abstract class `Problem`. The four abstract methods used by the low-level heuristics must then be overridden (presented in Section 7.3.2). An instance of `CCUS3O` must declare a list of objectives and constraints so that the TSP is defined by the mathematical program of an AFOP (Equation 7.2, see more detail in Section 7.3.1). In this implementation, no constraint is given and three objectives are defined, hence the classes `CoverageRate`, `CoverageTime` and `Connectivity` inheriting from `Objective` and overriding the abstract method `compute_value`. Finally, a list of state variables for nodes and edges must be declared to be used as an input of the GNN (described in Section 7.3.3). In the proposed implementation, we declare three state variables for nodes, identified by classes `Visited`, `DistanceBase` and `Neighbourhood` inheriting from the class `NodeVariable` and overriding the abstract method `compute`. We also declare one state variables for edges, *i.e.*, we write the class `DistanceEdge` inheriting from `EdgeVariable` and overriding the abstract method `compute`. In addition, the class `CCUS3O` defines three methods to utilise elements present in the definition of CCUS3O and could not be mapped into the definition of an AFOP. In CCUS30, the position of an UAV is a node of the environment graph. From a certain solution `s`, that position can be retrieve by calling `get_position(s, a)` where `a` is an agent, *i.e.*, an UAV in CCUS3O context (see Algorithm 7.1). The shortest path between two nodes `n1` and `n2` from a graph `g` is computed by calling `get_shortest_path(g, n1, n2)` (see Algorithm 7.2). Finally, given a solution `s`, the connected components within the communication graph is obtained by calling `get_connected_components(s)` (see Algorithm 7.3).

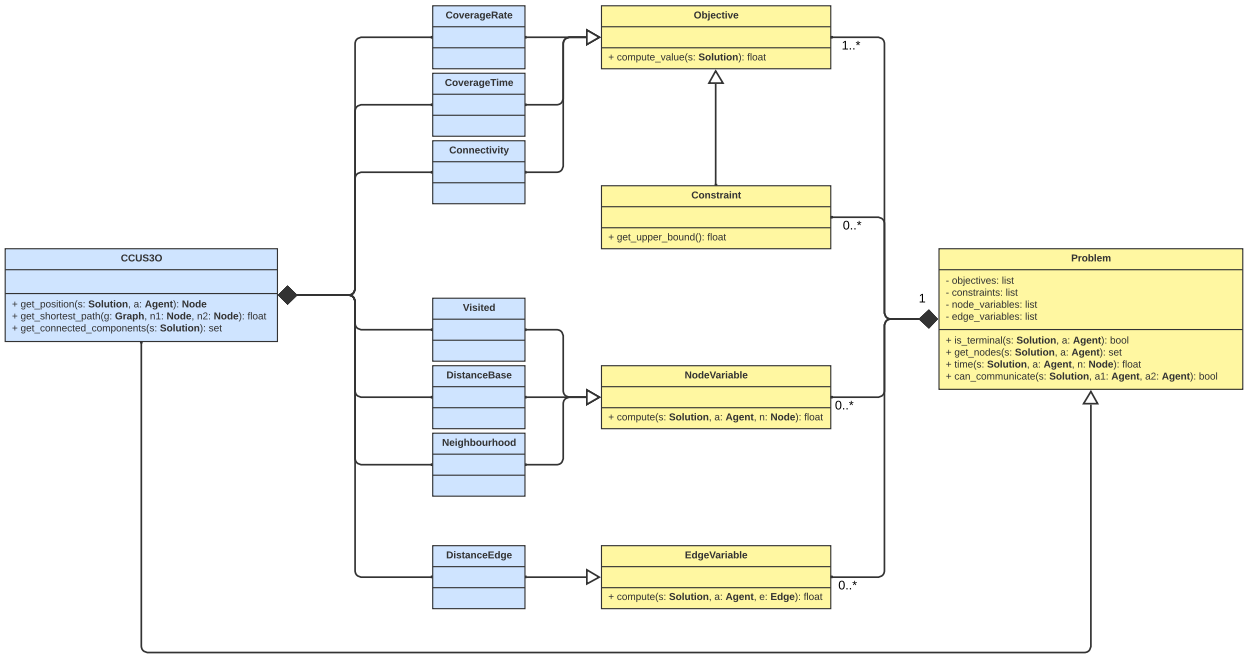


FIGURE 7.1: UML diagram showing the implementation of CCUS3O.

We remind that the classes `Graph` and `Solution` represent respectively an AFOP instance, *i.e.*, $I = \left(G_{(N,E)}, A\right)$, and an AFOP solution, *i.e.*, $S \in \mathcal{S}_I$. The class `Graph` has three attributes: a list of `Node` objects, a list of `Edge` objects and a list of `Agent` objects. The class `Solution` is seen as a set of `Item` objects, where `Item`, corresponding to a tuple $(a, n)_t \in A \times N \times \mathbb{R}_+$, has three attributes of type `Agent`, `Node`, `float`.

The position of an UAV is a node from the environment graph. In the context of an AFOP, it is the last node added into the solution by the corresponding agent. The method `get_position(s, a)` then returns the last node added by agent `a` into solution `s`.

Algorithm 7.1 `CCUS3O::get_position(s, a)`

```
Input: s: Solution, a: Agent
Output: Node
 1: t ← -1
 2: for item ∈ s.filter_agents({a}) do
 3:   if t < item.time then
 4:     n ← item.node
 5:   end if
 6: end for
 7: return n
```

As an initialisation step, the distance between all nodes from $G_{(N,E)}$ is computed with $|N|$ Dijkstra's algorithm. The result is stored in a matrix as an attribute of the instance of `Graph` (we remind that our framework allows the user to add attributes to objects instantiating `Graph`, `Node`, `Edge` and `Agent`). We name that attribute `d`. From that point, to compute the shortest path between two nodes `n1` and `n2` in a graph `g`, the method `get_shortest_path(g, n1, n2)` simply processes an A* search algorithm. The method is used when an UAV choose a destination node which is not adjacent to its current position.

Algorithm 7.2 `CCUS3O::get_shortest_path(g, n1, n2)`

```
Input: g: Graph, n1: Node, n2: Node
Output: list
 1: path ← [ n1 ]
 2: n_curr ← n1
 3: while n_curr ≠ n2 do
 4:   min_length ← ∞
 5:   for n ∈ n_curr.neighbours() do
 6:     if g.d[n1][n] + g.d[n][n2] < min_length then
 7:       n_next ← n
 8:       min_length ← g.d[n1][n] + g.d[n][n2]
 9:     end if
10:   end for
11:   path ← path + [ n_next ]
12:   n_curr ← n_next
13: end while
14: return path
```

Given a solution `s`, representing a solution $S \in \mathcal{S}_I$, the method `get_connected_components(s)` returns a partition of A, in the form of a set of sets of agents. The idea is to represent an edge from the communication graph, linking two agents `a1` and `a2`, with `can_communicate(s, a1, a2) = TRUE`.

7.3.1 Optimisation model

Given a solution `s`, the three objective values are computed by calling `compute_value(s)` from an instance of classes `CoverageRate`, `CoverageTime` and `Connectivity` respectively.

Given a solution `s`, `CoverageRate::compute_value(s)` returns the difference between the number of nodes in the graph and the number of nodes added into the solution (see Algorithm 7.4).

Algorithm 7.3 `CCUS3O::get_connected_components(s)`

```
Input: s: Solution
Output: set
 1: graph ← s.get_graph()
 2: components ← ∅
 3: visited ← ∅
 4: for agent ∈ graph.agents do
 5:   if agent ∉ visited then
 6:     component ← ∅
 7:     waiting ← { agent }
 8:     while waiting ≠ ∅ do
 9:       a1 ← select(waiting)
10:       waiting ← waiting \ { a1 }
11:       component ← component ∪ { a1 }
12:       visited ← visited ∪ { a1 }
13:       for a2 ∈ graph.agents do
14:         if a2 ∉ visited and ccus.can_communicate(s, a1, a2) then
15:           waiting ← waiting ∪ { a2 }
16:         end if
17:       end for
18:     end while
19:     components ← components ∪ { component }
20:   end if
21: end for
22: return components
```

Algorithm 7.4 `CoverageRate::compute_value(s)`

```
Input: s: Solution
Output: float
1: graph ← s.get_graph()
2: return |graph.nodes| - |s.get_nodes()|
```

Given a solution `s`, `CoverageTime::compute_value(s)` computes the length of the tour made by each agent and returns the maximal one (see Algorithm 7.5). The length of such a tour is the time at which an agent has added its last node into the solution added to the distance to its initial node.

Algorithm 7.5 `CoverageTime::compute_value(s)`

```
Input: s: Solution
Output: float
1: graph ← s.get_graph()
2: max_time ← 0.0
3: for a ∈ graph.agents do
4:   t ← max(s.filter_agents({a}).get_times()) + graph.d[ccus.get_position(s, a)][a.first]
5:   if t > max_time then
6:     max_time ← t
7:   end if
8: end for
9: return max_time
```

Given a solution `s`, `Connectivity::compute_value(s)` computes the number of connected components in the communication graph at different times. It then returns the average of them (see Algorithm 7.6). The time step between two measures is a parameter of CCUS3O. That value is hence obtained by calling `get_time_step()` on an object of type `CCUS3O` (line 8).

Algorithm 7.6 `Connectivity::compute_value(s)`

Input: `s: Solution`
Output: `float`

```
1: count ← 0
2: mean ← 0.0
3: t ← 0
4: while t ≤ max(s.get_times()) do
5:   n ← |ccus.get_connected_components(s.filter_times(t))|
6:   mean ← (count * mean + n) / (count + 1)
7:   count ← count + 1
8:   t ← t + ccus.get_time_step()
9: end while
10: return mean
```

7.3.2 Low-level heuristics

The implementation about low-level heuristics corresponds to the overriding of abstract methods from the class `Problem` which are used by the low-level heuristic.

A solution is terminal for an agent if all nodes from the graph have been added into it, and the agent's position is its initial node (see Algorithm 7.7).

Algorithm 7.7 `CCUS3O::terminal(s, a)`

Input: `s: Solution, a: Agent`
Output: `bool`

```
1: graph ← s.get_graph()
2: return |s.get_nodes()| = |graph.nodes| and ccus.get_position(s, a) = a.first
```

As long as some nodes from N are not added into the solution, an agent can choose any node from the graph but its current position. If all nodes have been added into the solution, the agent has no choice but going to its initial position (see Algorithm 7.8).

Algorithm 7.8 `CCUS3O::get_nodes(s, a)`

Input: `s: Solution, a: Agent`
Output: `set`

```
1: graph ← s.get_graph()
2: if |s.get_nodes()| = |graph.nodes| then
3:   candidates ← { a.first }
4: else
5:   candidates ← ∅
6:   for node ∈ graph.nodes do
7:     if node ≠ ccus.get_position(s, a) then
8:       candidates ← candidates ∪ { node }
9:     end if
10:  end for
11: end if
12: return candidates
```

The time for an agent to add a node into the solution, is the time for an UAV to reach the node in the environment graph. It is hence equivalent to the distance between the agent's position and the node to add (see Algorithm 7.9).

Algorithm 7.9 `CCUS3O::time(s, a, n)`

Input: `s: Solution, a: Agent, n: Node`
Output: `float`

```
1: graph ← s.get_graph()
2: return graph.d[ccus.get_position(s, a)][n]
```

Two agents can communicate if the distance between their position is lower than a certain communication range (see Algorithm 7.10). The value of this communication range is a parameter of CCUS3O. Its value is thus obtained by calling `get_com_range()` on an object of type `CCUS3O` (line 2).

Algorithm 7.10 `CCUS3O::can_communicate(s, a1, a2)`

Input: `s: Solution, a1: Agent, a2: Agent`
Output: `bool`

```
1: graph ← s.get_graph()
2: return graph.d[ccus.get_position(s, a1)][ccus.get_position(s, a2)] ≤ ccus.get_com_range()
```

7.3.3 State variables

The implementation of state variables is used as an input to the GNN. We should then provide any information that we want the GNN to consider in order to return a relevant embedded solution. Three state variables are defined for nodes, one for each CCUS3O objective.

The coverage rate objective is depicted by the class `Visited`, where `compute(s, a, n)` returns `1.0` if `n` has been added into `s`, regardless of agent `a` (see Algorithm 7.11).

Algorithm 7.11 `Visited::compute(s, a, n)`

Input: `s: Solution, a: Agent, n: Node`
Output: `float`

```
1: if n ∈ s.get_nodes() then
2:     return 1.0
3: else
4:     return 0.0
5: end if
```

The class `DistanceBase` represents the coverage time objective. the method `compute(s, a, n)` simply returns the distance between the position of `a` in `s` and node `s` (see Algorithm 7.12).

Algorithm 7.12 `DistanceBase::compute(s, a, n)`

Input: `s: Solution, a: Agent, n: Node`
Output: `float`

```
1: graph ← s.get_graph()
2: return graph.d[n][a.first]
```

Finally, the class `Neighbourhood` provides an information useful for the connectivity objective with its method `compute(s, a, n)`. The latter returns the number of agents (exluding `a`) within the communication range of node `n` (see Algorithm 7.13).

Only one state variable is defined for edges. In the class `DistanceEdge`, the method `compute(s, a, e)` returns the weight of `e`, regardless of `s` and `a` (see Algorithm 7.14).

Algorithm 7.13 `Neighbourhood::compute(s, a, n)`

Input: `s: Solution, a: Agent, n: Node`
Output: `float`

```
graph ← s.get_graph()
count ← 0
for agent ∈ graph.agents do
  if agent ≠ a and graph.d[ccus.get_position(s, agent)][n] ≤ ccus.get_com_range() then
    count ← count + 1
  end if
end for
return count
```

Algorithm 7.14 `DistanceEdge::compute(s, a, e)`

Input: `s: Solution, a: Agent, e: Edge`
Output: `float`

```
return e.w
```

7.4 Experimental Setup

This section introduces the setup used to conduct the experiments. In the following, the metrics used to assess the performance of the generated heuristic with ALGO are presented. Then, the manually designed heuristic used as basis of comparison is introduced. Finally details on the whole experimental process are provided.

7.4.1 Performance metrics

Two types of metrics are presented in this section: the metrics used to evaluate the performance of a swarm according to the coverage and the connectivity; the metrics used for comparing two sets of non-dominated solutions in multi-objective optimisation. As depicted in Figure 7.2, a heuristic H_1 provides different solutions $S_1, S_2, \cdots, S_k$. The swarm metrics evaluate these solutions according to the coverage and the connectivity. For a solution S_i, a couple of values (cov_i, con_i) is therefore returned and can be displayed in a two-dimensional coordinate system. The obtained set of 2D points thus represents the heuristic H_1 whose performance can finally be evaluated using the three MO metrics.

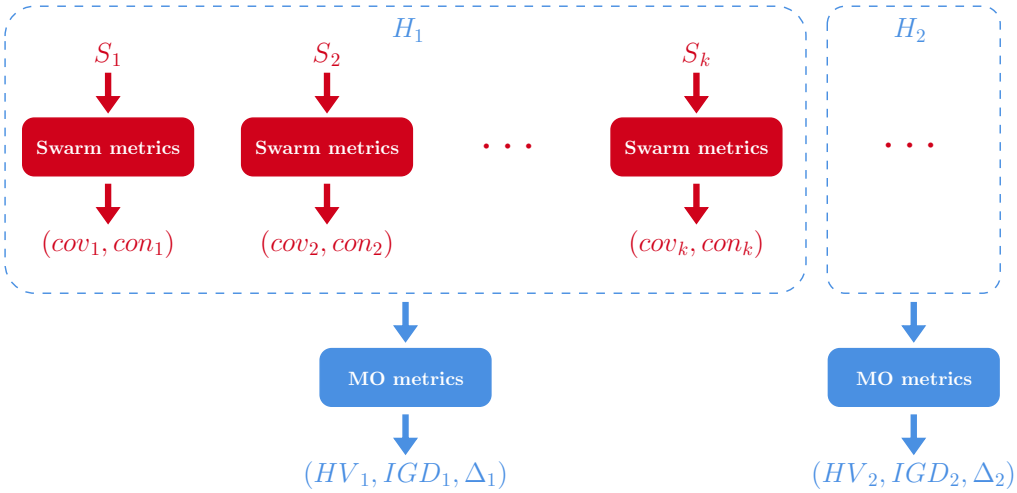


FIGURE 7.2: Usage of both swarm metrics and MO metrics to assess the performance of a heuristic.

7.4.1.1 Swarm metrics

Two state-of-the-art metrics Brust et al. [2017] are defined here to assess the performance of the swarm. The coverage speed evaluates the quality of the coverage while the number of connected components assesses the connectivity.

Coverage speed Let $n_v^{(t)}$ be the number of times the vertex $v \in V$ has been visited at the time step $t \in \{0, \cdots, T\}$. Then, $\texttt{speed}(r)$ is the time needed by the swarm to cover a rate $r \in [0, 1]$ of the environment graph.

$$\texttt{speed}(r) = \underset{0 \leq t \leq T}{\arg\min} \left\{ \frac{\left| \left\{ v \in V \mid n_v^{(t)} > 0 \right\} \right|}{|V|} \geq r \right\}$$

Number of connected components This metric is similar to the definition of the connectivity objective of CCUS. The value of `number` is then the average number of connected components in the communication graph.

$$\texttt{number} = O^{(conn)}(S_T)$$

7.4.1.2 Multi-objective metrics

For evaluating the quality of CCUS solutions, three metrics considering the multi-objective aspect have been used: the Hyper-Volume (HV), the Inverted Generational Distance (IGD) and the spread (Δ). For a set of solutions, let $P = \{p_i\}_i$ be the subset of non-dominated ones, and $P^* = \{p_i^*\}_i$ be the optimal Pareto front. These two sets are ordered according to one of their objective value. The Euclidean distance in the objective space between two points from both fronts is given by $d : \{1, \cdots, |P^*|\} \times \{1, \cdots, |P|\} \to \mathbb{R}_+$.

$$d(i, j) = \|p_i^* - p_j\|_2$$

Hyper-Volume (HV) HV represents the volume in the objective space covered by the points in P, relative to a reference point w.

$$HV = \texttt{volume}\left(\bigcup_{i=1}^{|P|} v_i \right)$$

with v_i the hypercube in the objective space made by p_i and w.

Inverted Generational Distance (IGD) IGD measures the proximity of the front P with the optimal front P^*.

$$IGD = \frac{1}{|P^*|} \sqrt{\sum_{i=1}^{|P^*|} \left(\min_{1 \leq j \leq |P|} d(i, j) \right)^2}$$

Spread (Δ) Δ indicates how well the non-dominated solution are spread on the front P. It also take into consideration the width of P compared to the optimal front P^*.

$$\Delta = \frac{d_f + d_l + \sum_{i=1}^{|P|-1} \left| d_i - \bar{d} \right|}{d_f + d_l + (|P| - 1)\,\bar{d}}$$

with $d_f = d(1,1)$ and $d_l = d\left(|P^*|, |P|\right)$ the distances in the objective space between the first edges of P and P^* and their last edge. Moreover, $d_i = \|p_i - p_{i+1}\|_2$ is the distance between two adjacent points in P, with $\bar{d} = \frac{1}{|P|} \sum_{i=1}^{|P|-1} d_i$ the average distance.

7.4.2 Comparison heuristics

In order to evaluate the performance of the heuristic generated by ALGO, it has been executed on different instances along with other heuristics which are presented in this section. The first heuristic has been specifically designed manually for tackling CCUS3O, the second one has been automatically designed by a hyper-heuristic specifically designed for CCUS3O, while the other two are state-of-the-art pheromone-based heuristics aiming to cover an area. One of the two latter not considering the connectivity of the swarm, it is used as a bound for the coverage objective.

7.4.2.1 Manually-designed heuristic

The *Weighted Objective* (WO) heuristic has been designed to tackle CCUS3O instances. It belongs to the space of low-level heuristics (defined in Algorithm 4.1), where the scoring function f_u depends on three values (one per CCUS3O objective).

$$f_u(v) = \sum_{o \in \mathcal{O}} a_{uv}^{(o)}$$

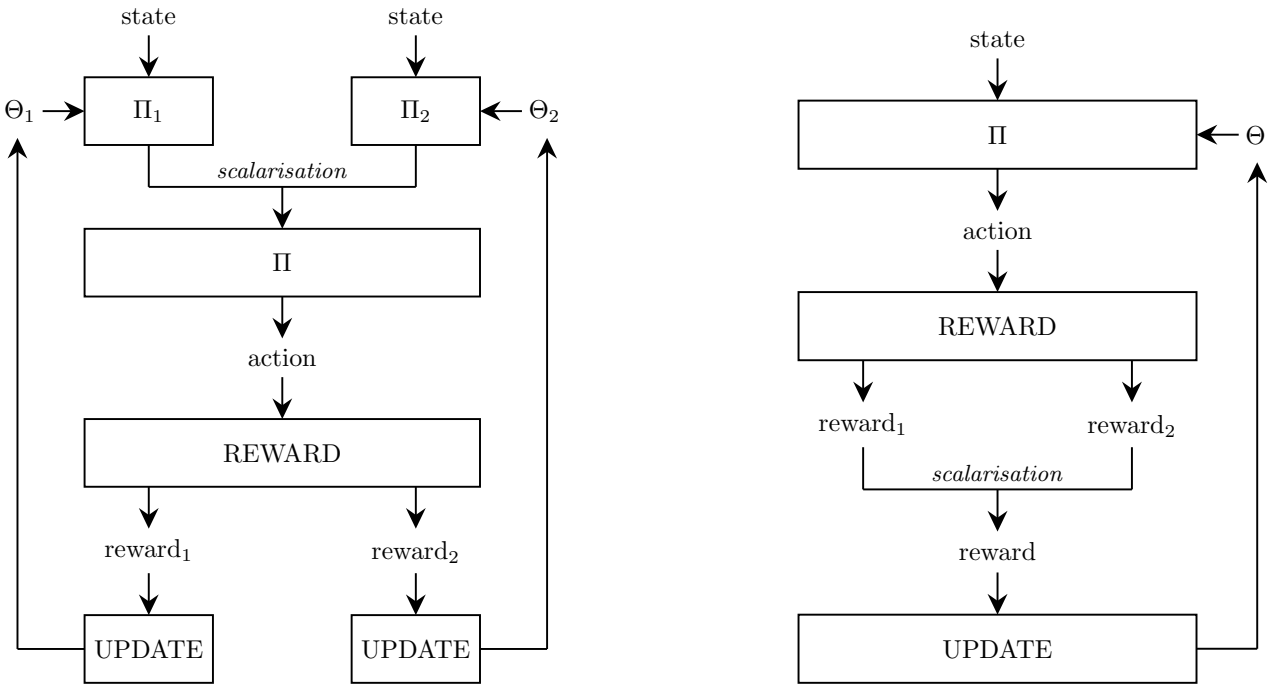
with

$$\begin{aligned}
a_{uv}^{(time)} &= W - dist(pos(u), v) - dist\left(v, P_u[0]\right) \\
a_{uv}^{(rate)} &= W \cdot x_{uv}^{(rate)} \\
a_{uv}^{(conn)} &= \min\left(W, W + D^{com} - \min_{u' \in U \setminus \{u\}} dist\left(v, pos\left(u'\right)\right)\right)
\end{aligned}$$

The value of W is set arbitrarily. For the coverage time objective, the weight linearly decreases according to the distance of the path from the current position to the initial position passing by the vertex to evaluate. The weight for the coverage rate objective is equivalent to the state variable of the vertex to evaluate. For the connectivity objective, the weight depends on the distance to the closest UAV from the vertex to evaluate. If the latter distance is lower than D^{com}, *i.e.*, is in the communication range of a UAV, then the weight is maximum; otherwise, it linearly decreases according to that distance. Even though the quality of *WO* is not guaranteed, it makes it possible to compare the performance of the heuristic generated by ALGO with a behaviour more relevant than a random process.

7.4.2.2 Automatically-designed heuristic

We compared the heuristic generated by ALGO with a previous Q-Learning-based Hyper-Heuristic (QLHH) that we designed specifically to tackle CCUS [Duflo et al., 2021]. Both hyper-heuristics have a similar principle but they differ in their use of scalarisation (see Figure 7.3).



(A) Process of QLHH. (B) Process of ALGO.

FIGURE 7.3: Difference between the workflow of QLHH and ALGO. For QLHH (a), the policy is scalarised from multiple policies. The multiple reward obtained by a choice of action are all used to update their corresponding policy. For ALGO (b), the scalarisation occurs on the reward. The resulting single reward is then used to update the single policy.

QLHH provides multiple policies that are scalarised into a single one, while ALGO uses a single policy and scalarises the multiple rewards into a single one. The multiple policies of QLHH have their own learning process from their own reward. This independence between objectives has shown difficulty in balancing them in the generated heuristic, which is an issue in itself. If it turns out that connectivity has more emphasis, UAVs are flying together rather than visiting new vertices, which can make episodes very long in the context of CCUS(3O) since the terminal condition for UAVs to return to their base is that every vertex is visited. To overcome this issue, the idea with QLHH is to update the weights during the scalarisation according to objectives "needing support" but it obliges the user to define higher and lower bounds for each objective, which can be time consuming. The single policy of ALGO is intended to break this independence and improve the balance of objectives in the generated heuristics.

7.4.2.3 Pheromone-based heuristics

Different techniques based on pheromone have been used in the context of area coverage by a swarm of robots/UAVs. Two have been implemented here in order to assess the quality of the coverage provided by the generated heuristic. For both heuristics, the behaviour is very similar to the one described in Algorithm 4.1. They however do not belong to the space of low-level heuristic since the choice of the next position v for a UAV at a position v_{curr} is stochastic and thus depends on a probability $\mathbb{P}(v \mid v_{curr})$.

Heuristic Φ The first heuristic, referred to as Φ, is based on repulsive pheromones Kuiper and Nadjm-Tehrani [2006], *i.e.*, UAVs drop pheromones on their way to indicate visited places. The probability of a UAV to go to a position is then inversely proportional to the amount of pheromone at that position (i.e., the more pheromones, the less chance). The environment on which UAVs are evolving is a grid, and at each iteration, UAVs assign the current coverage time to their current position. So each position is assigned a value which corresponds to the last time a UAV has visited it. As with repulsive pheromone, the lower the value assigned at a position, the higher the chance have UAVs to choose that position. Given a position v, let t_v be the time assigned to that position and $\mathcal{N}(v)$ be the set of adjacent positions. The probability to go to a position v for a UAV at position v_{curr} is

$$\mathbb{P}(v \mid v_{curr}) = \begin{cases} \dfrac{T - t_v}{(|\mathcal{N}(v_{curr})| - 1) \cdot T} & \text{if } v \in \mathcal{N}(v_{curr}) \\ 0 & \text{otherwise} \end{cases}$$

with $T = \sum_{v \in \mathcal{N}(v_{curr})} t_v$.

Heuristic Φ-K The drawback of heuristic Φ is that it does not consider the connectivity of the swarm. That is the purpose of the work of Danoy et al. [2015] which adds a dynamic clustering algorithm, so called KHOPCA Brust et al. [2008], in order to maintain the connectivity inside the swarm of UAVs. Let this heuristic be Φ-K. KHOPCA algorithm dynamically assigns a state value $s(u)$ to each UAV u which represents the distance to the cluster head ($s(u) = 0$ if u is the cluster head). For a UAV u, let $\mathcal{N}(u)$ be the set of UAVs in a communication range of u. Then, $s(u)$ is iteratively updated by following four simple rules:

$$s(u) = \begin{cases} \min\limits_{u' \in \mathcal{N}(u)} s(u') + 1 & \text{if } \min\limits_{u' \in \mathcal{N}(u)} s(u') < s(u) \\ 0 & \text{if } \min\limits_{u' \in \mathcal{N}(u)} s(u') = k \\ s(u) + 1 & \text{if } s(u) \neq 0 \wedge s(u) < \min\limits_{u' \in \mathcal{N}(u)} s(u') \\ s(u) + 1 & \text{if } s(u) = 0 \wedge \min\limits_{u' \in \mathcal{N}(u)} s(u') = 0 \end{cases}$$

where $k + 1$ is the number of UAVs (k is the biggest possible distance from the cluster head). At initialisation, $s(u) = k$ for every UAV u. At each iteration, UAVs which are not cluster heads follow their cluster with a certain probability PK, otherwise they choose their next position according to the probability defined in heuristic Φ. When a UAV follows its cluster, it decides to go to the position of its neighbour (a UAV in its communication range) with the lowest state value. Since the value of PK has a high impact on the quality of the coverage, let $\Phi\text{-K}_{\text{PK}}$ denote this heuristic with a certain value for PK.

7.4.3 Experimental process

For the experiments, all of the instances consider a grid graph as an environment graph. An instance class is then defined by its grid dimension and the number of UAVs in the swarm (see Section 6.4.1.3). The set of parameters is presented in Table 7.1.

The maximal distance of communication is the distance in which two UAVs can communicate (see Section 6.4.1.2). The exploration rate is the chance of choosing a random action (defined as a percentage) instead of following the policy (see Section 4.3.1). The discount factor is the importance given to the final reward

(see Section 4.3.3). The size of movement frame is the number of movements considered for one reward (see Section 4.3.2).

The next three parameters cannot be chosen empirically while they strongly affect the learning process. In order to have a relevant value for them, a factorial experiment is processed (detailed in Section 7.5.1). The learning rate defines how much of the gradient is used for a step of SGD (see Section 4.3.3). The embedding dimension is the length of the vectors representing both states and actions (see Section 4.3.1). The mini-batch size is the number of items selected from the memory to process the SGD (see Section 4.3.3).

When the parameterisation is defined, a stability experiment is processed where the purpose is to train ALGO on instances from a certain class, and to execute the generated heuristic on other classes (detailed in Section 7.5.3). The training then occurs on instances with 10 UAVs moving on a 20×20 grid. 50 instances are randomly chosen from the latter instance class and the ALGO algorithm is executed 10 times on each. The generated heuristic is finally executed on instances from other classes along with heuristics presented in Section 7.4.2 in order to assess its performance.

Parameter name	**Notation**	**Value**
maximal distance of communication	D^{com}	4
exploration rate	ϵ	0.05
discount factor	γ	0.9
size of movement frame	τ	10
Factorial experiment		
learning rate	α	
embedding dimension	p	
mini-batch size	$\lvert\mathcal{B}\rvert$	
Stability experiment		
number of epochs		10
number of instances		50
grid width		20
grid height		20
number of UAVs		10

TABLE 7.1: Experimental parameters used for training ALGO.

7.5 Experimental Results

This section presents the experimental results of ALGO on the CCUS3O problem which have been conducted on the High Performance Computing (HPC) platform of the University of Luxembourg Varrette et al. [2022]. The factorial experiment is first done to determine the best learning rate, embedding dimension and size of the mini-batch. Then, ALGO performance is evaluated against QLHH. Finally, the experimental study on the stability of ALGO with regard to state-of-the-art heuristics is depicted.

7.5.1 Factorial experiment

In order to assess the best parameterisation of ALGO, an analysis of the sensitivity to its three parameters has been conducted. To this end, four values have been considered for each parameter. The size of the mini-batch can be 16, 32, 48 or 64; the learning rate can be 0.01, 0.05, 0.1 or 0.2; and the embedding dimension can be 8, 16, 24 or 32. The objective is to learn a heuristic for each possible combination. The training is identical to the stability experiment, *i.e.*, on 50 instances from the class `(20x20/10)` (with a 20x20 grid graph as an

environment and 10 UAVs) with 10 epochs. After the training, the generated heuristic for each parameterisation is executed on 30 instances from the same class. These instances are the same for every heuristic generated. A distribution of 30 solutions is thus obtained and the non-dominated solutions are retained so that the Pareto fronts are compared with the three multi-objective metrics presented in Section 7.4.1.2. The results are depicted in Table 7.2.

Parameters			Multi-Objective Metrics		
$\|\mathcal{B}\|$	α	p	HV	IGD	Δ
16	0.01	8	3.563e+00	6.942e-01	8.617e-01
16	0.01	16	-	-	-
16	0.01	24	6.645e+00	5.965e-01	9.522e-01
16	0.01	32	7.532e+00	5.956e-01	8.200e-01
16	0.05	8	2.991e+00	6.576e-01	8.408e-01
16	0.05	16	-	-	-
16	0.05	24	2.969e+00	8.368e-01	9.098e-01
16	0.05	32	3.298e+00	7.574e-01	8.167e-01
16	0.1	8	1.602e+00	9.394e-01	9.086e-01
16	0.1	16	3.311e+00	7.473e-01	7.297e-01
16	0.1	24	2.809e+00	7.216e-01	8.673e-01
16	0.1	32	3.656e+00	4.539e-01	8.551e-01
16	0.2	8	3.104e+00	7.455e-01	8.504e-01
16	0.2	16	2.076e+00	1.251e+00	7.994e-01
16	0.2	24	1.086e-01	1.626e+00	1.000e+00
16	0.2	32	7.462e-02	1.490e+00	7.501e-01
32	0.01	8	7.883e+00	5.409e-01	7.600e-01
32	0.01	16	3.361e+00	5.832e-01	8.480e-01
32	0.01	24	2.367e+00	8.676e-01	9.251e-01
32	0.01	32	7.830e+00	5.136e-01	9.184e-01
32	0.05	8	6.486e+00	6.043e-01	6.703e-01
32	0.05	16	2.977e+00	8.042e-01	6.723e-01
32	0.05	24	4.586e+00	6.299e-01	7.706e-01
32	0.05	32	2.358e+00	9.694e-01	8.213e-01
32	0.1	8	3.419e+00	7.671e-01	7.991e-01
32	0.1	16	2.775e+00	6.690e-01	9.633e-01
32	0.1	24	9.577e+00	4.979e-01	8.683e-01
32	0.1	32	5.263e+00	4.419e-01	7.582e-01
32	0.2	8	2.270e-01	1.294e+00	1.000e+00
32	0.2	16	6.113e+00	4.186e-01	9.379e-01
32	0.2	24	5.449e+00	1.268e+00	9.234e-01
32	0.2	32	7.062e+00	5.676e-01	9.237e-01

Parameters			Multi-Objective Metrics		
$\|\mathcal{B}\|$	α	p	HV	IGD	Δ
48	0.01	8	4.419e+00	7.434e-01	8.857e-01
48	0.01	16	2.702e+00	7.619e-01	8.098e-01
48	0.01	24	4.650e+00	5.815e-01	8.373e-01
48	0.01	32	2.289e+00	8.671e-01	9.658e-01
48	0.05	8	8.769e+00	4.804e-01	8.861e-01
48	0.05	16	6.567e+00	7.979e-01	6.383e-01
48	0.05	24	2.957e+00	5.162e-01	8.830e-01
48	0.05	32	4.494e+00	5.918e-01	6.276e-01
48	0.1	8	7.024e+00	1.597e+00	9.207e-01
48	0.1	16	6.233e+00	4.777e-01	8.452e-01
48	0.1	24	9.667e-01	9.502e-01	8.728e-01
48	0.1	32	3.356e+00	8.064e-01	8.829e-01
48	0.2	8	2.997e+00	8.235e-01	6.844e-01
48	0.2	16	-	-	-
48	0.2	24	1.871e+00	6.959e-01	8.636e-01
48	0.2	32	9.217e+00	5.140e-01	7.940e-01
64	0.01	8	2.151e+00	6.347e-01	6.748e-01
64	0.01	16	8.979e-01	1.046e+00	9.482e-01
64	0.01	24	7.462e+00	5.675e-01	8.355e-01
64	0.01	32	2.342e+00	8.683e-01	8.608e-01
64	0.05	8	9.521e+00	4.592e-01	7.856e-01
64	0.05	16	6.634e+00	5.055e-01	9.591e-01
64	0.05	24	5.551e+00	5.164e-01	8.548e-01
64	0.05	32	2.186e+00	9.269e-01	9.302e-01
64	0.1	8	1.866e+00	1.917e+00	9.719e-01
64	0.1	16	1.215e+01	4.513e-01	8.143e-01
64	0.1	24	3.701e+00	8.365e-01	9.691e-01
64	0.1	32	2.638e+00	9.070e-01	7.825e-01
64	0.2	8	3.982e+00	4.634e-01	9.389e-01
64	0.2	16	8.946e+00	4.437e-01	6.733e-01
64	0.2	24	4.195e+00	6.778e-01	8.196e-01
64	0.2	32	9.566e+00	4.821e-01	9.516e-01

TABLE 7.2: Results of the factorial experiment.

For each column of Table 7.2, *i.e.*, for each metric, values are normalised and a color gradient is used to enhance the quality of a parameterisation according to metrics (the more red the better). By looking at the table, it can be noted that the model is very sensitive to the parameterisation. This is shown by the high disparity of values. Among parameters, the size of the batch $|\mathcal{B}|$ seems to have an impact independently from other parameters. Among the range of values, the bigger is $|\mathcal{B}|$, the better are results in general. The size of the batch has an impact for the IGD. Selecting less elements for the gradient descent indeed enhances the exploration. It is not obvious to analyse the impact of the other two parameters independently according to Table 7.2. The embedding dimension p has a direct impact on the size of the searching space (the higher p, the wider searching space). For a certain learning rate α, the model would struggle to converge with a lower searching space, *i.e.*, a lower p. It is thus relevant to admit that the lower is p, the lower should be α. Among the best five parameterisations, the convergence is observed and at the end, the selected paramterisation is the following: $|\mathcal{B}| = 48$; $\alpha = 0.2$; $p = 32$.

7.5.2 Comparison with QLHH

This section aims at comparing the current hyper-heuristic ALGO to the one it extends, *i.e.*, QLHH. In order to compare both models, QLHH and ALGO have been trained on instances from class `(15x15/10)`. QLHH uses the parameterisation provided used by Duflo et al. [2021] while ALGO uses the parameterisation obtained after

the factorial experiment described above. The generated heuristic has then been executed on several instances, ten times, from the same class. The solutions are represented according to both swarm metrics for two instances in Figure 7.4.

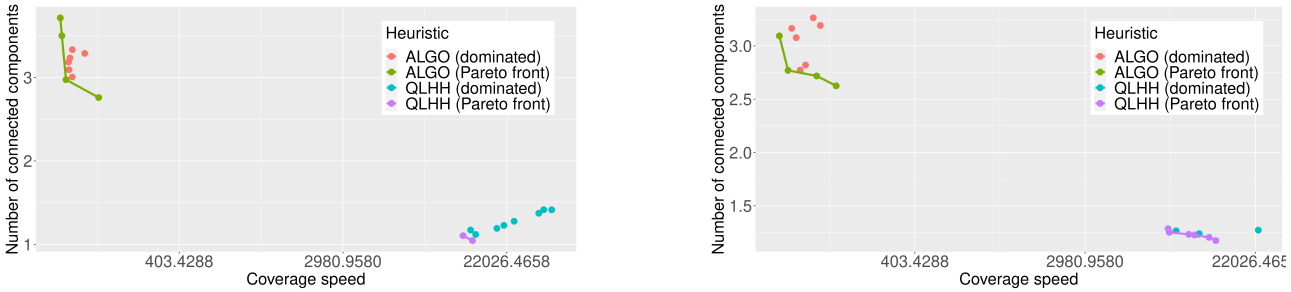


FIGURE 7.4: Solutions obtained with heuristics generated by QLHH and ALGO on two instances (the x axis uses a logarithm scale due to the huge gap between heuristics).

Results shown in Figure 7.4 are representative of the behaviour on other instances. It shows the Pareto fronts obtained by executing both generated heuristics ten times on two instances from the class `(15x15/10)`. QLHH generated a heuristic which provides solutions with a low number of connected components, around 1.5. This is due to the fact that the scalarisation weights enhance the connectivity objective without making UAVs cover new vertices, until a point where the weight for the coverage rate becomes too large. This explains why the coverage speed is extremely high for solutions obtained by the heuristic generated by QLHH. The heuristic generated by ALGO provides a worse connectivity since the number of connected components is around 3, but also a much better coverage speed. For any instance, UAVs need between 100 and 150 units of time to cover 95% of the environment graph, while they need from 10 000 to 30 000 with the heuristic generated by QLHH.

By looking at Figure 7.4, ALGO may seem to focus on the coverage objective only, but this is not the case. This idea is induced by the good connectivity obtained by the heuristic generated by QLHH in comparison, but the solutions produced by the latter heuristic are practically unusable. Consequently, heuristics generated by QLHH will not be used as a comparison for the stability experiment detailed below.

7.5.3 Stability

ALGO has been trained on instances from the `(20x20/10)` instance class. This class has been chosen since the number of UAVs is high enough to have a wide range of values for the connectivity objective. Moreover, the ratio of the number of vertices over the number of UAVs is good so that UAVs are not constrained to fly together by the lack of free room. The generated heuristic has then been executed on other instance classes, *i.e.*, `(15x15/10)`, `(20x20/5)`, `(20x20/10)`, `(20x20/15)` and `(25x25/10)`. The objective is not only to demonstrate its good performance on other instance classes with different numbers of UAVs and environment grid sizes, but also to validate its good stability against heuristics from the state-of-the-art.

7.5.3.1 Training

This section presents the training process along with the evolution of the Pareto front which depicts the convergence of the model, and therefore justifies the training time. ALGO has been trained on 50 instances over 10 epochs, resulting in 500 episodes. The evolution of the three MO metrics during the training is illustrated in Figure 7.5. In terms of IGD, the algorithm features a very fast convergence in the first episodes and then keeps improving at a slower pace. When considering HV, it appears that ALGO improves solutions steadily

with a major increase around the 300th episode. With regard to the third metric, Δ, it converges until the 50th episode and then gets worse. This might be explained by the fact that adding any new non-dominated to the front may completely disrupt the diversity.

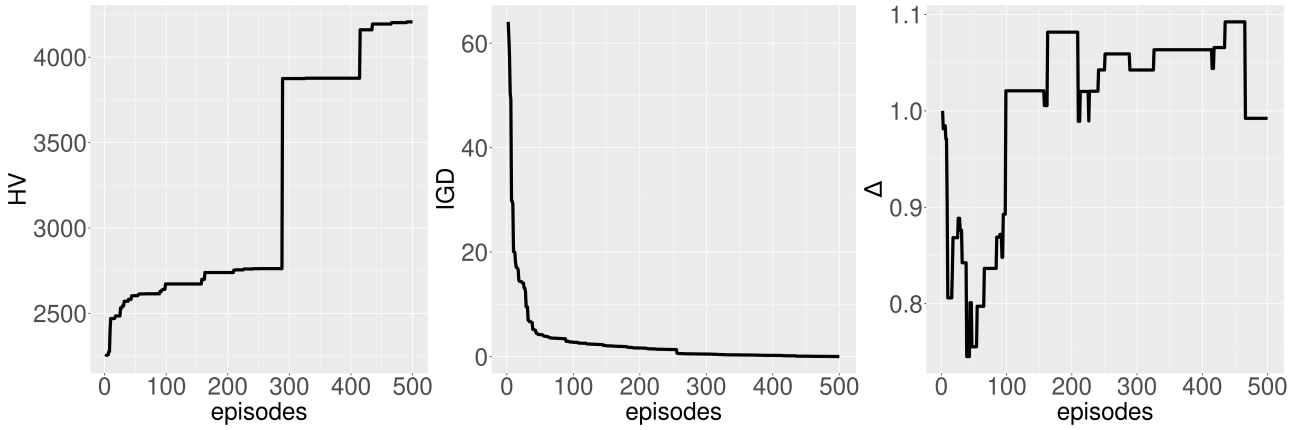


FIGURE 7.5: Evolution of the front during the training according to HV, IGD and Δ.

7.5.3.2 Testing

For both WO and the generated heuristics, each UAV has a deterministic behaviour so they are not stochastic as are heuristics Φ and Φ-K. However, since UAVs move asynchronously, two executions on a same instance may provide two different solutions which makes these two heuristics non-deterministic. All heuristics have thus been executed 30 times per instance, resulting in 30 solutions per instance. These distributions are compared according to their Pareto front using the three MO metrics defined in Section 7.4.1. A Wilcoxon signed-rank test is then used to assess the statistical significance of the results. The results of the latter test are displayed in three tables, one per MO metric (HV in Table 7.3, IGD in Table 7.4 and Δ in Table 7.5). For each table, a cell in dark blue means that the heuristic generated by ALGO outperforms the specific heuristic (column name) for the specific instance class (row name) with a 95% confidence. A light-blue cell means that the heuristic generated by ALGO provides better results in average but without statistical confidence.

Instance class		**Heuristic**										
grid	#UAVs	**ALGO**	**WO**	*p*-value	**Φ**	*p*-value	**Φ-K$_{0.05}$**	*p*-value	**Φ-K$_{0.10}$**	*p*-value	**Φ-K$_{0.20}$**	*p*-value
15x15	10	2.03e+01 ±3.97e+00	1.92e+01 ±3.54e+00	4.37e-04	5.42e+00 ±1.27e+00	7.11e-15	7.94e+00 ±1.81e+00	7.11e-15	1.01e+01 ±2.03e+00	7.11e-15	1.40e+01 ±2.93e+00	7.11e-15
20x20	05	1.35e+01 ±3.01e+00	1.48e+01 ±2.92e+00	1.20e-02	2.52e+00 ±6.82e-01	7.63e-06	3.56e+00 ±8.02e-01	7.63e-06	4.53e+00 ±1.03e+00	7.63e-06	6.83e+00 ±1.59e+00	7.63e-06
20x20	10	1.60e+01 ±4.61e+00	1.75e+01 ±4.21e+00	4.57e-02	3.27e+00 ±9.60e-01	1.19e-07	4.81e+00 ±1.27e+00	1.19e-07	6.39e+00 ±1.78e+00	1.19e-07	9.40e+00 ±2.52e+00	1.19e-07
20x20	15	1.67e+01 ±3.69e+00	1.63e+01 ±3.59e+00	1.40e-01	3.24e+00 ±8.12e-01	1.86e-09	5.22e+00 ±1.01e+00	1.86e-09	7.14e+00 ±1.50e+00	1.86e-09	1.13e+01 ±2.50e+00	1.86e-09
25x25	10	9.88e+00 ±2.62e+00	1.40e+01 ±2.14e+00	1.53e-05	1.62e+00 ±4.86e-01	7.63e-06	2.72e+00 ±5.71e-01	7.63e-06	3.74e+00 ±7.10e-01	7.63e-06	5.65e+00 ±1.22e+00	7.63e-06

TABLE 7.3: Comparison between heuristics according to HV.

According to the HV metric, the heuristic generated by ALGO outperforms all of pheromone-based heuristic. However, for most of instance classes, WO heuristic provides better results with 95% confidence. This is an unexpected result when looking at its results in terms of IGD and Δ metrics. It indeed clearly appears that the generated heuristic outperforms WO for every instance class according to IGD and Δ. While these metrics respectively measure the convergence and the diversity of the front, the results in terms of HV values are not similarly competitive with respect to WO. This behaviour can be visualised in Figure 7.6 which displays the non-dominated solutions for two different instances. It can be observed that despite a clear lack of diversity in

the front of WO, its lowermost point is very distant from the other solutions (including dominated ones) which in turn drastically increases the value of HV.

Instance class		**Heuristic**										
grid	#UAVs	**ALGO**	**WO**	*p*-value	**Φ**	*p*-value	**Φ-K$_{0.05}$**	*p*-value	**Φ-K$_{0.10}$**	*p*-value	**Φ-K$_{0.20}$**	*p*-value
15x15	10	3.74e-01 ±1.46e-01	8.99e-01 ±2.92e-01	3.54e-11	2.40e+00 ±1.62e-01	7.11e-15	1.99e+00 ±1.98e-01	7.11e-15	1.64e+00 ±1.70e-01	7.11e-15	1.11e+00 ±1.50e-01	7.11e-15
20x20	05	6.08e-01 ±3.23e-01	1.25e+00 ±4.08e-01	1.58e-03	2.57e+00 ±2.21e-01	7.63e-06	2.35e+00 ±2.09e-01	7.63e-06	2.16e+00 ±1.85e-01	7.63e-06	1.69e+00 ±1.72e-01	7.63e-06
20x20	10	6.68e-01 ±3.27e-01	1.15e+00 ±5.13e-01	1.26e-02	2.58e+00 ±2.55e-01	1.19e-07	2.26e+00 ±2.51e-01	1.19e-07	1.97e+00 ±2.49e-01	1.19e-07	1.50e+00 ±2.52e-01	1.19e-07
20x20	15	5.09e-01 ±2.00e-01	1.32e+00 ±3.25e-01	9.31e-09	2.51e+00 ±2.12e-01	1.86e-09	2.14e+00 ±1.58e-01	1.86e-09	1.78e+00 ±1.55e-01	1.86e-09	1.14e+00 ±1.57e-01	1.86e-09
25x25	10	8.60e-01 ±3.73e-01	1.52e+00 ±5.57e-01	6.58e-03	2.72e+00 ±1.95e-01	7.63e-06	2.47e+00 ±1.85e-01	7.63e-06	2.20e+00 ±2.18e-01	7.63e-06	1.74e+00 ±2.40e-01	7.63e-06

TABLE 7.4: Comparison between heuristics according to IGD.

In terms of IGD, the heuristic generated by ALGO outperforms all other heuristics as presented in Table 7.4. It means that the non-dominated solutions provided by the generated heuristic are closer to the optimal front than other heuristics. This result is particularly good since the front provided by the generated heuristic has more points than other ones for most of instances (see Figure 7.6).

Instance class		**Heuristic**										
grid	#UAVs	**ALGO**	**WO**	*p*-value	**Φ**	*p*-value	**Φ-K$_{0.05}$**	*p*-value	**Φ-K$_{0.10}$**	*p*-value	**Φ-K$_{0.20}$**	*p*-value
15x15	10	8.26e-01 ±6.43e-02	8.68e-01 ±8.06e-02	6.31e-03	9.15e-01 ±4.05e-02	6.46e-10	8.74e-01 ±6.32e-02	1.15e-04	8.67e-01 ±7.12e-02	9.31e-03	8.53e-01 ±8.43e-02	6.68e-02
20x20	05	8.95e-01 ±8.04e-02	9.00e-01 ±5.45e-02	8.99e-01	9.18e-01 ±3.54e-02	3.69e-01	8.88e-01 ±5.11e-02	5.51e-01	9.29e-01 ±4.40e-02	1.54e-01	8.47e-01 ±4.78e-02	4.32e-02
20x20	10	8.34e-01 ±4.33e-02	8.56e-01 ±1.94e-01	1.15e-02	9.18e-01 ±3.16e-02	3.58e-07	9.16e-01 ±4.22e-02	3.58e-07	9.01e-01 ±6.23e-02	2.05e-04	8.79e-01 ±7.06e-02	1.38e-02
20x20	15	8.64e-01 ±5.73e-02	8.91e-01 ±6.26e-02	1.58e-01	9.27e-01 ±3.61e-02	7.99e-06	9.13e-01 ±3.26e-02	7.98e-04	8.88e-01 ±5.89e-02	1.64e-02	8.48e-01 ±6.70e-02	5.70e-01
25x25	10	8.42e-01 ±2.84e-02	8.64e-01 ±8.88e-02	2.46e-01	9.46e-01 ±2.84e-02	7.63e-06	9.40e-01 ±3.88e-02	7.63e-06	9.24e-01 ±4.21e-02	7.63e-06	9.26e-01 ±3.80e-02	7.63e-06

TABLE 7.5: Comparison between heuristics according to Δ.

This diversity in the Pareto fronts obtained with the ALGO generated heuristic is confirmed by the results according to the Spread metric (Δ) presented in Table 7.5. ALGO obtained fronts are in the majority of cases more diverse than with the state-of-the-art heuristics. For some class, heuristic Φ-K however provides more diverse fronts, but with a 95% confidence only for the class `(20x20/5)`.

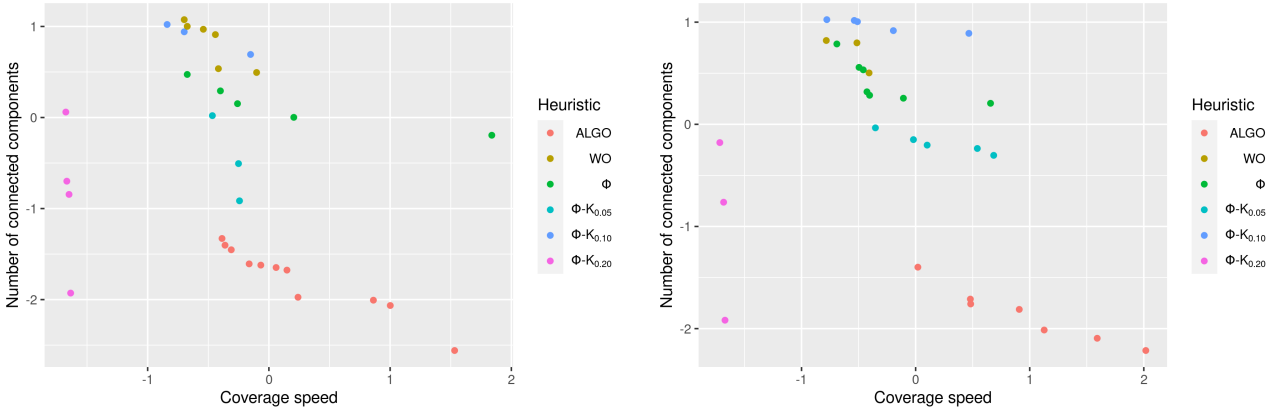


FIGURE 7.6: Example of fronts obtained with different heuristics for two instances, from the classes `(20x20/10)` on the left and `(25x25/10)` on the right.

Overall, except for some specific cases, the heuristic generated by ALGO outperforms the state-of-the-art heuristics according to the three MO metrics. By observing the fronts obtained with different heuristics (see

Figure 7.6 for example), several aspects can be pointed out. First, in terms of convergence of the Pareto front, all of pheromone-based heuristics are completed outperformed by the generated heuristic. Most of solutions provided by heuristics Φ and Φ-K are dominated by solutions from the front of the generated heuristic. Another point is that WO heuristic provides the best coverage results for every instance. It however does not seem to consider the connectivity aspect of the swarm while that heuristic has been designed for balancing CCUS3O objectives. It shows that the limitation of manual design is even more apparent when trying to deal with multi-objective optimisation problems. The last aspect is that the front obtained with the ALGO generated heuristic is generally the one containing the most solutions. This is an important feature since it shows that it is able to provide a larger set of well performing solutions.

The main purpose of this experiment was to demonstrate the stability of the model. After training ALGO on instances from `(20×20/10)`, the generated heuristic has showed very similar results after being executed on other instance classes. The class where the generated heuristic seems to be the least efficient is `(20×20/5)` in term of diversity of the front. This is however not due to the density of the swarm compared to the swarm of the grid (less UAVs than during training), since the generated heuristic provides good results on instances from `(25×25/10)` where the grid is bigger than during training. The model is thus able to generate a heuristic without being overfitted by the dimension of the instance. The stability of the model is a very promising aspect since it shows that the generated heuristic is able to perform well not only on unknown instances, but also in unknown situations and therefore dynamic environments.

7.6 Conclusion

In this chapter, we introduced a novel approach for learning to optimise the coverage of an area by a swarm of UAVs, i.e., automating the design of swarming behaviours in the context of area coverage missions. As a first step, a multi-objective optimisation problem has been defined to formally describe the objectives of the task to cover an area with a swarm of UAVs. This problem, called the Coverage of a Connected-UAV Swarm with 3 Objectives (CCUS3O), considers the coverage rate, the coverage time and the swarm connectivity. A CCUS3O instance represents a scenario for an area coverage mission. To automatically obtain the distributed heuristics necessary to tackle CCUS3O, we used our generic hyper-heuristic ALGO. It has been experimented on a set of CCUS30 instances. 50 instances from the class `20×20/10)` (10 UAVs on a 20×20 grid) have been used for the training. The generated heuristic has then been executed on 50 new instances of five different classes. ALGO has revealed a better convergence than existing techniques by generating a heuristic not only with better performance but also with more consideration of the multi-objective aspect. After processing a factorial experiment to determine an efficient parameterisation, experimental results demonstrate that while trained on a single instance class, the generated heuristic has shown to outperform on other classes a manually designed problem-specific heuristic along with state-of-the-art techniques designed for the coverage by a swarm of UAVs. In addition, empirical evidence of the good stability of the model and the better balance obtained on both objectives, coverage speed and swarm connectivity, has been provided.

Part IV

Conclusion

Chapter 8

Conclusion

Contents

8.1 Summary

This PhD work lies at the intersection of two research domains: optimisation and learning. This dissertation thus starts by presenting an overview of this wide domain in Chapter 2. To achieve this, a formalism to describe both the optimisation and learning concepts is introduced. For each process, the format of the required input and the returned output is defined, and generic components are extracted out of them. The purpose is to have a formal representation of both processes independent to specific existing techniques. The first aim of this categorisation is to understand the fundamental differences between optimisation and learning. This formalism is then utilised to analyse hybrid techniques, *i.e.*, combining two levels of optimisation or learning, with the high-level process improving the quality of the low-level process. Four categories can be drawn depending on whether optimisation or learning is used at the high or low level: *optimise optimisation*; *learning to optimise*; *optimise learning*; *learning to learn*. For each of them, we present how both processes can interact through their components defined in the formalism introduced beforehand.

Since in this work a learning algorithm (high-level) is used to improve an optimisation process (low-level), the latter belongs to the *learning-to-optimise* category. Thereby, Chapter 3 provides a state of the art about these hybrid techniques, hyper-heuristics to be more specific. An emphasis is made on works using Reinforcement Learning (RL) as the high-level learning algorithm. This makes it possible to draw the specific area in which this PhD work takes place.

The second part is the central point of this dissertation. Our model of generic hyper-heuristic based on RL, named Algorithm Learner for Graph Optimisation problems (ALGO), is presented in Chapter 4. Its purpose is to tackle different optimisation problems without considering a time-consuming redesigning task. ALGO can indeed be used to generate heuristics for different optimisation problems as long as they can be defined as an abstract problem, named ALGO-Friendly Optimisation Problem (AFOP). Chapter 4 is divided into two main sections, each presenting one level within our generic hyper-heuristic. In the first place, the low-level aspect

of ALGO, specific to the problem to tackle, is presented. An AFOP which is based on a graph structure, and a set of rules that an user must follow in order to map a problem to an AFOPis provided. The space of low-level heuristics to tackle such an abstract problem is then described. Secondly, the high-level RL algorithm is presented. The modular aspect of ALGOis demonstrated, with abstract components that the user can reimplement, *e.g.*, the graph neural network to encode the graph information and the scalarisation to tackle multi-objective problems.

Chapter 5 validates our model of generic hyper-heuristic by applying it to a classical optimisation problem: the Travelling Salesman Problem (TSP). An implementation of the TSP as an AFOP is first provided, which consists of a concrete example for an user who wants to use ALGO on a new problem. The results of the conducted experiences are finally presented. These consisted in generating TSP heuristics with ALGO and to compare their performance with state-of-the-art heuristics.

The real-world use case is finally presented in the third part. ALGO was used to automate the design of a swarm of UAVs for covering an area. A state of the art about the design of robot/UAV swarm is first given in Chapter 6, where existing techniques to manually and automatically design swarming behaviours are presented. A description of a new optimisation problem, so called Coverage of a Connected-UAV Swarm (CCUS) is also provided. CCUS is defined to illustrate the coverage of an area by a swarm of UAVs with the aim to remain connected. The latter problem has the distinction of being multi-objective and considering multiple agents running in a distributed way.

In Chapter 7, ALGO is used to generate distributed heuristics for CCUS. The resulting heuristic represents the behaviour of UAVs within the swarm. Similarly to Chapter 5, An implementation of CCUS as an AFOP is first presented, followed by a description of experiments and obtained results. We compared the generated heuristic to state-of-the-art techniques, including pheromone-based algorithms.

8.2 Contributions

Our contributions are summarised in Figure 8.1. They are classified according to the part of this thesis to which they belong.

The first objectives of this PhD thesis were to categorise techniques combining both learning and optimisation processes, by putting an emphasis on those aiming at *learning to optimise*. To this end, we presented a formalism to describe learning and optimisation concepts. It is helpful to categorise all of works combining both at different layers. Thanks to this formalism, we could gather different techniques into one paradigm by identifying which components of optimisation or learning are involved. Within this wide area, a focus was made on approaches using learning to improve the performance of an optimisation process. We thus present a state of the art of hyper-heuristics which are the mainly used *learning-to-optimise* techniques.

The main objective of this work was to design a generic hyper-heuristic based on RL to tackle different optimisation problems, which led to our main contribution, *i.e.*, the design of ALGO. ALGO makes it possible to generate heuristics for any compatible optimisation problem without the need to redesign the space of low-level heuristics. The strength of ALGO is its modularity and flexibility. It can easily tackle different compatible problems, regardless of whether they have a multi-objective aspect, a dynamic environment or a multi-agent context. There is even the possibility to adjust inner components. For instance, a Graph Neural Network (GNN) is used in ALGO's workflow. The user can therefore use any GNN. Similarly, the treatment of multi-objective optimisation problems is done with a scalarisation process which can be easily adapted by the user. In order to

apply ALGO on a new problem, the latter must override an abstract problem, AFOP. We then define an AFOP and detail the steps of the overriding task which is thought to be as intuitive and straightforward as possible. The model of ALGO was validated by applying it on the TSP. Experiments demonstrate that the TSP heuristic generated by ALGO is not only efficient, but also stable, *i.e.*, it performs well on instances with a bigger or smaller size than the training instances. The generated heuristic moreover outperforms most of state-of-the-art classical heuristics, and has similar performance than a 2-opt heuristic on big instances.

Since the heuristic generated by ALGO is limited by its greedy aspect (together with existing hyper-heuristic so far), we believe that it cannot be competitive when applied on classical optimisation problems which have already been deeply studied. We have thus thought ALGO to be mainly applied on dynamic problems, even considering a distributed aspect. We hence use ALGO to tackle a real-world problem. For that purpose, we defined CCUS which describes the task to cover an area with a swarm of UAVs by considering the connectivity within the swarm. Applying ALGO to CCUS then makes it possible to automate the design of behaviours within the swarm of UAVs. In the literature, the vast majority of existing techniques to automate the design of a swarm are based on predefined rules or actions that a UAV must choose at each iteration. By using ALGO in that context, we completely go beyond the state of the art by proposing a novel way to automate the design of a swarm of UAVs.

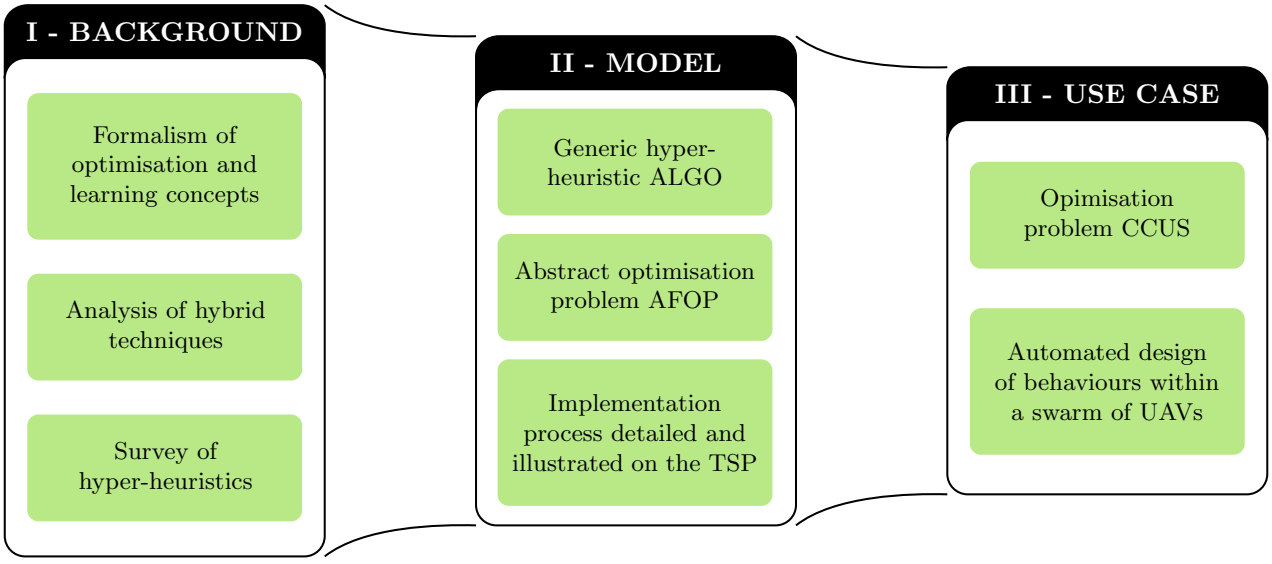


FIGURE 8.1: Summary of contributions.

All the publications related to this PhD work are listed below.

- Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. Learning to Optimise a Swarm of UAVs. *Applied Sciences*, 12(19):9587, January 2022b. ISSN 2076-3417. doi: 10.3390/app12199587
- Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. A Generative Hyper-Heuristic based on Multi-Objective Reinforcement Learning: the UAV Swarm Use Case. In *2022 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, July 2022a. doi: 10.1109/CEC55065.2022.9870223
- Emmanuel Kieffer, Gabriel Duflo, Grégoire Danoy, Sébastien Varrette, and Pascal Bouvry. A RNN-Based Hyper-heuristic for Combinatorial Problems. In Leslie Pérez Cáceres and Sébastien Verel, editors, *Evolutionary Computation in Combinatorial Optimization*, Lecture Notes in Computer Science, pages 17–32, Cham, 2022. Springer International Publishing. ISBN 978-3-031-04148-8. doi: 10.1007/978-3-031-04148-8_2

- Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. A Framework of Hyper-Heuristics based on Q-Learning. In *2022 International Conference on Optimisation and Learning (OLA)*, Syracuse, Italy, 2022c
- Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. A Q-Learning Based Hyper-Heuristic for Generating Efficient UAV Swarming Behaviours. In Ngoc Thanh Nguyen, Suphamit Chittayasothorn, Dusit Niyato, and Bogdan Trawiński, editors, *Intelligent Information and Database Systems*, pages 768–781, Cham, 2021. Springer International Publishing. ISBN 978-3-030-73280-6. doi: 10.1007/978-3-030-73280-6_61
- Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. Automating the Design of Efficient Distributed Behaviours for a Swarm of UAVs. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 489–496, Canberra, Australia, December 2020b. IEEE. ISBN 978-1-72812-547-3. doi: 10.1109/SSCI47803.2020.9308355
- Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. Automated design of efficient swarming behaviours: a Q-learning hyper-heuristic approach. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 227–228, Cancún Mexico, July 2020a. ACM. ISBN 978-1-4503-7127-8. doi: 10.1145/3377929.3390026
- Gabriel Duflo, Grégoire Danoy, and Pascal Bouvry. A Q-Learning Hyper-Heuristic for UAV Swarming. In *2020 International Conference on Optimisation and Learning (OLA)*, Cadiz, Spain, 2020c
- Gabriel Duflo, Emmanuel Kieffer, Matthias R. Brust, Grégoire Danoy, and Pascal Bouvry. A GP Hyper-Heuristic Approach for Generating TSP Heuristics. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 521–529, Rio de Janeiro, Brazil, May 2019a. IEEE. ISBN 978-1-72813-510-6. doi: 10.1109/IPDPSW.2019.00094
- Gabriel Duflo, Emmanuel Kieffer, Grégoire Danoy, and Pascal Bouvry. GP Hyper-Heuristic for the Travelling Salesman Problem. In *2019 International Conference on Optimisation and Learning (OLA)*, Bangkok, Thailand, 2019b

8.3 Perspectives

We showed that ALGO is able to generate a heuristic to tackle any problem that can be represented as an AFOP. We therefore propose to study the efficiency of ALGO according to the AFOP to tackle. We indeed suppose that the nature of the problem may influence the performance of ALGO. This analysis would be useful to draw more precisely the area of application of ALGO, illustrated in Figure 8.2. In a more general way, it would be interesting to specify which optimisation problem is an AFOP. A deep study could then help determining the feasibility of the mapping to an AFOP.

In a later stage, ALGO could be used for analytical purposes thanks to its modularity. The impact of a graph neural network on the generation of a heuristic could indeed be studied. In the same way, given a multi-objective problem, we could observe the convergence of the learning process according to the scalarisation chosen. Finally, some part of ALGO could become modular, especially the multi-agent aspect. At this point, we consider a scenario with 100% collaboration between agents, all agents sharing the same policy with a centralised learning. Other approaches are however conceivable. We could introduce some competition between agents with techniques from game theory. Moreover, the possibility to use distributed or federated learning can be added in ALGO.

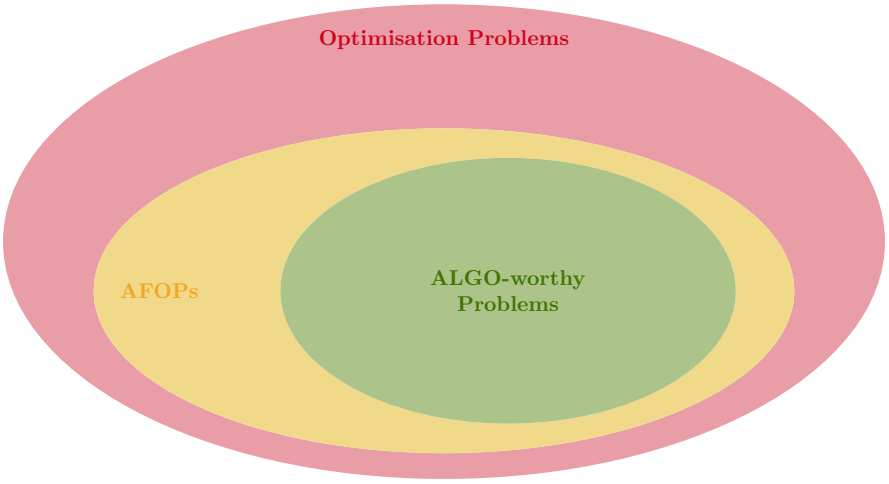


FIGURE 8.2: Area of application of ALGO. Among AFOPs, some optimisation problems may not be well suited to be tackled by ALGO.

Appendix A

Other Contributions

A GP Hyper-Heuristic Approach for Generating TSP Heuristics

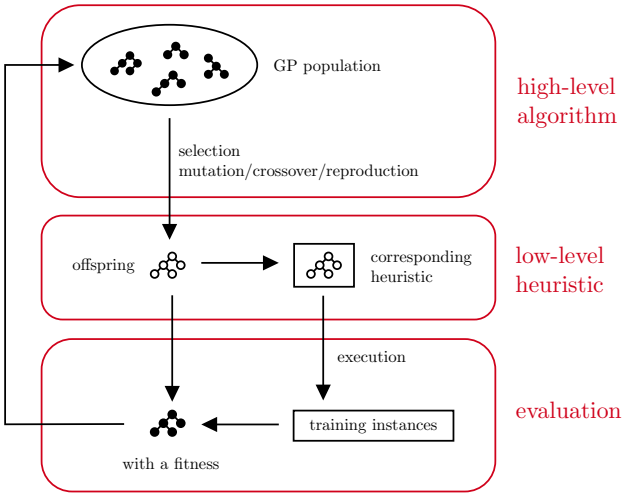


FIGURE A.1: Overview of the proposed GP hyper-heuristic.

Name	Description
set of operators	
+	Add two inputs
-	Subtract two inputs
*	Multiply two inputs
/	Divide two inputs with protection
%	Modulo between two inputs with protection
set of terminals	
Nn	Number of nodes in the graph
Nrn	Number of remaining nodes to visit
Dcn	Distance from the current node
Din	Distance from the initial node
Dc	Distance from the centroid of the nodes
Pd	Predicted distance from the initial node
Dle	Distance left estimation

TABLE A.1: GP nodes of the proposed hyper-heuristic

Test instances	**GPHH-best**		**Nearest neighbour**			**Nearest insertion**		**Greedy**		**Christofides**	
	mean	std dev.	mean	std dev.	p-value	value	p-value	value	p-value	value	p-value
ts225.tsp	136412.4	3387.212	147941.8	4076.552	7.11886e-17	151884.6	0	133459.7	0.7994262	133823.6	0.7745219
rat99.tsp	1381.677	29.24415	1474.923	59.65596	1.902147e-14	1465.884	0	1481.095	0	1325.171	0.9695625
rl1889.tsp	383303.7	5823.827	391697	7447.168	7.584303e-08	393573.5	0.05338111	378068	0.7889766	340583	1
u1817.tsp	69334.72	964.5484	69901.17	1181.565	0.01636283	70970.14	0.04076182	68517.05	0.7897301	65293.93	1
d1655.tsp	73740.45	1122.372	76950.71	1233.387	8.498443e-16	75390.58	0.07961539	72263.02	0.8977983	69989.38	1
bier127.tsp	136781.2	3841.962	145784.9	4241.502	2.435149e-13	145544.1	0.01623342	141351.1	0.1258296	133690.6	0.7487836
lin318.tsp	48039.78	1244.466	52865.57	1520.733	5.862237e-17	52299.12	0	49910.5	0.05434877	47830.88	0.535396
eil51.tsp	469.4567	12.76915	562.1576	32.86984	1.374741e-17	494.7529	0.02874414	481.5186	0.1386394	490.6541	0.04784139
d493.tsp	40453.72	590.4293	43403.9	893.9308	1.454267e-17	42140.47	0	40838.79	0.2611938	38333.65	1
kroB100.tsp	25254.54	813.0222	27955.27	1247.078	1.669134e-15	26908.61	0.02022685	25815.22	0.2484944	24316.05	0.8711435
kroC100.tsp	24114.56	616.4476	26094.22	1364.316	5.733966e-13	25780.57	0.02193806	23432.89	0.8747064	22632.53	1
ch130.tsp	7012.582	154.5253	7677.6	334.9915	3.362306e-17	7283.954	0.04391182	7844.935	0	6726.231	0.9570527
pr299.tsp	56980.64	1519.462	63334.8	2760.383	4.107702e-17	60263.85	0.02175183	63334.73	0.0008912489	53600.48	1
fl417.tsp	14555.84	666.7139	15706.24	504.8887	9.612834e-12	14887.62	0.2587874	13360.69	0.9739537	13707.55	0.9028805
d657.tsp	56882.87	947.1399	63456.26	952.5889	1.397338e-17	60081.63	0	56620.43	0.5641447	54004.06	1
kroA150.tsp	30660.12	587.0475	33440.39	702.5531	2.082407e-17	31588.4	0.05047068	31891.85	0.01281602	30089.76	0.8268097
fl1577.tsp	26163.75	581.1847	27813.25	734.2875	4.275012e-16	27625.77	0	25719.48	0.7662481	24216.51	1
u724.tsp	48423.29	620.0185	53834.65	1162.882	1.397428e-17	52629.51	0	49119.81	0.1179759	46955.06	0.9763688
pr264.tsp	60908.02	1293.462	57915.59	1261.642	1	65978.21	0	54974.84	1	54675.13	1
pr226.tsp	92837.77	4173.648	100178.3	5764.498	2.094478e-10	102887.2	0	97599.68	0.1807949	91753.45	0.5477194
pr439.tsp	130114.3	3009.391	136546.5	3981.135	2.973125e-12	133663.8	0.144199	128749.3	0.6465775	119181.3	1
xpr2308.tsp	8672.572	128.7851	9076.422	122.0759	5.465112e-11	9339.083	0	8632.146	0.5640092	8150.111	1
bcl380.tsp	1918.307	53.37217	2014.562	61.36226	2.590956e-07	2121.398	0	1876.972	0.7658752	1826.491	1
dkd1973.tsp	7792.836	68.11825	8520.187	126.6609	1.508983e-11	7990.916	0	7953.549	0.001037765	7247.629	1
pbd984.tsp	3281.114	48.26593	3582.584	71.60797	8.455617e-18	3480.203	0	3225.61	0.8011905	3139.56	1
dkg813.tsp	3931.528	61.91603	4044.898	81.07598	6.265391e-08	4082.176	0.01668589	3782.176	1	3576.115	1
rbx711.tsp	3632.072	86.08304	3821.43	78.24727	2.982211e-09	3941.668	0	3519.191	0.9126105	3501.695	0.9436677
rbu737.tsp	3934.535	43.39618	4284.951	116.2969	1.505204e-11	4157.718	0	3931.356	0.4997586	3715.348	1
dea2382.tsp	9927.748	171.8224	10202.24	229.9204	1.119734e-06	11074.04	0	9724.705	0.8341924	8845.206	1
bck2217.tsp	8038.587	74.88316	8567.246	84.23271	1.508038e-11	8777.92	0	7750.358	1	7679.882	1
xit1083.tsp	4262.549	55.46561	4677.546	61.75847	1.508038e-11	4374.092	8.207333e-05	4262.886	0.4651877	4073.27	1
icw1483.tsp	5334.868	89.92863	5725.362	99.80521	1.843717e-11	5629.58	0	5470.071	0.06932323	4969.913	1
dcb2086.tsp	7905.548	78.11186	8413.836	84.02766	8.455617e-18	8662.829	0	8076.356	0	7429.72	1
dka1376.tsp	5606.346	108.5535	5869.187	72.48792	4.561721e-13	6006.792	0	5761.826	0.08493456	5300.494	1
bva2144.tsp	7501.32	88.32525	8098.402	153.0135	8.455617e-18	8074.56	0	7441.435	0.6504908	7195.558	1
djc1785.tsp	7288.886	64.58055	7772.853	107.8659	1.508983e-11	7825.042	0	7158.487	0.9496844	6841.044	1
pka379.tsp	1603.686	29.45304	1630.845	34.44348	0.001749445	1580.694	0.6919587	1605.738	0.3806091	1530.744	1
fra1488.tsp	5124.243	61.52747	5572.661	129.8294	1.508983e-11	5151.302	0.338401	5407.792	0	4889.843	1
xqc2175.tsp	8179.526	122.6775	8610.238	107.738	2.299928e-15	8783.816	0	8140.881	0.578951	7775.502	1
fnb1615.tsp	5974.971	80.01615	6439.474	157.3768	1.508983e-11	6123.954	0.03469967	6034.764	0.2239577	5599.346	1
xqf131.tsp	647.8742	23.01932	708.0489	42.14538	3.791539e-07	687.1351	0.0381198	686.7936	0.04009066	612.0861	0.9075182

TABLE A.2: Comparison between the results obtained with GPHH-best and the other heuristics on the instances from TSPLIB

Appendix B

Experimental Results of QLHH on CCUS

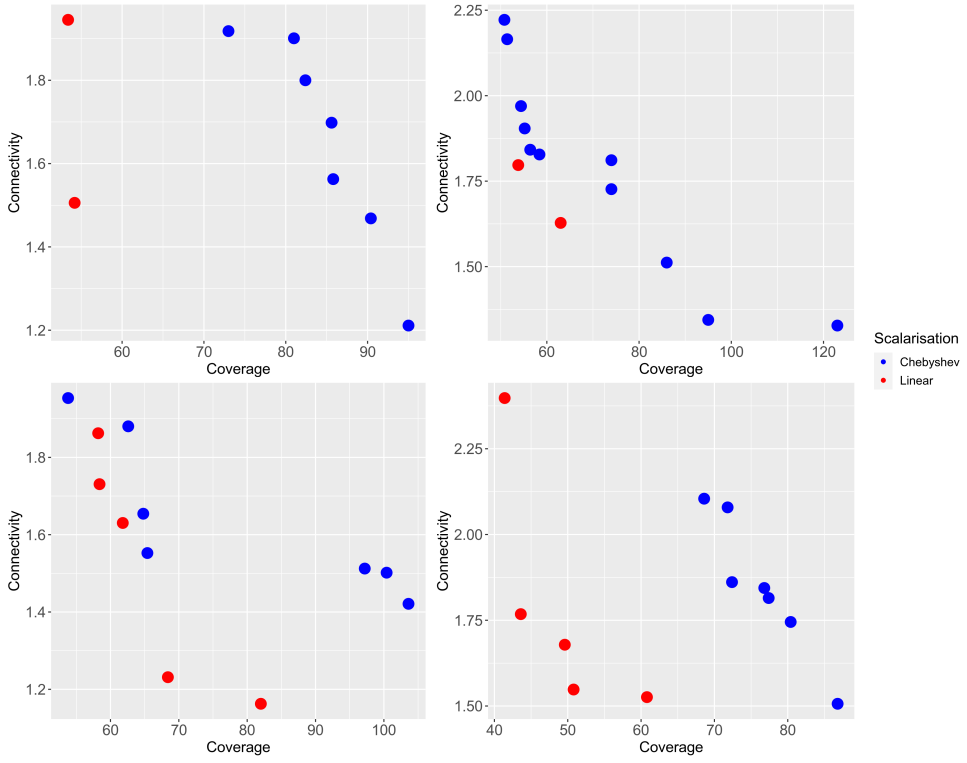


FIGURE B.1: Comparison of non-dominated heuristics obtained with linear and Chebyshev scalarisations.

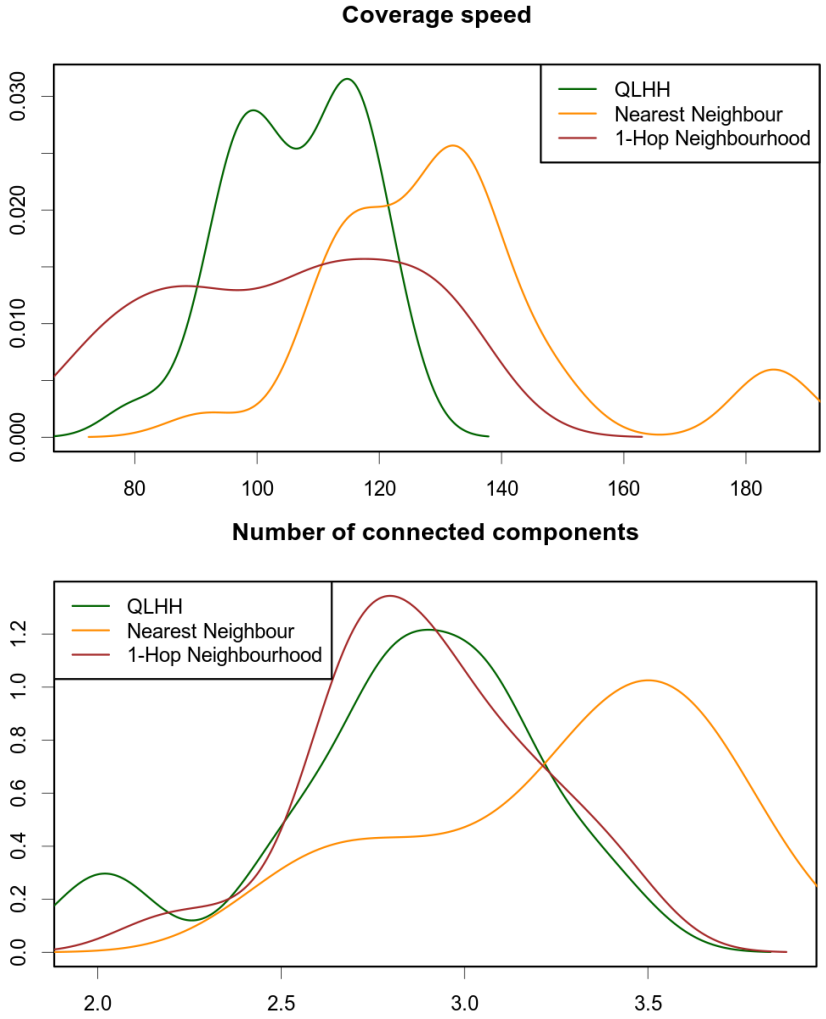


FIGURE B.2: Distributions of objective values obtained after running heuristics designed manually and the heuristic generated by QLHH on testing instances.

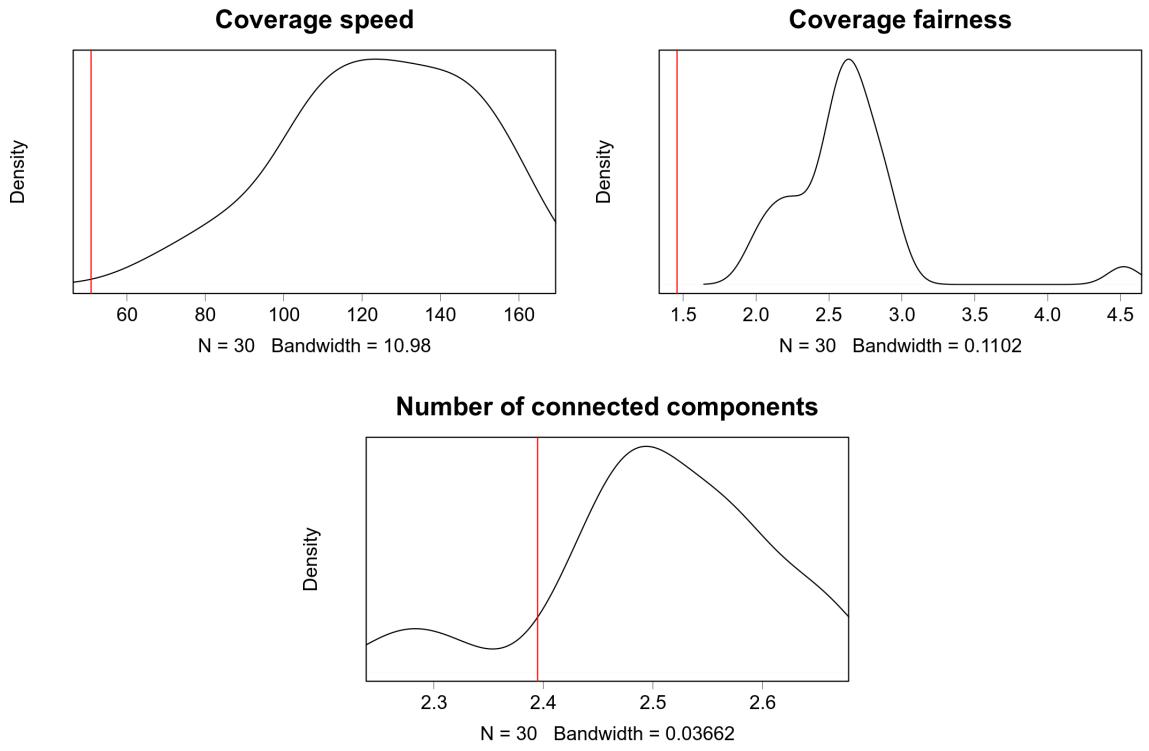


FIGURE B.3: Comparison of the generated heuristics with a random walk for one instance.

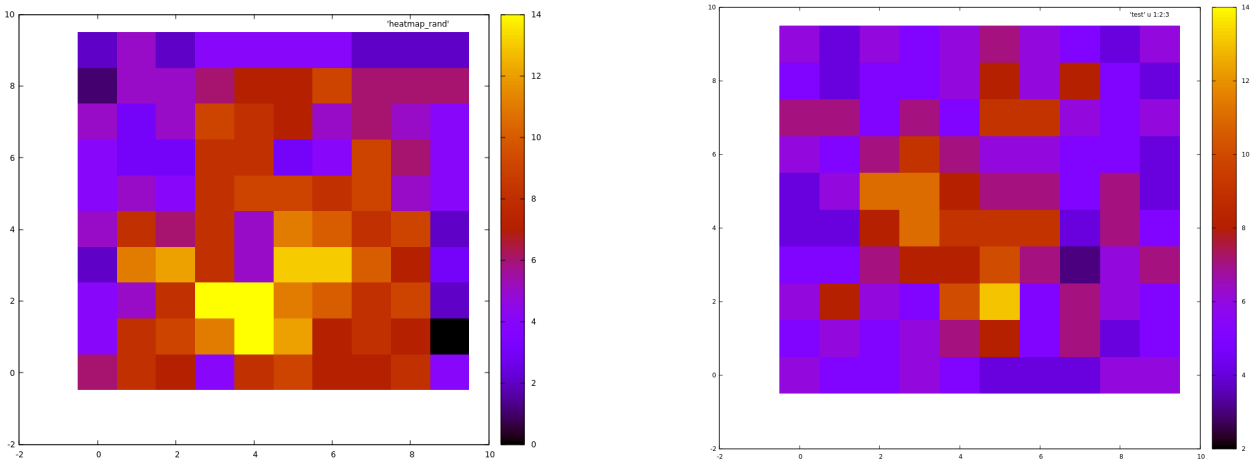


FIGURE B.4: Heatmaps of the number of visits of vertices with a random walk (on the left) and the generated heuristic (on the right).

Appendix C

Implementation of ALGO

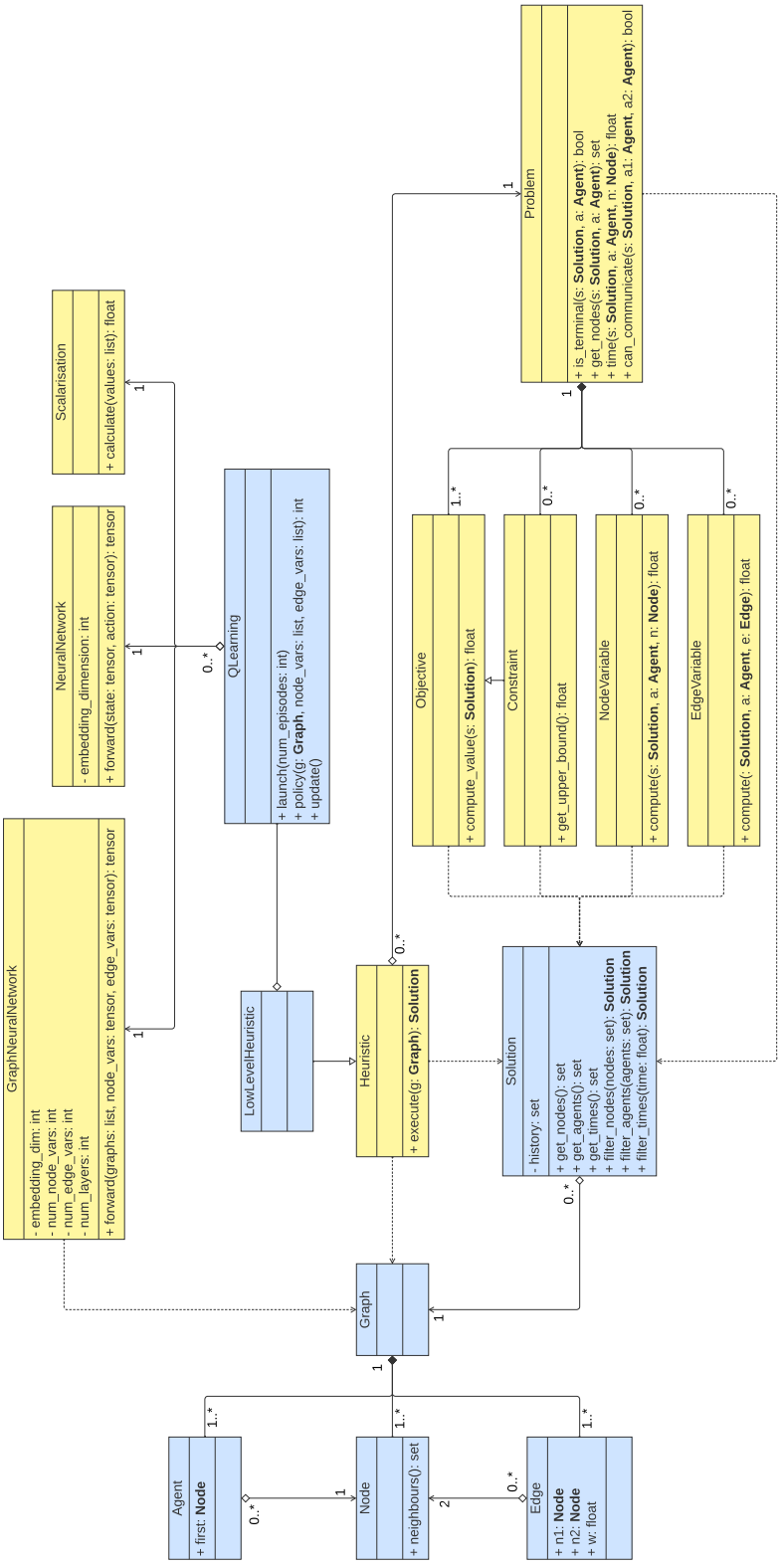


FIGURE C.1: Complete UML diagram for the implementation of ALGO.

Bibliography

Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64 (12):1695–1724, December 2013. ISSN 0160-5682, 1476-9360. doi: 10.1057/jors.2013.71.
5 citations in pages viii, 2, 22, 26, and 66.

Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.
2 citations in pages ix and 65.

Melanie Schranz, Martina Umlauft, Micha Sende, and Wilfried Elmenreich. Swarm Robotic Behaviors and Current Applications. *Frontiers in Robotics and AI*, 7:36, April 2020.
2 citations in pages ix and 65.

Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, October 2021. ISSN 0305-0548. doi: 10.1016/j.cor.2021.105400.
2 citations in pages 2 and 24.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning Combinatorial Optimization Algorithms over Graphs. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
3 citations in pages 2, 26, and 41.

Xingxing Hao, Rong Qu, and Jing Liu. A Unified Framework of Graph-Based Evolutionary Multitasking Hyper-Heuristic. *IEEE Transactions on Evolutionary Computation*, 25(1):35–47, February 2021. ISSN 1941-0026. doi: 10.1109/TEVC.2020.2991717.
2 citations in pages 2 and 23.

Fuqing Zhao, Shilu Di, Jie Cao, Jianxin Tang, and Jonrinaldi. A Novel Cooperative Multi-Stage Hyper-Heuristic for Combination Optimization Problems. *Complex System Modeling and Simulation*, 1(2):91–108, June 2021. ISSN 2096-9929. doi: 10.23919/CSMS.2021.0010.
2 citations in pages 2 and 25.

Yuchang Zhang, Ruibin Bai, Rong Qu, Chaofan Tu, and Jiahuan Jin. A deep reinforcement learning based hyper-heuristic for combinatorial optimisation with uncertainties. *European Journal of Operational Research*, 300(2):418–427, July 2022. ISSN 0377-2217. doi: 10.1016/j.ejor.2021.10.032.
2 citations in pages 2 and 25.

Hao Lu, Xingwen Zhang, and Shuang Yang. A Learning-based Iterative Method for Solving Vehicle Routing Problems. In *International Conference on Learning Representations*, February 2022.
One citation in page 2.

Kristof Van Moffaert, Madalina M. Drugan, and Ann Nowe. Scalarized multi-objective reinforcement learning: Novel design techniques. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement*

Learning (ADPRL), pages 191–199, Singapore, Singapore, April 2013. IEEE.
2 citations in pages 2 and 43.

Kristof Van Moffaert and Ann Nowé. Multi-Objective Reinforcement Learning using Sets of Pareto Dominating Policies. *Journal of Machine Learning Research*, 15(107):3663–3692, 2014.
One citation in page 2.

Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms. In Kyriakos G. Vamvoudakis, Yan Wan, Frank L. Lewis, and Derya Cansever, editors, *Handbook of Reinforcement Learning and Control*, Studies in Systems, Decision and Control, pages 321–384. Springer International Publishing, Cham, 2021.
2 citations in pages 2 and 38.

Ross Arnold, Kevin Carey, Benjamin Abruzzo, and Christopher Korpela. What is A Robot Swarm: A Definition for Swarming Robotics. In *2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference*, pages 0074–0081. IEEE, 2019.
2 citations in pages 2 and 64.

Daniel H. Stolfi, Matthias R. Brust, Grégoire Danoy, and Pascal Bouvry. A cooperative coevolutionary approach to maximise surveillance coverage of uav swarms. In *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6, 2020. doi: 10.1109/CCNC46108.2020.9045643.
One citation in page 15.

Maryam Karimi-Mamaghan, Mehrdad Mohammadi, Patrick Meyer, Amir Mohammad Karimi-Mamaghan, and El-Ghazali Talbi. Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *European Journal of Operational Research*, 296(2):393–422, 2022. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2021.04.032.
One citation in page 16.

Yuan Zheng, Xiaogang Fu, and Yanwen Xuan. Data-driven optimization based on random forest surrogate. In *2019 6th International Conference on Systems and Informatics (ICSAI)*, pages 487–491, 2019. doi: 10.1109/ICSAI48974.2019.9010547.
One citation in page 16.

Gabriel Duflo, Emmanuel Kieffer, Matthias R. Brust, Grégoire Danoy, and Pascal Bouvry. A GP Hyper-Heuristic Approach for Generating TSP Heuristics. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 521–529, Rio de Janeiro, Brazil, May 2019a. IEEE. ISBN 978-1-72813-510-6. doi: 10.1109/IPDPSW.2019.00094.
2 citations in pages 17 and 23.

Basheer Qolomany, Majdi Maabreh, Ala Al-Fuqaha, Ajay Gupta, and Driss Benhaddou. Parameters optimization of deep learning models using particle swarm optimization. In *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1285–1290, 2017. doi: 10.1109/IWCMC.2017.7986470.
One citation in page 18.

Joaquin Vanschoren. Meta-learning. *Automated machine learning: methods, systems, challenges*, pages 35–61, 2019.
One citation in page 18.

Peter Cowling, Graham Kendall, and Eric Soubeiga. A Hyperheuristic Approach to Scheduling a Sales Summit. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Edmund Burke, and Wilhelm Erben, editors, *Practice and Theory of Automated Timetabling III*, volume 2079, pages 176–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-42421-5 978-3-540-44629-3. doi: 10.1007/3-540-44629-X_11.
2 citations in pages 21 and 23.

D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. ISSN 1089778X. doi: 10.1109/4235.585893.
One citation in page 21.

Jian Lin, Lei Zhu, and Kaizhou Gao. A genetic programming hyper-heuristic approach for the multi-skill resource constrained project scheduling problem. *Expert Systems with Applications*, 140:112915, February 2020. ISSN 09574174. doi: 10.1016/j.eswa.2019.112915.
One citation in page 23.

Mohamed Abd Elaziz, Ahmed A. Ewees, and Diego Oliva. Hyper-heuristic method for multilevel thresholding image segmentation. *Expert Systems with Applications*, 146:113201, May 2020. ISSN 09574174. doi: 10.1016/j.eswa.2020.113201.
One citation in page 23.

Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. Exploring Hyper-heuristic Methodologies with Genetic Programming. In *Computational Intelligence*, pages 177–201. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-01798-8. doi: 10.1007/978-3-642-01799-5_6.
One citation in page 23.

Emmanuel Kieffer, Gregoire Danoy, Matthias R. Brust, Pascal Bouvry, and Anass Nagih. Tackling Large-Scale and Combinatorial Bi-Level Problems With a Genetic Programming Hyper-Heuristic. *IEEE Transactions on Evolutionary Computation*, 24(1):44–56, February 2020. ISSN 1089-778X, 1089-778X, 1941-0026. doi: 10.1109/TEVC.2019.2906581.
One citation in page 23.

Giovani Guizzo, Federica Sarro, Jens Krinke, and Silvia Regina Vergilio. Sentinel: A Hyper-Heuristic for the Generation of Mutant Reduction Strategies. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2020.3002496.
One citation in page 23.

Emmanuel Kieffer, Gabriel Duflo, Grégoire Danoy, Sébastien Varrette, and Pascal Bouvry. A RNN-Based Hyper-heuristic for Combinatorial Problems. In Leslie Pérez Cáceres and Sébastien Verel, editors, *Evolutionary Computation in Combinatorial Optimization*, Lecture Notes in Computer Science, pages 17–32, Cham, 2022. Springer International Publishing. ISBN 978-3-031-04148-8. doi: 10.1007/978-3-031-04148-8_2.
One citation in page 23.

John H Drake, Matthew Hyde, Khaled Ibrahim, and Ender Ozcan. A genetic programming hyper-heuristic for the multidimensional knapsack problem. *Kybernetes*, 43(9/10):1500–1511, November 2014. ISSN 0368-492X. doi: 10.1108/K-09-2013-0201. URL http://www.emeraldinsight.com/doi/10.1108/K-09-2013-0201.
One citation in page 23.

Yaroslav Pylyavskyy, Ahmed Kheiri, and Leena Ahmed. A Reinforcement Learning Hyper-heuristic for the optimisation of Flight Connections. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, Glasgow, United Kingdom, July 2020. IEEE. ISBN 978-1-72816-929-3. doi: 10.1109/CEC48606.2020.

9185803.
One citation in page 24.

Mourad Lassouaoui, Dalila Boughaci, and Belaid Benhamou. A synergy Thompson sampling hyper-heuristic for the feature selection problem. *Computational Intelligence*, page coin.12325, April 2020. ISSN 0824-7935, 1467-8640. doi: 10.1111/coin.12325.
One citation in page 25.

Wei Qin, Zilong Zhuang, Zizhao Huang, and Haozhe Huang. A novel reinforcement learning-based hyper-heuristic for heterogeneous vehicle routing problem. *Computers & Industrial Engineering*, 156:107252, June 2021. ISSN 0360-8352. doi: 10.1016/j.cie.2021.107252.
One citation in page 25.

Hanjun Dai, Bo Dai, and Le Song. Discriminative Embeddings of Latent Variable Models for Structured Data. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 2702–2711. PMLR, June 2016. ISSN: 1938-7228.
3 citations in pages 26, 40, and 41.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, Learn to Solve Routing Problems! February 2022.
One citation in page 26.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.
2 citations in pages 26 and 40.

Sahil Manchanda, AKASH MITTAL, Anuj Dhawan, Sourav Medya, Sayan Ranu, and Ambuj Singh. GCOMB: Learning Budget-constrained Combinatorial Algorithms over Billion-sized Graphs. In *Advances in Neural Information Processing Systems*, volume 33, pages 20000–20011. Curran Associates, Inc., 2020.
One citation in page 26.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
2 citations in pages 26 and 40.

Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. Automated design of efficient swarming behaviours: a Q-learning hyper-heuristic approach. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 227–228, Cancún Mexico, July 2020a. ACM. ISBN 978-1-4503-7127-8. doi: 10.1145/3377929.3390026.
2 citations in pages 26 and 67.

Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. Automating the Design of Efficient Distributed Behaviours for a Swarm of UAVs. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 489–496, Canberra, Australia, December 2020b. IEEE. ISBN 978-1-72812-547-3. doi: 10.1109/SSCI47803.2020.9308355.
2 citations in pages 26 and 67.

Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. A Q-Learning Based Hyper-Heuristic for Generating Efficient UAV Swarming Behaviours. In Ngoc Thanh Nguyen, Suphamit Chittayasothorn, Dusit Niyato, and Bogdan Trawiński, editors, *Intelligent Information and Database Systems*, pages 768–781, Cham, 2021. Springer International Publishing. ISBN 978-3-030-73280-6. doi: 10.1007/978-3-030-73280-6_61.
4 citations in pages 26, 67, 83, and 86.

Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. A Generative Hyper-Heuristic based on Multi-Objective Reinforcement Learning: the UAV Swarm Use Case. In *2022 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, July 2022a. doi: 10.1109/CEC55065.2022.9870223.
2 citations in pages 26 and 67.

Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. Learning to Optimise a Swarm of UAVs. *Applied Sciences*, 12(19):9587, January 2022b. ISSN 2076-3417. doi: 10.3390/app12199587.
3 citations in pages 26, 67, and 68.

S. Varrette, H. Cartiaux, S. Peter, E. Kieffer, T. Valette, and A. Olloh. Management of an Academic HPC & Research Computing Facility: The ULHPC Experience 2.0. In *Proc. of the 6th ACM High Performance Computing and Cluster Technologies Conf. (HPCCT 2022)*, Fuzhou, China, July 2022. Association for Computing Machinery (ACM). ISBN 978-1-4503-9664-6.
2 citations in pages 54 and 85.

Gerhard Reinelt. TSPLIB—A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384, November 1991. ISSN 0899-1499. doi: 10.1287/ijoc.3.4.376.
One citation in page 59.

Tauã M. Cabreira, Lisane B. Brisolara, and Paulo R. Ferreira Jr. Survey on Coverage Path Planning with Unmanned Aerial Vehicles. *Drones*, 3:4, March 2019.
One citation in page 65.

Barbara Siemiatkowska and Wojciech Stecz. A Framework for Planning and Execution of Drone Swarm Missions in a Hostile Environment. *Sensors*, 21(12):4150, January 2021.
One citation in page 65.

Fatih Semiz and Faruk Polat. Solving the area coverage problem with UAVs: A vehicle routing with time windows variation. *Robotics and Autonomous Systems*, 126:103435, April 2020.
One citation in page 65.

Shervin Nouyan, Alexandre Campo, and Marco Dorigo. Path formation in a robot swarm: Self-organized strategies to find your way home. *Swarm Intelligence*, 2(1):1–23, 2008.
One citation in page 66.

F. Ducatelle, G. A. Di Caro, C. Pinciroli, F. Mondada, and L. M. Gambardella. Communication assisted navigation in robotic swarms: Self-organization and cooperation. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4981–4988. IEEE, 2011.
One citation in page 66.

Xuelong Sun, Tian Liu, Cheng Hu, Qinbing Fu, and Shigang Yue. ColCOS Φ: A Multiple Pheromone Communication System for Swarm Robotics and Social Insects Research. In *2019 IEEE 4th International Conference on Advanced Robotics and Mechatronics (ICARM)*, pages 59–66. IEEE, 2019.
One citation in page 66.

Seongin Na, Yiping Qiu, Ali E Turgut, Jiří Ulrich, Tomáš Krajník, Shigang Yue, Barry Lennox, and Farshad Arvin. Bio-inspired artificial pheromone system for swarm robotics applications. *Adaptive Behavior*, 2020.
One citation in page 66.

Tian Liu, Xuelong Sun, Cheng Hu, Qinbing Fu, Hamid Isakhani, and Shigang Yue. Investigating Multiple Pheromones in Swarm Robots - A Case Study of Multi-Robot Deployment. In *2020 5th International Conference on Advanced Robotics and Mechatronics (ICARM)*, pages 595–601. IEEE, 2020.
One citation in page 66.

Tian Liu, Xuelong Sun, Cheng Hu, Qinbing Fu, and Shigang Yue. A Multiple Pheromone Communication System for Swarm Intelligence. *IEEE Access*, 9:148721–148737, 2021. ISSN 2169-3536. doi: 10.1109/ACCESS.2021.3124386.
One citation in page 66.

Erik Kuiper and Simin Nadjm-Tehrani. Mobility Models for UAV Group Reconnaissance Applications. In *2006 International Conference on Wireless and Mobile Communications (ICWMC'06)*, pages 33–33, Bucharest, Romania, 2006. IEEE.
2 citations in pages 66 and 84.

Martin Rosalie, Grégoire Danoy, Serge Chaumette, and Pascal Bouvry. Chaos-enhanced mobility models for multilevel swarms of UAVs. *Swarm and Evolutionary Computation*, 41:36–48, 2018.
One citation in page 66.

Grégoire Danoy, Matthias R. Brust, and Pascal Bouvry. Connectivity Stability in Autonomous Multi-level UAV Swarms for Wide Area Monitoring. In *Proceedings of the 5th ACM Symposium on Development and Analysis of Intelligent Vehicular Networks and Applications - DIVANet '15*, pages 1–8, Cancun, Mexico, 2015. ACM Press. ISBN 978-1-4503-3760-1. doi: 10.1145/2815347.2815351.
2 citations in pages 66 and 84.

Matthias R. Brust, Maciej Zurad, Laurent Hentges, Leandro Gomes, Gregoire Danoy, and Pascal Bouvry. Target Tracking Optimization of UAV Swarms Based on Dual-Pheromone Clustering. In *3rd IEEE International Conference on Cybernetics*, pages 1–8. IEEE, 2017.
2 citations in pages 66 and 81.

Edmund R. Hunt, Simon Jones, and Sabine Hauert. Testing the limits of pheromone stigmergy in high-density robot swarms. *Royal Society Open Science*, 6(11):190225, 2019.
One citation in page 66.

Mauro Birattari, Antoine Ligot, Darko Bozhinoski, Manuele Brambilla, Gianpiero Francesca, Lorenzo Garattoni, David Garzón Ramos, Ken Hasselmann, Miquel Kegeleirs, Jonas Kuckling, Federico Pagnozzi, Andrea Roli, Muhammad Salman, and Thomas Stützle. Automatic Off-Line Design of Robot Swarms: A Manifesto. *Frontiers in Robotics and AI*, 6, 2019.
4 citations in pages 66, 67, 73, and 74.

Michael G. Epitropakis and Edmund K. Burke. Hyper-heuristics. In Rafael Martí, Panos M. Pardalos, and Mauricio G. C. Resende, editors, *Handbook of Heuristics*, pages 489–545. Springer International Publishing, Cham, 2018. ISBN 978-3-319-07123-7 978-3-319-07124-4. doi: 10.1007/978-3-319-07124-4_32.
One citation in page 66.

Edmund K. Burke, Matthew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R. Woodward. A Classification of Hyper-Heuristic Approaches: Revisited. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 272, pages 453–477. Springer International Publishing, Cham, 2019. ISBN 978-3-319-91085-7 978-3-319-91086-4. doi: 10.1007/978-3-319-91086-4_14. Series Title: International Series in Operations Research & Management Science.
One citation in page 66.

Ke Li and Jitendra Malik. Learning to Optimize. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.
One citation in page 66.

Maryam Kouzehgar, Malika Meghjani, and Roland Bouffanais. Multi-agent reinforcement learning for dynamic ocean monitoring by a swarm of buoys. In *Global Oceans 2020: Singapore – U.S. Gulf Coast*, pages 1–8, 2020. doi: 10.1109/IEEECONF38699.2020.9389128.
One citation in page 66.

Mauro Birattari, Antoine Ligot, and Gianpiero Francesca. AutoMoDe: A Modular Approach to the Automatic Off-Line Design and Fine-Tuning of Control Software for Robot Swarms. In Nelishia Pillay and Rong Qu, editors, *Automated Design of Machine Learning and Search Algorithms*, Natural Computing Series, pages 73–90. Springer International Publishing, Cham, 2021. ISBN 978-3-030-72069-8.
One citation in page 67.

Antoine Ligot, Andres Cotorruelo, Emanuele Garone, and Mauro Birattari. Towards an Empirical Practice in Off-line Fully-automatic Design of Robot Swarms. *IEEE Transactions on Evolutionary Computation*, pages 1–1, 2022. ISSN 1941-0026. doi: 10.1109/TEVC.2022.3144848.
One citation in page 67.

Shuang Yu, Aldeida Aleti, Jan Carlo Barca, and Andy Song. Hyper-heuristic Online Learning for Self-assembling Swarm Robots. In Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2018*, volume 10860, pages 167–180. Springer International Publishing, 2018.
One citation in page 67.

Shuang Yu, Andy Song, and Aldeida Aleti. A Study on Online Hyper-heuristic Learning for Swarm Robots. In *IEEE Congress on Evolutionary Computation*, pages 2721–2728, 2019.
One citation in page 67.

Sasanka Nagavalli, Nilanjan Chakraborty, and Katia Sycara. Automated sequencing of swarm behaviors for supervisory control of robotic swarms. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2674–2681. IEEE, 2017.
One citation in page 67.

Fernando Silva, Miguel Duarte, Luís Correia, Sancho Moura Oliveira, and Anders Lyhne Christensen. Open issues in evolutionary robotics. *Evol. Comput.*, 24(2):205–236, jun 2016.
One citation in page 73.

Gianpiero Francesca and Mauro Birattari. Automatic Design of Robot Swarms: Achievements and Challenges. *Frontiers in Robotics and AI*, 3, 2016.
One citation in page 73.

Matthias R. Brust, Hannes Frey, and Steffen Rothkugel. Dynamic multi-hop clustering for mobile hybrid wireless networks. In *Proceedings of the 2nd International Conference on Ubiquitous Information Management and Communication*, ICUIMC '08, page 130–135, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939937. doi: 10.1145/1352793.1352820.
One citation in page 84.

Gabriel Duflo, Grégoire Danoy, El-Ghazali Talbi, and Pascal Bouvry. A Framework of Hyper-Heuristics based on Q-Learning. In *2022 International Conference on Optimisation and Learning (OLA)*, Syracuse, Italy, 2022c.

Gabriel Duflo, Grégoire Danoy, and Pascal Bouvry. A Q-Learning Hyper-Heuristic for UAV Swarming. In *2020 International Conference on Optimisation and Learning (OLA)*, Cadiz, Spain, 2020c.

Gabriel Duflo, Emmanuel Kieffer, Grégoire Danoy, and Pascal Bouvry. GP Hyper-Heuristic for the Travelling Salesman Problem. In *2019 International Conference on Optimisation and Learning (OLA)*, Bangkok, Thailand, 2019b.