Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

School of Engineering and Sciences

Analysis of a Feature-independent Hyper-heuristic Model for Constraint Satisfaction and Binary Knapsack Problems

A thesis presented by

Xavier Fernando Cuauhtémoc Sánchez Díaz

Submitted to the
School of Engineering and Sciences
in partial fulfillment of the requirements for the degree of

Master of Science

in

Intelligent Systems

Monterrey, Nuevo León, December, 2017

Dedication

For Angie.
Your determination fills me with hope.

For Xiomara.
An aesthete of transcendent sensibility. No doubt.

For Karen.
You came into my life at just the right moment.

Acknowledgements

Firstly, I would like to thank my thesis advisors, Hugo Terashima and José Carlos, whose guidance and encouragement have been key to the development of this dissertation. For their patience and all the challenges and opportunities they gave me. I am very grateful.

Working closely with the Research Group with Strategic Focus in Intelligent Systems has been enlightening, so I would like to thank all members who in any way have been involved with the progression of this research, but especially to the thesis committee members, Iván Amaya and Santiago Conant, for their invaluable feedback and recommendations. Every word of advice is appreciated, as it continuously shape the quality of my work.

The director of the Master Program, Ramón Brena, for trusting me and giving me the opportunity of proving to myself what I am capable of.

I would also like to thank Emma Hart, for her time and recommendations for this dissertation. The insightful discussion we had back in April 2017 was of immense impact for the design of the framework of this work.

To Patricia Salinas, for introducing me to the world of academic research. I must say I was fortunate enough to experience the rigorous side of mathematics research, which in turn has reflected in my life choices; something I am really grateful for.

To my classmates; especially Gilberto Ayala, Gabriel González and Pedro Fonseca for the long sessions of midnight work, to Luis Felipe González for the mathematical discussions, and to Victoria Esqueda for the food and the notebooks. An special mention for Homero Hernández, as his role as a programmer had a tremendous impact on this project. Part of this achievement is yours.

I must also thank my family, for their unconditional support throughout the years. To Xiomara, my sister, for the emotional support even from afar. To my parents, Guillermo and María Eugenia, for the encouragement and life guidance. I should listen to you more often. To my grandfather, since we all have been the focus of your love and support. To my cousins, Angie and Caro, for their daily support. Your presence here makes my life easier. You two look like Mariana. And Calvin.

And thank you to `Frodxx`, Carlos Hernández and `ArturoZero` for the laughs—a much needed medicine from time to time, and to `ScarletSteel` and `Steveecake` for being there at night where no one else was.

Thank you all for your support.

Analysis of a Feature-independent Hyper-heuristic Model for Constraint Satisfaction and Binary Knapsack Problems

by

Xavier Fernando Cuauhtémoc Sánchez Díaz

Abstract

This dissertation is submitted to the Graduate Programs in Engineering and Information Technologies in partial fulfillment of the requirements for the degree of Master of Science with a major in Intelligent Systems.

This document describes and analyzes empirical results of an evolutionary-based hyper-heuristic model applied to Binary Knapsack Problems (0/1 KP) and Constraint Satisfaction Problems (CSP). Hyper-heuristics are high-level methodologies that either select among existing algorithms or generate new ones for solving complex problems. The objective of this dissertation is to contribute to the knowledge about hyper-heuristics and propose a model which is feature-independent in order to obtain competing solutions on a variety of fields.

The CSP is a classical example of a complex problem where the solution is a valid assignment of variables in order to satisfy a set of constraints. It has many applications including knowledge representation, scheduling and optimization. Although there could be many solutions to a single CSP instance, finding them could represent a hard challenge—its complexity is, in general terms, computationally intractable.

With many real-life applications like allocation and cargo loading, the 0/1 KP is another example of complex problems, and is one of the most studied in the optimization branch. Its description can be summarized as finding a selection of items packed in a container which yields the highest profit without violating a capacity constraint.

Both KPs and CSPs can be stated as classical search problems, undergoing through a search tree associated to a problem instance. Each node in the search tree is a decision point—a variable in CSP or an item in KP. For each of these decision points many different heuristics can be used. Finding the right heuristics at the right decision points requires of efficient strategies, since an exhaustive search may be impossible due to the exponential growth of the variables involved in the problem. Many heuristics that are feasible and efficient exist for both CSPs and KPs. Nevertheless, these methods usually require delicate setup and constant monitoring as it may be necessary to adjust some parameters in order to get a solution within an acceptable time frame.

The concept of hyper-heuristics holds high relevance in this context since new solutions can come up from already known heuristic methods. In order for the hyper-heuristic to appropriately map a method to a state of the search problem, it needs some criteria. Usually problem characterization is useful in this process—two problem instances with similar features are very likely to be solved efficiently in the same manner. However, problem characterization may be especially difficult in some cases, as most real-life situations usually feature hyper-dimensional search spaces which are non-calculus friendly. In this dissertation we are interested in developing a solution model able to come up with general methods that show

good performance even when dealing with no problem characterization whatsoever. Having a more flexible solution model allows for an easier adaptation of the framework to other domains, where problem characterization may be somewhat difficult.

This document explores an evolutionary approach to generate hyper-heuristic from a pool of already known heuristics. Experiments suggest that the model is able to find and exploit key decision points in both KP and CSP. The model can also find better solutions from mixing heuristics that perform poorly on their own. However, the method is prone to overfitting under certain conditions. The general idea of this dissertation is to provide a better understanding on the generality aspect of hyper-heuristics and provide a feature-independent model to generate hyper-heuristics in order to tackle a variety of combinatorial optimization problems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In computer science, problems are classified according to their complexity. Although finding the solution to some problems could be computationally difficult, some of them are easily verifiable if a solution to the problem is known. These problems are called *Non-deterministic polynomial* problems, and are represented with the acronym NP. The term *Non-deterministic polynomial* suggests that the amount of resources needed to solve one of these problems is polynomial when using a non-deterministic Turing machine. If a polynomial-time method for solving a problem is known, then the problem is assigned to the *polynomial* class, represented with P. However, if no polynomial-time solution method is known, an exhaustive search may be necessary (in a worst-case scenario) to find the optimal solution for such problem [69].

Some other problems may not be easily verifiable (not NP), but may still be hard to solve nonetheless. The halting problem is a well-known example, which cannot be solved by any computer no matter how much time one allows the algorithm to run. If an algorithm that could solve it existed, then all NP problems could be solved in a similar manner. These types of problems are in the so-called NP-hard class [68]. The halting problem is NP-hard but not NP.

However, if a problem is both NP and NP-hard, it is referred to as being *Non-deterministic Polynomial Complete*, often abbreviated as NP-C [11], [56], [67]. These problems are considered the hardest of all NP problems [21]. The amount of possible states (and thus, the time needed to find a solution) grows faster than the number of variables involved in these problems. Because of this wide gap in complexity, proposed solutions for NP-C problems are actually approximations most of the time; doing an exhaustive search of all possible combinations may not be feasible.

The constraint satisfaction problem (CSP) is an important problem in the field of artificial intelligence and optimization. CSPs have many real-life applications such as scheduling, knowledge representation, diagnosis, resource allocation and network optimization [15]. The CSP is NP-complete—Finding a solution is not a trivial task. A CSP solution is an assignment of a value to each variable in order to comply with all the problem constraints.

The 0/1 Knapsack Problem (KP, also referred to as binary KP) is another significant problem in optimization. KPs are used to model cargo loading, cutting stock, allocation and cryptography problem situations. The KP belongs to the NP-Hard class, which makes finding an optimal item combination as hard as deciding if there is any such combination which yields a certain minimal value without exceeding the capacity constraint.

Many methods to solve a CSP exist, but none is equally good for all problem instances. This behavior suggests that each instance is more suitable to be solved by a given technique. The same could be said for the binary KP, where the performance of a certain operator varies greatly from instance to instance. A higher-level method which selects appropriate techniques may be used to tackle on most problem instances, and a hyper-heuristic is a good candidate for this task [5].

This work includes the review of feature-independent hyper-heuristic model applied to both the CSP and the 0/1 KP. Results are analyzed and contrasted in order to generalize the proposed method to other areas of combinatorial optimization.

1.1 Problem Definition and Motivation

A CSP consists of three components: a set of variables, a set of domains (one for each variable) and a set of constraints which specifies allowed combinations of assigned values for variables.

Each domain consists of a set of allowed values for each variable. Each constraint is an ordered pair that contains a relation between two variables and their limitations, either in the form of a list of allowed values or a condition describing these constraints.

A variable assignment that satisfies all the constraints is called a consistent or legal assignment. If a consistent assignment is also complete (when all variables have been assigned) then a solution has been found. There could be many solutions for a single instance of a CSP [52]. When looking for a solution in a CSP, the assignment of each variable should be checked against the whole set of constraints. This process ensures the assignment is consistent, and it is usually called *consistency check* (CC). Consistency checks are a good way to measure the efficiency of a given algorithm on a problem instance.

Another complex problem that will be covered in this thesis is KP. The 0/1 KP consists on packing a selection of items inside a container (a *knapsack*, hence the name of the problem) with limited capacity, in such a way that the capacity constraint is not violated and the profit of the items in the knapsack is maximized.

There are many algorithms that use rules and trial and error approaches to find a solution for these problems. These algorithms are called heuristics. Generally, heuristics are specifically designed for a given problem and often require some expertise to design and implement them in everyday-use computers [8]. Most heuristics rely on precise adjustments of their parameters and components, and most of this setup is usually done off-line and manually.

Different approaches to tackle on computationally hard problems exist. A first approach consists on finding an approximate solution, using methods with some random elements at certain degree, and reviewing possible solutions until finding one that meets a given performance criteria. These algorithms using stochastic elements are amongst the best options to solve complex problems [27].

In 1997, Wolpert and Macready described the *No Free Lunch for Optimization* theorem [70]. This theorem states that all algorithms are equally efficient when evaluated on a complete series of problems, no matter the evaluation criteria. Nevertheless, empirical studies by Gomes and Selman show that, in many cases, the performance of an algorithm dominates over all other methods for a given problem [27]. Analyzing these algorithms could lead to an

automatic selection of suitable solvers for problems in various fields of knowledge, and this notion suggests a new concept to review: hyper-heuristics.

Hyper-heuristics can be defined as "algorithms to select or generate algorithms" [8]. As opposed to heuristics, hyper-heuristics search the set of available heuristics to select or generate a solution for a given problem [50]. The process of selecting or efficiently designing a hybrid heuristic is a search problem by itself [5], and results in an increment of the computation time. Under what circumstances is the use of hyper-heuristics viable when applied to combinatorial optimization problems?

In order to either select or generate a solution, a hyper-heuristic must check all heuristic methods available to decide which of them offers the best solution for a given problem instance. Hyper-heuristics can be applied to CSPs or KPs in a similar manner. During this learning process, each heuristic method is tested against an instance for a short amount of time. There are different learning models for hyper-heuristics, and the learning phase may vary its length depending of the particular hyper-heuristic model. For example, the learning phase of a selection hyper-heuristic based on a generational genetic algorithm [31], roughly consists of the following:

- An initial population of n heuristics.
- A set of k problem instances for testing.
- A maximum allowed runtime of t units of time for each available heuristic method.

These are the variables for each of the m generations, so the whole learning process would take up to $mnkt$ units of time.

At first sight, the cost of solving a problem with a hyper-heuristic may not appear desirable. Nevertheless, if the proposed solution by a hyper-heuristic method actually requires fewer CCs than a simple heuristic (when applied to CSPs), then there could be an improvement in computing time for each of the problem instances that were solved. In the case of KP, if the hyper-heuristic method yields better results than an operator run in isolation, it may be worthwhile to go with a high-level method instead. This improvement may manifest in different forms. For example, when solving a large number of KP problem instances, the total profit of such instance set is expected to be greater for an efficient hyper-heuristic if it has an edge over a simple heuristic on most instances. A significant improvement in profit may also be relevant if packing expensive materials, where the profit gains outweigh the hyper-heuristic training cost.

Hyper-heuristics have been previously applied to both CSPs (see [8], [43], [44], [45],) and packing problems (see [7], [16], [17], [28], [33] and [54]). Nevertheless, little has been described about their use and the interaction between lower-level selected or generated heuristics. This "automated" aspect of hyper-heuristics represents an unexploited potential for finding solutions to these problems.

It is of high importance to determine if the learning cost of a hyper-heuristic is viable for the development of CSP and KP solvers. First of all, many optimization problems in the industry can be represented using a CSP or the binary KP. For example, bin packing and cutting-stock problems, vehicle routing, time-tabling and scheduling [14], [34], [38].

Characterizing the learning behavior in hyper-heuristics is a key factor for a better understanding of these high-level algorithms. Identifying these conditions may have a great impact on solution methods for both decision and optimization problems such as CSPs and KPs.

1.2 Objectives

The main objective of this work is to conduct an experimental analysis of a feature-independent hyper-heuristic model applied to both constraint satisfaction and knapsack problems. We expect to define key factors that are relevant for the implementation and improvement of this hyper-heuristic model to see if its application is possible for other domains in combinatorial optimization.

The analysis focuses on the efficiency of the learning mechanism, as well as the interaction between selected heuristic methods. To achieve this general goal, the following particular objectives are considered:

- Identify heuristic interaction patterns in the application of the hyper-heuristic model to several instances of CSPs and 0/1 KPs.
- Determine the feasibility and repeatability of the hyper-heuristic model by comparing it against isolated, simple heuristics on both CSPs and KPs.
- Justify the hyper-heuristic approach for 0/1 KPs and CSPs.
- Identify the advantages and disadvantages of the model in order to determine its viability to tackle other combinatorial optimization problems.

1.3 Hypothesis

Evolutionary algorithms can deal with optimization problems even when no problem knowledge is available. An experimental analysis of an evolutionary based, feature-independent hyper-heuristic model applied to binary knapsack problems and constraint satisfaction problems will help identify factors that are key to improve algorithm design and application.

This work intends to answer the following research questions:

1. Does a combination of heuristics guarantee a more efficient solution than using standalone methods?
2. Is it possible to obtain a good solution from combining heuristics that perform poorly in isolation?
3. Can the method be applied to decision problems as well?
4. Which conditions are necessary for the hyper-heuristic method to obtain quality solutions in other combinatorial optimization problems?

1.4 Solution Overview

This document explores the application of an evolutionary hyper-heuristic model for both KP and CSP.

Overall, this investigation revolves around the fact that many combinatorial optimization problems can be formulated as sequential search processes. Because an exhaustive approach for these processes is infeasible, heuristic methods are used to determine which problem state to search next. Many heuristics exist, but are often specialized in certain types of problems. Another way to guide the search is to analyze problem features. However, real-life problem situations usually contain noisy search spaces which are hyper-dimensional and non calculus-friendly—concise problem characterization is not always possible.

The solution presented in this research provides a more flexible approach, based on the idea of using a combination of different methods at appropriate decision points and thus create more general solvers without the need of problem characterization. The extensibility of the model allows for an easy cross-domain adaption at a certain extent.

This study explores different aspects related to optimization and hyper-heuristics. Firstly, the need of appropriate training examples and objectives functions as they are crucial for guiding the search. The second one is the repeatability of the algorithm, since it is essential for the method to be used in different domains. The final aspect, which is perhaps the most important, was the capability of the method to generate acceptable solutions from combining heuristics which perform poorly when used in isolation.

For KP, bad decisions are easily identifiable and avoidable, since the objective function is clearly defined—maximize profit. However, a bad decision in a CSP may still get to any of the solutions, making it impossible to determine its performance as in other decision problems. The hyper-heuristic in this dissertation considers maximizing profit as the objective function for KP, and minimizing the amount of consistency checks for CSPs.

The hyper-heuristic method described throughout this investigation moves to states of the problem which seem promising. The algorithm escapes from local optima using stochastic ‘jumps’ in the form of disruptive mutations, resembling the (1+1) Evolutionary Algorithm. The resulting solution methods were tested on a wide set of instances, both from synthetic datasets as well as instances from the literature.

1.5 Main Contributions

This dissertation delves into certain aspects of hyper-heuristics with little to no research available in the literature at the time of its writing. The most important contribution is the experimental analysis of a feature-independent hyper-heuristic model based on an evolutionary algorithm. Results showed that even though the model was tested on KPs and CSPs, the model can be applied to other domains of optimization with a few modifications. In a more descriptive way, the contributions of this investigation are:

The analysis of a feature-independent hyper-heuristic model

The hyper-heuristic model analyzed in this work showed that mutation operators can be used to tackle on difficult instances even if no problem features are known. The model

revolves around an evolutionary metaphor, making it easy to implement and extend to combinatorial optimization problems with concise objective functions.

Suitable solvers can be obtained from non-promising heuristics
When heuristic performance is lacking, or no known method is able to solve a problem instance, a mixture of these operators may behave better than any of the components alone. Diversity of operators, as well as taking the right decisions at the right time, makes a hyper-heuristic a powerful tool to deal with complex problems.

A frequency analysis of heuristic interaction for CSP and KP
A frequency analysis was conducted for 2-place combinations of heuristics for both CSP and KP. The evolution of the hyper-heuristics during the learning phase was analyzed to determine which combinations of heuristics are favored by the learning method.

1.6 Outline of the Thesis

The ensuing chapters of this dissertation detail technical information relevant to this investigation. Chapter 2 presents the background and state of the art of the topics related: CSP, KP, Hyper-heuristics and Evolutionary Algorithms. In Chapter 3, the methodology used during this research is described. All phases of the investigation, including the experimentation methodology, are detailed in this chapter along with their justification. Chapter 4 presents the hyper-heuristic framework used and its technical and implementation details. This chapter serves as a preamble of the two subsequent chapters. Chapter 5 shows the application of the framework to the 0/1 Knapsack Problem and the analysis for that particular problem. The application of the framework to the Constraint Satisfaction Problem and its analysis is presented in Chapter 6. Finally, Chapter 7 provides conclusions derived from the investigation. In addition, different alternatives for future work are presented and their challenges described.

Chapter 2

Related Background

This chapter presents a detailed description of the concepts and topics related to this research. A brief description of the problems at hand—both CSP and binary KP—is included, as well as some heuristics used to solve them. The concept of hyper-heuristic is defined next– their definition, characteristics and some background on their application to both CSPs and KPs. Finally, a review of the state of the art is presented, where related work and research is described.

2.1 Constraint Satisfaction Problems

This section covers a formal definition of the constraint satisfaction problem. A brief introduction to key definitions is presented, followed by some measure concepts and common solution methods used in CSPs.

2.1.1 Definitions and Formal Representation

A classic CSP can be represented as a set of three sets: a set of *variables*, a set of *variable domains* and a set of *constraints*.

Variables are objects or items that can take on a variety of values. The set of all values that a single variable can take is referred to as its *domain*. A *constraint* is a rule that imposes a limitation on variables and the values they can take simultaneously [14]. A crossword puzzle is one of the simplest examples of a CSP: there is a set of spaces we must fill (variables), a set of available letters to choose from (values), and some rules we need to stick to (constraints).

A formal definition is as follows.

A CSP $\mathcal{P}$ is a triple $\mathcal{P} = \langle X, D, C \rangle$, where

- X is an n-tuple of variables $X = \langle x_1, x_2, \ldots, x_n \rangle$,
- D is a corresponding n-tuple of domains $D = \langle D_1, D_2, \ldots, D_n \rangle$ such that $x_i \in D_i$ and
- C is a t-tuple of constraints $C = \langle C_1, C_2, \ldots, C_t \rangle$

A constraint C_j is a pair $\langle R_{S_j}, S_j \rangle$, where R_{S_j} is a relation on the variables in S_i, i.e. a subset of the Cartesian product of the domains of the variables in S_i [22]. The number of affected variables by a constraint C_j is known as the arity of the constraint [57].

The set of variables X in a CSP must be finite. Variable domains D_i, however, could be infinite (e.g. any numerical variable). Discrete domains are also common in CSPs, when variables can take categorical or boolean values.

A solution to the CSP $\mathcal{P}$ is an n-tuple $A = \langle a_1, a_2, \ldots, a_n \rangle$ where $a_i \in D_i$ and each C_j is satisfied in that R_{S_j} holds on the projection of A in S_j [22].

A CSP may consist in finding any of the following three cases [64]:

- Any possible solution.
- All solutions of the problem.
- The optimal solution according to some domain knowledge.

This thesis focuses on CSPs as decision problems: is it possible to find a consistent assignment for $\mathcal{P}$, given X, D and C?

An specialization of the CSP where the domains are restricted to truth values $D = \{T, F\}$ and each constraint is a clause of k variables is known as the k-SAT problem, the first problem proved to be NP-Complete [11]. Hence, CSP is also NP-Complete.

2.1.2 Searching in a CSP

Heuristics for solving CSPs look for values for each variable so that their assignment does not violate any constraint. This solution can be found either with local search algorithms (which do not assure finding a solution in all cases); or using complete methods that guarantee finding a solution if at least one exists. Nevertheless, when bigger and more complex CSPs appear, using exact methods is not feasible as the search space grows exponentially with the number of variables, and so does the computing time to find a solution.

Search in a CSP is usually represented in an abstract way by using a *depth search tree* [52]. When a variable is assigned in a CSP, all the constraints must be checked. If there is any conflict between the variables given the existing constraint, then another value in the domain of such variable should be assigned. If there is no value that does not violate any constraint, then previously assigned variables must also be reassigned. There are many ways to do this variable reassignment. *Backtracking* is one of these methods, which goes to the immediately preceding level to modify the value of a variable already assigned [27]. *Back-jumping* is a somewhat more efficient approach, as it jumps back many previous levels in the search tree [45].

Since there could exist more than one solution that satisfies all conditions in the objective function, it is important to note that some of these solutions could be easier to find than others. This means that possible improvements of solution methods for CSPs are actually related to improvements of the search method itself. The number of *Consistency Checks* (CC) can be seen as a way to measure the efficiency [44] of such searching strategy.

2.1.3 Measure Concepts in CSPs

There are some components that can be analyzed in order to measure the hardness of a CSP. One of these characteristics is the *size of the problem*, which varies according to the number of variables and subsequently their domain size. Because the size of a CSP consists of all possible variable assignments, the size of the search space can be represented by the product of the domain sizes

$$\prod_{x_i \in X} m_{x_i} \tag{2.1}$$

The size of the problem $\mathcal{N}$ is the number of bits used to describe a point in the search space [24], and it is expressed as:

$$\mathcal{N} = \sum_{x_i \in X} \log_2 m_{x_i} \tag{2.2}$$

Another meaningful concept to measure is the *solution density*. Solution density considers that each constraint c_i prohibits a fraction p_{c_i} of possible values assigned to variables, and thus, a fraction $1 - p_{c_i}$ of assignments are allowed for that variable. Assuming all constraints are independent, then we can calculate the average solution density ρ as the average fraction of allowed assignments for all variables [24]:

$$\rho = \prod_{c_i \in C} (1 - p_{c_i}) \tag{2.3}$$

The *expected number of solutions* is derived from the previous concepts and it is represented as $E[S]$, which is the product of the size of the search space multiplied by the probability that any element in this search space is a solution [24]:

$$E[S] = \prod_{x_i \in X} (m_{x_i}) \times \prod_{c_i \in C} (1 - p_{c_i}) \tag{2.4}$$

It is expected that computationally hard problems appear when $E[S] \approx 1$ [42].

Constraint density (p_1) represents the proportion of edges in the graph of constraints, while *constraint tightness* (p_2) denotes the proportion of conflicts in the problem instance. For any CSP with arity a, the number of maximum edges in a constraint graph of n variables is given by $\binom{n}{a}$. If e constraints exist in the problem instance, then the constraint density is denoted as [42]:

$$p_1 = \frac{e}{\binom{n}{a}} \tag{2.5}$$

It is important to notice that the number of constraints in a CSP instance cannot exceed $\binom{n}{a}$, and that the number of conflicts is defined by the number of involved variables and their domains. Assuming variable domains to be of equal size, and that the problem instance contains c constrained tuples [42], the tightness of a CSP instance is:

$$p_2 = \frac{c}{\binom{n}{a} m^a} \tag{2.6}$$

The factor κ derives from both the size and density of the solution, and is a general measurement of how constrained a problem is. With lower values of κ, problems usually have more solutions for their size. On the other hand, when κ is large, instances tend to have fewer solutions or none at all. κ is defined as follows [24]:

$$\kappa = \frac{-\sum_{c_i \in C} \log_2(1 - p_{c_i})}{\sum_{x_i \in X} \log_2(m_{x_i})} \tag{2.7}$$

In this equation, c_i represents a constraint in which the variable x is involved, m_x represents its domain, and p_{c_i} is the fraction of tuples that cannot be satisfied with the c_i constraint.

2.1.4 Phase Transition in CSP

There exists a threshold under which the structure of a random graph $\mathbb{G}(n, p)$ (with n vertices and a probability p of every edge occurring independently) changes from a scattered collection of trees to a dense lump of components, and it is usually referred to as "the giant component". The short period where the giant component emerges is known as the phase transition.

Erdős and Rényi [19] showed that the size of the largest random tree, varying certain parameters little by little, grew from $\mathcal{O}(\log n)$ through $\Theta(n^{2/3})$ to $\Omega(n)$—in other words, a sharp increase in complexity [23].

Cheeseman et al. found critical values for many NP-complete problems: areas in the search space where feasible solutions are clustered [9]. This was also the case for CSPs. For CSP instances with few constraints, finding a solution is easy as there are lots of alternatives to choose from. On the other hand, for CSPs with too many constraints there is a high chance that no solution exists at all—this is also relatively easy to determine. Nevertheless, there is a threshold where the value of some parameters can greatly influence the hardness of the problem. These values are known as *critical values*, and are described in [9] and [10]. In the case of boolean satisfiability (SAT), Mitchel et al. demonstrated that if the number of clauses is around 4.3 times the number of variables, then the problem is nearly impossible to solve for high number of variables [40].

Cheeseman et al. [9] also reviewed the order of assignment of variables in CSPs: choosing a variable may allow the assignment of other variables that were previously constrained. This forms a chain reduction that has a great impact on algorithm performance. Crawford and Auton later confirmed the existence of a *crossover point*, where half of the randomly generated constraint satisfaction problems are actually solvable [12]. Below this crossover point, complexity of 3-SAT problems (boolean satisfiability) grows almost linearly with the number of variables involved. On the contrary, complexity grows exponentially for 3-SAT instances above the crossover point. This behavior suggests that the order of variable selection in CSPs heavily impacts computing time.

Studies by Gomes and Selman in [27] show that performance of backtracking drastically varies depending on how the next variable is selected, and in which order the possible values are assigned to such variable. The first condition is referred to as the variable selection strategy, while the latter is known as the value selection strategy.

2.1.5 Common Heuristics for CSPs

There exist many heuristic methods for selecting the order in which variables are assigned in a CSP. These heuristics may be based on variable domains, constraint levels or number of conflicts. Some of these heuristics are described briefly in this section.

- The **Max-Conflict** heuristic selects the variable which is involved in most conflicts amongst the constraint set. This will create a sub-problem that minimizes the number of conflicts between the remaining unassigned variables.
- The **Saturation degree** method measures the number of already assigned variables connected to a single node. This way, the heuristic will always select the variable connected to the most assigned variables at a given time.
- **Min-domain** (or simply *dom*) prefers variables with the smallest domains. It is highly likely that a variable fails to satisfy a constraint if it has a small domain.
- The **dom/deg** heuristic combines two parameters to decide which variable to assign first: the size of its domain and the number of uninstantiated variables connected to each node (known as the forward degree). The variable that maximizes the quotient of the domain size over the forward degree will be the first variable selected.
- In the case of the **kappa** heuristic, the κ function is used to measure the difficulty of the remaining instances. For this reason, the variable x_i that maximizes Equation 2.7 will be selected.
- The **Rho** heuristic was proposed by Gent [24], and uses the ρ parameter (Eq. 2.3 in Section 2.1.3) to guide the search towards variables with less constraints, so that resulting sub-problems are under-constrained and thus easy to solve.
- **Weighted Degree** (Wdeg) is another graph-based heuristic which directs the search towards *hard* components of the search tree. A counter is assigned to every constraint in the instance and is updated every time a dead-end is found. Variables with the highest weighted degrees are preferred.
- **Maximum forward degree** is a heuristic which favors the variable with most connections to unassigned variables. This heuristic is also known as Deg.

It is important to note that the heuristic methods previously described are useful to determine the order for assigning the variables in a CSP. There are also heuristics used to determine the order to assign the values of these variables. The most common heuristic for this task is **Min-conflicts**. Min-conflicts will prefer values involved in the least conflicts in order to leave the next assignments as flexible as possible. This way, resulting sub-problems should be easier to solve, as each sub-problem will have more values to choose from [45].

An extended list of many other strategies for both variable and value ordering can be found in [42], [65].

Selection strategies can greatly reduce computing time for CSPs. Understanding these strategies is key to develop new solvers.

2.2 The Binary Knapsack Problem

This section includes the formalization of KP and a brief description of common heuristics used in this domain.

2.2.1 Definitions and Formal Representation

The 0/1, also called binary, KP is defined by Martello and Toth [38] as follows. Given a knapsack of capacity c and a set of n items with profits p_j and weight w_j of item j; select a subset of the items in order to

$$\text{maximize} \qquad z = \sum_{j=1}^{n} p_j x_j \tag{2.8}$$

$$\text{subject to} \qquad \sum_{j=1}^{n} w_j x_j \leq c, \tag{2.9}$$

$$x_j = 0 \text{ or } 1, \qquad j \in N = \{1, \ldots, n\}, \tag{2.10}$$

where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases} \tag{2.11}$$

The term "binary" in this case refers to the actual state of an item inside the knapsack: it is either packed entirely or not included at all. This notion arises from other variants of KP, for example the *Fractional Knapsack Problem* (FKP). In FKP, each item generates a cost and thus the objective is to maximize the ratio of the total profit over the total costs [32]. This research focuses on the binary variant of KP.

Looking at Eq. 2.8, one can see that KP is a linear integer programming problem, which objective is to maximize a numerical value—it is looking for a single solution rather than any of the feasible solutions available. In the case of KP, the solution is the subset of the set of items with maximal profit and less than a capacity constraint. This notion contrasts the idea of answering whether or not a solution exists, which is the case of the studied CSPs according to the definition in Section 2.1.1.

2.2.2 Complexity and Solution Methods for KP

As Kellerer et al. describe, only *pseudopolynomial* algorithms are known for exact solutions of KP: algorithms asymptotically bounded by a polynomial both in n (the size of the problem) and in one (or several) of the input values. Even problems with small values for n may have large coefficients and thus long running times when solved in this fashion [34]. As well as the CSP, KP belongs to the NP-hard class of problems—finding the optimal solution of a KP is at least as hard as deciding whether or not a CSP instance is solvable.

Despite the KP being an NP-hard problem, it is considered to be on the "easy" side of these problems. Falkenauer considers KP as a grouping problem [20], which is characterized

by a cost function where one item taken in isolation has little or no meaning for the entire solution. Since the combination of appropriate decisions is what leads to an acceptable result, it is of high importance to decide which item to pack next. After each item is packed, one could represent the remaining subproblem as a new instance of KP with a reduced capacity and a smaller partition of the set of items. These new instances could be solved either using exact or approximate methods.

The *Greedy Algorithm*, perhaps the most natural of the approximate solution methods for solving KPs, looks to pack the "best" item that can fit in the knapsack. If this item cannot be packed, then it approaches the second best, and so on until no more items can be included in the knapsack. The "greedy" idea is to sort the items in descending order according to certain criteria (usually a ratio of cost over weight, more heuristics are described in Section 2.2.3) , and go one by one up to the end [34]. This approach runs in linear time after sorting the items, which can be done efficiently in $\mathcal{O}(n \log n)$ units of time.

A linear programming approach to get exact solutions is to include a "relaxation" for the integrality constraint, so that "fractions" of the items can be included. The solution given by this *linear programming relaxation* (LKP) is based on the local optima obtained by the greedy approach: items are sorted in descending order and every item is subsequently packed into the knapsack. When the greedy algorithm stops for the first time, there may be space still available but not enough to pack a whole item. The optimal solution is then achieved packing a corresponding fraction of the last item. As well as the greedy algorithm, LKP can be solved in $\mathcal{O}(n \log n)$ time units [34].

Optimal solutions for the KP can also be obtained using the *dynamic programming* approach: consider an optimal solution of a KP instance and then remove an item r. The remaining solution is an optimal for the subproblem of capacity $c - w_r$. This shows the presence of an optimal substructure [11], which is the foundation of a dynamic programming application. The solution method is then build using the following idea: Assume an optimal solution has been found to a subproblem of KP (with a given subset of items). For each item added to the set of items in the KP instance, the optimal solution must be checked. This check can be performed using information of the previously solved KP subproblems, to see which is the item that needs to change. The process is repeated for all items in capacity c, an the optimal solution is then found in pseudopolynomial time.

An extensive collection of solution methods for the KP can be found in [34] and [38].

2.2.3 Common Heuristics for KP

As mentioned in Section 2.2.2, the greedy algorithm, the one covered in this work, sorts the items of a KP according to certain criteria. This section lists the heuristics (also referred to as packing operators) used in this research.

- **Def** (Default). Items are packed in their default order, as long as they fit in the knapsack. This heuristic runs in $O(n)$ time, and the quality of the solutions obtained by this operator are as inefficient as randomly packing items in the knapsack.
- **Max-P** (Max-Profit). Items are sorted from most valuable to less valuable, and then packed in this order as long as they fit in the knapsack.

- **Max-PW** (Max-Profit per Weight unit). For each item, a profit-over-weight ratio is obtained. All items are then sorted from most valuable-per-weight-unit to less valuable. Items are then packed in this order as long as the capacity constraint is not violated.
- **Min-W** (Minimum Weight). This heuristic favors lighter items. They are sorted in ascending order according to their weight, and then packed in this order as long as the capacity constraint is not exceeded.

All the *ordering* operators (Max-P, Max-PW and Min-W) run in $O(n \log n)$ time. While all these three packing operators take longer to compute, they are expected to yield better results than using no ordering at all (e.g. using the Default heuristic). These approximation schemes may not seem attractive by the fact that the boundary between the optimum and the proposed solution could be quite large. However, any approximation algorithm (e.g. Greedy) can also compute an optimal solution in linear time [34]. Hence, the attractiveness of using these approaches.

2.3 Hyper-heuristics

Another important concept is that of hyper-heuristics. To understand their significance, an introductory take on learning mechanisms is studied along the hyper-heuristic definition. Hyper-heuristic classification and a brief description of different models are presented later throughout this section.

2.3.1 Hyper-heuristics and Learning Mechanisms

Hyper-heuristics can be considered high-level heuristics that select or construct algorithms to solve complex problems [4]. They do not solve problems directly. Instead, they browse a set of available solvers, searchers and heuristics that can solve the problem at hand [50]. To determine which heuristic to use in which problem instance, a hyper-heuristic method may undergo a learning process. This is one of the main problems tackled in artificial intelligence: learning.

Although the term *learning* in computer science is usually associated with classification, clustering, prediction and planning; the word itself describes any process which improves performance of future tasks after making observations about the environment [52]. Alanazi and Lehre present a mathematical definition of a learning mechanism within a selection hyper-heuristic in [1]:

Let $\mathscr{X}$ be a finite search space and $f\colon \mathscr{X} \to \mathbb{R}$ be a cost function. Let m be the number of low-level heuristics, and $h_j^{(t)}$ be the selected heuristic in iteration t. Let $p(h_k^{(t+1)})$ be the selection probability of heuristic k in iteration $t+1$. A learning mechanism can be defined as a function ℓ:

$$((h_j^{(i)}, f(x_i)))_{i=1,\dots,t} \to (p(h_k^{(t+1)}))_{k=1,\dots,m} \tag{2.12}$$

where

$$\sum_{k=1}^{m} p(h_k^{(t+1)}) = 1$$

In other words, a learning mechanism is a function mapping a pair of *what-to-do* and *how-well-it-went* to a probability of doing *what-to-do* in the future. A hyper-heuristic is, in this sense, a learning mechanism which comes up with a probability of using promising algorithms when facing a given problem.

2.3.2 Classification of Hyper-heuristics

There are different ways to classify hyper-heuristics. Perhaps the most common is what Burke et al. proposed in [6], where hyper-heuristics are categorized according to their learning method or search models.

By learning type, hyper-heuristics are classified as follows:

- **On-line learning**. The learning process of these hyper-heuristic methods are executed at runtime, when the algorithm is actually solving a problem instance. This way, characteristics of the problem can be used to determine which heuristic method to use.
- **Off-line learning**. Hyper-heuristic methods obtain information from solving a set of training instances, from which a general idea of how to solve the problem is learned so that it is later applied to similar, but not known problems.

Hyper-heuristics can also be classified according to their solution approach:

- **Constructive hyper-heuristics**. All those hyper-heuristics that construct a solution in an incremental way.
- **Perturbation hyper-heuristics**. These methods start with a complete solution, and slowly alter some of its components in order to improve the algorithm on each iteration.

However, the most fundamental distinction is to separate hyper-heuristics according to their methodology:

- **Heuristic selection**: methodologies for choosing or selecting existing heuristics.
- **Heuristic generation**: methodologies for generating new heuristics from the components of existing ones.

Despite the *natural* differences between these methodologies, some elements in hyper-heuristics may be generated while others being selected from existing components, so a completely *orthogonal* classification is far from ideal: Swan et al. further suggest that a hard distinction between on-line and off-line learning may be a significant obstacle to progress in hyper-heuristic design as it may exist some information gathered on-line that is useful off-line and vice versa [61].

2.3.3 Hyper-heuristic Models

As seen on Section 2.3.2, there are many approaches to the search problem when using a hyper-heuristic. And, as a matter of fact, which element in the search space to look for and deciding what to do next is an optimization problem by itself [61].

The hyper-heuristic search process may vary from one approach to another. Generation hyper-heuristics, for example, search a space of *construction blocks* instead of the whole solution space, as opposed to what selection strategies do [5]. The search process in a selective hyper-heuristic can be divided, in general, into two phases: heuristic selection and move acceptance—the former focuses on choosing an algorithm to solve the problem while the latter deals with the solution quality [46].

In order for a selective hyper-heuristic method to choose amongst one of the available heuristics, it needs a performance measure $P(a)$. The heuristic method that maximizes the expected performance $E[P(a)]$ will be selected to solve that specific problem instance [50]. The solution can be, then, accepted or discarded according to an acceptance criteria.

Many classical learning mechanisms have been used as hyper-heuristic models, but soft computing algorithms are perhaps the most common.

For constraint satisfaction problems, Poli and Graff used Genetic Programming (GP) as a hyper-heuristic model [50] as well as Sosa-Asencio et al. [60], Lourenço et al. propose a Grammatical Evolution algorithm (GE) [37], Ortiz-Bayliss et al. utilize a Genetic Algorithm (GA) [43], as well as Terashima-Marín et al. [62]. A comparison of GP, GA and Neural Networks can be found in [42].

Packing problems have also been tackled using hyper-heuristics. Hart and Sim describe an Artificial Immune System (AIS) used as a hybrid hyper-heuristic for the Bin Packing Problem (BPP) in [28], [30], and [54]. Falkenauer [20] proposed a GA-based hyper-heuristic model, and Burke et al. [7], Hyde [33] and Drake et al. [16], [17] studied GP rules for bin packing and multi-dimensional KP.

There are some other, non-nature inspired approaches—Terrazas and Krasnogor use grammatical inference in a hyper-heuristic constructive model [63], Arbelaez et al. use a Support Vector Machine (SVM) as a learning mechanism [3], while O'Mahony et al. propose a case-based approach for algorithm portfolios [41]. Additionally, Alanazi and Lehre cover limitations on additive reinforcement learning methods when used as hyper-heuristic models [2]. However, a thorough search of the relevant literature has shown that both selection approaches and evolutionary algorithms are far more common in the state of the art.

This work focuses on a hybrid approach: mainly a selection hyper-heuristic with added elements which are inherent to generation approaches, using an evolutionary algorithm. A complete description of the framework is included in Chapter 4.

2.4 Heuristic Assessment and Selection

This section contains information which revolves around the idea behind the heuristic selection phase of selective hyper-heuristics: which algorithm should be used next? The word *next* is crucial, since both in KP and CSP the solutions are obtained in a sequential way.

Hyper-heuristic methods applied to CSPs have been studied previously. The research conducted by Ortiz-Bayliss et al. in [44], [45], and Wallace in [66] suggest that the use of

different heuristic methods can reduce computing time in CSPs.

Given a set of k heuristics and a CSP instance with n variables, the maximum number of sequences of heuristics that can be formed is k^n, assuming that the problem is actually satisfiable. However, combining heuristics is not enough to improve the search process—not all heuristics work well when combined.

Ortiz-Bayliss et al. described some sequences of heuristics that work better than others [45]. The sequential combination of kappa and dom/deg (both being variable order heuristics for CSPs, detailed in Section 2.1) seems to work particularly better than other sequences of heuristics. On the contrary, min-domain does not show good results when combined with dom/deg. It is clear that selecting appropriate heuristics heavily impacts runtime for CSPs. Likewise, KP also presents this behavior. As mentioned in Section 2.2.2, deciding which item to pack next in the knapsack is a computationally hard problem—despite its natural feel, KP is an NP-hard problem after all. The selection of the most suitable operator to use on the next item (or variable, for CSPs) matches a well-studied problem in the literature, known as the algorithm selection problem. The next section covers a more formal definition on this topic.

2.4.1 The Algorithm Selection Problem and Heuristic Performance

The idea of selecting the right algorithm for a given problem is known as *the algorithm selection problem* (ASP), and was first described by Rice [51] in 1976. Formally, the algorithm selection problem can be defined as [58]:

> For a problem instance $x \in P$, with features $f(x) \in F$, find the assignment $S(f(x))$ in an algorithm space A, so that the selected algorithm $\alpha \in A$ maximizes the performance of the assignment $y(\alpha(x)) \in Y$.

This also applies to the algorithm selection problem of hyper-heuristic methods: a selection hyper-heuristic needs to find an α algorithm in its available operators A, in order to maximize the performance of such α algorithm when solving the problem instance x of its list of available problem instances P.

It is here where heuristic performance and interaction comes into play. Smith-Miles [58] presented a detailed review about several learning models for different problems: optimization, constraint satisfaction, and regression and classification problems. The research also proposes a framework which aims to achieve automatic selection of an algorithm no matter its application or knowledge domain. This framework comprises an empirical learning approach that is divided into three phases:

1. Phase one requires meta-data generation for the domain of the problem, which is the set of P, A, Y, F, as previously defined in the formal definition of ASP.

2. Phase two focuses on learning the relations between problem features F, and performance measures of the algorithm Y, in order to create empirical rules and rankings for each algorithm.

3. In phase three, empirical results of the learning process are compared against theoretical results obtained in phase two, so that the learning process can be refined and hence improvement is achieved.

An implementation of this framework (applied to graph coloring problems) is presented in [59]. Graph coloring problems can be represented as CSPs.

However, performance on hyper-heuristics is also a matter to consider. Defining parameters and resources used by hyper-heuristic methods is always a difficult task. Most of the time, learning *success* is usually proportional to computing time or the storage capacity defined for a hyper-heuristic method. In this way, this allocation of resources can be considered a high-level control mechanism, instead of being a static dimension of the problem [61]. Nevertheless, the running time over a particular problem instance cannot be guaranteed in most solvers [53].

Because of this, it is common that users try to optimize the rate of how hyper-heuristic search operators are applied—using a trial and error approach, for example—in order to obtain a hyper-heuristic method that is able to find feasible algorithms which should be associated with acceptable performance values, and using the least algorithms as possible. This manual adjustment is too, in a sense, a searching process—one in which humans are *the searchers* [50].

Likewise, Bai et al. suggested that the amount of allocated memory can influence the performance of hyper-heuristics, and that a good decision (resource-wise) could improve the quality of the solution [4]. Petrovic supports this idea: a full restart seems to improve performance of the search, as well as the use of random subsets of variables (when dealing with CSPs) [48].

Mısır et al. presented an empirical study in which the set of heuristics influences the performance of hyper-heuristics, as well as the execution time: a hyper-heuristic throwing good results in a short amount of time can reduce its performance if it runs for longer than needed [39].

Lourenço et al. propose short learning phases for hyper-heuristics: a long learning phase would imply that the performance of a hyper-heuristic would depend on solving small instances or adopting parameters that minimize computational cost (e.g. population size or number of iterations, etc.). Adopting excessively simple conditions can compromise results, as differences between feasible and unfeasible strategies could be erased and therefore lead to an inaccurate evaluation of the proposed solutions [37].

These findings suggest that problem features and heuristic interactions are key factors to take into account when designing high-level methodologies. As problem features are difficult to extract (and are domain or even instance-specific), there has been a trend to automate this process (see [1], [29], [30], [36] and [55]). The proposed framework in this research takes into account this notion, and employs an evolutionary approach in order to deal with combinatorial problems in a more general way.

2.5 The (1+1) Evolutionary Algorithm

What is the easiest way to solve a problem? Is there a solution or method which feels *natural*? Nature has always been a source of inspiration for humans when faced by problems, and optimization is not the exception.

Pavlus presented an article on Scientific American in 2012 which included brief descriptions of some nature-inspired systems like soap bubbles and DNA structures applied to

NP-hard problems [47]. However, this idea of optimization using natural metaphors as data structures dates back from way before. Focusing on evolution and natural selection, Holland's famous *Adaptation in Natural and Artificial Systems* book in 1975 introduced the simple GA [31]. This algorithm was designed with robustness and flexibility in mind, something that exact methods and optimizers lack. The idea behind this approach is because, according to Goldberg, the real world of search is populated with discontinuities and noisy search spaces which are common characteristics of non calculus-friendly functions [25].

Later on, Droste et al. proposed a variant on the classical GA which the authors refer to as the (1+1) Evolutionary Algorithm ((1+1) EA) [18]. As the name suggests, it is a variation featuring a population of one individual which generates a single offspring. The (1+1) EA is based on mutation only (since the population is of just one individual), strongly resembling asexual reproduction of primitive cells. A formalization of the algorithm is as follows:

Algorithm 1 (1+1) Evolutionary Algorithm

1 $p_m \leftarrow 1/n$ ▷ Set probability of mutation
2 Choose randomly an initial bit string $x \in \{0,1\}^n$
3 **while** Stopping criteria not satisfied **do**
4 | Compute x' by flipping independently each bit x_i with probability p_m
5 | Replace x by x' iff $f(x') \geq f(x)$
6 **end while**

Algorithm 1 can be seen as a variant on the hill-climbing method in which the neighborhood is any point in the search space. Though convergence may seem slower using only mutation and escaping from local optima seems easier with crossover approaches [13], some modifications could be applied to enhance exploration.

Lehre and Özcan presented a variant of (1+1 EA) in [36] which chooses one of m mutation operators based on a probability distribution $\bar{p}$. This variant was tested using two mutation operators for simple problems like ONEMAX and GAPPATH. In this research, we use this approach with additional mutation operators. The whole framework, including this variant, is formally described in Chapter 4.

2.6 Summary

This chapter gave a general glimpse over the topics related to this investigation: constraint satisfaction problems, the 0/1 knapsack problem, hyper-heuristics, the algorithm selection problem and the (1+1) Evolutionary Algorithm.

The constraint satisfaction problem is a classic decision problem which may become intractable due to its high complexity. The 0/1 knapsack problem is a combinatorial optimization problem which, as well as the CSP, has many real-life applications despite its complexity. This is why both problems are important in the field of combinatorics.

A hyper-heuristic is a high-level algorithm which solves a problem by either selecting low-level heuristics or generating new algorithms from a set of operators. When deciding which operator to use for which part of the search, the hyper-heuristic faces a problem known

as the algorithm selection problem. Problem features and objective functions are used to determine which heuristic is the best suited for the subproblem at hand.

Evolutionary approaches have been widely used to tackle combinatorial problems, and also as learning models for hyper-heuristics. The (1+1) EA is an evolutionary algorithm which employs the metaphor of an individual as a set of features. The individual is cloned and then undergoes a mutation process which may alter its features. The clone and the original individual are compared using an objective function, and the best of the two is kept for the next generation. The process is repeated until a stopping criteria is met.

Chapter 3 reviews the methodology used during this research, while Chapter 4 includes the specific details of the hyper-heuristic model.

Chapter 3

Methodology

This chapter provides a detailed description of the methodological process used during this research, which is separated into three different phases and presented in chronological order.

3.1 Phase I. Understanding Background

This first phase of the methodology consisted of two branches of theoretical computer science areas: combinatorial optimization problems, their properties and how to solve them; and Hyper-heuristics—their generation and application to computationally complex problems.

Combinatorial problems usually come in the form of optimization: find the best arrangement of items according to certain criteria; but another common representation is that of a decision query: is this arrangement possible? Could an arrangement like this be found under these conditions? To exemplify these concepts, two key problems in combinatorics were considered: CSP and KP—the former being a decision problem and the latter an optimization problem.

Phase I included a vast study of the state of the art in these fields. Regarding CSP and KP, the study focused on solution methods, heuristics and their complex nature. In the hyper-heuristic domain, we studied their definition and their classification, the different models used for training and the impact of appropriately selecting the algorithm when solving an instance—the algorithm selection problem. Basic concepts of evolutionary computation were also reviewed to understand the implications of the parameter selection when training the hyper-heuristic. This information was compiled and is included in Chapter 2 so to let the reader learn the basic concepts about optimization problems, but more importantly, to emphasize that the impact of choosing the right operator at each state of the problem should be understood as the basis of a whole line of research.

3.2 Phase II. Justification of the Hyper-heuristic Approach

Since previous work on CSP has shown that combining heuristics yields good results, can one found any sequence of heuristics that works better than others? The same inquiry was brought to a different field in combinatorics: the binary Knapsack Problem.

During this phase, efficiency tests were conducted on synthetically generated 0/1 KP instances and then compared against random methods. Results were satisfactory: a hyper-heuristic method yielded, most of the time and for most of the instances, more efficient solutions than random methods. This sparked curiosity over the idea of a hyper-heuristic model for obtaining better solutions than those from greedy packing heuristics run in isolation. A set of experiments was designed in order to test if there was something valuable that the model could learn from the test instances.

3.3 Phase III. Experiments and Analysis

The experimentation phase covered two branches: KP and CSP, in order for it to cover both a decision problem and an optimization problem. This section briefly outlines the experiments carried out on each branch. Instance information is provided (if relevant), as well as a general overview of the programming framework. The hyper-heuristic framework is discussed in the next chapter.

3.3.1 Experiments on the 0/1 KP

To deal with 0/1 KP instances, the hyper-heuristic model was implemented as a separate class inside the *Hyper-heuristic Engine for Reusable Multi-domain Enhanced Search* (HERMES) framework. The implemented generator hyper-heuristic starts with a fixed-size solution of twelve blocks. Each block represents "a twelfth" of an instance, meaning that roughly, a twelfth of the items in a problem instance will be packed using the heuristic chosen for the first block, and so on. An integer division is performed according to the number of items of the instance and the number of blocks in the hyper-heuristic. The remaining items are added sequentially from end to start (i.e. backwards). An instance with 20 items and 12 blocks, for example, will have one item in each block, and an additional item. Fig. 3.1 illustrates this case.

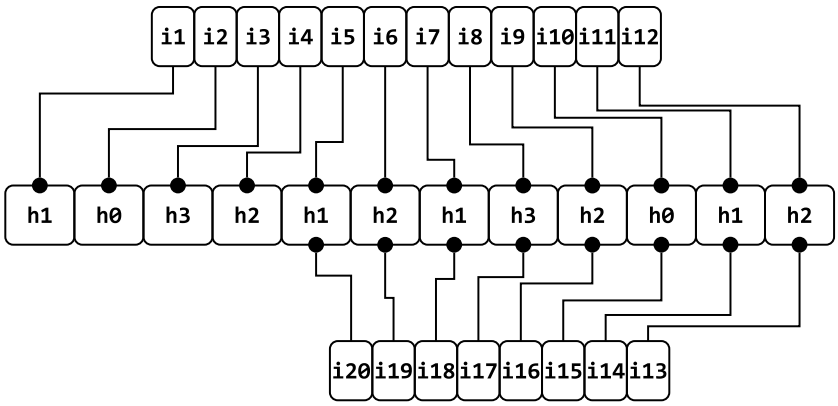

Figure 3.1: Item distribution for a 20-item knapsack instance with a 12-block long hyper-heuristic. Items are equally distributed, and remaining items are dropped one by one from end to start.

The constructive solver handles hyper-heuristics of different size, since the sequence structure could be altered during the mutation phase in the learning process. Over time, a hyper-heuristic may end-up with different distributions of items per block: some blocks could potentially contain more than one item, and sometimes a block may contain a single item. These strategies which vary the size of the hyper-heuristics over time generate more interactions between packing heuristics. The packing heuristics used in this research are those mentioned in Section 2.2.3, in Chapter 2.

The Knapsack Instances

For this research three main datasets were used, which are briefly described in Table 3.1.

Table 3.1: 0/1 Knapsack datasets used.

ID	Dataset	Features		
		Instances	Items	Capacity
SetA-Train	Synthetic	400	50	50
SetA-Test	Synthetic	400	50	50
SetB	knapPI11-16	600	20	Variable
SetC	knapPI11-16	600	50	Variable

The first dataset comprises 800 knapsack problem instances which were synthetically generated. This dataset is referred to as SETA. SETA is a balanced problem set: 25% of the dataset is best solved by each of the four packing heuristics described in Section 2.2.3. Additionally, SETA was split into a learning and a testing subset, both comprising 50% of the whole dataset, i.e. 400 problem instances each. Each instance in this dataset contains a knapsack with both capacity c and a set of items n of 50 units.

The second dataset, referred to as SETB, consists of 600 hard instances proposed by Pisinger in [49]. Instances in this dataset feature a set of items n of 20, but a different capacity each.

SETC consists on 600 additional hard instances also proposed by Pisinger. All instances in this dataset have 50 items, with different capacities from one problem instance to another.

Methodology overview for 0/1 KP

The experimentation in the KP branch was conducted in three phases: preliminary, confirmatory and complementary. Fig. 3.2 shows a summary of each phase.

Preliminary Testing

In this phase, 30 hyper-heuristics were generated, all of them starting at a fixed length of twelve blocks. Each hyper-heuristic was trained using a balanced training subset of SETA (400 instances), as mentioned previously in Section 3.3.1. In order to find an appropriate number of iterations for the training phase, we conducted 30 runs of 1000 training iterations each. For each run, a stagnation point (SP) was found. We define an SP as the iteration at

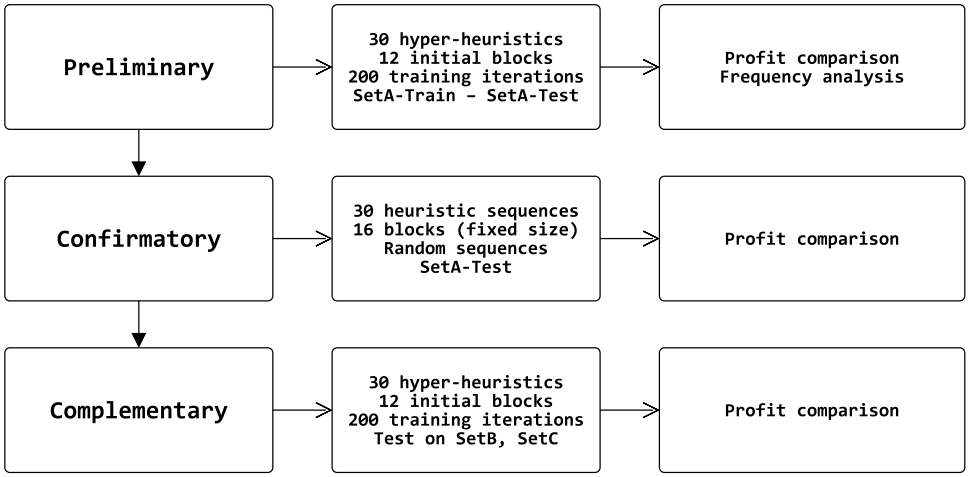

Figure 3.2: Outline of the methodology used during the 0/1 KP experiments.

which the best solution was first encountered and for which profit showed no improvement during subsequent iterations. Table 3.2 shows the stagnation points of these 30 runs.

Table 3.2: Stagnation points for the first thirty runs. No better solution could be found during subsequent iterations.

Run	SP	Run	SP	Run	SP
1	162	11	46	21	675
2	127	12	51	22	41
3	743	13	114	23	30
4	39	14	38	24	47
5	682	15	18	25	40
6	38	16	317	26	28
7	68	17	56	27	152
8	21	18	146	28	78
9	405	19	176	29	37
10	64	20	30	30	34

These stagnation points were used to calculate a simple average of the iterations. The average stagnation point found was 150.1, so we rounded down to 150 iterations. A small gap of 50 extra iterations was added so that the average convergence of training was limited to 200 iterations.

After training, the resulting hyper-heuristics were tested on SETA-TEST. Results were recorded and compared against solutions yielded by each heuristic run in isolation on the same set.

Confirmatory Testing

Additional tests were conducted to verify the consistency of the results on the first phase. 30 heuristic sequences were randomly generated. These sequences were all of the same size,

which was fixed at 16 blocks as it was the average length of the 30 hyper-heuristics in the previous phase.

The random sequences were then tested on SETA-TEST, and the results stored. A comparison was made against both low-level heuristic methods and hyper-heuristics obtained on the previous phase.

Complementary Testing

In order to test the stability of the hyper-heuristic model, additional testing was conducted using problem instances from the literature.

This phase is pretty similar to phase one: it was comprised of 30, 12-blocks long hyper-heuristics, each trained for 200 iterations as well. However, this process was repeated two times. First training with SETA-TRAIN and testing on SETB and SETC, and once again but with both training and testing done on the new datasets.

Additionally, the evolution history of the 30 hyper-heuristics obtained in the Preliminary Testing phase was used in a frequency analysis in order to check for heuristic interactions. For this, 2-block long subsequences were considered for all four packing heuristics, so that 16 different sequences were counted and compared.

Results of all three phases are presented in Chapter 5.

3.3.2 Experiments on CSP

As in the case for KP, the implemented hyper-heuristic model was set up to work with the CSP solvers and instances available in the HERMES framework. Each hyper-heuristic starts with a fixed-size, which varies from phase to phase. Back in the KP experiments, each block contained a subset of items to pack using the heuristic assigned to such block. In CSP, each block contains a subset of decision points in the search tree: pairs of variable/value decisions, so a heuristic is really a compound heuristic—one part deals with variable selection and another with value selection. As in the case for KP, decision points are distributed in the sequence using an integer division: number of items of the instance over the number of blocks in the hyper-heuristic. Remaining items are added sequentially from end to start (i.e. backwards).

In a similar manner to the previous branch, the hyper-heuristics proposed may vary in size throughout the training phase due to the presence of length-modifying mutators. It is expected that these length-modifying mutators generate big changes when solving the CSP instances, as each decision point frees up subsequent decision points that are constrained while restricting new ones on each step. Heuristic combinations used for the CSP experimentation are shown in Table 3.3:

The only value ordering heuristic used for the experimentation phase in this branch was **Min-Conflicts**, as is one of the simplest yet more powerful methods for deciding which value to assign next. Both variable and value ordering heuristics are described in Section 2.1.5, back in Chapter 2.

Table 3.3: Compound Heuristics used for the CSP experimentation.

ID	Variable selection	Value selection
0	Dom	Min-C
1	Deg	Min-C
2	Dom/Deg	Min-C
3	Wdeg	Min-C
4	Dom/Wdeg	Min-C

The CSP instances

For this research, instances from multiple datasets from the literature were used to train and test the resulting hyper-heuristics [35]. The datasets used are presented in Table 3.4.

Table 3.4: Datasets used for CSP experimentation. Number of domains, variables and constraints are presented as $|D|, |X|$ and $|C|$ respectively.

Dataset	Instances	Relations	$\|D\|$	$\|X\|$	$\|C\|$
QCP-10a	12	12	11	100	900
QCP-10b	3	11	10	100	900
QCP-15	15	17	16	225	3150
EHI-85	100	(+80)	1	297	(+4120)

The acronym *QCP* (which can be seen in the name of the datasets) refer to the **Quasigroup Completion Problem**. A quasigroup, which is also known as Latin Square, is defined as an $n \times n$ multiplication in which each row and each column is a permutation of the n symbols used. The QCP consists on determining if the $n^2 - p$ empty cells can be filled to obtain a complete Latin Square, and is an NP-Complete problem [26].

As described in Table 3.4, the QCP-10 dataset consists of fifteen instances of 900 constraints. Each of the 100 variables in the instance have one of 11 (or 10 for some cases) domains: the set of available values for the variables in the problem. Instances in QCP-15 contain, on the other hand, over 3000 constraints and 200 variables, showing a sharp increase in complexity from the QCP-10 set.

The EHI-85 instance set (which name derives from *Exceptionally Hard Instances*) features 100 instances which are all unsolvable. Though the number of relations and constraints vary from one instance to another, EHI-85 problems feature over 4000 constraints across almost 300 variables. These instances were originally 3-SAT problems, which can also be formulated as CSPs (see Section 2.1.1 back in Chapter 2.)

Methodology overview for CSP

As in the KP branch, the experimentation on the CSP focuses on two main analyses: learning efficiency and heuristic interaction. Since CSP instances studied in this research are decision problems which may have multiple solutions, the comparison was made using both

consistency checks (CC) and milliseconds. Tests were run on a 64-bit Windows 10 personal computer, with 8GB of RAM and an Intel® Core™ i5-2500K CPU @ 3.30GHz processor.

The three-phase approach used for KP had to change in order to tackle CSP, as it is a much more complex problem to handle. The learning efficiency analysis consisted in two sequential phases along a parameter adjustment approach: preliminary testing and confirmatory testing. Fig. 3.3 shows a summary of each phase.

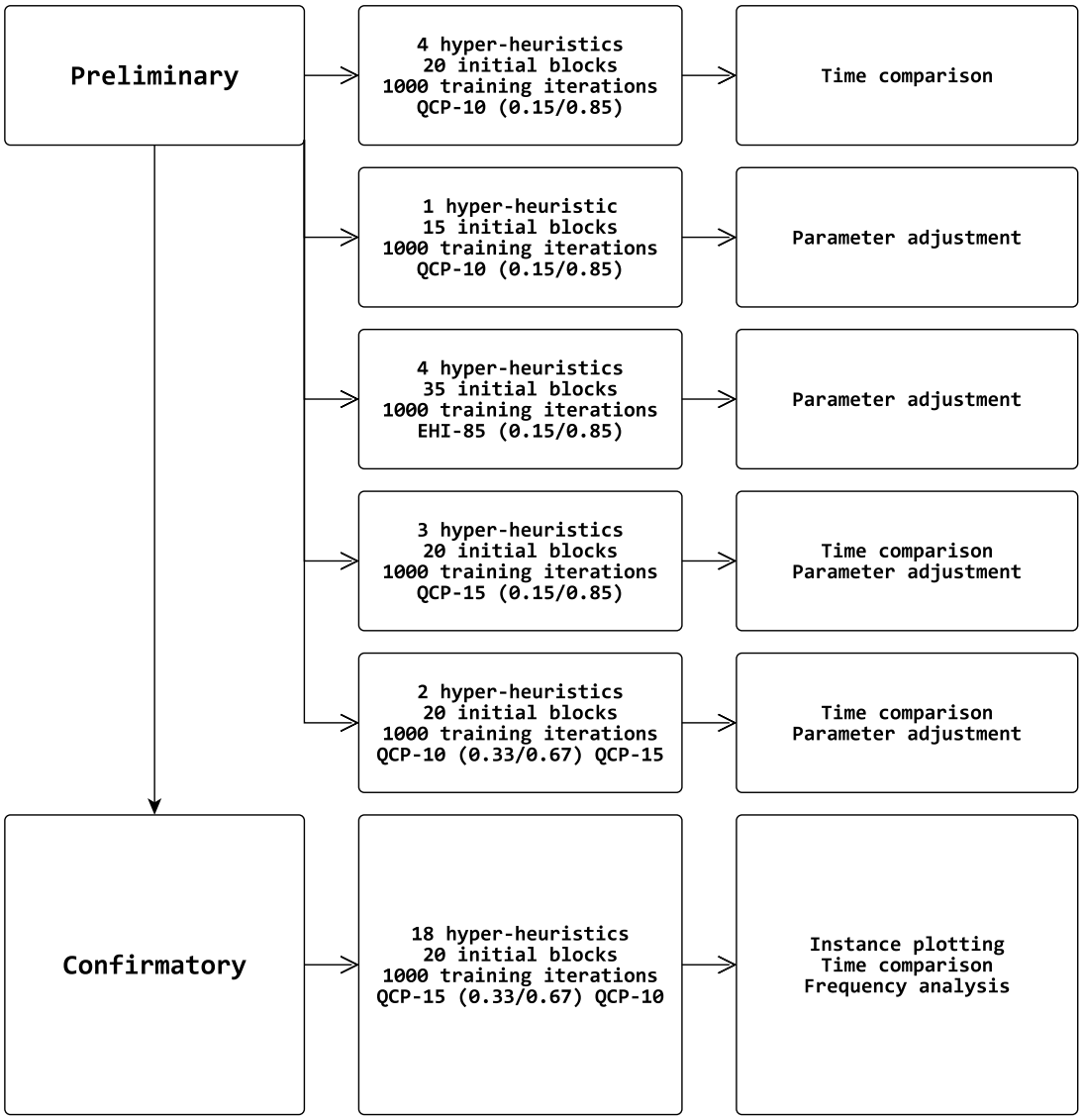

Figure 3.3: Methodology outline for the CSP experiments.

Preliminary Testing

This phase can be broken down into 5 sub-phases, each with its own hyper-heuristic initial length and training parameters. It is important to note that parameter selection for experimenting with CSP instances could easily become a line of research on its own, as some instances

may be quite diverse and highly complex. Nevertheless, many setups were considered until finding a balanced point between training time and solution quality.

The first sub-phase featured four runs of 20 initial blocks each. The model was trained using 15% of the QCP-10 dataset, and tested on the remaining 85% of the same instance set. These results were used for time comparison. Results shown that some parameters were potentially off, so the length of the initial hyper-heuristic was modified. This modification corresponds to the second sub-phase, in which one single run was conducted using 15-block sequences.

As the resulting hyper-heuristic yielded results far from those expected, it seemed that reducing the length of the sequence was detrimental for the learning process for that dataset. The third sub-phase was conducted using 35-block sequences (up from 15), running four times using 15% of the EHI-85 dataset as the training set and the remaining 85% as the testing set.

The increase in length resulted in a much slower convergence, as the search space grew exponentially, so the length of heuristic sequences was reduced to 20 blocks again during the fourth sub-phase. The training consisted in 15% of the dataset, but this time QCP-15 instances were used instead. The testing was conducted on the same dataset for this sub-phase.

The final sub-phase of the preliminary testing tried a different learning/testing approach, as it kept the same running parameters than the previous sub-phase (20-block hyper-heuristics, 1000 training iterations) but featured a learning phase on one third of the instances available on QCP-10. Testing was performed on QCP-15.

Confirmatory Testing

With a wide variety in the results of the previous phase, it was difficult to settle with a single best way to tackle on the learning problem in CSPs. However, some decisions were made taking into the account many features and findings on the previous phase. 20-block sequences seemed appropriate, while the partition of the training and testing subsets needed a tweak. Again, one third of a dataset was used to train a single hyper-heuristic, however the datasets were exchanged: training was now conducted on QCP-15 and testing was done on QCP-10. This change was necessary, as QCP-15 presents lots of instances which are far more complex than those in QCP-10, so we expected better results if using hard instances for the learning process.

An additional phase was conducted during the confirmatory testing, which was preparing the input data for the frequency analysis of the heuristic sequences. As well as in the KP branch, simple 2-digit sequences were analyzed using the evolution history of all 18 hyper-heuristics on this phase.

The results of these two experimentation phases are presented in Chapter 6. It is important to note that though 32 runs is a decent number of observations, additional sampling would be needed to fully determine performance issues and to propose improvements for the learning model. For a more detailed description of proposed future work, see Chapter 7.

Run Representation

Since the preliminary phase contained many different parameter adjustments, minute changes between each sub-phase were difficult to locate and reference. For this reason, each run was labeled with a human-readable label depending on the seed used to generate the subset of training instances. Table 3.5 show the hyper-heuristic method and their respective IDs.

Table 3.5: Hyper-heuristic IDs and labels.

ID	Seed	Label	ID	Seed	Label
1	1101813	Surprised Man	17	41529	BLSting
2	18523	Tower	18	75263	BRSting
3	71593	X	19	18523	Tower
4	74123	BLCorner	20	74852	q
5	74123	BLCorner	21	85296	p
6	74123	BLCorner	22	748263	Bracket
7	71593	X	23	48526	Cross
8	18523	Tower	24	71593	X
9	12963	BRCorner	25	74182963	Square
10	48526	Cross	26	85263	b
11	718293	2OHBars	27	7415963	H
12	6666	Beast	28	741963	2OVBars
13	415263	2HBars	29	741963	2OVBars
14	741852	2VBars	30	745296	UpTrident
15	18596	TRSting	31	41852	d
16	74853	TLSting	32	418563	DownTrident

A visual representation of the seed was needed in order to understand what happened and at which level of the experimentation process. For this, a "run chart" was generated, which can be reviewed in Appendix B, along with an explanation on how to read it.

3.4 Summary

This chapter reviewed the methodology used throughout this research, which can be broadly separated into three phases. Phase 1 considered the background and theoretical framework construction. Phase 2 consisted in generating the research questions which fueled the investigation and conformed its justification. Phase 3 comprised many experiments in two different branches: KP and CSP. The KP branch of the experimentation phase consisted in three main sub-phases—preliminary, complementary and confirmatory testing—where the both the learning efficiency of the model and heuristic interaction were analyzed. The CSP branch of the experimentation consisted in 32 experiments distributed in two phases: preliminary and confirmatory. Preliminary results considered five sub-phases, many of which were used for parameter adjustment. The confirmatory sub-phase comprised 18 hyper-heuristics and was used for both learning efficiency analysis as well as the heuristic interaction frequency analysis. KP and CSP results can be found in Chapters 5 and 6 respectively.

Chapter 4

Framework Description

In this chapter, the hyper-heuristic learning model is described as well as the rationale behind it. An extensive description of the mutation operators used by the model is presented along with graphical examples on how the algorithm works.

4.1 Hyper-heuristic Model

The hyper-heuristic model used in this research is a selection hyper-heuristic with a learning mechanism based on a variant of the (1+1) Evolutionary Algorithm originally proposed by Droste et al. in [18]. The (1+1) algorithm works by generating a random individual—a set of features—which is cloned and then mutated over time. In this sense, the hyper-heuristic is an ordered set of features—a sequence of heuristics. An abstract representation of an individual can be found in Fig. 4.1, which shows a sequence of heuristics, each enclosed in a block.

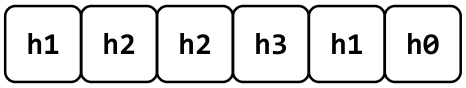

Figure 4.1: Abstract representation of an individual as a set of heuristics. Each feature is enclosed in a block.

The mutated clone is compared against its parent and the model preserves the best of the two individuals. If the two individuals are equally fit (objective function-wise), then the mutated clone is preferred. Mutation in the (1+1) EA algorithm uses a single mutation operator which flips each feature with a probability of $1/n$, where n is the number of heuristics in an individual.

The modified version used in this researched was first presented by Lehre and Özcan in [36]. This variant chooses one of m mutation operators based on a probability distribution $\bar{p}$. The pseudo-code of this learning mechanism is presented in Algorithm 2, and though it is shown as a maximization problem, the algorithm is able to operate on minimization problems as well.

Algorithm 2 Learning phase of the Hyper-heuristic

$s \sim \text{UNIF}(\{0, 1, \ldots, h\}^n)$
while termination criteria is not satisfied **do**
    op $\sim D_{\bar{p}}(\text{OP}_1, \text{OP}_2, \ldots, \text{OP}_m)$
    $s' \leftarrow \text{op}(s)$
    if $f(s') \geq f(s)$ **then**
        $s \leftarrow s'$
    end if
end while

As described in Section 2.5 in Chapter 2, this model works in a similar fashion to the well-known hill climbing technique, exploring the surrounding search space looking for a better solution but with an additional stochastic component added during the mutation phase. If the *fitness* of the new proposed solution is greater or equal (for maximization, and less or equal for minimization) than that of the previous one, then the solution is replaced and the searching process continues.

A UML sequence diagram of the model is presented in Fig. 4.2. The hyper-heuristic component interacts with the heuristic space to select a mutation operator which is then applied to the proposed solution. The solution is tested on the problem set (solution space) and a fitness level is obtained. Any hyper-heuristic which is not worse than the current candidate is accepted, and the process starts anew until a number of iterations or a stopping criteria is met.

A brief explanation on how to interpret a UML sequence diagram can be found in Appendix A.

4.2 Mutation Operators

In order to exploit the proposed solution, an individual must undergo through a mutation process. This mutation process alters the structure of the individual and changes its features, so that a new set of features is generated and evaluated. For the model used in this dissertation different mutation operators were used. Mutators can be grouped into different classes according to their effects. Mainly, three broad groups were proposed:

- **Length modifiers**. Those mutators that add or remove blocks to the set of heuristics.
- **Feature modifiers**. Mutators which replace the heuristic in a block by another heuristic.
- **Neighborhood modifiers** Mutators affected by adjacent blocks.

These groups are not orthogonal, since there are mutation operators which modify the length of the set of heuristics according to their neighbors, for example. Eight mutators were used in this research, and are described throughout this section.

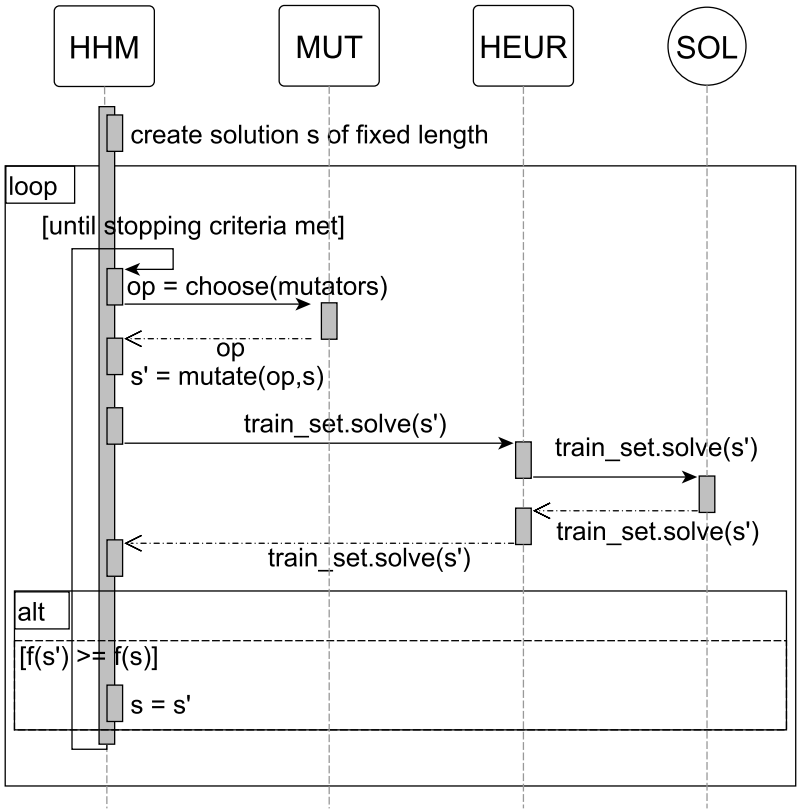

Figure 4.2: UML sequence diagram of the proposed Hyper-heuristic model. HHM stands for Hyper-heuristic model, MUT for Mutators, HEUR for Heuristics and SOL for Solution space.

4.2.1 Length Modifiers

Length modifiers are perhaps the most disruptive of the mutation operators, since they alter the whole feature set by inserting or removing blocks at random positions.

- ADDBLOCK. This operator inserts a random feature at a randomly selected position.
- REMOVEBLOCK. This operator selects a random block and removes it from the hyper-heuristic.

Fig. 4.3 and 4.4 illustrate the behavior of ADDBLOCK and REMOVEBLOCK respectively.

4.2.2 Feature Modifiers

Feature mutators work by altering the blocks of a hyper-heuristic, directly changing the values instead of its structure.

- FLIPONEBLOCK. This operator takes a randomly selected block and assigns a new random heuristic from those heuristics available. There is a 1 out of h probability of

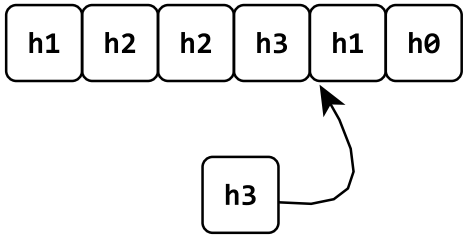

Figure 4.3: The ADDBLOCK mutation operator adds a block with a random feature at a randomly selected position, increasing the length of the hyper-heuristic sequence.

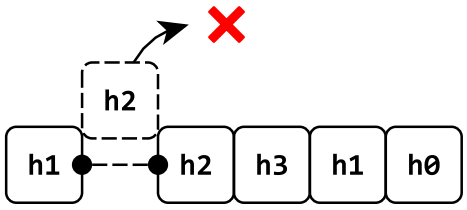

Figure 4.4: The REMOVEBLOCK mutator randomly selects a block in the hyper-heuristic and removes it, reducing its length.

the feature being the same as it was before, resulting in an unchanged sequence of heuristics.

- FLIPTWOBLOCKS. It works in the same fashion as FLIPONEBLOCK but choosing two blocks at random, one after the other. Because of this, there is a small chance of a block being overwritten twice, and even a smaller chance of an unchanged individual.
- SWAPBLOCKS. This operator swaps the values for two randomly selected blocks. Again, there is a possibility that the two selected blocks share the same value, so that swapping their positions has no effect.

As there may be situations where an individual remains unchanged after the mutation phase, these operators are considered less disruptive than length-modifying mutators. Fig. 4.5, 4.6 and 4.7 illustrate the process of FLIPONEBLOCK, FLIPTWOBLOCKS and SWAPBLOCKS respectively.

4.2.3 Neighborhood Modifiers

The neighborhood mutator group comprises neighborhood variants of previously described mutators. The neighborhood of a block is comprised by the two adjacent blocks. The hyper-heuristic sequences in this dissertation are considered circular when applying neighborhood operators. This means that if the block chosen to be modified corresponds to an end of the sequence, the other end is taken as its neighbor.

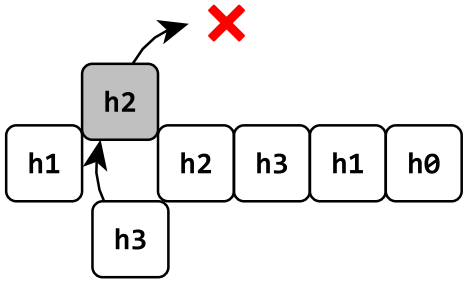

Figure 4.5: The FLIPONEBLOCK mutator changes the value of a block. There is a chance of the new value being the same as it was before.

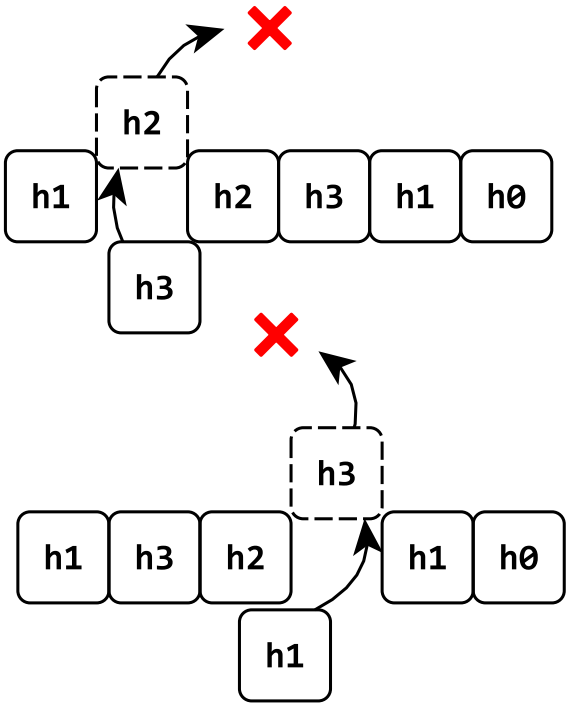

Figure 4.6: The FLIPTWOBLOCKS mutation operator. This mutator applies FLIPONEBLOCK twice. Though the possibility of an unchanged individual exists, it is not that common.

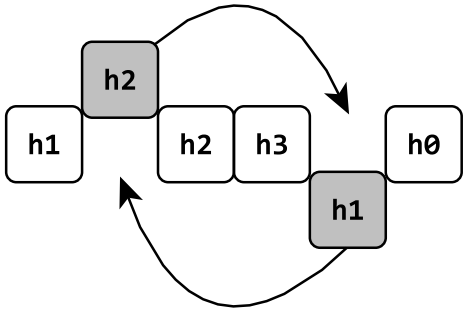

Figure 4.7: The SWAPBLOCKS mutator randomly selects two blocks and then swaps their values. Depending on the distribution of the features in an individual, there may be a high chance of an unchanged hyper-heuristic.

- ADDBLOCKNEIGH. This mutator selects a random location in the hyper-heuristic and inserts a block based on its neighborhood, i.e, the available heuristics to choose from are decided by the values of the neighbors.
- FLIPONEBLOCKNEIGH. This is a neighborhood variant of FLIPONEBLOCK, in which the randomly chosen block is replaced by one of its neighbors.
- FLIPTWOBLOCKSNEIGH. This mutation operator acts like FLIPTWOBLOCKS, but heuristics are chosen randomly from the neighborhood of each selected block.

Assuming that blocks that are in the same neighborhood represent points in the search space which are close from each other, mutation from neighborhood operators is expected to be less disruptive for the individual. Again, as in the feature-modifying mutators, some neighborhood operators have a small chance of not altering the hyper-heuristic. Fig. 4.8 shows how the ADDBLOCKNEIGH operator works. Fig. 4.9 and 4.10 present the mutation processes of FLIPONEBLOCKNEIGH and FLIPTWOBLOCKSNEIGH respectively. Fig. 4.11 illustrates the idea of a circular hyper-heuristic.

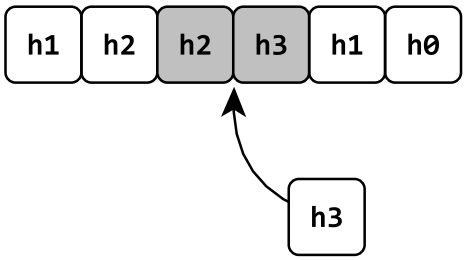

Figure 4.8: The ADDBLOCKNEIGH operator inserts a block at a randomly chosen position. The pool of available values for the new block is limited to the values of the adjacent blocks.

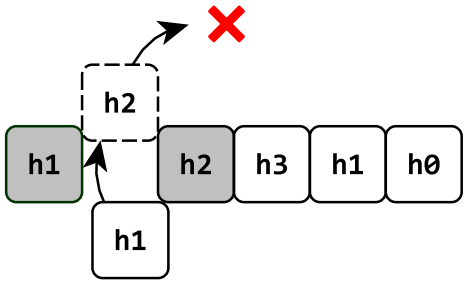

Figure 4.9: A FLIPONEBLOCKNEIGH example.

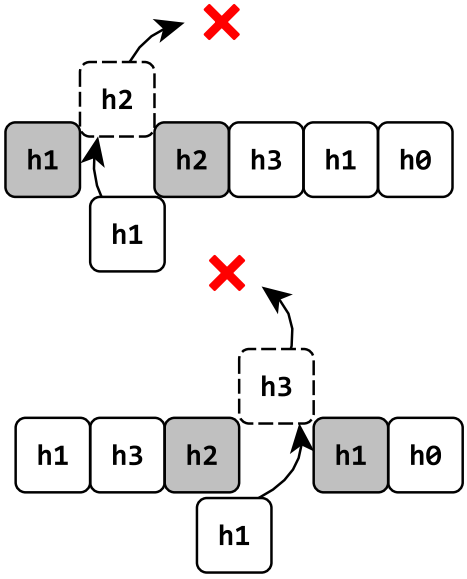

Figure 4.10: An example of the FLIPTWOBLOCKSNEIGH process.

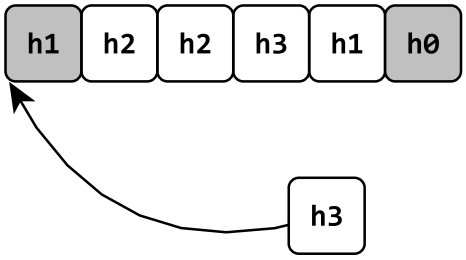

Figure 4.11: A case of a circular sequence for the FLIPONEBLOCKNEIGH process. Notice how the neighborhood include both ends of the hyper-heuristic.

At each iteration, one of these operators is randomly selected and applied to the candidate solution during the mutation phase. This mutation process may result in a different sequence of features, which in this study represents a sequence of heuristics to apply to the problem instance being solved. Thus, each block can be seen as a subproblem where the chosen heuristic h will be applied.

Both the FLIPONEBLOCK and FLIPTWOBLOCKS operators were previously used in [36]. The use of ADDBLOCK, SWAPBLOCKS and REMOVEBLOCKS can be found in [28], [29] and [54]. Having a wide range of mutation operators on different disruptive levels adds several mechanics to the hyper-heuristic for avoiding stagnation on local optima during the learning phase.

4.3 Summary

In this chapter we have explained the hyper-heuristic model, its inspiration and its learning process. The hyper-heuristic learning model was based on the (1+1) Evolutionary Algorithm, which clones an individual (a sequence of blocks, each containing a feature) and mutates it. If the proposed solution (mutated individual) performs better than its predecessor, then it is kept for the next generation. The process is then repeated until a stopping criteria is met.

The mutation phase of the model is accelerated using multiple mutation operators. These mutators can be broadly classified into three different groups, according to how much they alter the solution: length-modifying, feature-modifying and neighborhood mutators. Modifying the structure of the hyper-heuristic is more disruptive than altering their features, which in turn is more disruptive than modifying a block depending on the adjacent features. Graphical examples of the mutation operators can be found throughout the chapter.

Chapter 5 details the results of the experimentation on KP. Results on CSP are included in Chapter 6. Additional implementation details and framework modifications are included in the corresponding chapters.

Chapter 5

0/1 KP Analysis

This chapter discusses the results of the KP branch of the experimentation phase. In general, two aspects of the hyper-heuristic method proposed were analyzed: the efficiency of the learning method and simple interaction between packing operators. Section 5.1, 5.2 and 5.3 detail the results of the learning method analysis, while Section 5.4 describes heuristic interaction and sequence behavior.

5.1 Phase I. Performance against Traditional Packing Operators

A summary of the profit of the simple packing operators on SETA-TEST, as well as the profit of the hyper-heuristic methods, is presented in Table 5.1. Three hyper-heuristic methods were considered: the best, median and worst cases, which are referred to as Best-HH, Med-HH and Worst-HH respectively. Additionally the number of instances per rank, per method is reported. The Rank column shows the standing of a given method against the performance of all other methods tested in isolation on SETA-TEST. This ranking considers nine ranks (from R1 to R5, using decimal ranks to describe a tie for a given rank) and compares five methods: four low-level packing operators against the best hyper-heuristic method, and each of the hyper-heuristic cases versus all low-level operators.

A quick glance to the table reveals that the best hyper-heuristic method seems to be no significantly better than any of the low-level operators. In fact, the total profit of the best hyper-heuristic is exactly the same as the best single heuristic shown in Table 5.1, MAX-PW. This behavior shows that the hyper-heuristic method was able to learn from the best traditional operator available using the training instances, but could not find any improvement over the best traditional operator solution.

In order to understand these results, additional experiments were carried out. For each instance on the dataset, a ranking was obtained. The performance ladder including the whole SETA-TEST is also presented in Table 5.1. The ranking exhibits that the best hyper-heuristic method imitated the MAX-PW behavior. This may be due to the fact that MAX-PW operates appropriately even in cases when it is not the best heuristic for a given instance. However, looking at the median hyper-heuristic method column reveals that this high-level method obtained a better result in 7 out of the 400 instances available in the set. Therefore, it is possible

Table 5.1: Summary of experiments for Phase 1a. The method with the highest Total Profit is highlighted. Non-integer ranks refer to a tie for a given rank, e.g. R1.5 refers to a tie for Rank 1.

	Low-level heuristics				Hyper-heuristics		
Rank	Def	Max-P	Max-PW	Min-W	Best-HH	Med-HH	Worst-HH
R1	86	100	0	100	0	7	0
R1.5	0	0	111	0	111	103	109
R2	2	1	2	0	2	25	13
R2.5	1	0	285	1	285	258	274
R3	11	21	1	79	1	6	3
R3.5	0	1	1	1	1	1	1
R4	18	195	0	183	0	0	0
R4.5	0	0	0	0	0	0	0
R5	282	82	0	36	0	0	0
Total Profit	241081	343872	**467145**	376148	**467145**	467103	466711

to beat MAX-PW under some circumstances.

To explore different scenarios, this experiment was repeated four additional times using a random sample for the training phase: twice using 60% of the training set, once with 30% and finally once 15%. The test set remained intact.

The results confirmed the notion that the hyper-heuristic model can beat the best simple heuristic under some circumstances. On the second run, using only a random sample of 60% of the original training set, the total profit of the hyper-heuristic was higher that that of the MAX-PW operator. The results of the second run can be observed in Table 5.2, where the total profit and the corresponding rankings per method are reported. Additionally, Fig. 5.1 illustrates the performance of the best hyper-heuristic compared against the best and worst methods available for each problem instance on SETA-TEST.

As the results of the run for Phase 1b suggest, balancing the training set seems to be detrimental for the efficiency of the model, since it is a sequence based-method that accepts any solution which is not worse than the actual state. As shown in Table 5.2, the total profit of the best hyper-heuristic model for this run was better than that obtained by MAX-PW on Table 5.1. Although slight, improvement over a conventional method seems to be achievable when using the hyper-heuristic approach. This improvement is very likely to rise if solving large number of problem instances, or by adding more operators to choose from.

Since there exists room for improvement over the best heuristic analyzed, we decided to test further. The experiments were repeated as described in Section 3.3.1, back in Chapter 4: 30 instances of 12 initial blocks, but removing MAX-PW from the available operators to choose from during training. These results are shown in Table 5.3 and in a boxplot available in Fig. 5.2. Each box in the boxplot represents a series. Bold colored lines represent the statistic mean of each sample. Crosses are used to represent *outliers*: data between 1.5 and 3 times the interquartile range (IQR). The IQR is the actual height of the box. A circle is used

Table 5.2: Summary of experiments for Phase 1b. The method with the highest Total Profit is highlighted. As opposed to Table 5.1, these hyper-heuristics were trained using a random sample of 60% of the training set.

	Low-level heuristics				Hyper-heuristics		
Rank	Def	Max-P	Max-PW	Min-W	Best-HH	Med-HH	Worst-HH
R1	86	100	5	95	0	0	0
R1.5	0	0	106	5	111	111	109
R2	2	1	2	0	7	2	13
R2.5	1	0	280	1	280	285	274
R3	11	21	6	79	1	1	3
R3.5	0	1	1	1	1	1	1
R4	18	195	0	183	0	0	0
R4.5	0	0	0	0	0	0	0
R5	282	82	0	36	0	0	0
Total Profit	241081	343872	467145	376148	**467182**	467145	466711

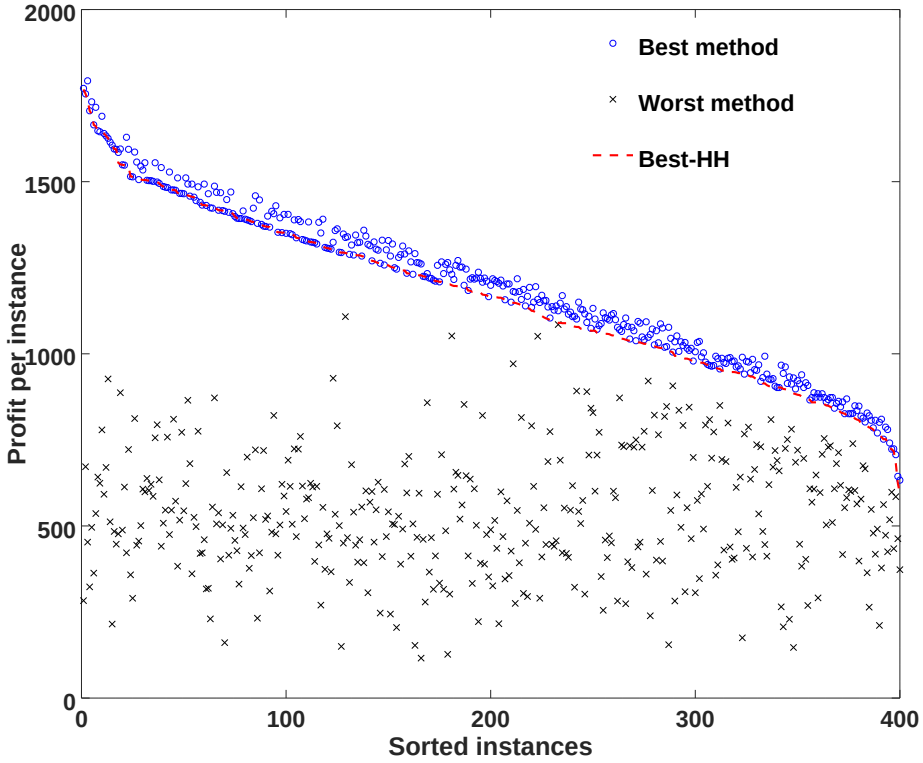

Figure 5.1: Performance of the best hyper-heuristic method, compared against oracle and worst methods available for each problem instance. These results correspond to Phase 1b, i.e. training with 60% of SETA-TRAIN and testing on all SETA-TEST.

to denote an abnormally large or small datum, lying outside the range described for crosses.

Table 5.3: Summary of experiments for Phase 1c. The method with the highest Total Profit is highlighted. The MAX-PW operator was not available during the learning during phase for this run.

	Low-level heuristics				Hyper-heuristics		
Rank	Def	Max-P	Max-PW	Min-W	Best-HH	Med-HH	Worst-HH
R1	85	99	111	97	0	1	1
R1.5	3	1	2	3	5	5	3
R2	12	1	238	2	129	121	116
R2.5	0	0	13	1	12	13	11
R3	1	40	36	105	187	207	210
R3.5	0	1	0	17	16	4	7
R4	17	178	0	141	37	41	44
R4.5	0	1	0	9	10	3	3
R5	282	79	0	25	4	5	5
Total Profit	241081	343872	**467145**	376148	397373	395586	391940

Comparing the total profit of the best hyper-heuristic method in Table 5.3 against the total profit of MAX-PW, it is seen that there was some profit loss due to the absence of the MAX-PW heuristic on the set of available heuristics to choose from during the learning phase. However, all three hyper-heuristics (best, median and worst cases) obtained second rank in terms of total profit. When comparing the results of the best hyper-heuristic case against those obtained from the available operators, the profit rises sharply, showing an improvement of nearly 6%, 16% and over 64% over the MIN-W, MAX-P and DEF operators respectively. This result exhibits that the model is capable of learning in harsh conditions, and thus obtain better results when no appropriate heuristic is known.

5.2 Phase II. Performance against Random Sequences of Operators

The second Phase of the experiments looks to answer a common interrogative: is learning really worthwhile in this problem? For this, SETA-TEST was solved using 30 randomly generated heuristic sequences. Each heuristic sequence was 16-blocks long, and for each block, a low-level operator was uniformly chosen at random. Again, three scenarios were considered—best, median and worst cases—and compared against the four heuristics defined. Table 5.4 presents a summary of the results of this experimentation phase.

As Table 5.4 shows, the best of the thirty random sequences did not beat the best of the low-level operators. There is actually a difference of about 7% of the profit obtained by the best random sequence. Although this difference may seem small, it can be considered a ‘lucky hit’. Looking at the median random sequence instead, shows a difference that rises

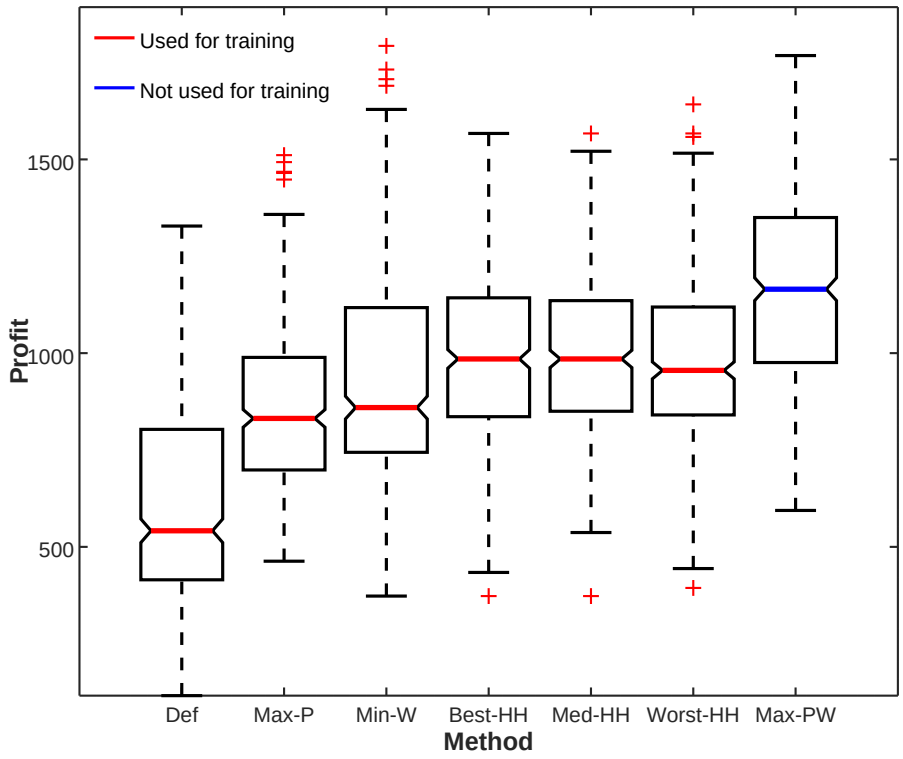

Figure 5.2: Boxplot of Phase 1c. The median of the best, expected and worst cases of the hyper-heuristic outperformed all heuristics used for its training. However, it could not obtain better results than MAX-PW, which median is shown in blue.

Table 5.4: Summary of experiments for Phase 2. The method with the highest Total Profit is highlighted. Best, median and worst cases of the random heuristic sequences are reported in the Best-Rnd, Med-Rnd and Worst-Rnd columns respectively.

	Low-level heuristics				Hyper-heuristics		
Rank	Def	Max-P	Max-PW	Min-W	Best-Rnd	Med-Rnd	Worst-Rnd
R1	80	100	102	98	1	0	0
R1.5	5	0	9	2	16	20	84
R2	6	1	127	0	114	66	2
R2.5	1	0	140	2	139	6	14
R3	8	25	21	84	118	140	0
R3.5	0	1	1	1	1	77	18
R4	18	191	0	177	11	72	0
R4.5	0	0	0	0	0	16	282
R5	282	82	0	36	0	3	0
Total Profit	241081	343872	**467145**	376148	436393	361284	241025

considerably to nearly 30%. The worst case exhibits a profit difference of almost 94%. And since the median result holds vital importance—for its closeness to the expected behavior when solving the problem completely at random—it then seems worthwhile to go with a learning method for this problem.

5.3 Phase III. Performance on Hard Problem Instances

Phase 3 of the experimentation focused on solving hard KP instances. As opposed to Phase 1, instances on this phase feature a variable capacity. Results for Phase 3a are presented in Table 5.5.

Table 5.5: Summary of experiments for Phase 3a. The method with the highest Total Profit is highlighted. All 30 runs in this phase trained on SETA-TRAIN and tested on SETB.

	Low-level heuristics				Hyper-heuristics		
Rank	Def	Max-P	Max-PW	Min-W	Best-HH	Med-HH	Worst-HH
R1	39	143	50	11	77	84	62
R1.5	20	30	119	12	103	60	47
R2	77	68	167	106	166	116	101
R2.5	19	8	93	40	62	58	62
R3	88	53	58	66	76	150	148
R3.5	18	5	64	29	36	23	34
R4	266	43	28	90	52	94	121
R4.5	13	7	15	17	8	3	5
R5	60	243	6	229	20	12	20
Total Profit	3804271	3724588	**4039708**	3867345	4031981	3958061	3905734

The model was trained using SETA-TRAIN for this phase, and was later tested on SETB. The overall profit of the hyper-heuristic ranked second in all three cases: best, median and worst cases were all behind MAX-PW. Since the training was handled in the same manner as in Phase 1a and results were somewhat similar, an additional phase was conducted.

Phase 3b comprises another 30 runs, but this time training was performed on 60% of SETB. Resulting hyper-heuristics were then tested on the remaining 40% of SETB. The results for this phase are shown in Table 5.6. Fig. 5.3 illustrates the performance of the best hyper-heuristic per instance, compared against both the best and worst available methods for these problem instances.

As opposed to the results of Phase 1b, Fig. 5.3 shows a sharp reduction of the gap between the oracle (best method available for a single instance, marked with blue circle) and the best hyper-heuristic method (marked with a red dotted line). This behavior is also present when facing problem instances of different number of items, as shown in the next phase.

To confirm the results obtained using the hard problem instances of 20 items from Pisinger, an additional phase was conducted. This phase, which we refer to as Phase 3c,

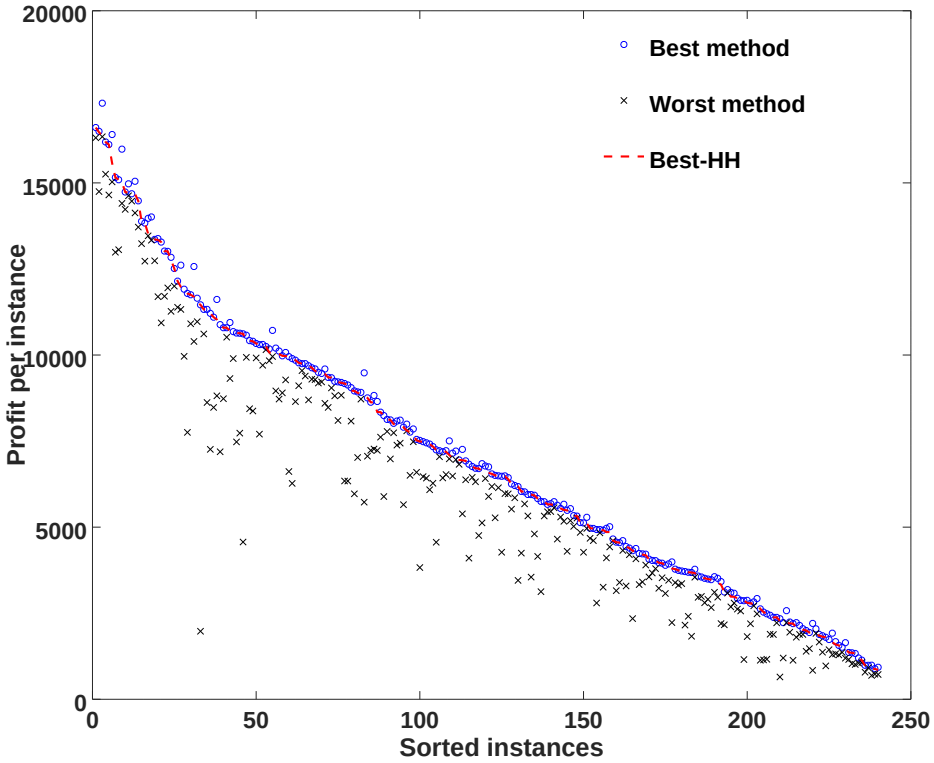

Figure 5.3: Performance of the best hyper-heuristic method per instance, compared against oracle and worst methods available.

Table 5.6: Summary of experiments for Phase 3b. The method with the highest Total Profit is highlighted. As opposed to Phase 3a, hyper-heuristics were trained and tested using hard problem instances from SETB.

	Low-level heuristics				Hyper-heuristics		
Rank	Def	Max-P	Max-PW	Min-W	Best-HH	Med-HH	Worst-HH
R1	16	54	4	3	47	35	48
R1.5	3	4	57	9	53	55	49
R2	27	34	57	36	63	77	62
R2.5	7	3	48	20	26	27	18
R3	43	20	17	20	16	14	19
R3.5	11	4	36	14	19	19	17
R4	105	17	13	31	14	10	22
R4.5	6	4	7	10	1	1	1
R5	22	100	1	97	1	2	4
Total Profit	1584007	1540100	1673387	1600015	**1681403**	1681127	1678375

encompassed 30 additional runs but both training and testing were conducted on SETC. The results for this phase can be observed in Table 5.7. Fig. 5.4 presents the performance of the best hyper-heuristic method, the oracle and the worst method available for each problem instance in SETC.

Table 5.7: Summary of experiments for Phase 3c. The method with the highest Total Profit is highlighted. As opposed to Phase 3a and 3b, hyper-heuristics were trained and tested using hard problem instances from SETC.

	Low-level heuristics				Hyper-heuristics		
Rank	Def	Max-P	Max-PW	Min-W	Best-HH	Med-HH	Worst-HH
R1	11	24	4	15	23	18	24
R1.5	1	5	97	3	94	96	83
R2	18	30	68	39	72	76	76
R2.5	1	1	36	10	26	27	28
R3	31	32	8	49	6	5	6
R3.5	2	2	14	6	10	11	11
R4	143	35	6	29	3	1	5
R4.5	1	1	6	0	6	6	7
R5	32	110	1	89	0	0	0
Total Profit	3752072	3631322	4046715	3930092	**4052501**	4052039	4045794

Phase 3c confirmed the results of the previous phases: the learning method seems to be quite stable. Despite the fact that the set comprised instances with different features, all three cases of the hyper-heuristic beat the best operator (MAX-PW) in isolation. Additionally, setting a good training set seems to impact the efficiency of the hyper-heuristic model. Training done on SETA-TRAIN seemed to negatively affect the results, as seen on both Phases 1a and 3a. It is important to note that SETA—both train and testing subsets—were balanced and synthetically made: 25% of the problem instances were designed to maximize a single low-level heuristic. This pattern was repeated for all low levels heuristic throughout this set. On the other hand, hard problem instances from SETB and SETC were randomly sampled (without replacement) using three different seeds. This training scheme is more representative of real-life applications, where often no balanced or ideal conditions are met.

5.4 Analysis of Packing Operators Sequences

This section focuses on the analysis of heuristic sequences for solving the binary Knapsack Problem. Data was collected from the results of the first 30 hyper-heuristics from the Preliminary Testing phase, trained for 200 iterations on SETA-TRAIN.

The objective of this small experiment was to try to understand what was happening behind the evolutionary curtain of the hyper-heuristic. We analyzed the frequency of appearance of simple heuristic sequences during the evolution of each hyper-heuristic.

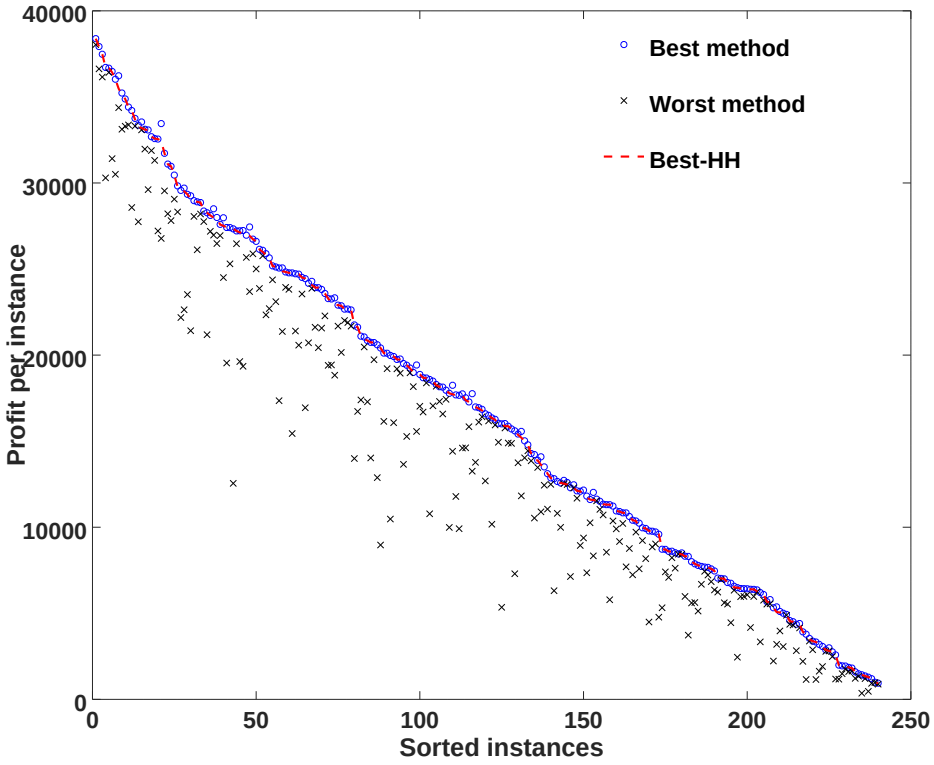

Figure 5.4: Performance of the best hyper-heuristic method per instance, compared against oracle and worst methods available. These problem instances belong to SETC, which are all hard instances of 50 items each.

At each iteration training process, the new proposed individual was stored along its profit in a comma separated value. For each run (200 iterations per hyper-heuristic), the frequency of all two-segment heuristic sequences was recorded. Table 5.8 shows the data obtained during this phase.

Table 5.8: Frequency of two-segment low-level heuristic sequences for the Preliminary Testing phase on KP

Run/Sequence	00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33
1	36	24	32	44	73	171	104	84	35	237	**259**	56	26	38	41	115
2	106	79	72	136	127	65	126	101	124	170	**752**	330	36	218	302	250
3	120	63	26	139	86	160	148	97	117	248	**372**	8	52	103	154	149
4	65	41	69	68	118	83	58	39	83	25	**492**	247	37	196	130	110
5	308	139	63	49	157	87	55	31	135	91	**614**	28	53	14	64	40
6	96	101	58	89	120	187	89	129	70	222	**483**	92	58	140	89	142
7	93	115	55	14	65	278	131	175	148	24	**508**	312	25	214	219	346
8	189	33	176	27	93	82	176	28	102	236	**588**	186	105	28	99	83
9	18	55	31	55	44	260	176	175	77	261	**535**	112	15	113	96	136
10	161	103	108	48	107	92	74	28	154	90	**827**	165	82	105	36	117
11	39	51	4	21	47	210	89	165	8	278	**322**	10	52	93	30	41
12	192	46	164	28	112	111	86	65	182	179	**526**	87	34	54	78	30
13	122	82	103	59	131	224	149	57	180	247	**559**	29	35	26	50	48
14	83	93	59	51	86	134	163	177	139	284	**1046**	279	93	88	303	105
15	76	85	68	53	131	423	195	147	63	317	**795**	177	42	183	153	134
16	97	132	79	21	161	346	274	47	155	336	**531**	75	4	50	162	25
17	74	35	63	39	6	18	60	11	117	60	**761**	157	45	8	181	45
18	89	75	22	86	89	106	69	143	38	**223**	202	20	79	43	72	173
19	371	64	130	118	250	119	24	38	22	323	**445**	21	95	14	68	69
20	31	29	113	39	110	302	217	70	37	437	**572**	21	35	34	61	49
21	125	10	92	85	77	174	145	193	74	180	**523**	215	93	190	305	216
22	48	50	77	31	86	191	124	113	96	276	**527**	103	50	81	127	160
23	2	19	42	46	51	101	102	267	20	227	**474**	149	38	177	183	304
24	49	124	0	27	92	391	159	177	43	275	**717**	32	30	161	87	154
25	104	16	50	25	91	115	93	99	36	229	**277**	46	33	47	48	82
26	86	104	72	55	104	287	176	86	125	234	**461**	61	45	69	101	92
27	71	115	97	51	141	274	174	135	53	195	**678**	275	77	157	185	116
28	367	178	83	58	141	258	123	173	106	238	**742**	69	116	108	44	93
29	293	36	46	182	150	41	52	127	15	188	**412**	83	128	83	46	262
30	157	44	122	93	28	24	98	84	142	76	**378**	185	224	71	128	154

Using the ordering in which they were presented back in Chapter 2, the numbers 0, 1, 2 and 3 correspond to the DEFAULT, MAX-P, MAX-PW and MIN-W operators respectively. Thus, sequence 13 is a short label for the MAX-P–MIN-W heuristic sequence. Numbers in bold font represent the most common heuristic sequence for that run. This information is also presented as a boxplot in Fig. 5.5.

As seen on the plot, a pure heuristic sequence was dominant: pure MAX-PW was the most common sequence amongst the hyper-heuristics. Pure heuristic sequences seem to have been preferred by the hyper-heuristics, especially the pure MAX-P sequence.

At a glance, the box plot reveals that the most common heuristic interaction is just using MAX-PW. On the contrary, sequences 30 and 03 were the least frequent: mixing DEF and MIN-W seems like a bad idea from an evolutionary perspective. The same can be observed

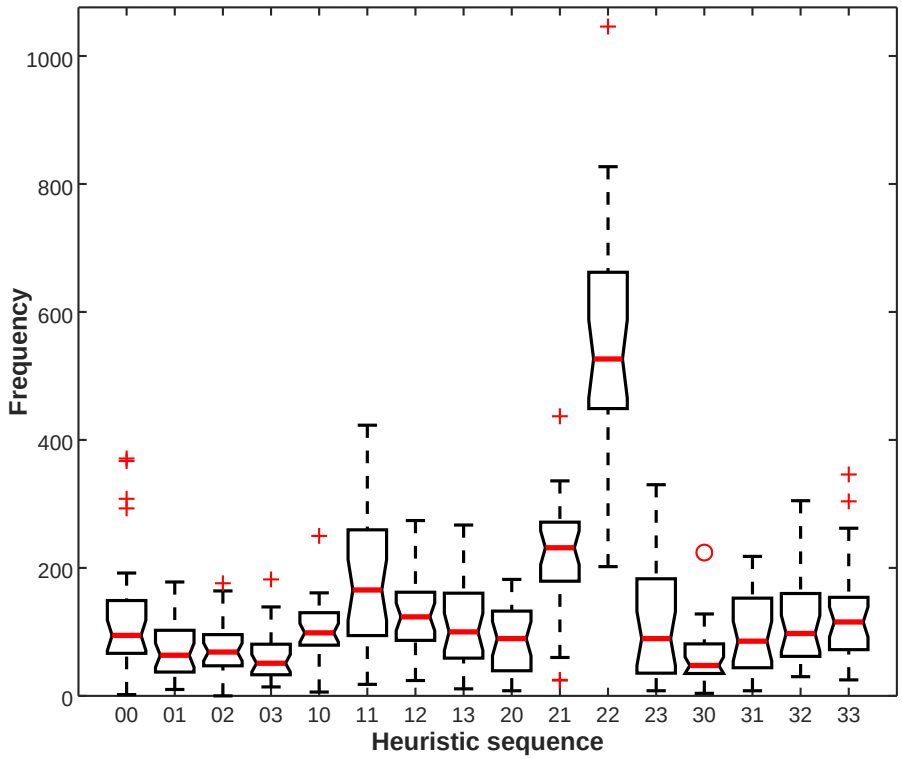

Figure 5.5: Boxplot of heuristic sequences.

for sequences 01 and 10 (DEF–MAX-PW). The mean of each sample was quite similar. In fact, by looking at the notches of all boxes, the means are not significantly different, except for MAX-PW–MAX-P (21) and pure MAX-PW (22). Different mixtures of heuristics do have an impact on how frequently the evolutionary algorithm selects them.

However, the frequency of a heuristic interaction alone cannot guarantee that a single packing heuristic is the best operator for all instances. Since the knapsack has limited capacity and not all items in the instance are supposed to fit in, heuristic sequences at the end of longer hyper-heuristics do not impact on the profit but were, nevertheless, taken into account for statistical purposes in this experiment. For a human, switching heuristics at certain decisions points may seem obvious. An interesting example would be a situation where only two items are left, and the knapsack has capacity for any of them, but not both. The best outcome would be to apply MAX-P and not MAX-PW in this case. A small detail, obvious for us humans was, to our surprise, learned by the proposed model. That is why the sequence MAX-PW–MAX-P has the second most frequent packing heuristic sequence in this examination. This suggests that, after all, there is something to learn from the heuristic interaction, even if it may be unnoticed at first glance.

5.5 Summary

This chapter reviewed the results of the KP branch of experimentation.

Preliminary results shown that the model could potentially outperform standalone heuristics, but it needed a parameter adjustment. Using the new parameters, hyper-heuristics beat single operators, both in synthetically generated instances and in hard instances from the literature.

Another interesting feature was the capability of the model to learn from heuristics that do not perform well on their own. Removing the best heuristic from the training examples, the hyper-heuristic model learned to combine low-level operators which resulted in a solution that outperformed its components but not the best heuristic. This behavior showed that the method is able to find decent results even when no good heuristic is known.

Additionally, a frequency analysis was conducted to identify heuristic interaction during the training phase of hyper-heuristics in the preliminary tests. Results shown that the MAX-PW heuristic is the one most frequently used in training. However, certain combinations like MAX-PW–MAX-P had a significant edge over many other heuristic interactions, revealing that using the best heuristic all the time does not guarantee maximal profit.

Next chapter presents the experimentation on the CSP.

Chapter 6

CSP Analysis

This chapter describes the results of the CSP branch of the experimentation phase. As with KP, two aspects of the hyper-heuristic method were analyzed: learning efficiency and simple interaction between variable and value heuristics.

6.1 Phase I. Preliminary Results

As mentioned in Section 3.3.2, back in Chapter 4, the preliminary testing phase underwent many parameter adjustments throughout the five sub-phases it encompassed. To have a better perspective, the results of each sub-phase is presented in this section in the form of a table. A summary of the standalone heuristics performance is presented in Table 6.1, which serves as a reference for those results obtained during Phase II and for the case of the QCP instances in Phase I.

At a glance, the DOM heuristic shows an edge over the rest of the variable selection operators for this dataset.

The first comparison was performed against results of the first sub-phase, which featured 20-block initial sequences and trained using 15% of the dataset. These results are presented in Table 6.2.

Sub-phase 1 shows interesting results. The first hyper-heuristic, *The Surprised Man* (see Run Representation in Section 3.3.2, back in Chapter 3), performed better than the best heuristic alone on QCP-10, as the average consistency checks for the set was found to be around 916,278 checks (as opposed to 992,757 for DOM in Table 6.1). However, this could have been a lucky hit: hyper-heuristics *The X* and *BLCorner* featured an average of over 2 million consistency checks, which is way worse than the best standalone heuristic.

Sub-phase 2 consisted on a single run using a different initial length: 15 blocks and training using 15% of the QCP-10 instance set. The results were quite impressive, as the hyper-heuristic model could not solve any instance in the set. The testing was transported to the EHI-85 set. Table 6.3 shows the first 15 rows of the full results table: both time and consistency checks for the whole dataset.

Sub-phase 3 shown that longer hyper-heuristics yielded good results, however the instance set itself provided little information to learn: the model learned what it was able to learn in 1000 iterations in 20 minutes, as opposed as the training in QCP where training was

Table 6.1: Performance of standalone heuristics on the QCP-10 instance set. Numbers displayed correspond to the number of consistency checks per method, per instance.

Instance	DOM	DEG	DOMDEG	WDEG	DOMWDEG
qcp-10-67-0_ext	68,177	8,558,850	6,220,676	10,738,844	7,747,438
qcp-10-67-10_ext	6,092,045	11,558,010	8,279,166	10,039,969	8,170,371
qcp-10-67-11_ext	126,634	11,566,901	8,101,622	11,648,799	8,717,308
qcp-10-67-12_ext	5,980,554	11,949,454	8,582,245	11,217,141	7,732,059
qcp-10-67-13_ext	63,749	11,729,419	9,155,159	11,109,788	8,517,710
qcp-10-67-14_ext	589,287	12,048,994	10,720,560	10,799,745	1,494,534
qcp-10-67-1_ext	67,910	11,898,715	267,973	11,380,221	412,486
qcp-10-67-2_ext	68,677	11,927,199	7,672,672	10,506,197	8,858,486
qcp-10-67-3_ext	67,973	11,805,366	8,290,158	10,115,238	2,325,723
qcp-10-67-4_ext	67,760	11,985,380	388,541	11,092,886	248,078
qcp-10-67-5_ext	69,391	10,975,728	8,064,617	10,213,332	517,341
qcp-10-67-6_ext	67,188	11,806,112	875,673	10,478,362	7,688,753
qcp-10-67-7_ext	67,107	11,826,825	442,408	11,246,398	1,297,216
qcp-10-67-8_ext	204,075	11,571,924	8,092,609	10,609,334	1,630,728
qcp-10-67-9_ext	1,290,825	11,895,018	611,457	10,921,404	8,077,074
Sum	**14,891,352**	173,103,895	85,765,536	162,117,658	73,435,305
Average	**992,757**	11,540,260	5,717,702	10,807,844	4,895,687

Table 6.2: Performance of hyper-heuristics in sub-phase 1 of preliminary testing. Time values are shown in milliseconds.

Instance	01-Surprised Man (1101813)		02-Tower (18523)		03-X (71593)		04-BLCorner (74123)	
	CCs	Time	CCs	Time	CCs	Time	CCs	Time
qcp-10-67-0_ext.xml	67,860	31	71,648	31	140,699	78	62,743	15
qcp-10-67-10_ext.xml	4,007,991	2,012	892,100	405	4,662,315	2,012	4,597,834	2,012
qcp-10-67-11_ext.xml	126,489	62	3,754,361	2,012	4,426,009	2,012	4,227,673	2,012
qcp-10-67-12_ext.xml	3,950,856	2,012	4,300,886	2,012	4,605,801	2,012	4,470,447	2,012
qcp-10-67-13_ext.xml	63,645	31	63,325	31	4,347,700	2,012	4,270,416	2,012
qcp-10-67-14_ext.xml	458,338	202	115,028	62	4,446,370	2,012	4,242,439	2,012
qcp-10-67-1_ext.xml	67,880	46	71,551	46	711,405	390	248,114	109
qcp-10-67-2_ext.xml	68,481	46	68,905	46	66,202	31	64,963	46
qcp-10-67-3_ext.xml	67,775	31	66,530	31	4,054,372	2,012	3,979,826	2,012
qcp-10-67-4_ext.xml	67,760	46	67,277	46	4,113,710	2,012	79,871	46
qcp-10-67-5_ext.xml	73,116	46	67,660	31	888,346	468	68,449	46
qcp-10-67-6_ext.xml	67,112	46	67,523	31	80,592	46	63,395	31
qcp-10-67-7_ext.xml	67,651	31	66,853	31	89,848	46	92,503	46
qcp-10-67-8_ext.xml	3,395,342	2,012	3,591,625	2,012	4,152,309	2,012	3,886,594	1,981
qcp-10-67-9_ext.xml	1,193,879	670	838,952	468	558,151	265	3,604,377	2,012
Sum	**13,744,175**	7,324	14,104,224	7,295	37,343,829	17,420	33,959,644	16,404
Average	**916,278.33**	488.27	940,281.60	486.33	2,489,588.60	1,161.33	2,263,976.27	1,093.60

Table 6.3: Performance of hyper-heuristics in Sub-phase 2 of preliminary testing. Time values are shown in milliseconds.

Instance	01-BLCorner (74123)		02-X (71593)		03-Tower (18523)		04-BRCorner (12963)	
	CCs	Time	CCs	Time	CCs	Time	CCs	Time
ehi-85-297-0_ext.xml	65,939	78	61,785	62	61,785	62	61,785	62
ehi-85-297-10_ext.xml	43,717	62	43,717	62	43,717	46	1,130,451	2,012
ehi-85-297-11_ext.xml	47,731	31	47,731	31	47,731	31	47,731	31
ehi-85-297-12_ext.xml	827,114	811	1,270,468	2,012	768,058	873	57,654	62
ehi-85-297-13_ext.xml	36,681	31	998,550	2,012	36,681	31	36,681	31
ehi-85-297-14_ext.xml	35,208	31	35,208	31	35,208	15	35,208	31
ehi-85-297-15_ext.xml	1,172,938	2,012	52,904	62	52,904	62	1,298,061	2,012
ehi-85-297-16_ext.xml	1,132,466	2,012	1,327,590	2,012	908,033	2,012	1,275,989	2,012
ehi-85-297-17_ext.xml	1,024,622	2,012	66,880	46	846,067	2,012	916,828	2,012
ehi-85-297-18_ext.xml	746,323	2,012	60,273	78	1,167,018	2,012	789,101	2,012
ehi-85-297-19_ext.xml	37,733	46	37,733	31	1,343,652	2,012	1,182,199	2,012
ehi-85-297-1_ext.xml	1,494,234	2,012	55,901	46	1,518,762	2,012	1,268,360	2,012
ehi-85-297-20_ext.xml	866,593	2,012	891,733	2,012	1,152,277	2,012	729,748	2,012
ehi-85-297-21_ext.xml	58,840	46	60,426	46	56,143	46	95,342	78
ehi-85-297-22_ext.xml	77,977	62	230,722	171	1,062,656	2,012	366,597	265
				⋮				
Sum	**42,547,375**	75,541	53,177,188	103,836	63,088,682	111,497	52,664,575	97,740
Average	**425,473.75**	755.41	531,771.88	1,038.36	630,886.82	1,114.97	526,645.75	977.40

conducted, in average, through a 170-minute time window. This may be due to the fact that all instances in EHI-85 are unsatisfiable and contain a high amount of constraints: if the search space is vastly populated by constraints, unsatisfiability is likely to appear.

For sub-phase 4, we turned our view to QCP again, so we performed a 1000 iteration learning process for 20-block-long hyper-heuristics on 15% of QCP-15. Results for this phase are presented in Table 6.4.

Table 6.4 shows quite a different story from that in Sub-phase 3. The instances were extremely difficult, as most of the instances show a timeout exception, which are those time values with more than 2000 milliseconds. This dataset represented a challenge, so for the next phase some of the parameters changed. Sub-phase 5 consisted in a different learning focus: the training was conducted on QCP-10 and the testing on both the QCP-10 and QCP-15 datasets. Results of this phase are shown in Table 6.5.

Table 6.5 shows some interesting remarks. First of all, no hyper-heuristic could solve any of the QCP-15 problem instances. This confirms the idea of the dataset being much more complex than its -10 counterpart. The second point of interest here is the great difference between results. The *2HBars* hyper-heuristic solved nine out of 15 instances in QCP-10, and no instance from QCP-15. On the other hand, *2VBars* solved only two of the QCP-10 instances. One can look at the average of consistency checks done by each method to notice that *2VBars* is clearly 'trapped' inside a very difficult part of the search. This efficiency gap may be alarming, but additional samples are needed if one were to conclude behavioral aspects like these. For that reason, we decided that it was best for the training to be performed on the hard instances of QCP-15, and then tested on both. This is the Confirmatory Phase of experimentation, which is detailed in the next Section.

Table 6.4: Performance of hyper-heuristics in Sub-phase 4 of preliminary testing. Time values are shown in milliseconds.

Instance	01-cross (48526)		02-2OHBars (718293)		03-Beast (6666)	
	CCs	Time	CCs	Time	CCs	Time
qcp-15-120-0_ext.xml	3,137,522	1,872	3,910,601	2,012	4,481,021	2,012
qcp-15-120-10_ext.xml	3,668,282	2,012	3,992,422	2,012	4,523,656	2,012
qcp-15-120-11_ext.xml	3,800,111	2,012	3,927,342	2,012	4,523,452	2,012
qcp-15-120-12_ext.xml	4,248,294	2,012	4,388,617	2,012	4,747,947	2,012
qcp-15-120-13_ext.xml	4,107,861	2,012	4,192,631	2,012	5,013,368	2,012
qcp-15-120-14_ext.xml	4,172,560	2,012	4,145,455	2,012	5,038,575	2,012
qcp-15-120-1_ext.xml	2,344,571	1,279	4,052,766	2,012	4,333,987	2,012
qcp-15-120-2_ext.xml	3,881,509	2,012	3,940,556	2,012	4,905,429	2,012
qcp-15-120-3_ext.xml	4,099,492	2,012	4,089,526	2,012	4,603,404	2,012
qcp-15-120-4_ext.xml	3,795,635	2,012	3,781,318	2,012	4,679,460	2,012
qcp-15-120-5_ext.xml	3,922,985	2,012	4,337,474	2,012	4,720,946	2,012
qcp-15-120-6_ext.xml	3,838,566	2,012	4,041,384	2,012	4,792,059	2,012
qcp-15-120-7_ext.xml	3,774,366	2,012	4,331,051	2,012	4,837,609	2,012
qcp-15-120-8_ext.xml	3,874,770	2,012	4,143,709	2,012	4,543,545	2,012
qcp-15-120-9_ext.xml	3,856,538	2,012	4,190,676	2,012	5,055,620	2,012
Sum	**56,523,062**	29,307	61,465,528	30,180	70,800,078	30,180
Average	**3,768,204.13**	1,953.80	4,097,701.87	2,012.00	4,720,005.20	2,012.00

6.2 Phase II. Confirmatory Testing

Phase II of the CSP experimentation consisted on confirming the notion that QCP-15 is computationally harder than the rest of the datasets, as well as showing that there is work to be done when dealing with search spaces of increased size.

Table 6.6 shows a summary of the experiments using consistency checks as a time unit. Table 6.7 shows the same results on milliseconds, where timeouts are easier to observe. Fig. 6.1 illustrates the consistency checks of all different methods per instance in the datasets.

Table 6.6 contains some interesting points. The first thing that comes into play seems to be the "efficiency" of the best hyper-heuristic method, BEST-HH (out of 18 runs of this test). This method obtained around 2.16 million of consistency checks on average for the 30 problem instances. The next method, the median hyper-heuristic, denoted as Median-HH and being the expected average case of the learning process, obtained second place in consistency checks with 3.1 million CCs. The DOM heuristic, by itself, ranked third with 3.5 million CCs. Again, the expected average case seems to be better than the best of the heuristics in isolation. Though it may seem like a small gain (by a 400 thousands-odds difference), half a million consistency checks less may not be that much for an everyday-use computer but could yield a considerable difference if solving for many problems instances.

Another interesting finding is that no heuristic is able to solve any instance on the QCP-15 dataset. However, the Best-HH (viz. *TLSting*, the second pattern in Fig. B.3) was able to solve one of the instances of QCP-15. This may be easier to observe in Table 6.7 where

Table 6.5: Performance of hyper-heuristics in Sub-phase 5 of preliminary testing. Time values are shown in milliseconds.

Instance	01-2HBars (415263)		02-2VBars (741852)	
	CCs	Time	CCs	Time
qcp-10-67-0_ext.xml	87,370	31	5,182,074	2,012
qcp-10-67-10_ext.xml	4,411,727	2,012	5,423,247	2,012
qcp-10-67-11_ext.xml	4,117,404	2,012	4,906,364	2,012
qcp-10-67-12_ext.xml	4,399,692	2,012	5,399,095	2,012
qcp-10-67-13_ext.xml	3,968,393	2,012	5,057,070	2,012
qcp-10-67-14_ext.xml	4,159,763	2,012	5,657,637	2,012
qcp-10-67-1_ext.xml	62,515	31	4,796,905	2,012
qcp-10-67-2_ext.xml	64,419	31	5,205,228	2,012
qcp-10-67-3_ext.xml	63,930	46	5,393,661	2,012
qcp-10-67-4_ext.xml	63,103	31	5,039,885	2,012
qcp-10-67-5_ext.xml	117,629	62	5,203,146	2,012
qcp-10-67-6_ext.xml	89,364	46	4,745,586	1,950
qcp-10-67-7_ext.xml	68,269	46	5,080,645	2,012
qcp-10-67-8_ext.xml	4,423,337	2,012	4,351,775	1,684
qcp-10-67-9_ext.xml	463,051	218	5,381,340	2,012
qcp-15-120-0_ext.xml	4,910,040	2,012	6,842,662	2,012
qcp-15-120-10_ext.xml	5,118,564	2,012	6,432,994	2,012
qcp-15-120-11_ext.xml	5,220,766	2,012	7,155,374	2,012
qcp-15-120-12_ext.xml	5,022,431	2,012	6,891,864	2,012
qcp-15-120-13_ext.xml	5,239,264	2,012	6,920,871	2,012
qcp-15-120-14_ext.xml	5,262,942	2,012	6,763,490	2,012
qcp-15-120-1_ext.xml	5,208,805	2,012	6,595,355	2,012
qcp-15-120-2_ext.xml	5,224,663	2,012	6,625,592	2,012
qcp-15-120-3_ext.xml	5,306,572	2,012	6,958,365	2,012
qcp-15-120-4_ext.xml	5,201,055	2,012	6,683,741	2,012
qcp-15-120-5_ext.xml	5,141,406	2,012	7,131,954	2,012
qcp-15-120-6_ext.xml	5,213,704	2,012	6,829,868	2,012
qcp-15-120-7_ext.xml	5,287,519	2,012	6,842,324	2,012
qcp-15-120-8_ext.xml	5,027,869	2,012	6,660,151	2,012
qcp-15-120-9_ext.xml	5,295,651	2,012	6,737,980	2,012
Sum	**104,241,217**	42,794	178,896,243	59,970
Average	**3,474,707.23**	1,426.47	5,963,208.10	1,999.00

Table 6.6: Comparison of heuristic and hyper-heuristic performance in terms of consistency checks. Entries marked with a star were solved. Best method per instance is highlighted in bold font.

Instance	DOM	DEG	DOMDEG	WDEG	DOMWDEG	BESTHH	MEDIANHH	WORSTHH
qcp-10-67-0_ext.xml	68,177 *	8,558,850	6,220,676 *	10,738,844	7,747,438	68,464 *	**64,813 ***	7,486,155
qcp-10-67-10_ext.xml	6,092,045	11,558,010	8,279,166	10,039,969	8,170,371	**1,961,439 ***	4,581,360	7,684,053
qcp-10-67-11_ext.xml	**126,634 ***	11,566,901	8,101,622	11,648,799	8,717,308	3,874,929	4,216,281	7,231,344
qcp-10-67-12_ext.xml	5,980,554	11,949,454	8,582,245	11,217,141	7,732,059	**305,971 ***	4,215,437	7,553,219
qcp-10-67-13_ext.xml	**63,749 ***	11,729,419	9,155,159	11,109,788	8,517,710	85,030 *	4,379,737	8,027,395
qcp-10-67-14_ext.xml	589,287 *	12,048,994	10,720,560	10,799,745	1,494,534	**673,405 ***	4,397,142	8,060,157
qcp-10-67-1_ext.xml	67,910 *	11,898,715	267,973 *	11,380,221	412,486	**67,243 ***	80,903 *	7,200,555
qcp-10-67-2_ext.xml	68,677 *	11,927,199	7,672,672 *	10,506,197	8,858,486	**67,500 ***	71,902 *	8,011,467
qcp-10-67-3_ext.xml	67,973 *	11,805,366	8,290,158	10,115,238	2,325,723	**67,593 ***	138,970 *	7,777,063
qcp-10-67-4_ext.xml	67,760 *	11,985,380	388,541 *	11,092,886	248,078	68,529 *	**66,778 ***	7,202,130
qcp-10-67-5_ext.xml	69,391 *	10,975,728	8,064,617	10,213,332	517,341	70,881 *	**68,333 ***	8,001,250
qcp-10-67-6_ext.xml	67,188 *	11,806,112	875,673 *	10,478,362	7,688,753	67,673 *	**66,831 ***	7,091,347
qcp-10-67-7_ext.xml	67,107 *	11,826,825	442,408 *	11,246,398	1,297,216	**65,590 ***	67,943 *	7,753,656
qcp-10-67-8_ext.xml	204,075 *	11,571,924	8,092,609	10,609,334	1,630,728	**68,326 ***	3,764,417	7,523,285
qcp-10-67-9_ext.xml	1,290,825 *	11,895,018	611,457 *	10,921,404	8,077,074	427,271 *	**67,958 ***	7,786,914
qcp-15-120-0_ext.xml	4,685,045	14,400,910	9,922,171	14,174,476	11,531,654	**3,608,267**	4,276,030	9,719,269
qcp-15-120-10_ext.xml	6,160,538	15,835,026	10,639,788	14,840,073	12,460,636	**3,645,490**	4,450,213	9,867,690
qcp-15-120-11_ext.xml	6,926,629	15,355,338	10,676,990	14,087,297	12,611,123	**3,935,798**	4,378,498	9,757,482
qcp-15-120-12_ext.xml	7,042,968	16,114,789	9,433,327	15,279,010	12,523,036	**3,910,956**	4,962,932	10,163,964
qcp-15-120-13_ext.xml	6,141,544	16,156,957	8,441,910	15,306,463	12,733,084	**4,099,821**	4,240,287	9,783,093
qcp-15-120-14_ext.xml	6,703,990	15,634,237	10,596,249	14,089,446	12,606,062	**4,194,603**	4,983,760	10,201,646
qcp-15-120-1_ext.xml	6,338,169	15,494,121	10,179,103	13,681,650	11,611,404	**3,628,326**	4,796,956	9,827,919
qcp-15-120-2_ext.xml	6,346,249	15,429,087	10,556,261	14,466,499	11,737,820	**3,659,073**	4,237,423	9,901,137
qcp-15-120-3_ext.xml	5,978,238	15,531,101	11,163,275	14,349,894	12,100,156	**3,710,173**	4,596,087	9,938,905
qcp-15-120-4_ext.xml	5,376,893	13,581,828	9,818,717	12,989,444	10,312,813	**3,765,607**	4,211,880	9,779,524
qcp-15-120-5_ext.xml	5,573,645	14,511,415	10,743,210	13,117,073	12,235,040	**3,803,083**	4,565,165	9,880,091
qcp-15-120-6_ext.xml	5,316,315	14,048,721	8,209,345	11,218,805	10,304,313	**3,901,690**	4,249,834	9,859,525
qcp-15-120-7_ext.xml	5,543,476	15,262,425	10,581,985	13,581,551	11,864,889	**3,296,722 ***	4,714,078	9,491,222
qcp-15-120-8_ext.xml	5,862,847	15,610,418	10,354,540	14,174,750	12,554,662	**3,784,736**	3,801,558	10,399,086
qcp-15-120-9_ext.xml	6,321,079	15,501,444	10,409,071	14,762,213	10,655,608	**4,057,007**	4,803,964	10,286,453
Sum of CCs	105,208,977	401,571,712	237,491,478	372,236,302	251,277,605	**64,941,196**	93,517,470	263,246,996
Average CCs	3,506,965.90	13,385,723.73	7,916,382.60	12,407,876.73	8,375,920.17	2,164,706.53	3,117,249.00	8,774,899.87
Rank	3	8	4	7	5	**1**	2	6

timeout events (method could not solve) are those with more than 2000 milliseconds.

Actually, there were some occasions where the hyper-heuristic model evolved into a sequence of heuristics that were able to solve some instances of QCP-15—six hyper-heuristics out of 18 managed to find a way to solve four instances: `qcp-15-120-1`, `qcp-15-120-6` and `qcp-15-120-7` twice, and `qcp-15-120-8` once.

This reinforces the notion found for KP, back in Chapter 5, that the method is able to come up with better solutions using a mixture of non-optimal operators.

Additionally, a single test run was conducted using a longer timeout: 3000 milliseconds up from 2000. The best heuristic, DOM, was able to solve the first instance of QCP-15 with this extra time. In the case of BEST-HH, TLSting, it could solve three instances from the dataset up from one. The solved instances were `qcp-15-120-0`, `qcp-15-120-1` and `qcp-15-120-2`. An interesting found, since it seems that training with a lower timeout was enough to achieve better results than the best heuristic if an extended timeout is allowed during the testing phase.

Table 6.7: Comparison of heuristic and hyper-heuristic performance in terms of computing time (milliseconds). Best method per instance is highlighted in bold font.

Instance	DOM	DEG	DOMDEG	WDEG	DOMWDEG	BESTHH	MEDIANHH	WORSTHH
qcp-10-67-0_ext.xml	125	2,000	1,687	2,000	2,000	31	31	2,000
qcp-10-67-10_ext.xml	2,000	2,000	2,000	2,000	2,000	998	2,012	2,000
qcp-10-67-11_ext.xml	46	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-10-67-12_ext.xml	2,000	2,000	2,000	2,000	2,000	156	2,012	2,000
qcp-10-67-13_ext.xml	15	2,000	2,000	2,000	2,000	62	2,012	2,000
qcp-10-67-14_ext.xml	156	2,000	2,000	2,000	328	343	2,012	2,000
qcp-10-67-1_ext.xml	15	2,000	62	2,000	78	31	31	2,000
qcp-10-67-2_ext.xml	15	2,000	1,671	2,000	2,000	31	46	2,000
qcp-10-67-3_ext.xml	15	2,000	2,000	2,000	546	46	78	2,000
qcp-10-67-4_ext.xml	15	2,000	93	2,000	62	31	31	2,000
qcp-10-67-5_ext.xml	15	2,000	2,000	2,000	140	31	46	2,000
qcp-10-67-6_ext.xml	15	2,000	234	2,000	2,000	46	31	2,000
qcp-10-67-7_ext.xml	15	2,000	93	2,000	312	46	31	2,000
qcp-10-67-8_ext.xml	62	2,000	2,000	2,000	406	31	2,012	2,000
qcp-10-67-9_ext.xml	468	2,000	140	2,000	2,000	265	31	2,000
qcp-15-120-0_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-10_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-11_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-12_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-13_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-14_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-1_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-2_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-3_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-4_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-5_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-6_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-7_ext.xml	2,000	2,000	2,000	2,000	2,000	1,872	2,012	2,000
qcp-15-120-8_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
qcp-15-120-9_ext.xml	2,000	2,000	2,000	2,000	2,000	2,012	2,012	2,000
Solved	13	0	7	0	7	15	9	0
Time	34,977	60,000	49,980	60,000	47,872	34,200	42,608	60,000
Rank (Solved)	2	7	4.5	7	4.5	1	3	7
Rank (Time)	2	7	5	7	4	1	3	7

However, taking a look at the worst case of learning (denoted as Worst-HH in Tables 6.6 and 6.7) shows a non-desirable behavior: the method was not able to solve any instance at all, nor the 'easy' ones in QCP-10 nor any difficult instances from the QCP-15 dataset. These alarming results can be confirmed comparing the amount of consistency checks between Worst-HH (namely, the *2OVBars* pattern—fourth row, fifth column in Fig. B.3 in Appendix B) and the Best-HH; the difference is over six million CCs. Therefore, one can notice that there are conditions under which learning seems impossible for a small amount of time. In this case, 1000 iterations seem to be insufficient to compensate for an unfavored spot at the start of the searching process.

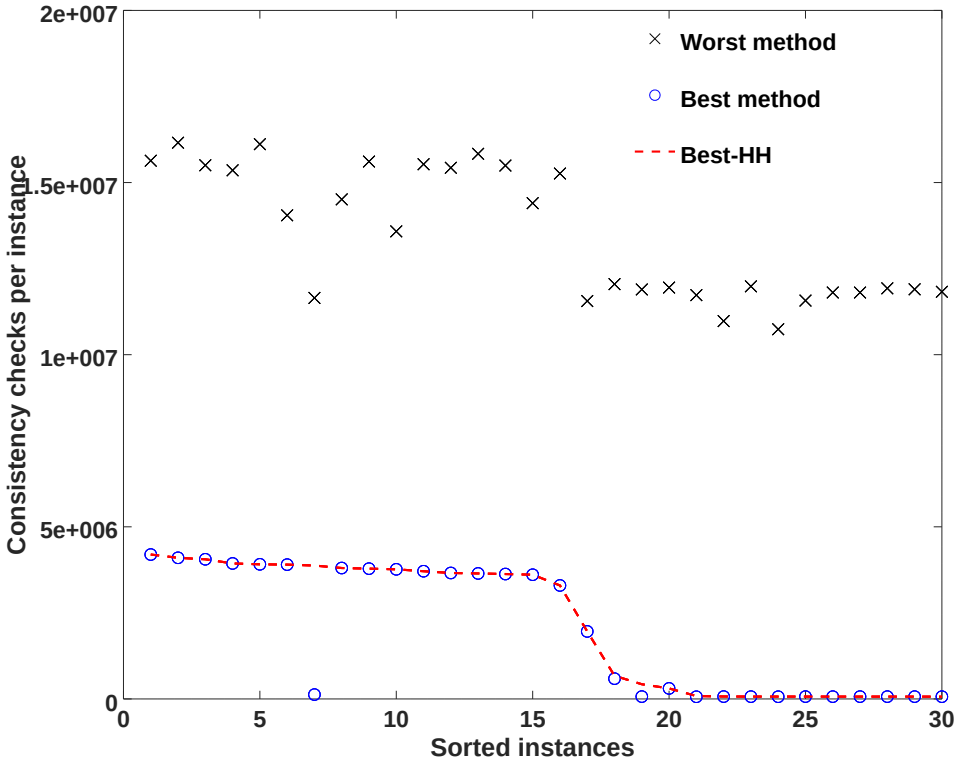

Figure 6.1: Performance of the best hyper-heuristic method per instance, compared against oracle and worst methods available.

Fig. 6.1, shows the best hyper-heuristic method in a red, dotted line, compared to the best and worst methods available (based on Table 6.6). Though it seems that the best-hyper-heuristic method works "as good" as the best method available, this behavior is actually a double-edged blade. Best-HH was the method which performed less consistency checks in average, including for those problem instances which it could not solve. The method is definitely learning to optimize its fitness function: minimize consistency checks. However, how could one measure the *depth* of a search? If two methods A and method B both failed when solving an instance, which one performed better? Which of these methods was closer to reach the goal of proving satisfiability or unsatisfiability? A bad decision in a CSP does not guarantee that the search is going in a bad direction, contrary to KP where the profit can be easily

compared to ensure taking the best decision. An alternative is to test the "generality" of the solver. Table 6.7 includes a row with information about the amount of problems solved by each method. One could compare the best hyper-heuristic method against the DOM heuristic and end up choosing the hyper-heuristic solver as a more general approach. A bit more expensive, since the training process of the hyper-heuristic consumes a decent amount of time, but it could find a solution to some problems which were not solved by heuristics alone. This is a good feature, which is pretty consistent for both the QCP-10 and QCP-15 datasets. Fig. 6.2 illustrates the stability of the learning mechanism on the QCP-10 dataset, and Fig. 6.3 shows the stability in QCP-15.

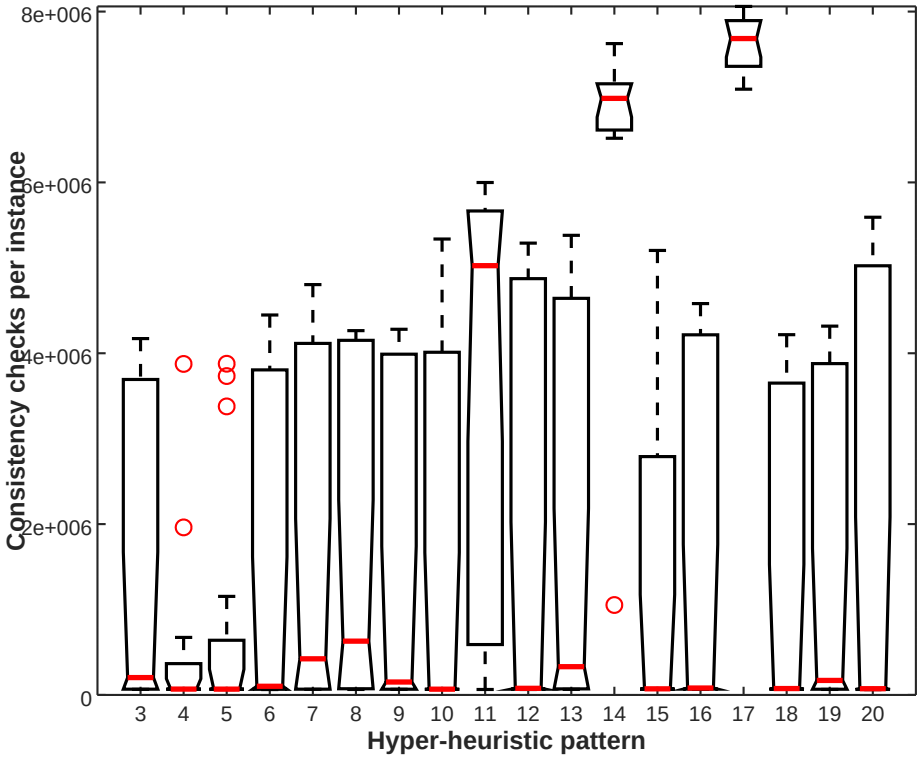

Figure 6.2: Boxplot of Hyper-heuristic methods based on the performance on all QCP-10 instances.

As shown in Fig. 6.2 and 6.3, there is no significant difference between the means for most of the hyper-heuristic methods. The stability is easier to observe in QCP-10, where the notches of the boxes overlap. QCP-15 results are a bit more unstable, but the medians on most methods show no statistical difference.

However, there are specially 'bad' looking methods. For example, hyper-heuristics 11, 14 and 17 were on the high-order of the amount of consistency checks. This behavior suggests that the time dedicated to the learning process may had been insufficient. Method number 17 (the infamous *2OVBars* pattern) could not solve not even a single instance of all the 30 available in the datasets. It is also clear that these methods are outliers, since their medians are significantly different from those of the rest of the hyper-heuristic methods.

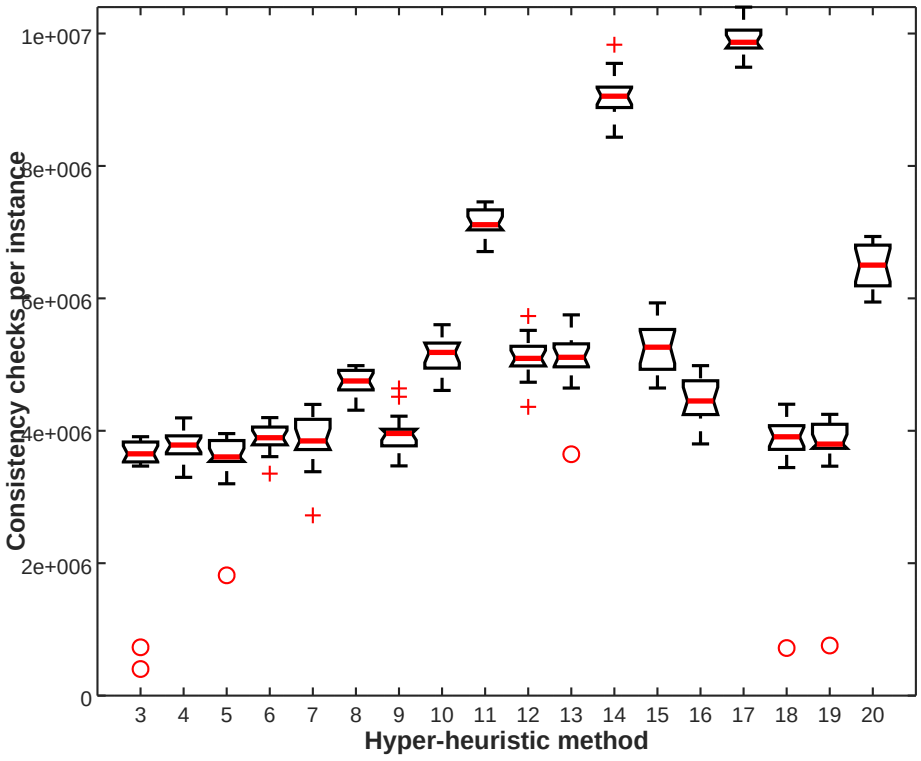

Figure 6.3: Boxplot of Hyper-heuristic methods based on the performance on all QCP-15 instances.

More on stability and local-optima escaping is discussed in the final Chapter. Next section details heuristic interaction analysis.

6.3 Analysis of Heuristic Sequences

As in Chapter 5, a frequency analysis was conducted using the evolution history of all hyper-heuristics from the Confirmatory Testing phase in order to find out if the model prefers or avoids certain combinations of heuristics. We analyzed the frequency of simple 2-digit heuristic sequences, namely all permutations of the heuristics presented in Table 3.3, back in Chapter 3. This experiment included all 18 instances from the Confirmatory Testing Phase, those shown in the run chart available in Fig. B.3, in Appendix B. Tables 6.8 and 6.9 show the frequency of each sequence, where 0 is DOM, 1 is DEG, 2 is DOMDEG, 3 is WDEG and 4 refers to DOMWDEG. Additionally, Fig. 6.4 illustrates this behavior in a boxplot.

Table 6.8: Frequency of two-segment low level heuristic sequences for CSP (I)

Run/Sequence	00	01	02	03	04	10	11	12	13	14	20	21	22
1	8456	57	1953	36	1169	0	0	8	49	0	2067	0	234
2	6579	0	91	0	1906	0	22	0	0	30	150	23	51
3	5314	104	1318	8	995	60	14	60	6	34	1423	0	76
4	6765	0	2205	589	1040	15	0	120	7	0	2437	0	1118
5	7948	0	1896	98	473	16	33	72	0	100	1877	72	327
6	5504	238	2832	962	1925	88	0	0	215	72	2737	67	46
7	7268	0	907	220	991	94	49	152	0	131	441	0	1117
8	7893	115	1078	59	2034	62	0	65	54	38	1253	5	140
9	3878	1000	1944	981	29	1022	21	13	0	10	952	13	1005
10	9153	371	1636	763	2602	281	271	0	271	9	1234	0	0
11	5389	78	141	81	1136	46	0	97	12	68	823	106	6
12	1692	0	932	56	1887	106	894	1740	254	1068	0	2937	1000
13	8862	112	238	42	1899	67	0	114	63	43	461	47	307
14	7508	0	1004	1030	964	0	0	24	0	0	1026	24	75
15	1000	1000	0	0	1000	1009	1986	2000	0	0	0	1990	2003
16	7311	242	946	0	1000	177	0	0	142	65	1028	0	248
17	7706	105	1237	8	29	117	26	47	70	0	1965	96	1015
18	3366	336	844	546	3321	336	0	0	307	0	1005	307	693

Fig. 6.4 shows a clearly dominant presence of the pure DOM–DOM sequence. Not only that, but all those heuristic interactions of DOM have an statistically different mean than all other heuristic interactions.

Another interesting thing regarding the sequence DOM–DOM is its standard deviation. The IQR is quite large, since the highest and lowest values are somewhat far from each other. This remark along with the information from Tables 6.6 and 6.7 can be used to conclude that, even if DOM–DOM is the hyper-heuristic's favorite, it does not guarantee finding a solution.

It may be worthwhile to mention that the method which performed better was Best-HH, which solved only half of the available instances in the set when testing with a 2000-millisecond timeout. As mentioned before, the number of solved instances increases when

Table 6.9: Frequency of two-segment low level heuristic sequences for CSP (II)

Run/Sequence	23	24	30	31	32	33	34	40	41	42	43	44
1	11	109	66	0	46	4	0	1255	0	17	16	63
2	0	66	0	0	0	0	0	1826	80	125	0	1262
3	0	54	4	10	0	0	6	1003	46	57	8	53
4	547	16	649	1	418	814	0	1023	141	284	33	197
5	179	15	104	95	48	37	92	372	21	127	62	59
6	27	77	1128	0	76	0	0	1999	0	0	0	82
7	230	842	239	142	0	230	117	1627	0	454	0	815
8	0	141	97	14	62	131	173	1856	85	194	132	978
9	0	1987	968	32	17	32	0	1048	0	978	4	2018
10	0	0	1019	281	0	430	145	2480	271	5	0	769
11	147	120	204	36	0	48	20	359	3	958	20	136
12	0	0	837	163	328	63	56	1932	68	937	1074	63
13	7	61	26	17	69	35	12	1791	43	155	14	229
14	0	135	1030	0	0	14	15	1034	0	65	15	1000
15	1000	1009	0	0	0	0	1000	991	10	1999	0	1
16	236	449	142	0	357	37	258	1065	142	186	379	808
17	28	1	58	25	23	41	0	21	8	1	0	0
18	723	0	2090	0	0	0	442	1616	0	1191	956	337

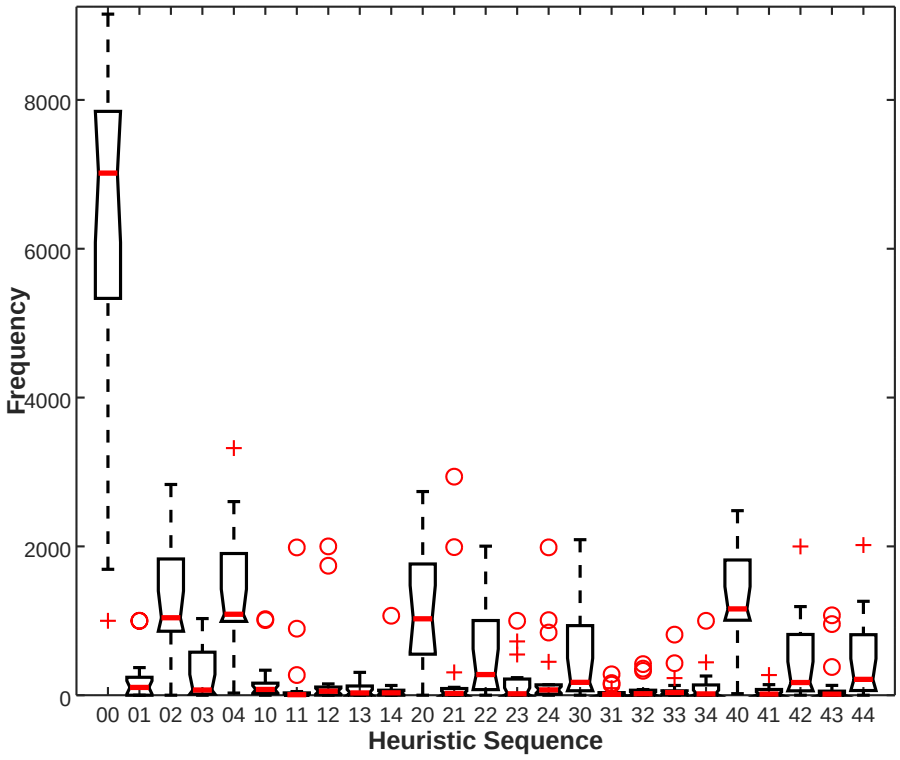

Figure 6.4: Boxplot of Heuristic sequences.

the timeout is increased during the testing phase, even if the training was done using a 2000-millisecond timeout. The presence of solutions found by the hyper-heuristic models in a set where no standalone heuristic could find its way suggests that there are key decision points where changing operators may result in great reductions in complexity.

6.4 Summary

This chapter reviewed the results of the CSP branch of experimentation.

32 hyper-heuristics were trained and tested in this branch. The preliminary phase consisted of 14 hyper-heuristics trained and tested over different instance sets from the literature, mostly for parameter adjustment and experiment design.

The confirmatory testing phase comprised 18 hyper-heuristics trained on QCP-15 and tested on both QCP-10 and QCP-15. The method was able to reduce the amount of consistency checks and solved most instances in QCP-10. QCP-15 was a hard challenge, but a third of the hyper-heuristics managed to obtain solutions in some instances despite the fact that no solution could be found by standalone heuristics before timeout.

Another interesting found was that the method performed better if the 2000-millisecond timeout was increased during the testing phase, as it could find solutions to additional problem instances even when the training process was carried out using a lower timeout limit. On the other hand, increasing the timeout limit did nothing for the best heuristic in isolation.

A frequency analysis was also conducted using evolution data from the 18 hyper-heuristics in the confirmatory testing phase. Results shown that, as it was the case of CSP, there was a significant difference between a single heuristic and the rest of the interactions, which showed no statistical difference between them. Along with the information on the number of instances solved, one can conclude that, again, using the best heuristic does not guarantee finding a solution—all the more so since there is no way to tell if the method is searching in the right direction without proper objective functions.

More about diversity and overfitting is summarized in Chapter 7.

Chapter 7

Conclusions and Future Work

This chapter aims to generalize the findings on Chapters 5 and 6, comparing the results of the performance of the learning mechanism in the Knapsack Problem and the Constraint Satisfaction Problem, summarizing and pointing out the advantages and disadvantages of the model. Additionally, the research questions originally described in Section 1.3 are to be answered.

7.1 On Performance and Quality of the Hyper-heuristic Model

As seen on Chapters 5 and 6, the method is doing what it was meant to: learning to identify key decision points where changing heuristics may be appropriate. This is easily seen in the KP results, where the total profit of the dataset was greater than the best heuristic available (Max-PW). Not that obvious in CSP, but this was also the case for some instances. When solving easy instances (like the ones in QCP-10), most hyper-heuristics performed less consistency checks than the best heuristic available (DOM). This shows that, clearly, the learning process is of great importance when trying to get closer to optimality. For optimization problems like Knapsack, when profit is more easily countable, this hyper-heuristic model may be suitable. However, decision problems like the Constraint Satisfaction Problem seems to be a hard challenge. The size of the search space is clearly difficult to traverse, but experiments showed that under some circumstances the method prevailed. This is because of the nature of the CSP: there is no way to ensure that making a decision will lead the method to find a solution. In contrast to this behavior, in KP there is a strong objective function which serves as a guidance: one can measure how impactful a decision is, since a given problem state could be distinguished from the rest by the profit it generates. Therefore, problems where the greedy approach works may be suitable for this method.

What is needed, then, to increase the exploration of new solutions? As pointed out in Section 2.4.1, back in Chapter 2, there is no algorithm that performs better than other when all problems are evaluated as a whole. Diversity, however, may be of use for exploitation strategies like the evolutionary algorithm used in this research. Having a wide range of heuristics to choose from is one of the many ways in which the model could improve.

Another approach is to "mix" acceptance operators, as Lehre and Özcan suggest in [36]. Accepting worse solutions (not only improvements) than the current candidate could lead to escape local optima faster. However, that represents an additional parameter to set up when

dealing with the training phase.

Increasing the number of iterations for the training phase could also be considered as a way to let the method explore more options when dealing with decent-sized problems (like constraint satisfaction). This could back-fire, of course, when dealing with small instances or problems more on the easy-side of NP-Complete problems—like Knapsack—since it could lead to overfitting.

Another way to explore different regions of the search space is to increase the diversity of mutation operators. From an evolutionary point of view, it may not make sense to have different mutators. Even though the mutation phase is "assured" on every iteration of the model, and is a bit disruptive, it may not be enough when dealing with combinatorial problems with random components. As mentioned in Chapter 4, some mutators are milder than others. Adjusting their chance of being selected by designing a probability distribution may also be a good idea.

A more evolutionary-friendly alternative to disruptive mutation is that of crossover. Dang et al. recently showed that using diversity during the crossover phase of a *simple* genetic algorithm $(\mu + 1)$ with μ individuals and just one offspring, escapes from local optima faster than using only mutation even when the probability of mutation tends to be 1 [13]. This is quite relevant if working with difficult problems, where μ *searchers* will probably converge to a single solution faster than only one searcher.

This brings up the idea of searching over actual high-level methods and using them as prospects for solving, for example, when metaheuristics or even hyper-heuristics could be part of the *building blocks* of the solution. However, adding all sorts of 'diverse' strategies to the selection pool is actually expanding the search space, and getting to a solution which is better than one which is easily obtainable using heuristics may represent an increase in resources and time; it may not be worth it for easy problems. And though it is true that no "best algorithm" exists for all problems, generalization is not implausible.

Problem characterization may represent a roadblock when dealing with complex problems. A flexible approach, like the hyper-heuristic model in this dissertation, may be suitable to obtain solvers which outperform low-level heuristics when no knowledge of the problem is known. Furthermore, as in most evolutionary approaches, the metaphor of the evolutionary algorithm makes it easy to implement, as adding heuristics, mutation operators and objective functions can be easily mapped to most optimization domains.

Is then appropriate the use of this hyper-heuristic model for combinatorial optimization problems? The answer, as in most questions of this matter, is: "it depends." Results on both CSP and KP exhibited numerous cases of hyper-heuristics beating the profit or consistency checks (whichever the case may be) obtained by a simple heuristic. However, overall, the method performed better in KP than in CSP. The CSP, being a decision problem, does not naturally come with an objective function. The minimization of consistency checks is actually just an approximated measure—a rough guess—about how well the searcher is doing. It is not an actual objective function. It can be then concluded that the method requires the presence of a precise objective function which guides the search in order to perform well. Decision problems may be more difficult to attack, but experimental results showed that the method can put out decent solutions nonetheless, which strengthens the generality and *portability* of the model. Without too much hassle, the model can be transported to other domains at a certain extent. Parameter setting, however, is still an aspect to consider, as in many optimization

algorithms, but as described by Poli in [50], this *quest* for the optimal parameter setting is a search problem on its own, where we—the algorithm designers—are the searchers.

Another interesting find pertaining the learning mechanism is the fact that *better* solutions can actually be constructed using bits and pieces of non-promising methods. This answer the question of the possibility to obtain a good solution from combining heuristics that perform poorly in isolation. Having found that "better decisions" are actually made of a sequence of "good decisions" in greedy approaches, will the learning method to venturing into including those non-promising methods into the selection pool in order to exploit their approaches to problem solving.

7.2 Future Work

The field of combinatorics is immense, and there are many approaches to take if one were to expand this research. On one hand, the idea of using additional mutators is quite attractive, since the model and the implementation allow this kind of "expansion" to be pretty easy. A mutator builder could be also considered. The idea seems not only fun, but also very ambitious.

On the other hand, mutation operators in this research were applied using a uniform probability distribution, but some mutation operators are more disruptive than others, and some others could lead to faster convergence. Analyzing adequate conditions and probability distributions for the model is an interesting challenge which could be an entire thesis on its own, just as the parameter setting on difficult decision problems like constraint satisfaction.

Expanding the model is another interesting idea, whether using some kind of crossover, or adding a bit of problem characterization; not that much to be a manual and tedious task, but not inexistent as these little reductions in search space could represent huge complexity jumps.

Combining diverse strategies for parameter setting, move acceptance and playing around with the fitness function is only a handful of the many ways in which this hyper-heuristic model can be expanded and transported to other areas of optimization.

Definitely more samples are needed. The statistical nature of the project sets up the opportunity of performing comparisons between instances in many areas, and such an invitation to explore heuristic interaction should not be wasted—there is way too much to learn from using evolutionary-based hyper-heuristics.

Appendices

Appendix A

Reading a UML Sequence Diagram

Sequence diagrams are used to describe interactions and operations between objects in a sequential, top-down manner.

The objects (or classes) involved in a process are depicted by rectangles on the top of the diagram. A dotted vertical line represents the lifetime of each object. Vertical rectangles along lifelines are used to represent actions performed by an object. Interactions (messages) between objects are represented using horizontal arrows. Solid arrow heads represent synchronous messages (e.g. function calls which require a response), while open arrow heads represent asynchronous messages (e.g. passing parameters). Dashed lines in arrows are used to represent reply messages (e.g. return values).

Fig. A.1 shows a UML Sequence Diagram of a simple sequence when sending an e-mail. A `checkEmail` task is started at the `:Computer` object. Short after this, a function `sendUnsetEmail` is called from the computer to the `:Server` object. The server object processes the requests and finishes without giving any response to the computer. A new task is created on the computer, `newEmail`, which calls the server and awaits for a response. An additional task is created on the computer shortly after getting a response from the server, `downloadEmail` (specifically the `newEmail` instance) and is passed on to the server to process. Finally, the computer asks the server to `deleteOldEmail` and the server completes this request. The user finishes using the computer, and so the whole process is terminated.

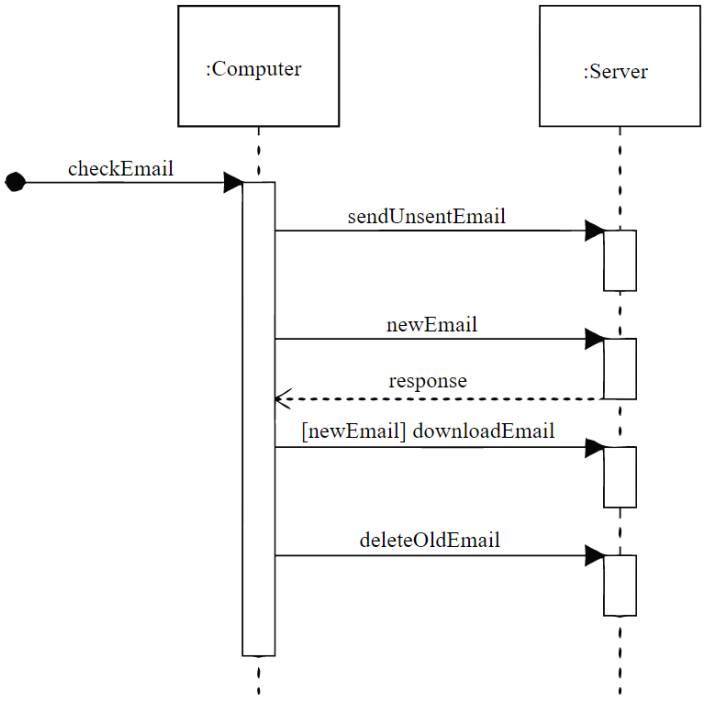

Figure A.1: Sequence diagram of e-mail message sequence. Picture freely available under a CC BY-SA 3.0 license, available at: `https://en.wikipedia.org/wiki/Sequence_diagram`.

Appendix B

The CSP Run Chart

During the preliminary phase of the CSP branch of experimentation, some parameter adjustments were made. These adjustments made many minute changes between sub-phases, and pinpointing which run had which characteristics was difficult. For this reason, each run was labeled with a human-readable label, a catchy "name" which was easier to remember. The name of a hyper-heuristic depended on the seed used to generate the subset of its training instances. Table 3.5, back in Chapter 3 lists all hyper-heuristic methods and their respective labels and seeds.

A visual representation was generated for this table, which was inspired by the arrangement of the keys in the *numeric keypad*, which is present in most personal computers keyboards. Fig. B.1 shows the layout of a *numpad*.

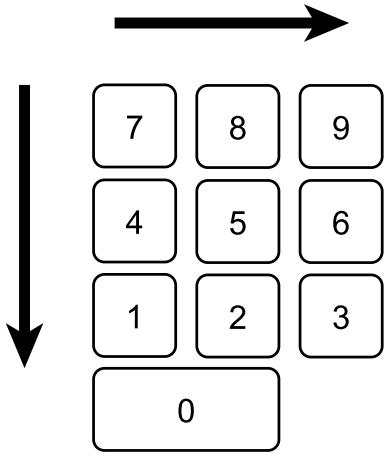

Figure B.1: An abstract representation of a numeric keypad (*numpad*).

This chart shows a group of squares where some of them may be filled. A filled square represents one or many key-presses at that number. The numbers are read in a top-down fashion, going one column at a time, from left to right.

Since a keypad could be used to show the information of a single run (one hyper-heuristic), many keypads were laid out one next another in order to group them. Each row

represents a sub-phase. Patterns in the same row indicate that they were part of the same sub-phase. Fig. B.2 shows the run-chart for the preliminary experimentation phase in CSP.

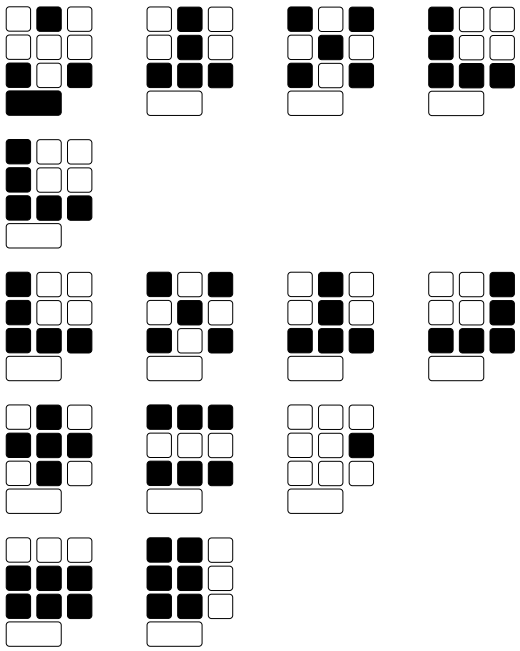

Figure B.2: Run chart for the preliminary experimentation testing phase of the CSP branch of the analysis.

Therefore, the second run of the first sub-phase (first row, second column of Fig. B.2) can be decoded as 18523, which is the seed used for that specific run. This notation is used throughout Chapter 6 to easily identify specific points of the experimentation phase. A similar chart is provided for the confirmatory testing phase as well, in which all patterns belong to the same sub-phase. The chart for the runs in the confirmatory testing phase is presented in Fig. B.3.

It is our intention to further improve and formalize the pattern for seed representation in order to use it in future occasions.

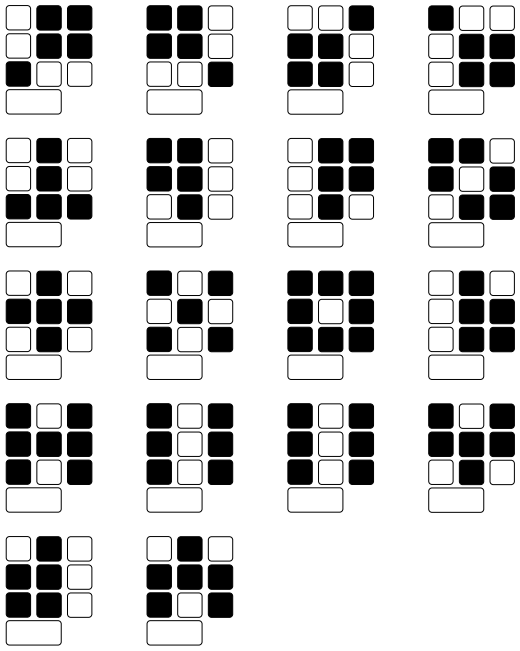

Figure B.3: Run chart for the confirmatory testing phase of the CSP branch of the analysis.

Bibliography

[1] ALANAZI, F., AND LEHRE, P. K. Runtime analysis of selection hyper-heuristics with classical learning mechanisms. In *2014 IEEE Congress on Evolutionary Computation (CEC)* (July 2014), pp. 2515–2523.

[2] ALANAZI, F., AND LEHRE, P. K. Limits to learning in reinforcement learning hyper-heuristics. In *Evolutionary Computation in Combinatorial Optimization: 16th European Conference, EvoCOP 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings*, F. Chicano, B. Hu, and P. García-Sánchez, Eds. Springer International Publishing, Cham, 2016, pp. 170–185.

[3] ARBELAEZ, A., HAMADI, Y., AND SEBAG, M. Online heuristic selection in constraint programming. https://hal.inria.fr/inria-00392752, 2009. International Symposium on Combinatorial Search - 2009.

[4] BAI, R., BURKE, E. K., GENDREAU, M., KENDALL, G., AND MCCOLLUM, B. Memory length in hyper-heuristics: An empirical study. In *Computational Intelligence in Scheduling, 2007. SCIS '07. IEEE Symposium on* (April 2007), pp. 173–178.

[5] BURKE, E. K., GENDREAU, M., HYDE, M., KENDALL, G., OCHOA, G., ÖZCAN, E., AND QU, R. Hyper-heuristics: a survey of the start of the art. *Journal of the Operational Research Society 64*, 12 (December 2013), 1695–1724.

[6] BURKE, E. K., HYDE, M., KENDALL, G., OCHOA, G., ÖZCAN, E., AND WOODWARD, J. R. A classification of hyper-heuristic approaches. In *Handbook of Metaheuristics* (Boston, MA, 2010), Springer US, pp. 449–468.

[7] BURKE, E. K., HYDE, M. R., KENDALL, G., AND WOODWARD, J. Automating the packing heuristic design process with genetic programming. *Evol. Comput. 20*, 1 (March 2012), 63–89.

[8] BURKE, E. K., KENDALL, G., NEWALL, J., HART, E., ROSS, P., AND SCHULENBURG, S. Hyper-heuristics: An emerging direction in modern search technology. *Handbook of Metaheuristics* (2003), 457–474.

[9] CHEESEMAN, P., KANEFSKY, B., AND TAYLOR, W. M. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1* (San Francisco, CA, USA, 1991), IJCAI'91, Morgan Kaufmann Publishers Inc., pp. 331–337.

[10] CHEESEMAN, P., KANEFSKY, B., AND TAYLOR, W. M. Computational complexity and phase transitions. In *Workshop on Physics and Computation* (October 1992), pp. 63–68.

[11] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to algorithms*. MIT Press, 2009.

[12] CRAWFORD, J. M., AND AUTON, L. D. Experimental results on the crossover point in satisfiability problems. In *AAI-93 Proceedings* (1993), pp. 21–27.

[13] DANG, D., FRIEDRICH, T., KÖTZING, T., KREJCA, M. S., LEHRE, P. K., OLIVETO, P. S., SUDHOLT, D., AND SUTTON, A. M. Escaping local optima using crossover with emergent or reinforced diversity. *to appear in IEEE Transactions on Evolutionary Computation abs/1608.03123* (2016).

[14] DECHTER, R. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[15] DECHTER, R., AND ROSSI, F. *Constraint Satisfaction*. John Wiley & Sons, Ltd, 2006.

[16] DRAKE, J. H., HYDE, M., IBRAHIM, K., AND ÖZCAN, E. A genetic programming hyper-heuristic for the multidimensional knapsack problem. In *11th IEEE International Conference on Cybernetic Intelligent Systems* (Limerick, Ireland, August 2012).

[17] DRAKE, J. H., ÖZCAN, E., AND BURKE, E. K. A case study of controlling crossover in a selection hyper-heuristic framework using the multidimensional knapsack problem. *Evolutionary Computation 24*, 1 (March 2016), 113–141.

[18] DROSTE, S., JANSEN, T., AND WEGENER, I. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science 276*, 1 (2002), 51–81.

[19] ERDŐS, P., AND RÉNYI, A. On the evolution of random graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences* (1960), pp. 17–61.

[20] FALKENAUER, E. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics 2*, 1 (1996), 5–30.

[21] FORTNOW, L. The status of the P versus NP problem. *Commun. ACM 52*, 9 (September 2009), 78–86.

[22] FREUDER, E. C., AND MACKWORTH, A. K. Constraint satisfaction: An emerging paradigm. In *Handbook of Constraint Programming*, F. Rossi, P. V. Beek, and T. Walsh, Eds. Elsevier, 2006, pp. 13–28.

[23] FRIEZE, A., AND KAROŃSKI, M. *Introduction to Random Graphs*. Introduction to Random Graphs. Cambridge University Press, 2015.

[24] GENT, I. P., MACINTYRE, E., PRESSER, P., SMITH, B. M., AND WALSH, T. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Principles and Practice of Constraint Programming – CP96* (1996), vol. 1118 of *Lecture Notes in Computer Science*, pp. 179–193.

[25] GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[26] GOMES, C., AND WALSH, T. Randomness and structure. In *Handbook of Constraint Programming*, F. Rossi, P. V. Beek, and T. Walsh, Eds. Elsevier, 2006, pp. 639–664.

[27] GOMES, C. P., AND SELMAN, B. Algorithm portfolio design: Theory vs. practice. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence* (San Francisco, CA, USA, 1997), UAI'97, Morgan Kaufmann Publishers Inc., pp. 190–197.

[28] HART, E., AND SIM, K. On the life-long learning capabilities of a NELLI*: A hyper-heuristic optimisation system. In *Parallel Problem Solving from Nature – PPSN XIII* (September 2014), vol. 8672 of *Lecture Notes in Computer Science*, pp. 282–291.

[29] HART, E., AND SIM, K. A hyper-heuristic ensemble method for static job-shop scheduling. *Evol. Comput. 24*, 4 (December 2016), 609–635.

[30] HART, E., AND SIM, K. On constructing ensembles for combinatorial optimisation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (New York, NY, USA, 2017), GECCO '17, ACM, pp. 5–6.

[31] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*, 2nd ed. MIT Press, Cambridge, MA, USA, 1992.

[32] HROMKOVIČ, J. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[33] HYDE, M. *A Genetic Programming Hyper-Heuristic Approach to Automated Packing*. PhD thesis, University of Nottingham, March 2010.

[34] KELLERER, H., PFERSCHY, U., AND PISINGER, D. *Knapsack Problems*. Springer-Verlag Berlin Heidelberg, 2004.

[35] LECOUTRE, C. Csp benchmarks. Internet. https://www.cril.univ-artois.fr/ lecoutre/benchmarks.html.

[36] LEHRE, P. K., AND ÖZCAN, E. A runtime analysis of simple hyper-heuristics: To mix or not to mix operators. In *Proceedings of the Twelfth Workshop on Foundations of Genetic Algorithms XII* (New York, NY, USA, 2013), FOGA XII '13, ACM, pp. 97–104.

[37] LOURENÇO, N., PEREIRA, F. B., AND COSTA, E. The importance of the learning conditions in hyper-heuristics. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation* (New York, NY, USA, 2013), GECCO '13, ACM, pp. 1525–1532.

[38] MARTELLO, S., AND TOTH, P. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[39] MISIR, M., VERBEECK, K., PATRICK DE-CAUSMAECKER, P., AND VANDEN-BERGHE, G. An investigation on the generality level of selection hyper-heuristics under different empirical conditions. *Applied Soft Computing 13*, 7 (July 2013), 3335–3353.

[40] MITCHELL, D., SELMAN, B., AND LEVESQUE, H. Hard and easy distributions of sat problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence* (1992), AAAI'92, AAAI Press, pp. 459–465.

[41] O'MAHONY, E., HEBRARD, E., HOLLAND, A., NUGENT, C., AND O'SULLIVAN, B. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Proceedings of the Irish Artificial Intelligence and Cognitive Sciences Conference* (2008).

[42] ORTIZ-BAYLISS, J. C. *Exploring Hyper-heuristic Approaches for Solving Constraint Satisfaction Problems*. PhD thesis, Instituto Tecnológico y de Estudios Superiores de Monterrey, Monterrey, NL, México, December 2011.

[43] ORTIZ-BAYLISS, J. C., MORENO-SCOTT, J. H., AND TERASHIMA-MARÍN, H. Automatic generation of heuristics for constraint satisfaction problems. In *Nature Inspired Cooperative Strategies for Optimization (NICSO 2013)* (2013), vol. 512 of *Studies in Computational Intelligence*, pp. 315–327.

[44] ORTIZ-BAYLISS, J. C., ÖZCAN, E., PARKES, A. J., AND TERASHIMA-MARIN, H. Mapping the performance of heuristics for constraint satisfaction. In *IEEE Congress on Evolutionary Computation* (July 2010), pp. 1–8.

[45] ORTIZ-BAYLISS, J. C., TERASHIMA-MARÍN, H., ÖZCAN, E., PARKES, A. J., AND CONANT-PABLOS, S. E. Exploring heuristic interactions in constraint satisfaction problems: A closer look at the hyper-heuristic space. In *2013 IEEE Congress on Evolutionary Computation* (June 2013), pp. 3307–3314.

[46] ÖZCAN, E., BILGIN, B., AND KORKMAZ, E. E. A comprehensive analysis of hyper-heuristics. *Intell. Data Anal. 12*, 1 (January 2008), 3–23.

[47] PAVLUS, J. Machines of the infinite. *Scientific American 307*, 3 (2012), 66–71.

[48] PETROVIC, S. V., AND EPSTEIN, S. L. Tailoring a mixture of search heuristics. *Constraint Programming Letters 4* (2008), 15–38.

[49] PISINGER, D. Where are the hard knapsack problems? *Comput. Oper. Res. 32*, 9 (September 2005), 2271–2284.

[50] POLI, R., AND GRAFF, M. There is a free lunch for hyper-heuristics, genetic programming and computer scientists. In *Genetic Programming: 12th European Conference, EuroGP 2009 Tübingen, Germany, April 15-17, 2009 Proceedings* (Berlin, Heidelberg, 2009), Springer Berlin Heidelberg, pp. 195–207.

[51] RICE, J. R. The algorithm selection problem. *Advances in Computers 15* (1976), 65–118.

[52] RUSSELL, S., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*, 3 ed. Prentice Hall, Upper Saddle River, NJ, USA, 2010.

[53] SAYAG, T., FINE, S., AND MANSOUR, Y. Combining multiple heuristics. In *STACS 2006: 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23-25, 2006. Proceedings* (Berlin, Heidelberg, 2006), Springer Berlin Heidelberg, pp. 242–253.

[54] SIM, K., AND HART, E. An improved immune inspired hyper-heuristic for combinatorial optimisation problems. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation* (New York, NY, USA, 2014), GECCO '14, ACM, pp. 121–128.

[55] SIM, K., HART, E., AND PAECHTER, B. A lifelong learning hyper-heuristic method for bin packing. *Evol. Comput. 23*, 1 (March 2015), 37–67.

[56] SKIENA, S. *The Algorithm Design Manual*, 2nd ed. Springer-Verlag London, 2008.

[57] SMITH, B. M. A tutorial on constraint programming. Tech. rep., University of Leeds, 1995.

[58] SMITH-MILES, K. A. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv. 41*, 1 (January 2009), 6:1–6:25.

[59] SMITH-MILES, K. A., BAATAR, D., WREFORD, B., AND LEWIS, R. Towards objective measures of algorithm performance across instance space. *Computers and Operations Research 45* (May 2014), 12–24.

[60] SOSA-ASCENCIO, A., OCHOA, G., TERASHIMA-MARIN, H., AND CONANT-PABLOS, S. E. Grammar-based generation of variable-selection heuristics for constraint satisfaction problems. *Genetic Programming and Evolvable Machines 17*, 2 (2016), 119–144.

[61] SWAN, J., WOODWARD, J., ÖZCAN, E., KENDALL, G., AND BURKE, E. K. Searching the hyper-heuristic design space. *Cognitive Computation 6*, 1 (2013), 66–73.

[62] TERASHIMA-MARÍN, H., ORTIZ-BAYLISS, J. C., ROSS, P. M., AND VALENZUELA-RENDÓN, M. Hyper-heuristics for the dynamic variable ordering in constraint satisfaction problems. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation* (New York, NY, USA, 2008), GECCO '08, ACM, pp. 571–578.

[63] TERRAZAS, G., AND KRASNOGOR, N. Grammatical rules for the automated construction of heuristics. In *IEEE Congress on Evolutionary Computation* (July 2010), pp. 1–8.

[64] TSANG, E., AND FRUEHWIRTH, T. *Foundations of Constraint Satisfaction: The Classic Text*. Books on Demand, 2014.

[65] WALLACE, R. J. *Factor Analytic Studies of CSP Heuristics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 712–726.

[66] WALLACE, R. J. Analysis of heuristic synergies. In *Recent Advances in Constraints: Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2005, Uppsala, Sweden, June 20-22, 2005, Revised Selected and Invited Papers* (Berlin, Heidelberg, 2006), Springer Berlin Heidelberg, pp. 73–87.

[67] WEISSTEIN, E. W. Np-complete problem. Internet. http://mathworld.wolfram.com/NP-CompleteProblem.html. From Mathworld, a Wolfram Web Resource.

[68] WEISSTEIN, E. W. Np-hard problem. Internet. http://mathworld.wolfram.com/NP-HardProblem.html. From Mathworld, a Wolfram Web Resource.

[69] WEISSTEIN, E. W. Np-problem. Internet. http://mathworld.wolfram.com/NP-Problem.html. From Mathworld, a Wolfram Web Resource.

[70] WOLPERT, D. H., AND MACREADY, W. G. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation 1*, 1 (April 1997), 67–82.