FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

Department of Informatics Engineering

Pedro Miguel Duque Rodrigues

Principled Modelling Of The Google Hash Code Problems For Meta-Heuristics

Intermediate Report

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems, advised by Professor Alexandre B. Jesus and Professor Carlos M. Fonseca, and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

January 2023

Abstract

The Google Hash Code programming competition is an yearly event that challenges teams to solve challenging engineering problems within a limited time frame using any necessary tools. These problems, which are inspired by real-world issues and can be approached from both practical and theoretical perspectives, are of particular interest to this work. In the context of this thesis, we hope to solve some of these problems in a principled manner, with a particular focus on the modelling aspect and the clear separation between the concept of models and solvers. Additionally, there is interest in exploring the impact of this strategy on the development of general-purpose meta-heuristic solvers that can tackle these problems in a black-box fashion, making them more accessible for practitioners, researchers and developers.

Keywords

Meta-Heuristics • Modelling • Local Search • Constructive Search • Combinatorial Optimization • Intelligent Systems • Software Engineering

Resumo

A competição de programação Hash Code da Google é um evento organizado anualmente onde equipas são convidadas a resolver problemas de engenharia complexos num curto espaço de tempo, usando qualquer ferramenta necessária. Estes problemas, que são inspirados em questões do mundo real e podem ser abordados tanto sob um ponto de vista prático como teórico, são de particular interesse para este trabalho. Esta tese visa resolver alguns destes problemas de uma forma fundamentada, com particular ênfase na vertente de modelação e na clara separação entre o conceito de modelos e algoritmos (solvers). Além disso, há interesse em explorar o impacto desta estratégia no desenvolvimento de algoritmos (solvers) meta-heurísticos genéricos que consigam atacar estes problemas numa perperspetiva "black-box", tornando-os mais acessíveis para profissionais, investigadores e programadores.

Palavras-Chave

Meta-Heurísticas • Modelação • Procura Local • Procura Construtiva • Otimização Combinatória • Sistemas Inteligentes • Engenharia de Software

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the motivation behind this project 1.1, the main contributions 1.2 we hope to make with this work and a brief outline 1.3 on this document's structure.

1.1 Motivation

The Hash Code programming competition is an yearly event held by Google where teams are asked to solve complex and challenging engineering problems using any tools and programming languages of their choice in four hours available. The problems are typically inspired by issues arising in real-world situations, such as vehicle routing, task scheduling, and Wi-Fi router placement. They are posed as "open" research problems for which there exists a variety of solutions of different qualities. In fact, the great majority of these are essentially Combinatorial Optimization (CO) problems concerning the search for the best solutions among a potentially large set of candidate solutions, thus being of utmost importance the adoption of efficient search strategies that take the available time budget into account. Moreover, in the context of the competition, the contestants must also read, understand the problem and find a suitable representation for it, which is to say that, not all the available time will be spent in the solution optimization stage.

For solving CO problems, a variety of algorithms exist that often offer a compromise between the time and quality of the solutions found. The heuristics are a set of procedures, often problem-specific, that attempt to quickly solve a problem and provide a helpful "rule of thumb" for achieving decent results, generally in a greedy fashion. A superset of these algorithms is called meta-heuristics, and contrary to regular heuristics, these are generic and can be applied to a broad range of problems. Natural processes and phenomena, such as collective behavior, natural selection, and some physical properties of materials, inspire several meta-heuristics search processes making them flexible and adaptable, although more computationally intensive than heuristics.

A suite of optimization tools of different natures is available to practitioners

for solving CO problems, thus constituting a challenge to enumerate them all. In particular, there exist a set of tools that rely upon more mathematical and exact approaches yielding optimal solutions although not being used in a competition context due to their poor time performance on challenging problems, such as the Hash Code challenges.

In the context of the Google Hash Code competition, competitors frequently make use of greedy and heuristic strategies tailored to the challenge. Moreover, meta-heuristics in this situation are not as popular due to the time constraints imposed by the competition format. At first instance, the majority of optimization problems are structurally different from each other, which makes it demanding to write general-purpose heuristic solvers that can be easily reused. On the other hand, it might not be worth the effort spent in the implementation of such solvers and provided that the benefit of using a simple heuristic strategy might outweigh the development and the running cost of meta-heuristic solvers e.g., evolutionary algorithms.

Algorithms such as meta-heuristics usually follow some notion of an optimization strategy that guides the procedure in the search for solutions. Some of the main strategies are constructive and local search. Constructive search algorithms work by starting with an empty or partially complete solution and building a complete and feasible solution by iteratively adding components based on the solution's current state and the problem's constraints. In contrast, local search algorithms develop a given initial solution to the problem by introducing small changes that aim to improve it to a local optimum rather than a global one. A common usage of these strategies in competition is to sequence them, starting with a constructive search stage, improving solutions to a certain stagnation point and then taking advantage of a local search strategy in an attempt to enhance the solutions even further.

Figure 1.1: Modelling and Problem Solving

It is evident that algorithms or solvers that utilize these search strategies are highly dependent on the specific problem they are trying to solve. The solver plays a crucial role in the problem-solving process as it is responsible for finding a solution. However, without a comprehensive model that can provide the solver with relevant information about the problem, the solver's effectiveness may be impaired.

The model serves as a means of presenting the various aspects of the problem to the computer, which will subsequently utilize a solver to find a solution (1.1). When constructing a model, it is important to include relevant features such as the representation of the problem and solution, a description of how components can be added or removed from the solution, and methods

for evaluating the solution, calculating bounds, and obtaining other heuristic information.

A good model should encode in it all the relevant information from the problem and ask the right questions to obtain it, adhering to the philosophy that — "*understanding the question is half the answer*". Additionally, if a model was to be built in a standardized way, this would allow the development of a suite of generic and reusable (meta-heuristic) solvers that could find solutions in a black-box fashion. This idea has already been explored to some extent in previous work [19, 22] and will be further deepened.

It is worth noting that the modelling aspect in this field has often been neglected by the community, which has been primarily focused on the development of meta-heuristics algorithms (solvers). This discrepancy can be contrasted with the mathematical perspective, where practitioners and researchers have emphasized the importance of clearly separating the model and the algorithm (solver), and have placed a significant emphasis on the modelling perspective. This gap highlights the importance of considering the modelling aspect in this field.

There has also been a recent growing interest within the community in the development of optimization benchmark problems that are both relevant in practical applications and amenable to theoretical analysis. The Hash Code problems may be suitable candidates for this purpose, as they pose significant challenges from a modelling perspective while also being easily describable. Furthermore, these problems have already been partially solved in a competitive setting and have a wealth of empirical data available on the most effective known solutions. Overall, these factors make the Hash Code problems an ideal testing ground for both modelling and the evaluation of meta-heuristics.

1.2 Contribution

The main goal of this work is to develop and implement effective heuristic and meta-heuristic approaches for solving Hash Code problems, with a particular focus on the modelling aspect and the clear separation between solvers and models. By doing so, we hope to develop more structured and efficient problem-solving strategies that can effectively address a range of challenges. Additionally, some effort will be made to address other key areas that are crucial to the success of this work, namely:

- Development and refinement of the frameworks that separate models from solvers currently materialized in an Application Programming Interface (API) designed only for constructive search [19]. The main goal is to optimize and "fine-tune" this API in order to improve its efficacy and utility, by using the Hash Code problems as benchmarks.
- Expansion of the aforementioned API to support local search strategies in a problem-independent manner. This will allow for its application to a wider range of problems and contexts.

- Implementation of a small set of general-purpose meta-heuristic solvers and utilities that can be used to not only generate and test solutions for the multiple Hash Code benchmark problems, but also to verify the correctness of the results.

Last but not least, the objective of this work is to engage in a critical examination of the strengths and limitations of our proposed approach to problem modelling and solver development. This discussion will be relevant to meta-heuristic researchers, software developers, and practitioners alike. The analysis will consider various performance dimensions, including the effort required for problem modelling and solver development, the computational efficiency of the implemented software, and the quality of the solutions obtained.

1.3 Outline

The remainder of this document is structured as follows:

- **Chapter 2:** Provides some background on some essential aspects of optimization, search strategies, meta-heuristics, and modelling. Moreover, it presents the modelling frameworks and current state of the art of the API [19] that supports this work.
- **Chapter 3:** Analyzes the main objectives that we hope to achieve, along with the methodology required to successfully accomplish them. In addition, a work plan is presented outlining the tasks to be carried out in the upcoming semester. Finally, some comments are made regarding the usability and ease of use of the tools that will be utilized, given their current state.
- **Chapter 4:** Gives some insight into the Google Hash Code competition and typical problems presented to contestants. Furthermore, it makes a brief categorization of all the problems from previous editions, with particular emphasis on the ones analyzed so far. Finally, it describes the modelling of the problems examined and the results obtained
- **Chapter 5:** Presents a brief reflection on this work.

Chapter 2

Background

This chapter presents a literature review of various optimization concepts relevant to the analysis of Hash Code problems as well as the framework detailed in [19], which will be used and potentially improved upon throughout this work. Additionally, it provides background on essential concepts such as modelling and meta-heuristics, which will be further discussed in subsequent chapters. In particular, Section 2.1 presents a series of combinatorial optimization concepts relevant to this work, whilst Section 2.3, delves into the definition of meta-heuristics and provides key concepts and examples. Finally, Section 2.4 presents the concept of modelling and offers insight into current frameworks and API [19, 22] based around this idea.

2.1 Optimization Concepts

Optimization involves finding the best solution to a given problem among a set of feasible solutions. Specifically, considering the single objective case involves finding the optimal configuration or set of parameters that maximize or minimize an objective function, possibly subject to constraints on its variables [17].Given that, and as defined by Papadimitriou and Steiglitz [20], an optimization problem can formally be described as follows:

Definition 2.1.1 (Optimization Problem [20]): *An optimization problem is a collection $\mathcal{I}$ of instances, typically generated in a similar manner. An instance ι of an optimization problem consists of a pair $(\mathcal{S}, f)$, where $\mathcal{S}$ is a set containing all feasible solutions, and f is an objective (cost) function, with a mapping such that:*

$$f : \mathcal{S} \longrightarrow \mathbb{R} \tag{2.1}$$

That is, for each solution $s \in \mathcal{S}$, a real value is assigned to indicate the quality of the solution. Thus, the problem consists in finding a global optimal solution $s^* \in \mathcal{S}$, for each instance ι.

Definition 2.1.2 (Global Optimal Solution [9, 20]): *Assuming, without loss of generality, an optimization problem with a maximizing objective function*

$f(s)$, a global optimal solution s^* is expressed by:

$$\forall s \in \mathcal{S} \colon f(s^*) \geq f(s) \tag{2.2}$$

Given that the Hash Code problems are designed with a maximizing objective function, only maximization problems will be considered in this work. Nonetheless, a minimizing objective function $f(s)$ can be reformulated for maximization by using the identity $\max -f(s) = \min f(s)$.

2.1.1 Combinatorial Optimization

There are two main categories of optimization problems based on the domain of the variables: discrete and continuous. In problems with discrete variables, the solutions are defined on a finite, or countably infinite, set of values. In contrast, for problems with continuous variables, the solutions take on any value on a continuous (infinite) subset of real numbers. Nonetheless, there are also problems that involve both categories commonly denominated as mixed [17].

Definition 2.1.3 (Combinatorial Optimization Problem [20]): *An instance ι of a combinatorial optimization problem is an instance of an optimization problem (2.1.1) where the set $\mathcal{S}$ of feasible solutions is finite or countable infinite.*

Combinatorial Optimization (CO) problems 2.1.3 are a subset of discrete optimization problems characterized by a discrete solution space that typically involves different permutations, groupings, or orderings of objects that satisfy certain criteria [22, 20]. As such, solutions for these problems are discrete objects related to the combinatorics e.g integers, permutations sets and graphs. [2, 25]

Typical examples of CO problems include network flow, matching, scheduling, shortest path and decision problems. Some representative examples of such problems are the Travelling Salesman Problem (TSP) and the Knapsack Problem (KP) [25]. In the case of the KP, given a set of items, each with a given weight and profit, and a knapsack with a given maximum weight capacity, the goal is to find the subset of items with the highest total profit that can be placed in the knapsack without exceeding its capacity. As for the KP, the goal is to find the shortest possible route that visits each city exactly once whilst returning to the starting city i.e an Hamiltonian cycle.

Definition 2.1.4 (Ground Set [19]): *The ground set $\mathcal{G}$ is a finite set of elements that represents all possible components that may integrate a feasible solution for a given instance ι of a CO problem.*

$$\mathcal{G} \colon \{c_1, c_2, c_3, \ldots, c_i\} \tag{2.3}$$

Thus, a feasible solution is a subset of the ground set, denoted as $s \in \mathcal{S} \subseteq 2^{\mathcal{G}}$, and which may be encoded as a binary string indicating the presence (1) or absence (0) of a given component c_i.

Given the discreteness of the decision space in CO problems, the solutions are constructed by combining objects present in a finite set containing all the individual components that fully characterize a solution. This set, commonly denominated "ground set" [19, 5, 15], can be defined as shown in 2.1.4:

Regarding the definition of a CO problem 2.1.3 and alluding to the solution representation as a binary string, we can then define both the feasible solution set $\mathcal{S}$ and the ground set $\mathcal{G}$ for the KP and the TSP.

- **Knapsack Problem**: The set of feasible solutions $\mathcal{S}$ consists of all possible subsets of items that can be placed in the knapsack without exceeding its weight capacity. The ground set $\mathcal{G}$ is a set of all possible items that may be included in a feasible solution [5]. As such, a solution $s \in \mathcal{S}$ can be represented as a binary string, where the i^th position of the string contains a value of 1 or 0 depending on whether the i^th item in $\mathcal{G}$ is present or absent in the final solution.
- **Travelling Salesman Problem**: The set $\mathcal{S}$ consists of all possible Hamiltonian cycles representing the order in which cities should be visited, typically represented as a permutation. The ground set $\mathcal{G}$ in this case can be thought of as a set containing all possible paths between two distinct cities, i.e. edges in a graph [15, 5]. Hence, a solution $s \in \mathcal{S}$ can be represented as a binary string where the i^{th} position of the string contains a value of 0 or 1 depending on whether a given edge connecting two cities is present or absent in the final solution.

In summary, since CO problems involve choosing a combination of objects any algorithm that is able to enumerate the entirety of the solution space can be used to solve these problems. However, finding optimal solutions can be difficult, and exhaustive search strategies may not be able to solve many of these problems, which are often NP-Hard [25, 5] and thus not approachable by algorithms in a reasonable amount of time. In these cases, approximation or heuristic/meta-heuristic methods present themselves as effective alternatives to be considered.

2.1.2 Global and Local Optimization

With regard to the search of solutions for optimization problems, there are two primary strategies: Global Optimization (GO) and Local Optimization (LO).

GO refers to the process of identifying a global optimal solution (2.1.2) to a given problem, regardless of its location within the solution space. In contrast, LO focuses on finding the best solution among those that are proximate in some sense. The concept of proximity is related to the definition of a neighborhood, which for a given solution is specified by a particular neighborhood structure defined as follows:

Definition 2.1.5 (Neighborhood Structure [20, 2]): *A neighborhood struc-*

ture for an optimization problem with instances $(\mathcal{S}, f)$ *is a mapping:*

$$\mathcal{N} : \mathcal{S} \longrightarrow 2^{\mathcal{S}} \tag{2.4}$$

Such that, a set of neighboring solutions $\mathcal{N}(\hat{s}) \subseteq \mathcal{S}$ *is assigned for each solution* $\hat{s} \in \mathcal{S}$*. This referred to as the neighborhood of* $\hat{s}$*.*

In general, the neighborhood structure refers to the set of rules that must be applied to a solution in order to generate all of its neighbors. Additionally, we consider the following definition of local optimal solution.

Definition 2.1.6 ((Strictly) Local Optimal Solution [2, 17]): *Assuming maximization without loss of generality, a solution* $\hat{s}$ *is local optimal with respect to a given neighborhood structure* $N(\hat{s})$ *iff:*

$$\forall s \in \mathcal{N}(\hat{s}) : f(\hat{s}) \geq f(s) \tag{2.5}$$

Furthermore, $\hat{s}$ *is considered strictly global optimal iff*

$$\forall s \in \mathcal{N}(\hat{s}) \setminus \{\hat{s}\} : f(\hat{s}) > f(s) \tag{2.6}$$

As an illustrative example, consider the objective function $f(s)$ shown in figure 2.1. With respect to the definitions 2.1.2 and 2.1.6 s^2, s^3 are (strictly) local optimal.

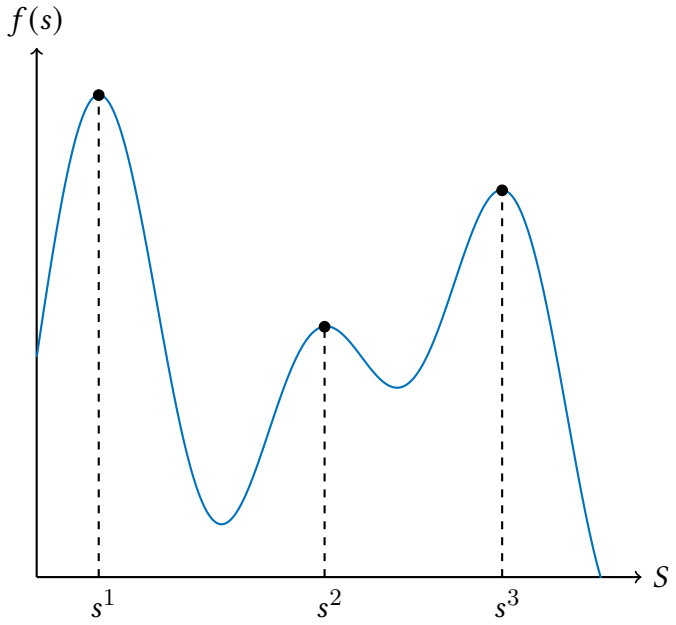

Figure 2.1: Global and Local Optimal Solutions

In practice, the decision to use either a global or local optimization strategy is often influenced by factors such as the available time budget and the preferences of the decision maker. While global optimization aims to find the optimal solution to a problem, the search process may be time-consuming or, in

some cases, computationally infeasible due to the size of the search space. On the other hand, local optimization, while lacking the optimality guarantees of global optimization, is able to quickly generate good solutions that may be acceptable to the decision maker. Nonetheless, the quality of the solutions may be poor due to the ruggedness of the objective function fitness landscape (many local optima). Ultimately, the performance of both methods is closely tied to the knowledge of problem-specific features.

In a setting such as the Hash Code competition, the problems presented are designed to resemble real-world situations. Additionally, due to the time constraints, it is not in the interest of the contestants to use global optimization methods, as they are unlikely to finish on more complex problem instances. Instead, a balance between global and local optimization is typically employed, with contestants utilizing the advantages of both methods. This approach involves establishing a baseline with global optimization methods that explore unseen regions of the search space for potentially good solutions, and then utilizing local optimization methods for exploiting solutions that have already been found.

2.1.3 Black-Box and Glass-Box Optimization

In the field of optimization, two settings are commonly recognized: Black Box Optimization (BBO) and Glass Box Optimization (GBO).

In BBO optimization settings there is no information about the landscape of the function being optimized, constraints defining the set of feasible solutions [1] or the objective function is too complex to be approached from an analytical perspective. As such, algorithms for achieving solutions for these problems do so only by interacting with the problem through the evaluation of potential candidate solutions [3]. Meta-Heuristics, as will be later detailed are examples of methods that follow this approach for finding/improving solutions. By contrast, in Glass Box Optimization (GBO) optimization, also known as “white box” optimization, there is a good understanding of the problem instance being optimized and the objective function properties [3]. Hence, the algorithms used may take advantage of more analytical properties of the problem since they are transparent to the optimizer.

For the purpose of clarification, with regard to the previously mentioned TSP, a black-box strategy would entail the utilization of a search heuristic such as simulated annealing [13] to obtain solutions. This is because the algorithm only necessitates knowledge of how to evaluate the quality of solutions through the objective function and not any specific information about the function being optimized. Alternatively, if the problem were to be formulated as an Integer Linear Programming problem [17, 20], it would become amenable to a white-box approach, as the objective function would be accessible, and additional information about the problem could be inferred from it and provided to the algorithm.

In the context of the Hash Code competition, contestants typically engage with

a problem from a black-box perspective, as the underlying objective function of the problem is not disclosed, and/or is too complex to formalize. Additionally, the process of formalization would be time-consuming and, as a result, the usage of white-box methods post-formalization would not be justified, given that they may be computationally slower. Despite this, it is important to note that white-box methods should not be overlooked as they construct a model of the problem in order to solve it, which could be applied to a certain extent in the context of black-box approaches.

2.2 Optimization Strategies

In the field of optimization, a vast array of algorithms have been extensively documented in the literature for resolving various types of problems. These algorithms can be broadly classified based on their time complexity, the strategy employed to discover solutions, and the quality of solutions obtained. Specifically, algorithms can be classified into three categories, based on optimality of the solution found as exact, approximate, or heuristic. Moreover, there are also general techniques for finding solutions, such as constructive and local search methods. Additionally, the concept of bounds, which serve to guide the search for solutions, is also a crucial factor that can be considered as a strategy to enhance the optimization process.

2.2.1 Exact, Approximation and Heuristic Approaches

Exact approaches are techniques that aim to find the optimal solution for a given problem. These approaches typically involve an exhaustive enumeration and evaluation of solutions belonging to the set $\mathcal{S}$ of feasible solutions. However, in complex real-world problem instances, this may prove to be computationally infeasible. In the context of combinatorial optimization problems, two general exhaustive search strategies are well-known and widely studied: Branch and Bound and Dynamic Programming. These strategies work by successively dividing the problem into smaller dependent or independent sub-problems, and then combining the solutions to these sub-problems to obtain the final solution.

Approximation approaches, in contrast, aim to find solutions that have a guarantee on the quality of the feasible solution being close to the quality of optimal solutions, within an approximation factor ε. Despite these approaches often solving problems in polynomial time and yielding solutions of relatively high quality, it is important to note that they require a mathematical proof of approximation that is specific to the problem at hand. There exists a significant amount of research in this field, particularly for combinatorial optimization problems [10].

Lastly, heuristic approaches work by finding solutions according to a general rule of thumb, the quality of which can be verified through experimentation. These methods do not provide any guarantees of optimality, as they are derived from intuition and their effectiveness is closely tied to the characteristics

of the problem at hand. Nevertheless, they are reliable means of finding solutions in difficult combinatorial optimization (CO) problems. A well-known class of algorithms that utilize this approach are meta-heuristics, as will be further discussed in this work, with examples such as Greedy Randomized Adaptive Search Procedures (GRASP), Simulated Annealing (SA) and Iterated Local Search (ILS).

2.2.2 Constructive Search

Constructive search (CS) is a procedure for optimization where from an empty or partially complete solution for a given problem instance a feasible complete solution is constructed by iteratively adding components extracted from the ground set. The construction process is guided by a pre-established set of rules, which may be heuristic in nature or informed by other relevant information. These rules determine, at each iteration, the set of components that are eligible for inclusion in the final solution.

CS approaches, as such, require certain information about the representation of solutions and specifically, how the addition or removal of a given component or set of components affects the overall quality of the solution, as measured by the objective function value or other metrics such as bounds or dominance relationships between solutions. It is crucial to note that the availability of information about the problem at hand directly impacts the capability of these strategies to optimize.

As an illustrative example, consider the aforementioned KP and TSP from section 2.1.1. In this scenario, an example of a constructive approach for the KP could involve selecting, in each step, the item with the highest ratio of profit to weight that does not exceed the weight capacity of the knapsack. Similarly, for the TSP, a constructive approach would involve selecting an edge to add to the tour that contributes the most, while ensuring that the primary constraint of constructing a Hamiltonian cycle is not violated.

2.2.3 Local Search

In contrast to constructive search, local search (LS) procedures begin with a feasible solution to a given problem instance, and then make small modifications to its components by applying transformations, such as adding or removing elements. These transformations aim to improve the solution within the neighborhood, as defined by the problem's neighborhood structure 2.1.5. The process terminates when further modifications cannot improve the quality of the solution, resulting in a local optimum 2.1.6.

Although the primary objective of the local search algorithm is to improve a solution in the direction of finding a local optimum, as defined in 2.1.6, it is common for such an approach to intentionally worsen a solution in order to allow for further exploration of previously unseen regions of the search space. Furthermore, this approach is frequently applied as a sequence following a constructive search phase, where the constructive search aims to construct

a new solution by exploring the search space, while the local search subsequently exploits the properties of the found solution.

2.2.4 Bounds

The use of bounds is a crucial tool in optimization. Formally, bounds of a (partial) solution s^p can be defined as shown in 2.2.1.

Definition 2.2.1 ((Lower and Upper) Bounds [19, 20]): *A lower bound of a partial solution $s^p \in 2^G$ for an optimization problem $(\mathcal{S}, f)$ is a numeric value given by a function $\Phi_{lb} : 2^{\mathcal{G}} \rightarrow \mathbb{R}$ such that:*

$$\forall s \in \mathcal{S} \wedge s \supseteq s^p : \Phi_{lb}(s^p) \leq f(s) \tag{2.7}$$

On the other hand, an upper bound is a numeric value given by a function $\Phi_{ub} :: 2^{\mathcal{G}} \rightarrow \mathbb{R}$ such that:

$$\forall s \in \mathcal{S} \wedge s \supseteq s^p : f(s) \leq \Phi_{ub}(s^p) \tag{2.8}$$

Particularly, in exact approaches such as Branch & Bound, bounds enable the pruning of the search space, thereby avoiding the exploration of solutions that are guaranteed to not improve the current best solution found by the algorithm at a given time. Furthermore, they provide a guarantee that a solution of inferior quality is not accepted during the search process.

In addition, bounds can also be employed in heuristic approaches to furnish additional information about the potential of a given partial solution s^p. For instance, in a constructive search setup, bounds can enable more effective guidance by providing insight into the best choices for components that can be added to improve a given partial solution.

Despite the objective function's role in guiding the search, the evaluation of a partial solution's quality may not be well-defined for a specific problem, or the objective function may be a bottleneck that does not allow the optimizer to gauge the significance of changes made to the solution's quality.

2.3 Meta-Heuristics

In the literature, the definition of meta-heuristic methods varies across different sources, resulting in a lack of consensus on a formal description [18, 2, 5, 13]. Nonetheless, one commonly accepted definition, provided by Osman and Laporte [18], captures the essence of these methods, which can be defined as iterative generation processes that *"guide and intelligently combine subordinate heuristics for exploring and exploiting solutions in the search space"*.

Given the nature of meta-heuristics as being based on heuristics, they inherit their underlying properties, making them a flexible tool for addressing complex problems that are intractable for exact algorithms in a reasonable amount

of time. Furthermore, the fine-tuning required for meta-heuristics is often specific to the problem at hand and is typically determined through experimental means.

In general, the majority of meta-heuristics can be characterized by the following properties [2]:

- **Search Strategy**: Meta-Heuristics can make use of different strategies for improving candidate solutions. As such, local and constructive approaches are common, with some methods using a combination of both.
- **Memory**: The utilization of memory mechanisms is an important aspect in meta-heuristics, as it allows for the storage of relevant information that can be utilized during the search process, such as previously visited solutions.
- **Population Based and Single-State**: Some meta-heuristics incorporate a learning aspect, thus keeping an archive of solutions (population) that is evolved during the optimization process. On the other hand, single state approaches work by improving a single candidate solution throughout the process.

In the context of this work, a concise examination of several representative meta-heuristic methods is conducted and presented in the subsequent sections. Specifically, we will delve into Hill-Climbing (HC), Iterated Local Search (ILS), Tabu Search (TS), Greedy Randomized Adaptive Search Procedures (GRASP), and Ant Colony Optimization (ACO).

2.3.1 Hill-Climbing

Hill Climbing (HC) methods [13] are single-state, stochastic, local search meta-heuristics that work by iteratively applying small perturbations to an initial candidate solution with the goal of finding a better one. The process updates the best solution s^* found so far at each iteration until a stopping criterion is met. Despite its simplicity, this approach is susceptible to getting trapped in local optima of the objective function. For illustration purposes the pseudo-code of a simple version of an HC algorithm is shown in Algorithm 2.1.

2.3.2 Iterated Local Search

The Iterated Local Search (ILS) [12, 13, 2] is a single-state, stochastic, local search meta-heuristic that builds upon the ideas of Hill Climbing (HC) by incorporating repeated local searches starting from different solutions, allowing for exploration of the search space. In its simplest form, ILS can be implemented according to the pseudo-code shown in Algorithm 2.2. Variants of the ILS algorithm also exist, such as the memory-based, which store previously visited starting solutions to avoid re-exploration of the same regions of the search space.

Algorithm 2.1: Hill-Climbing

Input: Problem Instance ($\mathcal{P}$), Objective Function (f), Stopping Criteria.
Output: Best solution found (s^*).
begin
  $s^* \leftarrow$ CandidateSolution($\mathcal{P}$); ▷ Initial Solution
  while ¬ *stopping criteria met* **do**
    $s' \leftarrow$ Perturb(s^*) ▷ Improvement Attempt
    if $f(s') > f(s^*)$ **then**
      $s^* \leftarrow s'$; ▷ Update Best
    end
  end
  return s^*
end

Algorithm 2.2: Iterated Local Search

Input: Problem Instance ($\mathcal{P}$), Objective Function (f), Stopping Criteria.
Output: Best solution found (s^*).
begin
  $s^* \leftarrow$ CandidateSolution($\mathcal{P}$); ▷ Initial Solution
  while ¬ *stopping criteria met* **do**
    $s \leftarrow$ Perturb(s^*); ▷ Starting Point
    $s' \leftarrow$ LocalSearch(s); ▷ Improve With Local Search
    if $f(s') > f(s^*)$ **then**
      $s^* \leftarrow s'$; ▷ Update Best
    end
  end
  return s^*
end

2.3.3 Simulated Annealing

Simulated Annealing (SA) [11, 16, 13, 2] is a single-state, stochastic, local search meta-heuristic that is inspired by the process of annealing in metallurgy. The algorithm works by iteratively applying small perturbations to a candidate solution with the aim of improving it, and accepting new solutions based on their quality and a probability function that simulates the cooling process of a metal. The probability function is controlled by a temperature parameter, which gradually decreases over time, thus reducing the acceptance probability of worse solutions. This allows the algorithm to initially explore a wider range of solutions and escape local optima, ultimately improving the exploration of the search space. The details of the process can be found in the pseudo-code provided in Algorithm 2.3.

Algorithm 2.3: Simulated Annealing

Input: Problem Instance ($\mathcal{P}$), Objective Function (f), Initial Temperature (T_0), Cooling Rate (α), Stopping Criteria.
Output: Best solution found (s^*).

```
begin
    t ← T_0;                                          ▷ Starting Temperature
    s* ← CandidateSolution(P);                        ▷ Initial Solution
    while ¬ stopping criteria met do
        s' ← Perturb(s*);                             ▷ Small Perturbation
        δ ← f(s*) - f(s');                            ▷ Check Improvement
        if δ < 0 ∨ Random(0, 1) < e^(-δ/t) then
            s* ← s';                                  ▷ Update Best
        end
        t ← t * α;                                    ▷ Update Temperature
    end
    return s*
end
```

2.3.4 Tabu Search

Tabu Search (TS) [8, 7, 13, 2] is a single-state, stochastic, local search meta-heuristic that incorporates the use of memory to guide the search process. The algorithm iteratively explores the solution space by performing neighborhood searches for the best solutions, while maintaining a tabu list $\mathcal{T}$ of recently visited solutions that are temporarily forbidden from being revisited. The main idea behind this approach is that by avoiding previously visited solutions, the algorithm can escape local optima and explore new regions of the search space. The size and duration of the tabu list, as well as the rules for adding and removing solutions from the list, are user-specified parameters that need to be fine-tuned for each problem. The pseudo-code in Algorithm 2.4 illustrates a simple version of this meta-heuristic.

2.3.5 Greedy Randomized Adaptive Search Procedures

Greedy Randomized Adaptive Search Procedures (GRASP) [21, 19, 2] are simple, single-state, stochastic meta-heuristics that iteratively build solutions by combining a greedy construction phase with a local search phase. The construction phase involves the selection of components from the ground set $\mathcal{G}$ based on their contribution to the objective function, using a greedy criterion, and subsequently incorporating them into a partial solution s^p. However, the constructed solution may not be feasible, requiring a repair process to improve its quality. The local search phase is then applied to the partial solution s^p obtained during the construction phase, with the goal of further completing and improving the solution. The pseudo-code provided in Algorithm 2.5 illustrates the functioning of a basic implementation of the GRASP algorithm.

Algorithm 2.4: Tabu Search

Input: Problem Instance ($\mathcal{P}$), Objective Function (f), Tabu List ($\mathcal{T}$), Tabu List Size ($|\mathcal{T}|_{max}$), Stopping Criteria.
Output: Best solution found (s^*).
begin
  $s^* \leftarrow$ CandidateSolution($\mathcal{P}$); ▷ Initial Solution
  $\mathcal{T} \leftarrow \{\emptyset\}$; ▷ Tabu List
  while ¬ *stopping criteria met* **do**
    $s' \leftarrow \text{argmax}_{s \in (\mathcal{N}(s) \setminus \mathcal{T})} f(s)$; ▷ Select Best Neighbor $\notin \mathcal{T}$
    if $f(s') > f(s^*)$ **then**
      $s^* \leftarrow s'$; ▷ Update Best
      $\mathcal{T} \leftarrow \mathcal{T} \cup \{s'\}$; ▷ Add Solution To $\mathcal{T}$
    end
    if $|\mathcal{T}| > |\mathcal{T}|_{max}$ **then**
      RemoveOldest($\mathcal{T}$) ▷ Keep $\mathcal{T}$ Updated
    end
  end
  return s^*
end

Algorithm 2.5: Greedy Randomized Adaptive Search Procedure

Input: Problem Instance ($\mathcal{P}$), Objective Function (f), Stopping Criteria.
Output: Best solution found (s^*).
begin
  while ¬ *stopping criteria met* **do**
    $s^p \leftarrow$ GreedyRandomizedSolution($\mathcal{P}$); ▷ Construction
    if ¬ isFeasible(s^p) **then**
      $s^p \leftarrow$ Repair(s^p); ▷ Repair Solution
    end
    $s' \leftarrow$ LocalSearch(s^p); ▷ Improve/Complete Solution
    if $f(s') > f(s^*)$ **then**
      $s^* \leftarrow s'$; ▷ Update Best
    end
  end
  return s^*
end

2.3.6 Ant Colony Optimization

Ant Colony Optimization (ACO) [4, 19, 13, 2] is a population-based, stochastic, constructive meta-heuristic that is inspired by the foraging behavior of ants.

The algorithm simulates the movement of "ants" through the search space, where each ant constructs a solution by making a sequence of probabilistic choices based on the "pheromone" trail left by previous ants. Notably, the

pheromones, associated with the components c_i of the ground set $\mathcal{G}$, weigh the relevance of the integration of a specific component in a solution during the construction process.

One of the key features of ACO is the incorporation of a learning component through the use of a pheromone update rule that adapts the pheromone trail based on the quality of the solutions constructed by the ants, with the aim of guiding the ants towards better solutions over subsequent iterations. As such, the algorithm requires the tuning of several parameters such as the pheromone evaporation rate, the choice of the pheromone update rule, and the initialization of the pheromone trail.

In summary, the ACO meta-heuristic can be described as a process that comprises of a solution construction phase, in which solutions (referred to as "ants") are constructed, followed by an optional phase of exploiting these solutions through local search, and culminating in a pheromone update phase. This process is then repeated for multiple iterations, as can be observed in the pseudocode provided in Algorithm 2.6.

Algorithm 2.6: Ant Colony Optimization

Input: Problem Instance ($\mathcal{P}$), Population ($\mathcal{A}$), Objective Function (f), Pheromone Update Rule ($\mathcal{R}$), Pheromone Values($\vec{\tau}$), Evaporation Rate (α), Stopping Criteria.
Output: Best solution found (s^*).
begin
  $\mathcal{A} \leftarrow \{\emptyset\}$; ▷ Ant Population
  while ¬ *stopping criteria met* **do**
    $\mathcal{A} \leftarrow$ AntBasedSolutionConstruction($\mathcal{P}$, $\vec{\tau}$)
    $\mathcal{A} \leftarrow$ LocalSearch($\mathcal{A}$); ▷ Optional ("Daemon Actions")
    $s' \leftarrow \text{argmax}_{s \in \mathcal{A}} f(s)$
    if $f(s') > f(s^*)$ **then**
      $s^* \leftarrow s'$; ▷ Update Best
    end
    $\mathcal{A} \leftarrow$ UpdatePheromones($\mathcal{A}$, $\mathcal{R}$, $\vec{\tau}$, α)
  end
  return s^*
end

2.4 Modelling

Modelling refers to the process of creating a simplified representation or approximation of a real-world system, process, or phenomenon in order to improve understanding and facilitate analysis. It is commonly used in the fields of physics and mathematics, where mathematical equations are used to depict reality and capture the important factors of a particular system in a manageable and understandable format [24].

In the field of optimization, there is an extensive body of literature on the development of mathematical programming models for a wide range of problems [20, 17, 23]. One of the key advantages of this approach is that it allows for the application of standard solvers that can be used to find solutions to diverse problems. This can be attributed to the fact that the model encapsulates all the information required by a generic solver to address any problem in a principled manner.

In the field of meta-heuristics, to the best of our knowledge, there is no established concept of a "model". Despite this, there is interest within the community in the development of such models that allow the development of solvers that tackle problems in a black-box fashion. However, there is also some skepticism about the feasibility of this. Nonetheless, if we attempt to identify the characteristics that must be encoded in a model in order for it to be available to a meta-heuristic solver of this kind, certain key aspects must certainly be considered, such as:

- **Instance Parameters**: Description of the problem instance.
- **Decision Space**: Description of the solution and component structure. Moreover, it should detail concepts such as: empty solution, partial solution, complete solution and feasibility.
- **Construction Rules**: Description of how components can be joined to form a feasible solution.
- **Objective Function & Bounds**: Description of how partial or complete solutions could be evaluated.

Therefore, an approach that incorporates these considerations could facilitate a principled approach to the problem, allowing for the use of meta-heuristics as solvers in the same manner as traditional solvers are utilized in mathematical programming.

2.4.1 Frameworks

In the literature, two noteworthy studies that adopt a practical approach to modeling in the context of meta-heuristics are those by Vieira et al. [22] which proposes a Python framework for experimentally testing meta-heuristics and the API developed by Outeiro et al. [19] which builds upon these concepts and applies them to CS approaches. The upcoming sections provide a succinct summary of the most relevant features and contributions of these works.

Python Optimization Framework (POF)

This framework, developed in the context of the work by Vieira et al. [22], is the first to implement the modeling principles for meta-heuristics. It does so by providing an "external" and "internal" interface. The external interface is designed for use by meta-heuristic developers who are interested in the implementation of solvers, while the internal interface is designed for individuals

who want to engage with the framework from a problem-solving perspective and are not interested in the implementation details of the algorithms.

Generally, the POF is implemented by them means of three main classes: Problem, Solver and Simulator:

- Problem: This class is where the modeling-related aspects are implemented. Specifically, the solution generation and evaluation are described by a series of classes and methods that, when implemented, comprise the "model". These classes and methods allow the user to specify how solutions are generated, how they can be modified to improve them, and how they are evaluated, thus constituting the "internal" interface.
- Solver: This class is where meta-heuristics can be implemented in a problem-independent manner. By calling upon the methods defined int the Problem class the development of these algorithms is standardized and constitutes the "external" interface of this framework.
- Simulator: This class servers as a general-purpose utility that allows the step-by-step execution of an implemented Solver with an implemented Problem, thus being responsible for the execution of the algorithms and gathering of solutions.

Furthermore, this work defines several primitives that a model must implement in order to be as generic as possible and be applied to different situations. These primitives include the enumeration of moves, such as the addition or removal of components. These concepts are further refined in the API for constructive search proposed by Outeiro et al. [19].

Not Another Software Framework for Nature-Inspired Optimisation — Constructive Search

In his work, Outeiro et al. [19] builds upon the concepts presented in the POF and an existing implementation of a framework for local search [6], further unifying the concepts and providing both a conceptual model and an implementation of an API (in the C Programming Language) for constructive search — "nasf4nio-cs".

Specifically, this API refines the model definition (the Problem class in the POF) by narrowing down the specifications into a small subset of operations and data structures. These elements, when combined, allow for the complete characterization of a model and the implementation of generic meta-heuristics.

In terms of the data structures, the API defines the following:

- Problem: This data structure is responsible for recording all the problem instance specific features and other relevant information that may be acquired and that pertains to the problem at hand and thus not being changed by the solver in any way
- Solution: This data structure is responsible for storing the data pertaining to a complete/partial solution for a particular problem instance.

- Component: This data structure stores the data relative to a component from the ground set $\mathcal{G}$ that may added, removed, permitted or forbidden w.r.t. a given solution.

With regards to the operations, by focusing on the ones that are specifically utilized for manipulating solutions and disregarding those used for implementation details such as inspection, assignment and memory management, we can categorize them into three primary groups.

- **Generation**: In this category we find operations such as: *emptySolution* and *heuristicSolution* which are used to generate solutions for a given problem.
- **Construction**: Under this category fall operation that allow a partial solution to be further improved constructively. These include the functions *applyMove*, *enumMove*, *heuristicMove*, *heuristicMoveWOR*, *randomMove* and *randomMoveWOR*, which allow for the application, enumeration and selection of moves. In this context, a move refers to the modification of a given solution by performing an action on its components.
- **Evaluation**: In this category appear functions such as *getObjectiveVector* and *getObjectiveLB* which allow for evaluation of the quality of the solution w.r.t. objective value and bounds.

In summary, this API is a comprehensive tool for modeling meta-heuristics in a principled manner. Although it is not fully developed to support local search approaches, it is in a state where it can be easily used and adapted, making it an essential component of our work. The current specification allows for the implementation of meta-heuristic solvers in a problem-independent fashion, adhering to the philosophy proposed by Vieira et al. [22], thus allowing for experimentally assessing the pros and cons of meta-heuristic approaches.

Chapter 3

Approach and Objectives

This chapter presents an overview of the current status and future direction of this work, with a focus on the developments and approaches to be undertaken in the upcoming semester. In Section 3.1, we summarize the overall objective of this work, discussing the progress that has been made, as well as the goals for the future. Section 3.2 provides a detailed timeline of the tasks that have been identified and their scheduling. Finally, in Section 3.3, we offer a succinct evaluation of the challenges encountered throughout the course of this semester.

3.1 Objectives

The primary objective of this project is to research and develop efficient heuristic and meta-heuristic methods for solving optimization problems, with a specific focus on the modeling aspect. By utilizing a modeling approach, we aim to thoroughly understand the problem by capturing as much information as possible, before applying meta-heuristic techniques to find solutions. Additionally, this modeling strategy will allow for the implementation of meta-heuristics in a problem-agnostic manner, which will be applied to a selection of Google Hash Code problems, as outlined in the scope of this work.

During the first semester, a survey of several Google Hash Code problems was conducted, and one problem was subsequently chosen for further analysis. The methodology and results of this preliminary work are presented in detail in the following chapter.

In alignment with the objective, a comprehensive review of relevant literature and existing frameworks was conducted. Based on the principles established in the literature review, the chosen problem was then modeled, and simple algorithms/utilities were developed to facilitate the verification of results. However, the analysis of the current problem is not yet complete, and further enhancement of the results obtained with the developed model is possible from both an implementation and conceptual perspective. This work will be continued during the early stages of the second semester.

Additionally, we aim to select two more Google Hash Code problems for which models will be developed. Furthermore, several state of the art meta-heuristics will be implemented in order to experimentally evaluate the performance of the models and to provide insight into the advantages and limitations of this approach from both a conceptual and implementation perspective.

As the problem tackled during the first semester, as well as the additional problems to be tackled in the future, will also be addressed through the usage of an API [19], this work has required significant investment in the learning and utilization of said API. While the API is generally complete, certain aspects require further refinement, and the local search paradigm has yet to be fully integrated into it, as it was primarily designed for constructive search. This represents a secondary objective of our work, which will be pursued in parallel to our efforts to model the problems in a principled manner.

3.2 Timeline

Given the objectives outlined in Section 3.1, several tasks were identified and are presented in the following diagram along with their corresponding timeline:

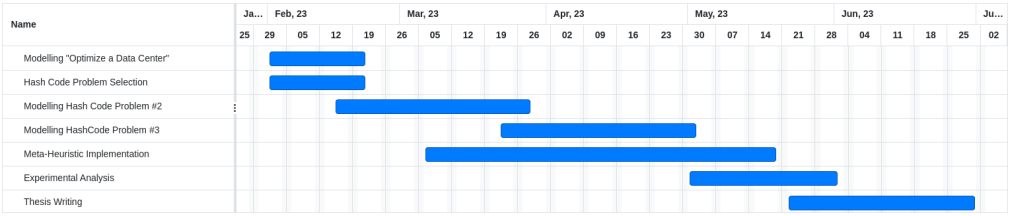

Figure 3.1: Project Timeline

- **Modelling "Optimizing a Data Center"**: Finalize and fine-tune the implementation of the model for this problem.
- **Google Hash Code Problem Selection**: Complete the survey of the Google Hash Code problems and select two problems for further analysis.
- **Modelling Google Hash Code Problem #2**: Develop a model for the first selected problem.
- **Modelling Google Hash Code Problem #3**: Develop a model for the second selected problem.
- **Meta-Heuristic Implementation**: Implement state-of-the-art meta-heuristics to test the approach of separating model development from solution development and to obtain competitive solutions for the modeled problems.
- **Experimental Analysis**: Conduct a thorough experimental analysis of the results obtained for all problems.

- **Thesis Writing**: Compose the final thesis report.

3.3 Concluding Remarks

The previous sections (Section 3.1 and Section 3.2) have outlined the current approach, current work, objectives, and tasks to be undertaken in the remaining time budget of this project. However, it is important to note that there have been some challenges encountered in the work already developed, and that are worth mentioning

From a personal perspective, the shift in paradigm towards a more principled approach to modeling presented a significant learning curve, as it is a concept that was not previously part of my problem-solving toolkit. This may have been due to the novel nature of the approach, which required a thorough understanding and familiarization of the underlying concepts.

Additionally, utilizing the API [19] required a significant investment in understanding its implementation and quirks that must be considered when implementing a model. Furthermore, the API was developed in C, while this work was conducted in C++, which required some adjustments to the existing code. The decision to use C++ was motivated by its familiarity and ease of use, as well as the availability of a standard library of support data structures, which was identified as a significant improvement that would aid in the development of models.

Lastly, in regards to the development of the API, it should be noted that the author [19] has implemented certain algorithms in a problem-independent manner. While these algorithms are functional, they still require certain components of the model to be fully implemented. Despite this, it is worth noting that this experience has highlighted the importance of clearly documenting the required operations for a model to be able to utilize a particular solver, which we intend to keep in mind as we develop our meta-heuristic solvers

Chapter 4

Preliminary Work

4.1 The Hash Code Problems

The Hash Code competition problems are modeled on complex engineering challenges faced by Google. Given the scale of the enterprise, it is evident that they represent complex engineering challenges that mirror real-world scenarios prevalent in the industry. As such, these problems not only function as valuable benchmark problems, but also provide an opportunity to establish a robust baseline for the development and testing of modeling based metaheuristic approaches. This is because numerous individuals with diverse backgrounds have attempted to solve these problems, resulting in readily accessible scores and solutions.

When examining the available information about the solutions for the problems, it is essential to take into consideration a few key points. Firstly, since only a select number of teams from the qualification round advance to the finals, the number of solutions available tend to be limited. However, it is reasonable to expect that the solutions found by the top teams will be excellent representatives of the best problem-solving processes, as they have undergone the selection process. Secondly, in the early years of the competition, it was only open to teams from Paris. In the subsequent three years, it was open to teams from Europe, Africa, and the Middle East before becoming a worldwide competition. Therefore, it is expected that there will be an increase in the number of results available in the later years and more challenging problems due to the increase in competition.

In order to fulfill the objectives of this work, it is crucial to gain a thorough understanding of the problem at hand before proceeding with the formulation of the model. This includes identifying the objective, classifying the problem type, understanding the constraints, and most importantly, determining its relationship to problems previously studied in literature. Through this analysis, valuable insights into the appropriate approach can be gained. Therefore, a comprehensive analysis of the Hash Code problems was conducted and a summary of the key aspects of each problem evaluated in this first semester is presented in the following sections

4.1.1 Google Hash Code 2014

Final Round: "Street View Routing"

In the context of constructing street view maps there is a need to collect imagery that is taken by specialized vehicles equipped for that purpose. This constitutes a challenging problem since given a fleet of cars which may only be available for a limited amount of time a route for each must be defined as to maximize the number of streets photographed. City streets are modelled as a graph where nodes are junctions and the edges are streets connecting said junctions. Moreover, streets are defined by three distinct properties: direction, length and cost that will take for the car to traverse the street.

The challenge consists of scheduling the routes for street view cars in the city, adhering to a pre-determined time budget. The goal is to optimize the solution by maximizing the sum of the lengths of the traversed streets, while minimizing the overall time expended in the process. The quality of the solution for this problem is evaluated by using the sum of the lengths of the streets as the primary criterion and the time spent as a tie-breaker.

The problem at hand bears a strong resemblance to a combination of the Vehicle Routing Problem and the Maximum Covering Problem. This is because the scheduling of routes for the fleet of cars must be done in a way that ensures that the combination of all sets of streets visited by each car encompasses the entire city, in the most time-efficient manner possible.

4.1.2 Google Hash Code 2015

Qualification Round: "Optimize a Data Center"

The optimization of server placement problem is a concern that pertains to the design of data centers, as various factors must be taken into account to ensure optimal efficiency. In this context, the 'optimizing servers" problem portrays a scenario in which contestants are in the position of designing a data center and seeking to determine the optimal distribution of servers. The data center is physically organized in rows of slots where servers can be placed. Hence, the challenge is to efficiently fill the available slots in a Google data center with servers of varying sizes and computing capacities, while also ensuring that each server is assigned to a specific resource pool.

Objectively, the goal is to assign multiple servers to available slots and resource pools in such a way as to maximize the guaranteed capacity for all resource pools. This metric serves as the criterion for evaluating solutions to this problem. The guaranteed capacity, in this context, refers to the lowest amount of computing power that will remain for a specific resource pool in the event of a failure of an arbitrary row of the data center. It is important to note that this objective function is considered a bottleneck, as small changes in a solution may not result in significant changes in the score, making the optimization process more difficult.

Notably, the problem of optimizing the placement of servers in a data center

can be thought of as a combination of a Multiple-Knapsack Problem and an assignment problem. This is because the servers must be placed within the constraints of the available space in the data center rows, and subsequently, they must be assigned to resource pools.

Final Round: "Loon"

Project Loon, which was a research endeavor undertaken by Google, aimed at expanding internet coverage globally by utilizing high altitude balloons. The problem presented in this competition drew inspiration from this concept, requiring contestants to devise plans for position adjustments for a set of balloons, taking into consideration various environmental factors, particularly wind patterns, with the objective of ensuring optimal internet coverage in a designated region over a specific time frame.

The objective of this problem was to develop a sequence of actions, including ascent, descent, and maintaining altitude, for a set of balloons with the goal of maximizing a score. In this case, the score is calculated based on the aggregate coverage time of each location, represented as cells on a map of specified dimensions, at the conclusion of the available time budget.

In summary, this problem can be classified as both a simulation and a coverage and routing problem, based on the properties previously described. It is important to note that the simulation aspect of this problem has a direct impact on the calculation of the score, and is not solely limited to constraints on the available time budget for operations. Furthermore, this problem can be represented in a forest, where the vertices represent spatiotemporal coordinates (x, y, z, t), and the edges symbolize changes in altitude and lateral movement (wind) for a given balloon.

4.1.3 Google Hash Code 2016

Qualification Round: "Delivery"

In today's world, with the widespread availability of internet, online shopping has become a prevalent activity. As a consequence, there is an ever-growing need for efficient delivery systems. This competition challenges participants to manage a fleet of drones, which are to be used as vehicles for the distribution of purchased goods. Given a map with delivery locations, a set of drones, each with a set of operations that can be performed (load, deliver, unload, wait), a number of warehouses, and a number of orders, the objective is to satisfy the orders in the shortest possible time, taking into consideration that the products to be delivered in an order may have product items stored in multiple different warehouses and therefore require separate pickups by drones.

In this problem, the simulation time $\mathcal{T}$ is given and the goal is to complete each order within that time frame. The score for each order is calculated as $\frac{(\mathcal{T}-t)}{\mathcal{T}} \times 100$, where t is the time at which the order is completed. The score ranges from 1 to 100, with higher scores indicating that the order was completed sooner.

The overall score for the problem is the sum of the individual scores for all orders, and it is to be maximized.

In summary, this problem can be classified as a variant of the Vehicle Routing Problem, specifically as a Capacitated, Pickup and Delivery Time Windowed Multi-Depot Vehicle Routing Problem. This classification takes into account the pickup and delivery of items, the time window for delivery, the multiple routes and warehouses that each vehicle may need to visit in order to fulfill the orders.

4.1.4 Outline

In summary, this section provided an overview and description of the key aspects of the Hash Code problems studied in the first semester. Furthermore, a categorization that links these problems to topics commonly found in combinatorial optimization literature was presented. The table 4.1 shows a summary of the analysis conducted.

Problem	Categories				
	Assignment	Knapsack	Coverage	Vehicle Routing	Simulation
Street View Routing			✓	✓	
Optimize a Data Center	✓	✓			
Loon			✓	✓	✓
Delivery				✓	

Table 4.1: Categorization of Google Hash Code Problems

4.2 Modelling “Optimize a Data Center”

In this section, we will analyze the model developed for the Hash Code 2015 qualification round problem, titled ”Optimizing a Data Center.” To begin, in Section 4.2.1, we will provide a comprehensive description of the problem and key concepts outlined in the problem statement, which are essential for understanding the context. In Section 4.2.2, we will delve into the model developed for this problem, highlighting the representation, objective function, bounds, and heuristic construction of solutions from both a theoretical and practical perspective, taking into account the framework proposed by Vieira et al.[22] and the practical implementation contained in the API for constructive search by Outeiro et al.[19]. Finally, in Section 4.2.3, we will present a brief overview of the results and in Section 4.3, we will offer some final thoughts and observations on the problem and the model developed for it.

4.2.1 Problem Description

As previously discussed in Section 4.1.2, the problem at hand entails optimizing the placement of servers in a data center. The data center is modeled as a series of rows, each containing a number of slots in which servers can be placed. However, it should be noted that certain slots may be unavailable due

to other installations within the data center. The servers available are characterized by a tuple containing both size, measured in slots, and computing capacity. Furthermore, servers are logically divided into pools, to which they can contribute their capacity, with the capacity of the pool being defined as the sum of the capacities of the servers assigned to it.

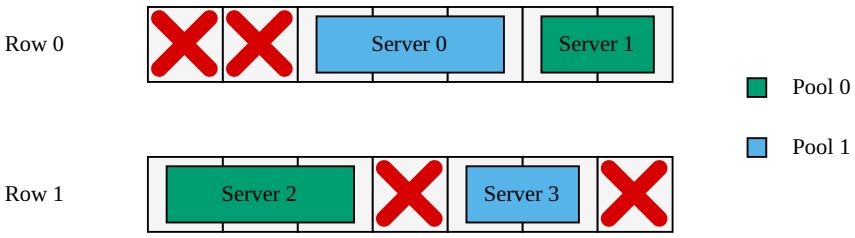

Figure 4.1: Example Data Center Layout

Server	Size	Capacity
0	3	2
1	2	5
2	3	10
3	2	3

Table 4.2: Server Properties

As an illustration the figure 4.1 provides an overview of a possible data center layout. In this particular case the data center is composed by 2 rows containing a total of seven slots each where in both cases only 5 are available for positioning of servers. Moreover the servers placed were assigned to two distinct resource pools.

Definition 4.2.1 (Guaranteed Capacity): *Given a resource pool p, the guaranteed capacity gc_p is a measure of the remaining computing capacity available in the event that at most one arbitrary row ($r \in \mathcal{R}$) of the data center becomes offline. This can be formally defined as follows:*

$$gc_p = \min_{r \in \mathcal{R}} \left(\sum_{s \in p} c_s - \sum_{s \in p \wedge s \in r} c_s \right) \tag{4.1}$$

Given the representation of a possible layout of a data center, the objective of this problem can be succinctly defined as finding an optimal placement of servers in the rows and assigning them to pools, with the goal of maximizing the guaranteed capacity 4.2.1, of all resource pools present in the data center. Formally, this objective can be described by the equation shown in 4.2.2.

Definition 4.2.2 (Objective Function): *Given a set of pools $\mathcal{P}$, the objective function for an instance of this problem is defined as:*

$$f(s) = \min_{p \in \mathcal{P}} (gc_p) \tag{4.2}$$

The objective function value resulting from the evaluation of a solution is, from a contestant’s perspective, the score that will be obtained for solving a particular problem instance. As an example, consider the data center layout shown in Figure 4.1 and the server properties displayed in Table 4.2.

Pool	Row 0	Row 1	Guaranteed Capacity
0	5	10	5
1	2	3	2

Table 4.3: Guaranteed Capacities For Each Pool In Example 4.1

In this scenario, the score of the server placement, as determined by the evaluation of the objective function 4.2.2, would be $\min(5, 2) = 2$. The table shown in 4.3 illustrates the calculation of the guaranteed capacities for this example.

Regarding the instances, the problem statement guarantees that the generated instances along with the description of the unavailable slots and the servers to be placed will have the following properties:

- $1 \leq \mathcal{R} \leq 1000$: Denotes the number of rows in the data center.
- $1 \leq \mathcal{S} \leq 1000$: Denotes the number of slots in a row.
- $0 \leq \mathcal{U} \leq \mathcal{R} \times \mathcal{S}$: Denotes the number of unavailable slots.
- $1 \leq \mathcal{P} \leq 1000$: Denotes the number of resource pools to be created.
- $1 \leq \mathcal{M} \leq \mathcal{R} \times \mathcal{S}$: Denotes the number of servers to be allocated.

On a final note, it is required that contestants generate a solution that includes an ordered list of servers, specified by the problem instance, where each server is assigned a row, pool, and starting slot (position) in the selected row. This information is critical to the modeling process, as it affects the representation of the problem, solutions, and components mainly from an implementation perspective.

4.2.2 Model

In this section, we will proceed to define the model for this problem. Specifically,we begin by addressing key aspects such as the instance parameters and the representation of this problem. Subsequently, we will delve into the representation of solutions, the ground sets, and the components of this problem. We will then present the objective function, giving particular attention to how solutions are evaluated within the context of our chosen representation. We will also explore the bounds established for this problem. Finally, we will discuss the heuristics and heuristic rules that are utilized to construct solutions.

It should be noted that the effort of modeling will be formulated within the context of the API for constructive search developed by Outeiro [19]. As such, the modeling will be described using the concepts and terminology specified in said work.

Instance Parameters

For the purpose of capturing the relevant information pertaining to the problem at hand, we initiated the process by defining the Problem data structure. This structure, in the context of modeling, encompasses the essential attributes of the instance that are required for, among other things, the construction of solutions. However, prior to specifying the data that was stored for the problem, it is necessary to provide clarification regarding the terms “segment” and “server”.

During the modeling phase, we identified that the rows of the data center could not be considered as a sort of “standard knapsack”, where placing a server would simply encompass the consumption of available slots. This is because the presence of unavailable slots in the row may prevent a server with a given size from being placed. Thus, a row having sufficient space to accommodate a server is not a sufficient requirement for its placement. Instead, we can view each row of the data center as a collection of multiple knapsacks, where each group of consecutive available slots is considered as a separate knapsack, which we refer to as a segment.

From a modelling perspective, a segment can be then be described as a tuple containing three defining fields. The first is the *size*, which pertains to the number of available slots. The second is the *row*, which obviously represent the row where that segment belongs. And last, the the *start*, which represents the starting slots (position) for the segment in that rows since it useful in the solution construction.

Furthermore, as previously mentioned, a server has two properties: the *size* and the *capacity*. From a modeling perspective, these are the fields of a tuple that defines the server, with the added *id* that serves as an indexing reference in our representation.

Having defined the terminology, our solution structure is comprised of the following fields:

- **pools**: A numerical value indicating the quantity of resource pools to be created.
- **servers**: An list of servers to be allocated to segments and resource pools.
- **sorted (servers)**: A list containing, indexes of the servers sorted by an heuristic (ratio between the size and capacity of the server). This is used in the solution-building process.
- **segments**: An list of segments within the data center, to be populated with servers.
- **rows**: An list of sets, each of which records the segments associated each row.

Decision Space

Having defined the problem the next step in the modelling process is to define the decision space. Namely the components and the solutions.

Given that the objective of this problem is to construct a data center layout by positioning servers in rows and assigning them to resource pools, it can be inferred that a solution will consist of a list of objects (components) that record this information. As such the component structure for this problem can be defined as follows by the following properties.

- **server**: The index (as represented in the ”servers” list) of the selected server.
- **segment**: The index (as represented in the ”segments” list) of the segment where the selected server will be placed.
- **pool**: The resource pool to which the selected server is to be assigned.

Thus, the ground set $\mathcal{G}$ of the problem becomes defined as the set containing tuples that represent all possible combinations of these properties.

In regards to the representation of solutions, it is important to note that in this problem, a partial solution, which is defined as a solution in which not all available servers have been assigned, is a feasible solution that can be properly evaluated through the objective function. With that in mind, the model represents solutions as structures containing the following properties:

- **score**: The value resulting from the evaluation of this solution through the objective function.
- **allocation**: A list of components mapping a server to a given pool and segment.
- **gc**: A list containing the guaranteed capacities of each resource pool.
- **capacity**: A matrix containing the capacities of each resource pool for each row of the data center.
- **sc**: The remaining space in each segment.

Additional properties, such as iterators and collections containing items ordered by multiple criteria, are also stored in the solution. This is done in order to maintain the state of the solution and to cache results from previous computations, thereby improving performance. However, these properties are not included in the present discussion, as they are not essential for a proper understanding of the modeling process and are considered to be implementation details.

Objective Function

As previously discussed in Section 4.2.1, the objective function for this problem can be calculated in a similar fashion by determining the minimum value within the range of values contained in the list designated as “gc”. The method

of constructing the solutions ensures that this list is continuously updated throughout the construction process. However, if the solution remains unmodified, the objective function value for a given solution can be accessed efficiently through the ”score” attribute defined in the solution’s model.

Bound

In contrast to the objective function, which serves as a metric for the quality of a solution within a given context, the bounds provide insight into the potential of a solution in the future.

In the context of this problem, where the objective is to address a bottleneck (objective function), it is intuitive that achieving optimal solutions requires distributing the servers for a given pool as evenly as possible across the rows of the server. This is because in the event of a failure, it would be detrimental if all the servers for a pool were located in the same row that failed. Therefore, this factor was taken into consideration when formulating the bounds.

Additionally, certain relaxations were made to the constraints of the problem when calculating the bound. Specifically, the bound disregards the requirement that the servers must fit within the total space available in the segments, and only takes into account the available space in the slots for each row and the size of the server to be placed.

Taking this into account, the upper bound for this model can be calculated in two steps:

1. **Row-Wise Bound**
   Let,

   - $\Theta_{\mathcal{R}\setminus r}$: Denote the remaining empty space in all the rows in the data center but the row r.
   - $\sum_{\Theta_{\mathcal{R}\setminus r}}$: Denote the maximum sum of the capacities of the available servers that can be fractionally placed into $\Theta_{\mathcal{R}\setminus r}$ w.r.t to the ratio between the capacity and the size of the server. This related to with a classical upper bound for the multiple-knapsack problem [14].

   Then, The row-wise upper bound is expressed by:

$$\Phi^{r}_{ub} = \frac{\sum_{\Theta_{\mathcal{R}\setminus r}}}{\mathcal{P}} \tag{4.3}$$

2. **Upper bound**
   The upper bound for a given solution s can then be calculated as:

$$\Phi_{ub}(s) = \min_{r\in\mathcal{R}} \Phi^{r}_{ub} \tag{4.4}$$

Furthermore, we can apply a correction to tighten each row-wise upper bound by discarding pools and servers that already have a guaranteed capacity, when dropping that row, greater than the value of the row-wise upper bound. After

discarding them, we can recompute the bound for a reduced server set and number of pools. We repeat this process until no further correction can be applied, i.e. until no considered pool has a guaranteed capacity greater than the bound.

Construction Rules

Having identified and defined all of the necessary elementary components of the model, the next step is to establish the methodology for integrating these components into functional solutions. To accomplish this, there are certain key concepts that must be addressed, such as enumeration and assignment/removal.

In the proposed model, enumeration is achieved through the utilization of the *enumMove* and *heuristicMoveWOR* operations. These operations function similar to iterators, continuously providing components that could be added to a solution. The specific order in which these components are presented, for each operation within the context of this model, is defined as follows:

- ***enumMove***
  This operation allows for the enumeration of all feasible components in the context of this problem. A feasible component could be interpreted as a component that can be added to a solution without violating the constraints that in this case are related to the available space for the placement of server given that there are no restrictions with respect to the assignment to resource pools.
- ***heuristicMoveWOR***
  The operation in question enables the sequential enumeration of all components in a specific order, determined by a heuristic. This order is consistent with that specified in the *heuristicMove* function, which enables the selection of components to be added based on a predefined heuristic criterion. Additionally, when a component is yielded during the iteration, it is no longer eligible for further consideration, hence the inclusion of “WOR” in the operation name, indicating that the enumeration is conducted “without replacement”.

  The heuristic employed in this model can be succinctly characterized as a strategy in which the server with the most favorable capacity-to-size ratio is assigned to the pool with the least guaranteed capacity in the row that possesses the highest available space.

In regards to the assignment and removal of components to and from solutions, this is accomplished through the implementation of the *applyMove* operation. In the context of this model, the addition of a component to a solution, denoted as “ADD’‘, involves decreasing the available space within the segment (referred to as *sc*) in which the server is placed by the size of the server, and updating the guaranteed capacity of the associated pool (referred to as *gc* and *capacity*) to reflect the inclusion of the server. Conversely, the removal of a component, denoted as “REMOVE”, follows a similar process but in the oppo-

site direction, returning the capacity to the segment from which it was originally allocated.

4.2.3 Results

From a practical perspective, the limited availability of instances for testing was a limitation of our work. However, we were able to achieve a score of 386 points on the instance that was available, which places us at 25th in the classification table. This was achieved without using a local search approach and with minimal fine-tuning of the algorithms.

In fact, the algorithm used was a simple testing utility that, during the solution construction process, enumerates the n best component insertions (moves) in a heuristic manner, and selects and stores the best one based on the calculated value of the bound. This utility can be considered general with respect to the model, as it was implemented in adherence to the principles of modeling. For clarity, the pseudo-code in Algorithm 4.1 illustrates its operation.

Algorithm 4.1: Narrow Guided Heuristic Construction

```
Input: Problem Instance (𝒫), Limit (N)
Output: Best solution found (s*).
begin
    s* ← ∅
    while True do
        updated ← False
        s' ← s*
        c* ← ∅
        for i = 0 to N do
            c' ← HeuristicMoveWOR(s*, ADD)
            ApplyMove(c', s', ADD)
            if Φ_ub(s') > Φ_ub(s*) then
                c* ← c'
                updated ← True
            end
            ApplyMove(c', s', REMOVE)
        end
        if ¬ updated then
            return s*;
        end
        ApplyMove(c*, s*, ADD))
    end
end
```

4.3 Concluding Remarks

In this chapter, we presented the preliminary work developed during this semester. We began by conducting an initial survey of the Hash Code problems, and categorizing them from a modeling perspective. This survey will be completed in the near future. From this survey, the ”Modelling a data center problem” was selected and modeled in a principled fashion.

In general, we achieved decent results. However, it is worth noting that the limited number of available instances presented a challenge for both testing and verifying our results. Additionally, the competition being restricted to a specific region (Paris) resulted in a lower number of participants attempting the problem than we had anticipated. Furthermore, our research yielded few references or solutions from other sources that could provide insight into achieving better scores, likely due to the problem being relatively old.

Given the limited availability of instances for testing, it was suggested during a discussion to explore the use of an exact method for solving this problem and comparing the results to our approach. The problem as defined in section 4.2.1 has relatively small constraints, which may allow for the generation of instances that can be solved by an exact method within a feasible amount of time. It would be informative to compare the results obtained by the exact method with our approach, in order to determine the quality of the solution reached through our methodology.

Lastly, it is worth noting that the results obtained for the analyzed instance can be further optimized. This is an objective that we aim to achieve in the upcoming semester.

Chapter 5

Conclusion

This document has highlighted the crucial elements of our research during the current semester. It has provided a motivation for the problem being addressed, recurring concepts in this area of study, a comprehensive review of relevant state-of-the-art methods and frameworks, and an examination of the progress made, obstacles encountered, and future directions of our work.

From our perspective, the research conducted thus far has yielded valuable insights into the potential of this approach, despite the challenges associated with the development of models and formalization of concepts. Additionally, the progress made has been productive, and the results obtained provide motivation to continue our work in the upcoming semester.

References

[1] Stéphane Alarie et al. "Two Decades of Blackbox Optimization Applications". In: *EURO Journal on Computational Optimization* 9 (Jan. 1, 2021), p. 100011. ISSN: 2192-4406. DOI: 10.1016/j.ejco.2021.100011.

[2] Christian Blum. "Metaheuristics in Combinatorial Optimization". In: *ACM Computing Surveys* 35.3 ().

[3] Carola Doerr. "Complexity Theory for Discrete Black-Box Optimization Heuristics". In: *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*. Ed. by Benjamin Doerr and Frank Neumann. Natural Computing Series. Cham: Springer International Publishing, 2020, pp. 133–212. ISBN: 978-3-030-29414-4. DOI: 10.1007/978-3-030-29414-4_3.

[4] Marco Dorigo and Thomas Stützle. "Ant Colony Optimization: Overview and Recent Advances". In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. International Series in Operations Research & Management Science. Boston, MA: Springer US, 2010, pp. 227–263. ISBN: 978-1-4419-1665-5. DOI: 10.1007/978-1-4419-1665-5_8.

[5] P. Festa. "A Brief Introduction to Exact, Approximation, and Heuristic Algorithms for Solving Hard Combinatorial Optimization Problems". In: *2014 16th International Conference on Transparent Optical Networks (ICTON)*. 2014 16th International Conference on Transparent Optical Networks (ICTON). July 2014, pp. 1–20. DOI: 10.1109/ICTON.2014.6876285.

[6] Carlos M. Fonseca. *Nasf4nio*. Oct. 27, 2021. URL: https://github.com/cmfonseca/nasf4nio (visited on 01/15/2023).

[7] Michel Gendreau and Jean-Yves Potvin. "Tabu Search". In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. International Series in Operations Research & Management Science. Boston, MA: Springer US, 2010, pp. 41–59. ISBN: 978-1-4419-1665-5. DOI: 10.1007/978-1-4419-1665-5_2.

[8] Fred Glover and Manuel Laguna. *Tabu Search I*. Vol. 1. Jan. 1, 1999. ISBN: 978-0-7923-9965-0. DOI: 10.1287/ijoc.1.3.190.

[9] J.-B. Hiriart-Urruty. "Conditions for Global Optimality". In: *Handbook of Global Optimization*. Ed. by Reiner Horst and Panos M. Pardalos. Nonconvex Optimization and Its Applications. Boston, MA: Springer US, 1995, pp. 1–26. ISBN: 978-1-4615-2025-2. DOI: 10.1007/978-1-4615-2025-2_1.

[10] David S. Johnson. "Approximation Algorithms for Combinatorial Problems". In: *Journal of Computer and System Sciences* 9.3 (Dec. 1, 1974), pp. 256–278. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(74)80044-9.

[11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (May 13, 1983), pp. 671–680. DOI: 10.1126/science.220.4598.671.

[12] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. "Iterated Local Search: Framework and Applications". In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. International Series in Operations Research & Management Science. Boston, MA: Springer US, 2010, pp. 363–397. ISBN: 978-1-4419-1665-5. DOI: 10.1007/978-1-4419-1665-5_12.

[13] Sean Luke. *Essentials of Metaheuristics*. Second Edition. 2013.

[14] Silvano Martello and Paolo Toth. "A Bound and Bound Algorithm for the Zero-One Multiple Knapsack Problem". In: *Discrete Applied Mathematics*. Special Copy 3.4 (Nov. 1, 1981), pp. 275–288. ISSN: 0166-218X. DOI: 10.1016/0166-218X(81)90005-6.

[15] Rafael Martí, Mauricio G. C. Resende, and Celso C. Ribeiro. "Multi-Start Methods for Combinatorial Optimization". In: *European Journal of Operational Research* 226.1 (Apr. 1, 2013), p. 2. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2012.10.012.

[16] Alexander G. Nikolaev and Sheldon H. Jacobson. "Simulated Annealing". In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. International Series in Operations Research & Management Science. Boston, MA: Springer US, 2010, pp. 1–39. ISBN: 978-1-4419-1665-5. DOI: 10.1007/978-1-4419-1665-5_1.

[17] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. ISBN: 978-0-387-30303-1. DOI: 10.1007/978-0-387-40065-5.

[18] Ibrahim H. Osman and Gilbert Laporte. "Metaheuristics: A Bibliography". In: *Annals of Operations Research* 63.5 (Oct. 1, 1996), pp. 511–623. ISSN: 1572-9338. DOI: 10.1007/BF02125421.

[19] Samuel Barroca do Outeiro. "An Application Programming Interface for Constructive Search". Msc Thesis. University of Coimbra, Portugal, Nov. 9, 2021.

[20] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Courier Corporation, Jan. 1, 1998. 530 pp. ISBN: 978-0-486-40258-1.

[21] Mauricio G.C. Resende and Celso C. Ribeiro. "Greedy Randomized Adaptive Search Procedures: Advances, Hybridizations, and Applications". In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. International Series in Operations Research & Management Science. Boston, MA: Springer US, 2010, pp. 283–319. ISBN: 978-1-4419-1665-5. DOI: 10.1007/978-1-4419-1665-5_10.

[22] Ana Vieira. "Uma plataforma para a avaliação experimental de metaheurísticas". Doctoral Thesis. University of Algarve, Portugal, 2009.

[23] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. 1st ed. Cambridge University Press, Apr. 26, 2011. ISBN: 978-0-511-92173-5. DOI: 10.1017/CBO9780511921735.

[24] Thomas Witelski and Mark Bowen. *Methods of Mathematical Modelling*. Springer Undergraduate Mathematics Series. Cham: Springer International Publishing, 2015. ISBN: 978-3-319-23041-2. DOI: 10.1007/978-3-319-23042-9.

[25] "Combinatorial Optimization". In: *Introduction to Evolutionary Algorithms*. Ed. by Xinjie Yu and Mitsuo Gen. Decision Engineering. London: Springer, 2010, pp. 263–324. ISBN: 978-1-84996-129-5. DOI: 10.1007/978-1-84996-129-5_7.