Pedro Miguel Duque Rodrigues

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

September 2023

Abstract

Acknowledgments

Contents

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

"Begin at the beginning", the King said gravely, "and go on till you come to the end; then stop."

— Lewis Carroll

1.1 Motivation

Optimization problems are ubiquitous in real-world scenarios. For example, when considering the task of planning a road trip, we are quickly faced with several optimization challenges arising from a seemingly simple task, such as, finding the shortest or cheapest route, and efficiently packing luggage. When solving these problems the goal is typically to find "good" solutions in a time-efficient manner.

Tackling an optimization problem can typically be seen as a two-phase process. First, we start by understanding the problem and modeling its details. Then, we can apply, or develop, a solver to find one or more solutions taking into account the given model. This approach serves, for example, as the foundation of most linear optimization software packages which are widely used to solve real-world problems, *e.g.*, *Gurobi*, *CPLEX*, or *GLPK*. In particular, such packages expect a mathematical formulation (model) describing the problem as a linear objective function and a set of linear constraints, and then use one or more algorithms designed to solve such linear optimization problems. This clear separation of concerns leads to some advantages. Notably, practitioners who want to solve a particular problem can focus on developing the model for that problem and easily use existing solvers to find solutions without needing to implement state-of-the-art algorithms themselves. Meanwhile, solver developers can take

advantage of existing problems to test and enhance their solvers' performance.

In this work, we are interested in tackling Combinatorial Optimization (CO) problems using a similar separation of concerns. Generally speaking, CO consists of finding an optimal solution, according to some objective function, from a discrete set of solutions. Several generic approaches have been developed to solve CO problems exactly, *i.e.*, to find an optimal solution. However, many CO problems are NP-Hard, meaning that, the time required to solve them via (problem-specific) exact approaches grows exponentially with the problem size, and consequently generic exact methods will also grow exponentially. In practice, this means that exact methods are often ineffective to solve "real-world" CO problems which have large problem sizes.

As a result, there has been a growing interest in the development of methods that can find "good" solutions for such problems. In this work, we focus on heuristic and meta-heuristic methods. Heuristic methods are search procedures, often problem-specific, that attempt to quickly solve a problem and provide a "rule of thumb" for attaining decent solutions, albeit without optimality guarantees. Meta-Heuristic (MH) methods employ several high-level strategies to construct and improve solutions. It is worth noting that such high-level strategies often depend on problem-specific details, *e.g.*, the neighborhood structure and search tree definition. However, meta-heuristics do not require knowledge about these problem-specific details and instead use the high-level strategies in a black-box fashion. As such, MH approaches are problem-independent and can be applied to a broad range of problems.

Given the nature of MH methods and the inherent diversity of problems, crafting universal MH solvers is a challenging task made harder due to the difficulty in separating the problem-specific details required by the high-level strategies from the MH problem-independent solving process. In fact, the abundance of MH optimization software that provides specific frameworks for implementing evolutionary, local or constructive search meta-heuristics for CO problems [13, 12, 23] and the lack of a unifying framework supporting all approaches can be regarded as a symptom of the difficulty of this endeavor. Still, it is worth remarking the works by Vieira [15] and Outeiro [35], which partially looked at the formalization of this objective.

The development of a unifying framework would standardize problem-solving approaches, facilitate the reuse of MH methods, and distinctly separate the tasks of problem modelling and solver development. Moreover, it would provide researchers and practitioners with a valuable tool to experimentally assess the performance of MH methods across a range of diverse problems.

Simultaneously, alongside the development and application of MH strategies to address CO problems, there exists a community interest in constructing a collection of benchmark optimization problems that hold both theoretical and practical significance [31]. The Google Hash Code competition problems, arguably, present themselves as suitable candidates.

The Hash Code programming competition, formerly hosted annually by Google, challenged teams of up to four members to solve intricate CO problems within a four-hour time frame using any tools, (online) resources, and programming languages of their choice. These problems often drew inspiration from real-world challenges, such as vehicle routing, task scheduling, and router placement. There is often some relation to classical problems found in CO literature, which may provide theoretical and practical insights on how to solve them but these seem to be harder to solve then their classical counterparts. Moreover, exact algorithms to solve these problems efficiently are not known, and up to our knowledge there is also no known heuristic or approximation method that prevails over other approaches.

Given the pertinence of these problems, and the wide range of challenges they present from both a theoretical and practical standpoint, they serve as interesting benchmarks for the evaluation of meta-heuristics offering ample research potential. Furthermore, they offer a suitable approach to assess the feasibility of the aforementioned unifying framework on more realistic and challenging problems beyond the ones commonly found in the literature.

1.2 Goals & Scope

The main goal of this work is the implementation and evaluation of meta-heuristic solution approaches for the Google Hash Code problems, using a principled approach that separates the modeling of the problems from the solvers.

In particular, we aim to expand upon the modelling approach for meta-heuristics that has been partially explored in previous research [15, 34, 35]. The objective is to solidify existing concepts while introducing additional functionality, both in conceptual understanding and practical application.

Furthermore, we aim to construct models for the Google Hash Code problems. These models will not only be described and discussed in this thesis but will also serve as illustrative examples documenting the modelling concepts. Furthermore, they will enable a critical evaluation of the merits and shortcomings of this principled approach in comparison to more ad-hoc and traditional methods

of problem-solving.

Finally, the implementation of state-of-the-art meta-heuristic solvers is a vital component of our work as it will enable us to assess the performance and quality of solutions found for the models of the Google Hash Code problems as well as the feasibility of the modelling approach for meta-heuristic solver development.

In summary, the main research questions we outline for this thesis are:

R1. Can we formalize the existing ideas explored by previous work on the modelling framework [15, 34, 35] and produce a practical implementation, potentially contributing with new features?

R2. Can we implement general-purpose meta-heuristic solvers with respect to the principled modelling framework implementation?

R3. Can Google Hash Code problems be solved effectively using this modelling approach?

1.3   Contributions

The main contributions of this thesis related to the aforementioned research questions, are as follows:

C1. Building upon existing research on well-structured modeling for meta-heuristics [15, 34, 35], this document aims to gather and formalize a comprehensive specification. Our goal is to bring together all the concepts and developments made so far. For this purpose, we've created a practical Python implementation of the framework, with the aim of summarizing the existing ideas related to modeling for both constructive and local search methods.

C2. We implemented several meta-heuristic solvers and utilities both for gathering the solutions and for testing the developed models. Given that these are general-purpose they can work with any model that is developed under the practical implementation of the framework we devised.

C3. We selected two Google Hash Code problems for which some models for each of the problems were developed that explore the different properties of the problems in an attempt to obtain the best solutions possible. Furthermore, these models provide a practical example on how to model

relatively complex problems and also allows us to think critically about the framework capabilities.

1.4 Software

The following software resulted from the development of this thesis is distributed under an open source license.

S1. Pedro Rodrigues. *Nasf4nio-Py*. GitHub. URL: https://github.com/pedromig/nasf4nio-py (visited on 08/31/2023) [40]

S2. Pedro Rodrigues. *Hascode-Models*. Github. URL: https://github.com/pedromig/hashcode-models (visited on 08/31/2023) [39]

1.5 Outline

The remainder of thesis is structured as follows. In Chapter 2, we provide an overview of optimization concepts, meta-heuristics, and modelling in the context of meta-heuristics. Moving to Chapter 3, we analyze the Google Hash Code competition, focusing on the characteristics of the problems and their relation to existing CO literature. In Chapter 4, we discuss the modelling framework and its role in meta-heuristic development. Chapters 5 and 6 present detailed studies of the Hash Code problems "Optimize a Data Center" and "Book Scanning", and the experimental results obtained. Finally, Chapter 7 summarizes findings in this work and suggest future research directions.

Chapter 2

Background

"If I have seen further than others, it is by standing upon the shoulders of giants."

— Isaac Newton

This chapter presents a comprehensive literature review of optimization, meta-heuristics and modelling. Additionally, it provides a background review of the state-of-the-art regarding the principled modelling approach. In particular, Section 2.1 describes fundamental CO concepts deemed relevant for better understanding this work. Section 2.2 discusses multiple well-known techniques for solving CO problems. Section 2.3 describes MH methods and presents an extensive review of MH solvers. Finally, Section 2.4 delves into the details of the modelling approach and describes the existing implementations.

2.1 Optimization Concepts

Optimization, as defined by Papadimitriou and Steiglitz [7], is the task concerning the search for an optimal configuration or set of parameters that maximizes or minimizes a given objective function. In other words, optimizing involves finding the best solution to a given problem among a set of feasible solutions. Formally, an optimization problem can be defined as follows:

Definition 2.1.1 (Optimization Problem [7]). *An optimization problem is a tuple $(\mathcal{S}, f)$, 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, each solution $s \in \mathcal{S}$, is assigned a real value representing its quality.

Definition 2.1.2 (Global Optimum [5, 7]). *Assuming, without loss of generality, an optimization problem with a maximizing objective function a global optimum $s^* \in \mathcal{S}$ is expressed by:*

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

Since Google Hash Code problems [37] have a single-objective maximizing objective function, we will only consider maximization in this work. However, it is possible to reformulate problems with a minimizing objective function for maximization [14] using the identity:

$$\min f(s) = \max -f(s) \tag{2.3}$$

2.1.1 Combinatorial Optimization

Combinatorial Optimization (CO) problems are a subset of optimization problems characterized by a discrete solution space that typically involves different permutations, groupings, or orderings of objects that satisfy some problem-specific criteria [7, 15, 11, 22]. Thus, regarding the previous definition of an optimization problem, a CO can be formally defined as follows:

Definition 2.1.3 (Combinatorial Optimization Problem [7]). *A combinatorial optimization problem is an optimization problem (2.1.1) where the set $\mathcal{S}$ of feasible solutions is finite or countably infinite.*

Typical examples of CO problems include network flow, matching, scheduling, shortest path and decision problems. Notably, the 0/1 Knapsack Problem (KP) [36, 22, 27] is a well-known example of a CO problem where the goal is to find the subset of items with the highest total profit that can fit in a knapsack without exceeding its maximum capacity.

Due to the combinatorial nature of CO problems, solutions are often defined in terms of a *ground set*.

Definition 2.1.4 (Ground Set [35, 27, 26]). *The ground set of a CO problem*

is a finite set of components $\mathcal{G} = \{c_1, c_2, \ldots, c_k\}$, such that every solution to the problem, feasible or not, can be defined as a subset of $\mathcal{G}$.

For this work, it is also relevant to define the notion of empty, partial and complete solutions.

Definition 2.1.5 (Empty Solution). *A solution $s \in 2^{\mathcal{G}}$, where $2^{\mathcal{G}}$ denotes the powerset of $\mathcal{G}$, is said to be an empty solution if $s = \emptyset$.*

Definition 2.1.6 (Partial Solution). *A solution $s \in 2^{\mathcal{G}}$ is said to be a partial solution if there is a feasible solution $s' \in \mathcal{S}$ such that $s' \supseteq s$.*

Definition 2.1.7 (Complete Solution). *A feasible solution $s \in \mathcal{S}$ is said to be a complete solution if there is no feasible solution $s' \in \mathcal{S}$ such that $s' \supset s$.*

It is worth noting that, according to our definition, a partial solution is not required to be feasible, unlike a complete solution.

To illustrate these concepts, let's consider the practical example of the KP. In this context, the ground set is the set of all the available items (components). As such, a feasible solution is one in which the sum of the weights of the items placed within the knapsack (select components) does not exceed its capacity. A partial solution is one where additional items (components) can be still be added to the current solution without exceeding the capacity of the knapsack. Note that, for the KP every partial solution is feasible. Finally, a complete solution is a feasible solution where no further items can be added due to capacity constraints.

In essence, since CO problems involve choosing a combination of components any algorithm that is able to enumerate all possible combinations can be used can be used to solve these problems. However, finding an optimal solution can be difficult, and exhaustive search strategies may not be able to solve many of these problems, which are often NP-Hard [22, 27]. In these cases, heuristic and MH methods present themselves as effective alternatives to be considered.

2.1.2 Bounds

In mathematics, the notion of bounds has its origins in set (order) theory. More precisely, upper and lower bounds are defined as the sets of *majorants* and *minorants* of a given parent set. Majorants are the elements greater or equal to the highest value within the parent set. Likewise, minorants are the elements

smaller or equal to the smallest value in the parent set. Furthermore, an upper bound is regarded as *tight* or *strong* when no smaller value can serve as an upper bound, a concept known as the *supremum* or *least upper bound* of a set. Similarly, a lower bound is considered tight when no higher value can function as a lower bound, which is known as the *infimum* or *greatest lower bound* of a set.

This concept holds significance in CO and is frequently employed when aiming to optimize the objective value of (partial) solutions. In essence, the upper and lower bounds of a (partial) solution encompass the range of values for an objective function that are either greater/smaller than or equal to the optimal value achievable through the partial solution. The same logic applies to the concept of tight upper and lower bounds.

Formally speaking, the upper bound and lower bound [7, 35] of a (partial) solution can be defined as follows.

Definition 2.1.8 (Upper Bound). *An upper bound of a (partial) solution $s' \in 2^{\mathcal{G}}$ is a numeric value given by a function $\Phi_{ub} \colon 2^{\mathcal{G}} \to \mathbb{R}$ such that:*

$$\forall s \in \mathcal{S} \wedge s \supseteq s' \colon f(s) \leq \Phi_{ub}(s') \tag{2.4}$$

Definition 2.1.9 (Lower Bound). *A lower bound of a (partial) solution $s' \in 2^{\mathcal{G}}$ is a numeric value given by a function $\Phi_{lb} \colon 2^{\mathcal{G}} \to \mathbb{R}$ such that:*

$$\forall s \in \mathcal{S} \wedge s \supseteq s' \colon \Phi_{lb}(s') \leq f(s) \tag{2.5}$$

The usage of bounds is often common in CO algorithms, especially in exact approaches, as will be later detailed. Nonetheless, concerning MH methods, bounds can be helpful tools for guiding the solution construction process. The rationale is that, when faced with a choice between adding one of two components, considering the upper bound's value becomes valuable as opting for the component having a lesser negative impact on the bound's value preserves the solutions' future potential for improvement. Note that the tightness of a bound is a determining factor for keeping this potential.

Moreover, while the objective function holds significance in directing the optimization process, there are situations where evaluating the quality of a partial solution might not be possible. Additionally, the objective function might pose a bottleneck, obstructing the optimizer's ability to measure the impact of changes on solution quality. Nevertheless, the value for an upper

bound can consistently be defined, thus emphasizing the importance of bounds in this context.

In essence, the upper bound value provides an optimistic look at the quality of a (partial) solution and its potential to improve. Conversely, the lower bound, provides a realistic perspective on the objective value of the solution. Moreover, for a complete solution, the upper bound value will always match the objective value.

2.1.3 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 involves the process of discovering the ultimate global optimum (2.1.2) for a given problem, regardless of where it might lie within the solution space. This search for the best solution is often called *exploration*. In contrast, LO concentrates on finding the most optimal solution among proximate that are close in some way, which is commonly referred to as *exploitation*. 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 2.1.10 (Neighborhood Structure [7, 11]). *A neighborhood structure for an optimization problem is a mapping:*

$$\mathcal{N} : \mathcal{S} \longrightarrow 2^{\mathcal{S}} \tag{2.6}$$

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 can define a local optimal solution or just local optimum as follows:

Definition 2.1.11 (Local Optimum [5, 11, 14]). *Assuming maximization without loss of generality, a solution $\hat{s}$ is a local optimum with respect to a given neighborhood structure $\mathcal{N}(\hat{s})$ iff:*

$$\forall s \in \mathcal{N}(\hat{s}) : f(\hat{s}) \geq f(s) \tag{2.7}$$

Furthermore, $\hat{s}$ is a considered a strict a local optimum iff:

$$\forall s \in \mathcal{N}(\hat{s}) \setminus \{\hat{s}\} : f(\hat{s}) > f(s) \qquad (2.8)$$

As an illustrative example, consider the objective function $f(s)$ shown in 2.1. With respect to the Definitions 2.1.2 and 2.1.11, the solution s^1 is a global optimum and s^2, s^3 are (strict) local optima.
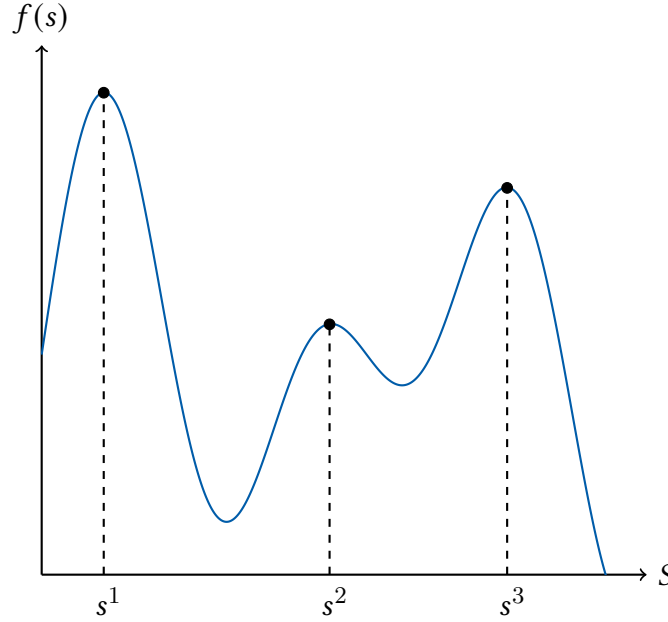
Figure 2.1: Global and Local Optima

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 GO 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, LO, while lacking the optimality guarantees of, 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 problem-specific knowledge.

In the Google Hash Code competition, due to the time imposed by the competition setting, it is often 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 (*exploration* and *exploitation*) is typically employed. To elaborate, the strategy typically

involves *exploring* the search space via GO methods to find "good" starting solutions, that LO methods can further *exploit*.

2.1.4   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 [33] 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 [32, 18]. Meta-Heuristics, as will be later detailed are examples of methods that follow this approach for finding/improving solutions. By contrast, in GBO optimization, also known as *white box* optimization, there is a good understanding of the problem instance being optimized and the objective function properties [32]. Hence, the algorithms used may take advantage of more analytical properties of the problem since they are transparent to the solver.

For the purpose of clarification, with regard to the previously mentioned KP, a BBO strategy would entail the utilization of a search heuristic such as simulated annealing [25] 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 Programing (ILP) [14, 7] problem, it would become amenable to a GBO 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 the problems from a BBO perspective, as the underlying objective function of the is too complex to formalize. Additionally, the process of formalization can be time-consuming and, as a result, the usage of GBO methods post-formalization would not be justified, as they could turn out to be computationally slower. However, it is in many cases, possible to use GBO methods, *e.g.*, ILP to tackle sub-problems that are more simple to formalize as more knowledgeable contestants have demonstrated in the past.

2.2   Optimization Strategies

Combinatorial optimization literature extensively documents a series of methods for solving multiple problems [7, 22, 25]. The approaches followed by these methods are diverse and typically defined by factors such as the time complexity, the quality and the strategy for finding solutions. Particularly, algorithms are often classified in the literature as exact, approximation, or heuristic based on the quality of solutions. Moreover, for each category exist procedures with varying complexities rely on different strategies to find solutions which, in essence, can be classified as constructive or local.

2.2.1   Exact, Approximation and Heuristic Methods

Exact methods are designed to find the optimal solution for a given problem. These typically involve an exhaustive enumeration and evaluation of solutions. However, in large problem instances, this may prove to be computationally infeasible. In the context of CO problems, two general exact algorithms are well-known and widely studied: *Branch and Bound* and *Dynamic Programming* [8, 27].

These algorithms operate by iteratively breaking down a problem into smaller, interconnected or standalone sub-problems. The solutions to these are then combined to form the final solution. However, each algorithm employs distinct techniques to enhance the exploration of the search space.

In Branch and Bound approaches, the strategy revolves around utilizing bounds to restrict the search space. Specifically, the upper bound facilitates pruning the search tree, effectively eliminating the need to explore solutions that are undoubtedly worse with respect to best solution found by the algorithm at a particular stage. Similarly, the lower bound guarantees that solutions of inferior quality are rejected during the search process. On the other hand, Dynamic Programming approaches leverage the optimal substructure property [27], thereby avoiding recomputing previous sub-problems.

Approximation methods are designed to find solutions that are provably guaranteed to be close to the optimal quality with respect to a given approximation factor. In fact, approximation methods are often able to solve problems in polynomial time and yield solutions of relatively high quality [24]. However, it is important to note that they require a mathematical proof of approximation that is specific to the problem at hand. Notably, a significant amount of research exists in this field concerning CO problems [1].

Heuristic methods 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 CO problems, typically yielding good solutions in a short time frame when compared to exact methods. A well-known class of algorithms that further extends these methods are meta-heuristics as will be further discussed and detailed in Section 2.3.

A MH is as a high-level heuristic method, as alluded by the word "meta", which describes a concept as an abstraction of other. In the literature, the definition of meta-heuristic varies across different sources, resulting in a lack of consensus on a formal description [6, 11, 27, 25]. Nonetheless, one commonly accepted definition, by Osman and Laporte [6], captures the essence of MH 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"*. Moreover, most, if not all heuristic and meta-heuristic approaches are defined in terms of constructive and local search approaches, which we describe next.

2.2.2 Constructive Search

Constructive Search (CS) is an approach for optimization where from an empty or partial solution for a given problem 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, *e.g.*, objective value and bounds. These rules determine what components from the ground set can be included in the solution at each iteration. The construction stops when no more components can be added to the solution, *i.e.*, the solution is complete. For clarification, a generic pseudocode for a CS procedure [26] is shown in Algorithm 2.1.

It's important to observe that a constructive search approach hinges on a strategy for selecting a component to add to a solution, represented in Algorithm 2.1 through the `SelectComponent` function. Moreover, various other problem-specific details come into play. These encompass activities such as enumerating components (C), creating an empty solution, adding a component to a solution, and evaluating its feasibility. Notably, these details directly influence the CS procedure's ability to construct good solutions. This highlights

---

Algorithm 2.1: Constructive Search Procedure

Input : Ground Set $(\mathcal{G})$
Output: Solution (s)

1   $s \leftarrow \emptyset$
2   $C \leftarrow \{\, c \in \mathcal{G} \setminus s \mid s \cup \{c\} \text{ is feasible} \,\}$
3   **while** $C \neq \emptyset$ **do**
4      $c \leftarrow \texttt{SelectComponent}(C)$
5      $s \leftarrow s \cup \{c\}$
6      $C \leftarrow \{\, c \in \mathcal{G} \setminus s \mid s \cup \{c\} \text{ is feasible} \,\}$
7   **end**
8   **return** s

---

the importance of having a solid problem *model*, a concept we will delve into
further in this thesis.

2.2.3   Local Search

Local Search (LS) approaches begin, with feasible solution to a given problem,
and then make modifications by adding or removing components in order
to improve it. These modifications, are typically localized and aim to exploit
the solution's neighborhood, as defined by the neighborhood structure of the
problem. The procedure concludes when no further alterations can enhance the
solution's quality, ultimately leading to a local optimum. In the scope of this
work, a transformation that can be applied to a solution within in the context
of a LS procedure will be referred to as *local move*.

The primary objective of a LS process is to improve a solution in the direction
of the local optimum. However, it is common in a LS approach to eventually
worsen a solution in order to allow for further exploration of previously unseen
regions of the search space. This action is commonly referred to as a *perturba-
tion* [19]. Furthermore, LS is frequently applied in sequence to a constructive
search phase, where a CS algorithm is used to construct a good initial solution,
which is then further improved through a LS approach.

A generic pseudocode for a local search procedure is outlined in Algorithm 2.2.
It's crucial to clarify that within the pseudocode, the Step function signifies the
execution of a local move, while the Perturb function represents an optional
action introducing a perturbation to the solution, if possible. Additionally, the
enumeration of possible local moves ($\mathcal{M}$) and the selection of a specific local
move to incorporate into the solution are abstracted through the LocalMoves
and SelectLocalMove functions, respectively. Note that, as in CS, the problem-
specific choices made regarding the implementation of each of these functions

will significantly influence the effectiveness of the LS procedure.

Algorithm 2.2: Local Search Procedure

Input : Solution (s')
Output: Solution (s)

1   $s \leftarrow s'$
2   $\mathcal{M} \leftarrow \mathsf{LocalMoves}(s)$
3   **while** $\mathcal{M} \neq \emptyset$ **do**
4     $m \leftarrow \mathsf{SelectLocalMove}(\mathcal{M})$
5     $s \leftarrow \mathsf{Step}(s, m)$
6     $s \leftarrow \mathsf{Perturb}(s)$ ▷ Optional
7     $\mathcal{M} \leftarrow \mathsf{LocalMoves}(s)$
8   **end**
9   **return** s

2.3   Meta-Heuristics

In the literature, a multitude of meta-heuristic algorithms have emerged over the years, exploring various ideas to guide the search process [6]. These encompass strategies that narrow the search space to promising regions, enhance solutions in a greedy manner, or utilize randomized and probabilistic techniques, some of which draw inspiration from natural phenomena like collective behavior, natural selection, and physical processes of materials.

However, the majority of state-of-the-art meta-heuristic algorithms can be described by a couple of distinctive traits [11] such as:

Search Strategy. This refers to the method used to construct a solution. It can be one of three main types: constructive, local, or a *composite* approach that combines both strategies.

Memoization. This concept involves maintaining a record or archive of previously explored solutions. This record helps in identifying solutions that may be revisited or disregarded in subsequent stages of the optimization process.

State Size. This pertains to the number of solutions being evolved during the construction phase. In *population methods*, multiple solutions are worked with at each iteration, while in *trajectory methods* (single-state), only a single solution improved at a time.

In this section, we will offer a brief overview of select state-of-the-art MH algorithms, which encapsulate all the above properties and will be utilized and implemented in the context of this work.

2.3.1 Beam Search

Beam Search (BS) [3, 35] is a CS trajectory Meta-Heuristic (MH) inspired by the *breadth-first search* algorithm [7]. However, it deviates from the conventional practice of expanding all solutions in the search tree during each iteration. Instead, this technique maintains a fixed-size archive of previous solutions, which are expanded at each step (*beam*). Subsequently, the expanded solutions undergo filtering, and only the best solutions, determined based on heuristic information or bound values, are retained within the archive. These selected solutions act as the starting points for the next iteration. It's important to acknowledge that the size of this archive, and consequently the number of solutions filtered at each step, is dictated by a parameter referred to as the *beam width*.

The construction process of BS concludes either when there are no more candidate solutions to expand or when other predetermined stopping criteria are satisfied. For illustration purposes, the pseudocode for BS is presented in Algorithm 2.3.

Algorithm 2.3: Beam Search

> **Input** : Beam Width (w), Objective Function (f), Upper Bound
>   Function (Φ_{ub})
> **Output**: Solution (s^*)

1   $s^* \leftarrow \emptyset$
2   $\mathcal{B} \leftarrow \{\emptyset\}$
3   **while** $\mathcal{B} \neq \emptyset \wedge$ `stopping criteria not met` **do**
4      $\mathcal{B}' \leftarrow \emptyset$
5   **foreach** $s \in \mathcal{B}$ **do**
6          $\mathcal{B}' \leftarrow \mathcal{B}' \cup$ `Branch`(s)
7   **end**
8   **if** $\mathcal{B}' \neq \emptyset$ **then**
9          $\mathcal{B} \leftarrow \text{argmax}_w\, \Phi_{ub}(s)$ ▷ `Select the "w" best solutions`
                   $s \in \mathcal{B}'$
10   **foreach** $s \in \mathcal{B}$ **do**
11   **if** s `is feasible` $\wedge f(s) > f(s^*)$ **then**
12                 $s^* \leftarrow s$
13   **end**
14   **end**
15   **end**
16 **end**
17 **return** s^*

The pseudocode assumes that the `Branch` function generates the set of all potential (partial) solutions achieved by incorporating components from the ground

set that are not currently part of the solution. It's worth noting that, since the solution is constructed incrementally, the solutions obtained through branching might not always be feasible. As a result, an additional step for feasibility verification is required before updating the best solution (s^*) identified during the search process. Moreover, it's important to emphasize that the *beam width* parameter directly influences the quality of the solutions discovered. Smaller values might lead to premature convergence to a local optimum, while larger values, although capable of producing better-quality solutions, could lead to increased memory and computational requirements.

2.3.2   Greedy Randomized Adaptive Search Procedure

Greedy Randomized Adaptive Search Procedure (GRASP) [21, 35, 11] is a stochastic, CS and LS, MH that iteratively build solutions by sequencing a construction phase with a local search phase. The construction phase begins with an empty solution and iteratively adds a new component at each step. This component is chosen at random from a *restricted candidate list* of components. This list consists of the best available components for extending the current (partial) solution, based on either a heuristic value or an upper bound. Additionally, there's an option to apply a local search phase to the partial solution obtained during the construction phase. The goal of this local search is to further exploit the solution, with algorithm concluding when some predefined stopping criteria are met.

The provided pseudocode in Algorithm 2.4 offers an overview of how the GRASP algorithm functions.

It's important to highlight that this meta-heuristic, GRASP, provides a means of controlling the balance between randomization and greediness in solution construction (`GreedyRandomizedConstruction` function). This control is achieved through the parameter α, which serves as a threshold for the quality of solutions within the candidate list. Specifically, when $\alpha = 1$, the construction process is entirely random, and all possible components are included in the candidate list, regardless of their quality. Conversely, with $\alpha = 0$, only the best component is considered at each step, rendering this MH purely greedy. It's worth noting that the inclusion of the optional local search step (`LocalSearch` function) can significantly enhance the quality of the randomized greedy process. However, the crux of fine-tuning lies in determining an appropriate value for α.

Algorithm 2.4: Greedy Randomized Adaptive Search Procedure

Input : Alpha (α), Objective Function (f), Upper Bound Function (Φ_{ub})

Output: Solution (s^*).

1 $s^* \leftarrow \emptyset$
2 **while** stopping criteria not met **do**
3      $s \leftarrow$ GreedyRandomizedConstruction(Φ_{ub}, α)
4      $s \leftarrow$ LocalSearch(s)
5   **if** s is feasible $\wedge f(s) > f(s^*)$ **then**
6          $s^* \leftarrow s$
7   **end**
8 **end**
9 **return** s^*

2.3.3 Iterated Greedy

Iterated Greedy (IG) [30, 35] is a CS trajectory MH that revolves around the concept of iteratively destroying a solution and subsequently reconstructing it to break free from local optima. In its basic form, this MH begins by constructing an initial solution, for example through a randomized greedy approach. Subsequently, it proceeds to remove one or more randomly chosen solution components before initiating the construction process anew. This sequence of operations continues until some predefined stopping criteria are satisfied.

Some less-common variations of this algorithm incorporate local search methods to further exploit the solution following the construction phase. Furthermore, certain versions of this MH, as detailed in the literature [30], implement an acceptance criterion that probabilistically permits the acceptance of solutions of inferior quality, akin to the mechanisms employed in SA, which will be elaborated upon in Section 2.3.7.

As an illustration, the pseudocode for an IG algorithm utilizing a randomized greedy strategy, is outlined in Algorithm 2.5. Here, the functions Construct and Destruct signify the approach used for (re-)constructing and destroying a solution, respectively.

2.3.4 Ant Colony Optimization

Ant Colony Optimization (ACO) [16, 10, 25, 11] is a stochastic population based, CS and LS, MH that is inspired by the foraging behavior of ants, which in the context algorithm represent (partial) solutions undergoing construction.

The algorithm simulates the movement of *ants* through the search space, where

---

Algorithm 2.5: Iterated Greedy

    Input   :Objective Function (f), Upper Bound Function(Φ_{ub})
    Output:Solution (s^*)

1 $s^* \leftarrow$ Construct $(\emptyset, \Phi_{ub})$
2 **while** stopping criteria not met **do**
3      $s \leftarrow$ Destruct(s^*)
4      $s \leftarrow$ Construct(s, Φ_{ub})
5   **if** s is feasible $\wedge\ f(s) > f(s^*)$ **then**
6         $s^* \leftarrow s'$
7   **end**
8 **end**

---

each ant constructs a solution by making a sequence of probabilistic choices based on the *pheromone* trail left by previous ants and other heuristic information. 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. Moreover, the *pheromones* are associated with each component of the ground set, and in practice consist in a parametrized probabilistic model.

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 model. Notably, there are different models which gave origin to many variations of this MH.

The pseudocode provided in 2.6 illustrates the basic functioning of this MH

In summary, the ACO meta-heuristic can be described as a process that comprises of a solution construction phase (AntBasedSolutionConstruction function), in which each solution of the population is constructed using both the *pheromone model* and heuristic information, followed by an optional phase of exploiting these solutions through local search (LocalSearch function), and culminating in a pheromone update phase (PheromoneUpdate function). This process is then repeated for multiple iterations, until certain stopping criteria is met.

Algorithm 2.6: Ant Colony Optimization

Input : Population ($\mathcal{S}$), Pheromone Update Rule ($\mathcal{R}$), Pheromone
Values ($\vec{\tau}$), Evaporation Rate (α)

Output: Solution (s^*)

1 $s^* \leftarrow s'$
2 $\mathcal{S} \leftarrow \{\emptyset\}$
3 **while** not stopping criteria met **do**
4      $\mathcal{S} \leftarrow$ AntBasedSolutionConstruction($\mathcal{P}, \vec{\tau}$)
5      $\mathcal{S} \leftarrow$ LocalSearch($\mathcal{P}$)   ▷ Optional
6      $s \leftarrow \underset{s \in \mathcal{P}}{\arg\max} f(s)$ ▷ Select best solution
7   **if** $f(s) > f(s^*)$ **then**
8        $s^* \leftarrow s;$
9   **end**
10     $\mathcal{S} \leftarrow$ PheromoneUpdate($\mathcal{P}, \mathcal{R}, \vec{\tau}, \alpha$)
11 **end**
12 **return** s^*

2.3.5 Hill-Climbing

Hill Climbing (HC) [25, 15] is a simple stochastic, LS trajectory MH that works
by iteratively attempting to improve a starting solution through a sequence of
incremental changes, *i.e.*, by selecting the solution in the neighborhood that
yields the best increment with respect to the objective value. This process
terminates when the best solution possible is found or another stopping criteria
is met. Despite the simplicity of this method it is worth noting that, due to the
inherent greedy choice of the best at each step this approach is susceptible to
getting trapped in local optima. For illustration purposes the pseudocode of
simple version of an HC algorithm is shown in Algorithm 2.7.

Algorithm 2.7: Hill Climbing

Input : Solution (s'), Objective Function (f)

Output: Solution (s^*)

1 $s^* \leftarrow s'$
2 **while** stopping criteria not met **do**
3      $s \leftarrow$ Perturb(s^*)
4   **if** $f(s) > f(s^*)$ **then**
5        $s^* \leftarrow s$
6   **end**
7 **end**
8 **return** s^*

It's worth noting that the presented algorithm's effectiveness is rooted in
the efficiency of a random process, which strives to improve solution quality

through a sequence of random perturbation attempts (`Perturb` function). Nevertheless, there exist two noteworthy variations that more thoroughly explore the solution's neighborhood and take steps in the direction of the most substantial improvement. These are commonly known as First Improvement (FI) and Best Improvement (BI). The latter is also known in the literature as *steepest ascent* HC [25].

In a FI approach, the first random neighboring solution that improves the current solution's quality is retained. Conversely, in a BI scenario, the entire neighborhood of the current solution is examined, and the best neighboring solution is selected for the subsequent iteration.

2.3.6 Iterated Local Search

Iterated Local Search (ILS) [19, 25, 11] is a stochastic LS trajectory MH that extends the HC method. This algorithm operates through a series of iterations, where each iteration aims to explore a solution by applying a LS procedure, typically FI. When an improvement is achieved, the best solution found is retained and serves as the reference solution for subsequent iterations. This iterative process continues until the algorithm either converges to the local optimum or meets predefined stopping criteria.

An integral feature of the ILS is the introduction of a perturbation step at the end of each iteration. This perturbation injects controlled randomness into the current solution, allowing for exploration of different regions within the search space. This exploration aids in preventing the algorithm from becoming stuck in local optima. Notably, other many variations of this algorithm exist, with certain versions incorporating an archive mechanism to store starting solutions for each iteration. This approach helps prevent repeated exploration of the same regions within the search space.

The core framework of the ILS algorithm is illustrated in Algorithm 2.8. The parameter *ks* in the `Perturb` function represents the *kick strength*, determining the intensity of the perturbation movement, and can be adjusted accordingly.

2.3.7 Simulated Annealing

Simulated Annealing (SA) [2, 20, 4] is a stochastic, LS trajectory MH that draws inspiration from the *annealing* process in metallurgy. The core concept adopted from metallurgy and applied to the algorithmic context is the notion of accepting poorer quality solutions in the early stages of the search and gradually tightening the acceptance criteria as the search progresses. This

---

Algorithm 2.8: Iterated Local Search

Input : Solution (s'), Kick Strength (k), Objective Function (f)
Output: Solution (s^*).

1 $s^* \leftarrow s'$
2 **while** stopping criteria not met **do**
3 $\quad\big|\quad s \leftarrow$ LocalSearch(s^*)
4 $\quad\big|\quad s \leftarrow$ Perturb(s, k)
5 $\quad\big|\quad$ **if** $f(s) > f(s^*)$ **then**
6 $\quad\big|\quad\big|\quad s^* \leftarrow s$
7 $\quad\big|\quad$ **end**
8 **end**
9 **return** s^*

---

strategy is based on the understanding that in the initial phases, accepting suboptimal solutions as the best available option promotes exploration of the search space. As the search advances towards its final stages, the emphasis shifts towards convergence to a local optimum, favoring exploitation.

Essentially, SA works by iteratively applying small perturbations to a candidate solution in order to enhance it. The algorithm accepts or rejects these new solutions based on their quality and a probability function that emulates the cooling process of a metal. This probability function depends on a temperature parameter, which decreases as the algorithm proceeds, causing the acceptance probability of worst solutions to decrease as well. Ultimately, the algorithm halts when further improvements to the solution are no longer feasible or when predefined stopping criteria are met.

The details of the SA algorithm are illustrated in the pseudocode provided in Algorithm 2.9.

Importantly, the SA algorithm is tunable, allowing for parameter adjustments to suit the specific problem at hand. Parameters such as the initial temperature, cooling function, and acceptance criteria can be fine-tuned for optimal performance. In Algorithm 2.9, the acceptance criteria is defined using an exponential function, while the cooling process is held constant and governed by the parameter α.

2.3.8 Tabu Search

Tabu Search (TS) [9, 17, 25] is a stochastic, LS trajectory MH that incorporates the use of a memory to aid the search process.

In essence,the TS algorithm operates by conducting an iterative exploration of

---

Algorithm 2.9: Simulated Annealing

Input : Solution (s'), Initial Temperature (t_0), Cooling Rate (α),
Objective Function (f)

Output: Solution (s^*)

---

1 $t \leftarrow t_0$
2 $s^* \leftarrow s'$
3 **while** `stopping criteria not met` **do**
4      $s \leftarrow \texttt{Perturb}(s^*)$
5      $\delta \leftarrow f(s^*) - f(s)$
6   **if** $\delta < 0 \vee \texttt{Random}(0,1) < e^{-\frac{\delta}{t}}$ **then**
7         $s^* \leftarrow s$
8   **end**
9      $t \leftarrow t \cdot \alpha$
10 **end**
11 **return** s^*

---

the solution space through the repeated application of a series of BI procedures. During each step, the best solution is updated and added to a list of recently visited solutions known as the *tabu list*. The tabu list has a predefined size and serves as a short-term memory mechanism. This approach aims to prevent the algorithm from revisiting solutions that have been explored before, thereby avoiding premature convergence to local optima. By incetivizing exploration of new regions in the search space, the algorithm enhances its ability to find better solutions. Similar to other LS meta-heuristics, the Tabu Search (TS) algorithm terminates when no further improvement to the solution is achievable.

The pseudocode in Algorithm 2.10 illustrates a simple version of this meta-heuristic.

Notably, the size and duration of the tabu list ($l_{\max}$), 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. Additionally, a common criterion for the removal of an element from the tabu list is based on its age. This aspect related to age-based removal is depicted in Algorithm 2.10 through the utilization of the `Oldest` function, which retrieves the oldest solution stored within the tabu list.

2.3.9   Outline

In this section, we provided a concise overview of well-known state-of-the-art meta-heuristic algorithms that hold relevance to our study. Our intention was to offer the reader a general understanding of the conceptual foundation behind

---

Algorithm 2.10: Tabu Search

Input : Solution (s'), Tabu Length (l_{max}), Objective Function (f)
Output: Solution (s^*)

1 $s^* \leftarrow s'$
2 $\mathcal{T} \leftarrow \{\emptyset\}$
3 **while** stopping criteria not met **do**
4      $s \leftarrow \underset{s \in \mathcal{N}(s)\backslash\mathcal{T}}{\mathrm{argmax}}\ f(s)$ ▷ Select best neighbor $s \notin \mathcal{T}$
5   **if** $f(s) > f(s^*)$ **then**
6          $s^* \leftarrow s$
7          $\mathcal{T} \leftarrow \mathcal{T} \cup \{s\}$
8   **end**
9   **if** $|\mathcal{T}| > l_{\max}$ **then**
10         $\mathcal{T} \leftarrow \mathcal{T} \setminus \mathrm{Oldest}(\mathcal{T})$
11   **end**
12 **end**
13 **return** s^*

---

each algorithm, without delving into an exhaustive exploration. Furthermore, we categorized each meta-heuristic based on the employed search strategy (CS or LS), the utilization of a memory archive for solutions, and the state size (*Population* vs. *Trajectory*).

The Table 2.1 presents a succinct summary of various meta-heuristic methods, organized according to these properties.

Meta-Heuristic	Search Strategy		State Size		Memoization
	Constructive Search	Local Search	Single	Population	
BS	✓		✓		✓
GRASP	✓	✓	✓		
IG	✓		✓		
ACO	✓	✓		✓	
HC		✓	✓		
ILS		✓	✓		
SA		✓	✓		
TS		✓	✓		✓

Table 2.1: Meta-Heuristics Summary

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 [29].

In the field of optimization, there is an extensive body of literature on the development of glass-box ILP models for a wide range of problems [7, 14, 24]. 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, *e.g.*,*Gurobi*, *CPLEX*, or *GLPK*, to address any problem in a principled manner.

As an example of this GBO modelling approach consider the KP for which an ILP model [22] is represented by the formulation 2.9. Herein, the parameter n signifies the count of items, while W designates the knapsack's maximum capacity. Additionally, the variables w_i and p_i correspond to the weight and profit, respectively, associated with each individual item i. Furthermore, the binary variable x_i is employed to denote whether a particular item is included (assigned a value of 1) or excluded (assigned a value of 0) from the knapsack, for a given solution.

$$
\max f(x) = \sum_{i=1}^{n} p_i \cdot x_i
$$
$$
\text{s.t} \sum_{i=1}^{n} w_i \cdot x_i \leq W \tag{2.9}
$$
$$
x_i \in \{0, 1\} \quad \text{for } i = 1, 2, \ldots, n
$$

It's crucial to recognize that this formulation highlights several significant aspects of the problem. These include the decision space, which defines the range of values each variable can take; the objective function, which expresses the optimization goal; the construction rules (constraints) that guide the creation of a valid solution; and the problem instance parameters necessary for defining and solving the problem instance effectively.

When approaching problems from a BBO perspective, particularly through MH solvers, the notion of developing a reusable model has yet to be established, as far as our knowledge extends. However, for such a model to come into existence, it would need to address, at the very least, the same details as the aforementioned KP model. Furthermore, considering the strategies employed by MH solvers to explore solutions (Section 2.3), certain recurring aspects emerge as essential questions that a model must address:

Problem Instance. What information is required to characterize a particular problem (instance)?

Solution. How do we characterize an empty, partial or complete solution for the problem, and its feasibility?

Objective Function & Bounds. Can we evaluate a partial or complete solution, and how? How do we calculate the bounds for a (partial) solution?

Combinatorial Strutcture (CS). How do we construct solutions? What is the ground set, components and how are they added/removed to/from the solution.

Neighborhood Strutucture (LS). How do we characterize neighborhood of a solution and the possible local moves?

Incorporating a standardized approach to address these fundamental questions can pave the way for a systematic problem-solving methodology. This enables the integration of meta-heuristics as powerful solvers, akin to traditional solvers in mathematical programming. This approach diverges from existing meta-heuristic frameworks [13, 12, 23], where custom meta-heuristic solvers are tailored for each specific problem. Instead, it establishes a problem-independent abstraction layer (model) that equips solvers with essential information across diverse meta-heuristics, while abstracting the intricacies of inherent to each specific problem.
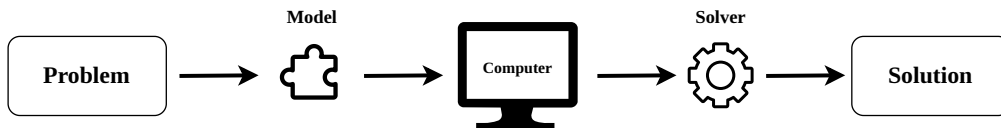
Figure 2.2: Principled Modelling Framework

The overall concept for a principled meta-heuristics modeling framework is to create a standardized model that is applicable to any problem domain. This model comprises a set of functions that encapsulate the essential aspects of the problem. This model is then provided to a computer system, which employs a range of meta-heuristics. These meta-heuristics utilize the set of functions within the model to effectively generate solutions, as illustrated in Figure 2.2.

Crucially, the set of functions represents the core of the standardized model. An essential consideration is to refine this set of functions to capture the fundamental characteristics that are universally present in all problems. This avoids the need for distinct functions for each problem, which would defeat the purpose of creating a standardized approach.

Notably, this framework for tackling problems in the context of meta-heuristics, which we refer to as the principled modelling approach, has been seldomly researched and constitutes an ongoing research effort. In fact, we identify two noteworthy studies in the literature and three attempts in practically developing an Application Programming Interface (API) for this modelling framework [15, 35, 34].

2.4.1 Software

The first attempt at developing a modeling framework for experimental testing of meta-heuristics can be traced back to the pioneering thesis work of Vieira [15]. In this work, the author introduced the concept of designing and implementing a modeling framework, specifically targeted at local search algorithms, while also alluding to ideas related to constructive search. The practical implementation of these modeling concepts was achieved through a Python framework, named Python Optimization Framework (POF). Regrettably, the source code is unavailable, leaving us with only a textual description for reference.

The work, that followed was in the sense of further refining the ideas proposed by Vieira [15] and culminated in a C API named Not Another Framework for nature-inspired optimization (nasf4nio). Then, in the context of the thesis work of Outeiro [35], a version of this API was implemented for constructive search meta-heuristics (nasf4nio-cs).

The upcoming sections provide a succinct summary of the most relevant features and contributions of these works.

Python Optimization Framework

This python framework, developed in the context of the work by Vieira [15], 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 MH 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 the means of three main classes:

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 serves as a general-purpose utility that allows the step-
by-step execution of a solver with respect to a given problem, thus being
responsible for the execution of the algorithms and gathering solutions.

Focusing on modelling perspective, the Problem class is the one that exposes
several modelling related functions that when implemented for a problem can
be used by a Solver to obtain solutions. Albeit, from a simple analysis of the
the function set we concluded that it is too complex and intricate possibly being
one of the reasons that motivated the simpler design of nasf4nio.

Not Another Software Framework for Nature-Inspired Optimization

This API began as an implementation of LS abiding by the concepts proposed
in the POF culminating in nasf4nio [34]. In general, this API follows the
same concepts as the nasf4nio-cs, which we will detail next. Regarding CS,
the work by Outeiro [35] was built upon the concepts presented in the POF
and the existing implementation of nasf4nio, further unifying the concepts
and providing both a conceptual model and an implementation of a C API for
constructive search (nasf4nio-cs).

Specifically, nasf4nio-cs, 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 core 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 that may added, removed, permitted or forbidden with

respect to a given solution.

In the context of nasf4nio, the majority of the data structures are retained, save for the substitution of the Component structure with the Move structure. The latter stores information pertaining to local moves that can be applied to a solution.

Focusing again in nasf4nio-cs and narrowing our focus to functions primarily concerned with Solution manipulation and excluding those related to implementation intricacies, such as, inspection, assignment, and memory management, we can classify them into three main categories:

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, *i.e.*,from a CS perspective.

Evaluation. In this category appear functions such as *getObjectiveVector* and *getObjectiveLB* which allow for evaluation of the quality of the solution with respect to the objective value and bounds.

Similarly, the same principles apply to nasf4nio. However, the "construction" operations are primarily designed to interact with solutions and (local) moves, as opposed to solutions and components. Importantly, there remains yet a connection to be made between both implementations of this API for constructive and local search. This connection will be further explored and formalized in Chapter 4.

Chapter 3

Google Hash Code Competition

"understanding a question is half an answer"

— Socrates

This chapter presents an overview of the Google Hash Code competition. In Section 3.1, we provide a concise review of the competition, encompassing both its historical background and format. Subsequently, in Section 3.2, we delve into the problems presented to participants over the years, attempting to categorize them and establish connections with well-known combinatorial optimization problems described in the literature. Moving forward, **??** sheds light on the design of competition instances. Concluding this chapter, Section 3.4 offers remarks that highlight key aspects of the competition problems, deemed pertinent to this work.

3.1   History & Format

The Google Hash Code competition, organized by Google from 2014 to 2022, consisted of two main phases: a qualifying round and a final round. During this competition, teams of 2-4 skilled individuals were tasked with solving complex problems that mirrored real-world engineering challenges faced by Google's own engineers. The primary aim of the competition was to attract talented individuals to the company. In the qualifying round, participants worldwide engaged in a 4-hour problem-solving session. Subsequently, around 40-50 select teams advanced to the final round, which took place at a Google headquarters. Additionally, participants gathered at designated hubs globally during the qualifying round, fostering a competitive environment.
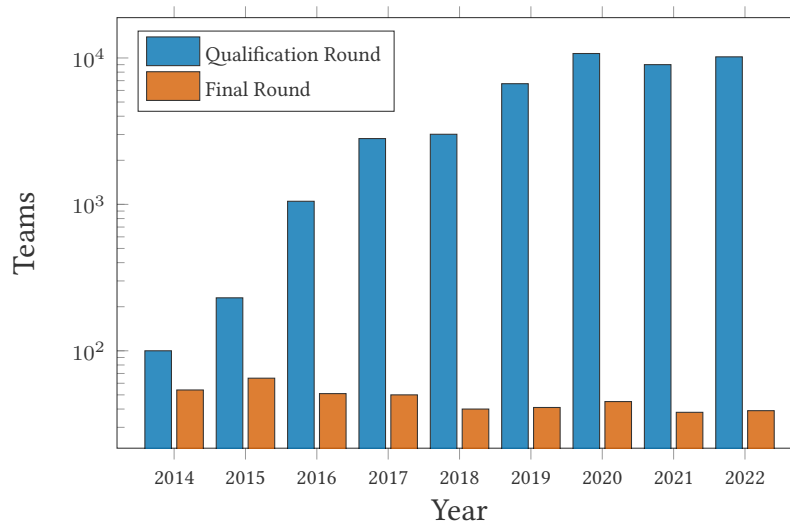
Figure 3.1: Google Hash Code Competition Attendance 2014-2022

In the early years of the competition, it was only open to teams from France. 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. Furthermore, the number of participants kept growing throughout the years which highlights the importance of this event as illustrated in Figure 3.1

Unfortunately, this year Google decided to cease all its coding competitions including Hash Code. Nevertheless, the competition generated a diverse collection of attempts at solving the problems, resulting in a valuable wealth empirical data accessible to the community. As the official coding competition website is no longer accessible, Google created a repository containing all problem statements and instances distributed under an open source license [37]. However, it's worth noting that the scores achieved by participants are not integrated into this repository. Instead, they are documented across various blog posts and third-party websites [38].

It is worth noting, that due to the nature of the problems and format of the competition, participants frequently made use of heuristic and meta-heuristic strategies to solve the problems as best as possible in the alloted time. Moreover, the majority of competition problems are structurally different from one other, which makes it demanding to write general-purpose heuristic solvers that can be easily reused. Hence, it's a common practice for competitors to use solvers that are easily implementable or readily available online, given that internet access is permitted.

3.2   Problems

The primary focus of this section is to provide a comprehensive understanding of the relevant details of the Google Hash Code problems and drive connections with well-studied topics in combinatorial optimization literature. For an exhaustive review of the problem specifics, the reader can refer to the problem statements made freely available in the Google Coding Competitions Archive [37]

3.2.1   Google Hash Code 2014

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.

3.2.2   Google Hash Code 2015

Optimize a Data Center

The optimization of server placement within data centers is a critical aspect, revolving around achieving optimal design efficiency. In this scenario, the problem presents participants with the task of designing a data center and devising the most effective server allocation. The data center's structure is

represented by a collection of rows, each containing slots where servers of varying sizes and computing capacities can be positioned. When placed, these servers are also assigned to specific resource pools, allowing them to contribute their computing capacity to those pools.

Objectively, the goal is to assign multiple servers to available slots and resource pools in a manner that maximizes the guaranteed capacity for all resource pools. This metric serves as the criterion for evaluating solutions to this problem and can be defined as the lowest amount of computing power that will remain for a specific resource pool in the event of a failure of any arbitrary row in the data center. Notably, this objective is considered a bottleneck, as even minor changes in a solution might not yield significant score changes, rendering the optimization process more challenging.

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

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.

3.2.3 Google Hash Code 2016

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.

Satellites

Terra Bella was a Google division responsible for managing and operating a constellation of satellites that collected and processed imagery for commercial purposes. Specifically, these satellites were tasked with capturing images in response to client requests.

The challenge presented to participants involves crafting schedules for individual satellites within the fleet. The goal is to secure image collections that match customer preferences. These collections are characterized by geographical coordinates on Earth and specific time windows for image capture. Each satellite,

originating from unique latitude and longitude coordinates and possessing a certain velocity, possesses the ability to make minor positional adjustments along both axes to access potential photography sites. The problem's score is determined by aggregating the points earned through the successful completion of customer collections. In this context, completion signifies capturing all images for a given collection within the designated time frame.

In essence, this problem falls into the categories of both an assignment and a maximum covering problem. It involves not only covering the maximum number of images with the available satellites to complete collections, but also making decisions about which satellites will capture each photo. Additionally, the simulation aspect is crucial as it directly affects scoring; images not taken within the specified time frame won't contribute to the collection, potentially influencing its completion and the overall score.

3.2.4   Google Hash Code 2017

Streaming Videos

In the era of online streaming services like YouTube, effectively distributing content to users is crucial. This challenge focuses on optimizing video distribution across cache servers to minimize transmission delays and waiting times for users. Contestants must strategize video placement within servers while considering space limitations.

With a roster of videos, each assigned a specific size, an collection of cache servers with designated space, and an index of endpoints initiating multiple requests for various videos, this challenge entails determining an optimal video assignment within servers. The time saved for each request is measured as the difference between data center streaming time and cache server streaming time with minimal latency. The overall score is computed by summing the time saved for each request, multiplied by 1000, and then divided by the total request count. It's important to note that the problem description offers transmission latencies between different nodes.

In general, we categorize this problem as a combination of assignment and knapsack problems. Contestants are tasked not only with determining the allocation of videos to servers but also with accounting for capacity limitations on the number of videos per server. It's worth noting that the calculation of time saved for each request may encounter a bottleneck effect, which can pose challenges when optimizing the overall score.

Router Placement

Strategically optimizing the placement of Wi-Fi routers to achieve optimal signal coverage is a challenge encountered by many institutions and individual users. This issue becomes particularly prominent in larger and complex buildings. Furthermore, in such scenarios, the task may involve setting up a wired connection to establish internet connectivity from the source point, facilitating the strategic positioning of routers for maximum coverage.

The challenge tasked participants with optimizing the arrangement of routers and fibber wiring within a building's cell-based layout, along with a designated backbone connection point. The aim was to strategically position routers and devise an effective wiring configuration. The primary goal encompassed achieving optimal coverage while adhering to a predefined budget. The problem's score comprised two components: the count of cells covered by routers, multiplied by 1000, and the remaining budget. Notably, the scoring approach emphasized both extensive coverage and economical budget allocation.

In essence, this problem falls under the category of a maximum covering problem, as the central aim is to ensure the coverage of as many cells as possible. Furthermore, considering the budget limitations and wiring arrangement, we observe that this challenge shares similarities with the Steiner Tree Problem. This likeness arises from the possibility of determining the optimal cost of wiring placement based on the router locations, which may hold significance for the problem's resolution.

3.2.5   Google Hash Code 2018

Self-Driving Rides

Daily car commuting is a ubiquitous practice globally, involving trips to homes, schools, workplaces, and more. As a means of travel, cars remain a common choice, with ongoing efforts to enhance safety through the advancement of self-driving technology. In this challenge, contestants assume the role of managing a fleet of self-driving cars within a simulated setting. The goal is to ensure commuters reach their destinations securely and punctually.

With a fleet of cars at disposal and a roster of rides defined by their starting and ending intersections on a square grid representing the city, along with the earliest start time and the latest end time to ensure punctuality, the task is to allocate rides to vehicles. The aim is to maximize the number of completed rides before a predefined simulation time limit is reached. The scoring is determined by the summation of the individual ride scores. A ride's score is computed

as the sum of a value proportional to the distance covered during the ride, augmented by a bonus if the ride commences precisely at its earliest allowed start time.

Generally, this problem can be categorized as an assignment and vehicle routing problem with time windows. This classification arises from the necessity to assign rides to cars within specific time constraints. Notably, the car's route is determined by the sequence of rides assigned to it. Moreover, this challenge falls under the simulation category, as it directly impacts the scoring mechanism and cannot be simplified or abstracted.

Google City Plan

With the world's population increasingly concentrating in urban areas, the demand for expanded city infrastructure is on the rise. This entails not only residential buildings but also the incorporation of essential public facilities and services to cater to the growing populace. This challenge mirrors a scenario where participants are tasked with planning a city's building layout, involving both the selection of building types and their strategic placement.

For this challenge, participants receive building projects with specific width and height dimensions, covering both residential and utility structures. The city is a square grid of cells. Overall, the goal is to create buildings from these plans, arranging them within the city to optimize space and create a balanced mix of structures. This minimizes residents' walking distance to reach essential services. The overall score is the sum individual residential building scores, calculated by multiplying the number of residents and the number of utility building types within walking distance of that building. Notably, the walking distance parameter is specific to each problem instance.

Essentially, this problem belongs to the category of packing problems. The core objective revolves around determining how to fit buildings within the city layout. Importantly, there is no predetermined limit on the number of buildings that can be constructed for each plan, granting contestants the flexibility to make choices accordingly.

3.2.6   Google Hash Code 2019

Photo Slideshow

Given the surge in digital photography and the vast number of images traversing the internet daily, this challenge delves into the interesting concept of crafting picture slideshows using the available photo pool.

In this scenario, participants were tasked with creating a slideshow composed of pictures, which could be oriented either vertically or horizontally in the slides. Notably, a slide could contain two photos if they were arranged vertically. Additionally, these photos could be tagged with multiple descriptors corresponding to their subjects. The scoring of this problem revolves around the slideshow's appeal, determined by a calculated value that depends on consecutive slide pairs. This value is computed as the minimum between the tags count of the first picture, the second picture in the sequence and the count of the common tags shared between the two images.

Overall, this challenge can be categorized as a scheduling problem, to be precise, a single-machine job scheduling problem. If we liken the jobs to photos, the goal is to sequence them to optimize a specific objective function in this context, the "appeal" factor. Additionally, the interactions between slides introduce elements resembling a grouping problem.

Compiling Google

Given Google's extensive codebase spanning billions of lines of code across numerous source code files, compiling these files on a single machine would be time-consuming. To address this, Google distributes the compilation process across multiple servers.

This challenge tasks participants with optimizing compilation time by strategically distributing source code files across available servers. Notably, the compilation of a single code file can depend on other files being compiled prior to it, involving dependencies. Given a certain number of available servers and specific deadlines for compilation targets, the problem's score is calculated by summing the scores for the completion of each compilation target. These scores are determined by a fixed value for meeting the deadline, with an additional bonus if the compilation is completed ahead of the expected time.

This problem can be categorized as a scheduling problem, as the primary objective involves distributing compilation tasks (jobs) among different machines while adhering to dependencies between files. In essence, this problem resembles a variation of the classical job-shop scheduling problem.

3.2.7   Google Hash Code 2020

Book Scanning

Google Books is project that aims to create a digital collection of many books by scanning them from libraries and publishers around the world. In this challenge,

contestants are put in the position of managing the operation of setting up a scanning pipeline for millions of books.

Given a dataset describing libraries and available books, the objective of this challenge is to select books for scanning from each library within a specified global deadline. Each library has a distinct sign-up process duration before it can commence scanning, and only one library can be signed up at a time. Moreover, each library has a fixed scanning rate for books per day, and each scanned book contributes to the final score. The problem's goal is to maximize the overall score, which is calculated as the sum of the scores for unique books scanned within the given deadline.

This problem exhibits a combination of characteristics from classical scheduling, assignment, covering, and knapsack problems. It resembles a scheduling problem as the order in which libraries are signed up needs to be determined. It involves assignment, since libraries can share books, necessitating a decision on which libraries will scan each book. The covering aspect is apparent in the scoring mechanism, where the aim is to maximize the number of unique books scanned. Lastly, the problem also incorporates a knapsack-like element. While the time-related simulation factor exists, it can be abstracted into a knapsack scenario where the goal is to optimize the overall score by considering the number of books a library can scan until the deadline as its capacity.

Assembling Smartphones

Constructing smartphones is a intricate process that entails assembling a multitude of hardware components. This challenge delves into the concept of creating an automated assembly line for smartphones, employing robotic arms to streamline the manufacturing process.

Contestants are tasked with placing robotic arms within a workspace depicted as a rectangular cell grid. The objective is to optimize the arrangement of these arms to allow the execution of assigned tasks. Each task involves specific movements that a robotic arm must perform, essentially traversing a designated number of cells to accomplish the task. Notably, robotic arms cannot cross each other, necessitating precise task assignment and arm positioning to ensure unobstructed task execution for all arms. The problem's score is derived from the summation of scores obtained by successfully completing tasks within the constraints.

In summary, this challenge falls under the category of both assignment and scheduling problems since it encompasses the assignment of robotic arms to

suitable positions and the scheduling of tasks across these arms to optimize the completion of tasks.

3.2.8   Google Hash Code 2021

Traffic Signaling

This challenge delves into the optimization of traffic light timers to enhance the travel experience in a city. While traffic lights inherently contribute to road safety, their built-in timers are important in regulating traffic flow. The focus here is to fine-tune these timers with the aim of optimizing overall travel time for all commuters within the city.

Contestants are presented with a city layout, complete with intersections housing traffic lights. The task is to strategically allocate time intervals to these traffic light timers, optimizing traffic flow to ensure the maximum number of car trips are successfully completed within a predefined simulation time limit. The problem's score is the cumulative sum of scores assigned to each completed trip. These scores comprise a fixed value for trip completion and a bonus proportional to how early the trip concludes relative to the simulation's time limit. While the challenge may seem complex due to its detailed rules and operational aspects, its core objective revolves around this fundamental optimization process.

In summary, this challenge can be categorized as a simulation problem. It's worth highlighting that this problem aligns closely with the Signal Timing problem in the literature of Control Optimization.

Software Engineering at Scale

This challenge addresses the complexity of managing Google's vast monolithic codebase, which has grown significantly alongside the expanding number of engineers. To overcome the hurdles of effective feature deployment, participants are tasked with creating a solution that optimally schedules feature implementation work among engineers.

In this challenge, there are three primary components to be considered: features, services, and binaries. Each feature may require certain services, which can be present in specific binaries. The main objective is to efficiently assign features to engineers, considering that their implementation might entail additional tasks such as service implementation, binary relocation, new binary creation, or waiting for a designated time. The challenge revolves around optimizing this workflow to minimize delays caused by multiple engineers working in the

same service. The scoring is based on the sum of scores awarded for feature completion. Each completed feature's score is determined by the product of the number of users benefiting from it, as specified in the problem statement, and the number of days between the maximum day (also defined) and the day the feature was launched.

In essence, this challenge falls within the realm of classic scheduling problems. It involves assigning tasks (jobs) to engineers with the aim of optimizing a quantity influenced by the order in which each engineer performs their tasks and the interactions of tasks among multiple engineers.

3.2.9   Google Hash Code 2022

Mentorship and Teamwork

This challenge delves into the concept of a teamwork environment, where knowledge sharing among peers and collaborative efforts are central to task completion. In this challenge, participants are tasked with orchestrating a team comprising individuals with diverse backgrounds to successfully execute projects that demand a variety of skills.

The main goal is to efficiently assign a list of contributors, each possessing specific skills and the potential to improve them through project involvement, mentoring, or being mentored. The challenge involves allocating contributors to projects with skill requirements to ensure timely completion. Notably, contributors can participate in multiple projects concurrently. The key factor here is the order in which contributors develop or enhance their skills, a decision that significantly impacts the overall project completion process. The score in this challenge is the sum of project scores achieved by completing them before the defined overall deadline. A project's score comprises a fixed value for completion, minus penalty points if it surpasses the deadline but is still within a tolerance window. Projects exceeding the deadline or tolerance won't add to the score but will still contribute to workers' training.

In summary, this challenge shares similarities with a scheduling problem, as it involves assigning projects to contributors while considering the order in which they are completed to maximize the overall score achieved through project completion.

Santa Tracker

The *Google Santa Tracker* is a project that visualizes the route taken by the famous *Santa Claus* character during his December gift distribution to children

globally. In this challenge, participants were tasked with optimizing the delivery route to enhance the efficiency of gift distribution.

The challenge scenario revolves around a 2D cell grid with no friction, symbolizing the world. Within this grid, children are located, and two types of items, carrots (providing speed boosts) and gifts, can be picked up by the cart. While the cart maintains its speed on the frictionless grid, the total weight affects the impact of carrot consumption. Thus, the main goal consists in devising a route that efficiently delivers the most gifts within the time constraints. The scoring metric for this problem involves summing the scores of successfully delivered items.

In summary, this challenge can be categorized as a type of Vehicle Routing Problem, specifically a Capacitated with Pick up and Delivery Vehicle Routing Problem. This is due to the presence of capacity constraints on the cart and the need to pick up and deliver items throughout the cart's journey.

3.2.10   Outline

In summary, this section provided an overview and description of the key aspects of the Hash Code problems. Furthermore, a categorization that links these problems to topics commonly found in combinatorial optimization literature was presented. The Table 3.1 shows a summary of the analysis conducted.

Problem	Categories						
	Assignment	Knapsack	Coverage	Vehicle Routing	Simulation	Scheduling	Packing
Street View Routing			✓	✓			
Optimize a Data Center	✓	✓					
Loon			✓	✓	✓		
Delivery				✓			
Satellites	✓		✓		✓		
Streaming Videos	✓	✓					
Router Placement			✓				
Self-Driving Rides	✓			✓	✓		
City Plan							✓
Photo Slideshow						✓	
Compiling Google						✓	
Book Scanning	✓	✓	✓			✓	
Assembling Smartphones	✓					✓	
Traffic Signaling					✓		
Software Engineering at Scale						✓	
Mentorship and Teamwork						✓	
Santa Tracker				✓			

Table 3.1: Categorization of Google Hash Code Problems

3.3   Instances

In the competition context, in combination with the problem statements, test case instances are provided to participants with the primary aim of providing a

mechanism for scoring teams, thus quantitatively assessing the efficacy of their strategies. These instances are carefully generated to conform to the stipulated limits and constraints inherent to the challenge, as described in upon in the problem statement.

The initial instance, commonly denoted as the "example", is routinely included within the problem statement for contestants' reference. This instance is included in the problem statement for contestants' reference, but is not solved optimally. Its purpose is to illustrate the input and output format for the instance and solution. However, the example is intentionally designed with small dimensions, making it approachable via exact brute force methodologies.

Subsequent instances are typically designed to push the boundaries of the problem. These instances are intentionally large and design to discourage exact methods and general heuristics, aiming to thoroughly examine various aspects of the problem and avoid that (non-exact) greedy approaches find an optimal solution. The ruggedness of their objective space introduces challenges for solvers, potentially rendering some of them ineffective or even unusable within the available time budget.

In the competition context, teams are allowed to provide unique solutions for each instance, thus becoming a common practice among participants to conduct thorough cross-instance analysis. This practice proves valuable in revealing patterns that can offer insights into tackling the challenge with greater efficiency. As such, participants have the flexibility to develop focused strategies for each instance. This can in fact be interesting for the study of general-purpose meta-heuristics, to understand whether they can achieve comparable results to instance-specific approaches.

Finally, given the articulated problem statements and the transparent instance generation process, participants can create customized test instances. This capability proves valuable for debugging purposes in a competition setting and further advocates these problems as interesting benchmarks for black-box optimization [31].

3.4   Concluding Remarks

In this chapter, we conducted a comprehensive exploration of the Google Hash Code competition, delving into its structure, problem descriptions, and instances. In particular, we established links between the challenges presented and well-known CO problems. Furthermore, we highlighted common techniques employed by participants, drawing from our own engagement over

several years.

We consider this analysis to be an important step that not only facilitated a deeper comprehension of the challenges, but also guided our choice of two specific problems (Optimize a Data Center and Book Scanning) for detailed exploration in this study. The particular choice of these problems is mainly motivated by the range of combinatorial optimization topics covered, leaving only vehicle routing and simulation as subjects to address in future work.

Moreover, based on the conducted analysis, we once again emphasize the importance of these problems as promising candidates for BBO [31]. However, we believe that for this potential to be realized, it is essential to establish a repository containing the scores achieved across various problems and instances. Ideally, this repository should be accompanied by the corresponding source code for reproducibility purposes. From the standpoint of experimentally evaluating meta-heuristics for these problems, we consider it vital to generate a diverse set of instances, eventually generated through different methods.

Having understood the Google Hash Code problems, in the ensuing chapters, we will discuss our modelling approach to solve them, analyze the chosen problems, and conclude with a reflection on the work carried out.

Chapter 4

Principled Modelling Framework

> *"Before software can be reusable, it first has to be usable."*
>
> — Ralph Johnson

In this chapter, we introduce a modeling framework designed to address the challenges of Google Hash Code problems. This framework integrates existing modeling concepts from the POF, nasf4nio, and nasf4nio-cs projects. In Section 4.1, we establish the model by defining a standard specification that supports the development of meta-heuristics. Subsequently, in Section 4.2, we provide examples illustrating how these operations can be utilized for crafting meta-heuristics solvers. Moving forward, in Section 4.3, we reflect on how the this model specification can aid in the (property) testing process. Finally, we conclude the chapter in Section 4.4, where we offer observation on the framework's usage within the context of this work.

4.1   Specification

This section, establishes a collection of operations that a model must support, all while motivating their necessity and applicability in the context of meta-heuristic solver development. To achieve this, our description will draw from the Python-based implementation that we've devised. However, it is crucial to underline that our implementation serves as a demonstration of a plausible way to materialize an API that facilitates modeling operations. As such, considerations such as naming conventions and programming language-specific concepts ought to be viewed as secondary.

Moreover, it is important to underscore that this framework represents a consolidation of prior efforts undertaken by Vieira [15], Fonseca [34], and Outeiro [35]. Consequently, it should be perceived as an enhancement of the prerequisites and specifications established in those works, tailored to encompass constructive and local search techniques, as well as the comprehensive modeling procedure.

4.1.1 Data Structures

This framework is based around a set data structures that are standard and utilized as a way to return information from the model. This design allows for data hiding and isolation of the problem-specific details, which any MH solver can query and manipulate in a black-box fashion. In the our implementation of the framework this corresponds to the following four classes.

- `class Problem: ...`
  Contains data that is known a priori and fully characterizes a problem instance. It is not meant to be modified by the solver.

- `class Solution: ...`
  Holds data that defines an empty or partially complete solution. It serves as the mutable state that a solver can modify during the optimization process.

- `class Component: ...`
  Characterizes any component within the ground set of a given problem.

- `class LocalMove: ...`
  Characterizes any local move that can be applied to a solution

Concerning the `Problem` class, since it holds problem-specific data, it might be advantageous to include methods within it for loading or parsing input data, in addition to the class constructor. However, the primary intent remains focused on storing problem-specific data and providing a away to initialize a solution for a specific problem instance. This can be achieved through the following method, which returns an empty solution.

```
def empty_solution(self: Problem) → Solution: ...
```

It is important to acknowledge that the definition of an empty solution is tied to the specific problem under consideration. Nevertheless, this method when invoked returns an encapsulated solution object that abstracts the problem's details.

As previously mentioned, the Component class serves as a representation of a component from the ground set, encapsulating its relevant data. Nonetheless, certain MH methods, such as, ACO, require a means for identifying specific components during solution construction. To address this requirement, the following method was incorporated:

```
def id(self: Component) → Hashable: ...
```

As can be inferred by the methods's type signature, this method should return an unique (Hashable) identifier for a given component.

Concerning the LocalMove class, it primarily encapsulates information about a specific local move applicable to a particular solution, and it does not necessitate any additional functionality. Notably, these two classes are frequently used in the model as they embody the dynamic aspects, encompassing actions capable of altering solution states. Thus, from a practical standpoint, their instances should be kept lightweight as they can have a significant impact on performance.

The Solution class is of extreme importance as it represents object undergoing optimization. Apart from housing data fields that correspond to the solution representation – which will be populated during solver execution – this class also contains the method definitions that we consider fundamental and which describe the *model*. Consequently, in the subsequent sections, we will comprehensively detail the methods within this class. These methods represent the operations that a model should implement to allow CS, LS procedures and other essential functionality.

4.1.2   Inspection Methods

The methods outlined in this section provide access to metrics that characterize the state a solution. In terms of implementation, these function as *getters* for the model's properties and should not be utilized for executing resource-intensive computations, given the high frequency at which they are meant to be invoked by solvers.

Objective Value

```
def objective(self: Solution) → Optional[T]: ...
```

This method returns the objective value of a solution. In the type signature for this method, the return type T signifies a generic type. While this type is usually numeric, it can be any other *comparable* type. This requisite is fundamental

from the solver's standpoint, as it's through comparing this value that a solver evaluates whether a particular solution outperforms others.

The wrapping `Optional` type indicates that a method has the option to either return a value or, if the value is not available or not defined, it can emit a sentinel value. In our implementation, this sentinel value is represented by `None`. This convention will be consistently applied to all methods where the `Optional` type is employed.

Upper Bound

```
def upper_bound(self: Solution) → Optional[T]: ...
```

This method returns the upper bound value of a given solution. Similar to the `objective` method, the parameter `T` in the function's type signature signifies a generic type, typically numeric.

Feasibility

```
def feasible(self: Solution) → bool: ...
```

This method can be used to assess the feasibility of a candidate solution, yielding a Boolean value of `True` when the solution is feasible, and `False` otherwise.

4.1.3 Constructive Search

The subsequent methods delineate modeling operations for CS. As such, these methods revolve around operations involving components. Specifically, these action involve enumeration, selection, application, and evaluation of their contribution with respect to the objective and upper bound values.

Component Enumeration

From a CS perspective, a central operation involves recognizing the components that constitute the problem's ground set – those that can be added, or removed from a specific solution. Therefore, it is imperative for a model to offer a means through which a solver can access this essential information. This is accomplished in our implementation via the following set of methods.

```
def add_moves(self: Solution) → Iterable[Component]: ...
```

```
def remove_moves(self: Solution) → Iterable[Component]: ...
```

Notably, all these methods yield a list of components specific to their corresponding operations, denoted as `Iterable[Component]` in the return type

signature. However, in our implementation, these methods are implemented as *iterators*, signifying that upon invocation, they provide the first available item and then pause until the next call to the same method is initiated, essentially maintaining state. This approach holds significance from a performance standpoint, both in conserving memory, as storing all components for a given problem could be infeasible, and in terms of execution speed, since there's no necessity to enumerate more components than what is required.

Furthermore, the intention behind these methods is for them to yield all conceivable components that can be either added or removed from a specific solution. However, in practice, this might become unfeasible if the ground set is excessively large. In such cases, an alternative strategy for component enumeration may need to be contemplated. It's important to note that this consideration is relevant to the implementation of the method itself by the user who models the problem, but it doesn't alter the method signature in any way.

Finally, within the context of ACO, there's a need to enumerate all components that are already included in a solution. This requirement is due to the necessity to update the *pheromone model.* As a result, this justifies the introduction of the following method.

```python
def components(self: Solution) → Iterable[Component]: ...
```

Heuristic Component Enumeration

In the context of component enumeration, it can be advantageous to follow a problem-specific heuristic order. With this in mind, we introduced the `heuristic_add_moves` method, designed to offer an iterator for components enumerated in accordance with the heuristic. Additionally, we introduced the `heuristic_add_move` method to provide a means for exclusively returning a single heuristic move, if available. The type signatures for both methods are shown below.

```python
def heuristic_add_moves(self: Solution) → Iterable[Component]:
    ↪  ...

def heuristic_add_move(self: Solution) → Optional[Component]:
    ↪  ...
```

Random Component Selection

As components can be enumerated in a heuristic order, there are situations where selecting components randomly is advantageous. This is particularly relevant, for instance, during processes like random solution construction or

destruction (IG). The subsequent methods, when called, provide randomized components that can potentially be added or removed, contingent upon the current state of the solution.

```python
def random_add_move(self: Solution) → Optional[Component]: ...

def random_remove_move(self: Solution) → Optional[Component]:
                                ↪ ...
```

Component Application

It is essential for all constructive search methods to possess a mechanism for incorporating or removing a component from a solution under construction. Therefore, the ensuing methods have been introduced, and upon invocation, allow the addition and removal of the specified component to/from the solution.

```python
def add(self: Solution, c: Component) → None: ...

def remove(self: Solution, c: Component) → None: ...
```

Objective Increment Calculation

When evaluating the contribution of a component to the objective value of a solution, there are two approaches to consider. The first approach involves recalculating the entire objective value from scratch each time a component is added or removed from the solution (full evaluation). The second approach involves incremental evaluation, where only the change in objective value due to the addition or removal of the component is computed. The latter method is more efficient as it avoids the potentially expensive operation of recomputing the entire objective value and should be preferred for its performance advantages.

This fact motivates the overall design of the `Solution` API and thus all operations involving evaluations are considered in function of their increments. Albeit, it is possible for a user to implement a full evaluation and return only the increment, but that aspect is practical and beyond the scope of this specification.

That being said, the following methods are exposed, granting a solver the capability to quantify the objective value contribution (increment) that is obtained from adding or removing a designated component with respect to the current state of the solution.

```python
def objective_increment_add(self: Solution, c: Component) →
                    ↪ Optional[T]: ...
```

```
def objective_increment_remove(self: Solution, c: Component) →
                      ↪ Optional[T]: ...
```

The return type of these methods is generic (denoted as T), with the precondition that it must be comparable and for all increment related methods it must support the basic arithmetic operations of addition and subtraction.

Upper Bound Increment Calculation

Just as there's a need to calculate increments concerning the objective value, the same functionality is essential for upper bounds. This leads to the introduction of the following methods, which serve the purpose of quantifying the increment in the upper bound resulting from the addition or removal of a component from a solution. Once more, the preference should lean towards incremental evaluation for determining this value, as it's often even more resource-intensive than calculating a contribution to the objective value.

```
def upper_bound_increment_add(self: Solution, c: Component) →
                      ↪ Optional[T]: ...

def upper_bound_increment_remove(self: Solution, c: Component)
                      ↪ → Optional[T]: ...
```

Component Heuristic Value

The concept of a heuristic value is associated with a component and offers a means to rank the importance of components with respect to the current solution state via a problem-specific heuristic. The method's type signature is as follows.

```
  def heuristic_value(self: Solution, c: Component) →
                      ↪ Optional[T]: ...
```

Notably, this method was introduced to offer an alternative approach for the GRASP algorithm to construct a randomized solution.

4.1.4   Local Search

The subsequent methods define modeling operations for LS approaches. These methods are centered on operations concerning local moves, encompassing their enumeration, selection, application, and evaluation.

Local Move Enumeration

In a manner similar to the approach adopted for CS, the same principle holds when addressing LS. Here, the necessity arises to enumerate all plausible alternatives for optimizing (exploiting) a candidate solution, which in turn pertains to the exploration of neighboring solutions. To fulfill this requirement, the following method serves as a mechanism for enumerating all the local moves that can be executed on a given solution. This method adheres to the same pattern as previous enumeration methods as it is meant to function as an *iterator* for feasible local moves.

```python
def local_moves(self: Solution) → Iterable[LocalMove]: ...
```

Random Local Move Enumeration

While local moves can indeed be enumerated in a specific order, as enabled by the `local_moves` method, the majority of local search strategies depend on random processes for selecting moves. This motivation gives rise to the subsequent two methods: one for systematically *iterating* all potential local moves in a randomized manner without repeating the same local move twice, *i.e.*, without replacement (`random_local_moves_wor` method), and the other for returning a random local move (`random_local_move` method).

```python
def random_local_moves_wor(self: Solution) →
                ↪ Iterable[LocalMove]: ...

def random_local_move(self: Solution) → Optional[LocalMove]:
                        ↪ ...
```

When it comes to random enumeration of a candidate solution's neighborhood without replacement, as performed by the `random_local_moves_wor` method, important insights can be obtained regarding its efficient implementation.

Conventionally, a naive method would involve shuffling all M possible local moves, selecting a local move, utilizing it, and then repeating the shuffling process minus the k moves that were rendered infeasible due to the use of the chosen move. This would leave the remaining $M - k$ moves for selection. Nonetheless, repeating this until exhausting all local moves would result in an additional space complexity of $O(M)$. However, as this approach can be memory-intensive, we propose an alternative method that achieves the same outcome with a space complexity of $O(1)$.

The alternative approach entails employing a Linear Congruential Generator (LCG) to achieve the same outcome with a constant spacial complexity, thereby

eliminating the need for additional space.

Definition 4.1.1 (Linear Congruential Generator [28]). *A linear congruential generator is a pseudo-random number generator described by the following recurrence relation:*

$$X_{n+1} = (aX_n + c) \bmod m \tag{4.1}$$

Where,

$$m, \quad 0 < m - \textit{is the modulus}$$
$$a, \quad 0 < a < m - \textit{is the multiplier}$$
$$c, \quad 0 \le c < m - \textit{is the increment}$$
$$X_0, \quad 0 \le X_0 < m - \textit{is the ``seed'' or ``starting value''}$$

For a LCG it is known [28] there is a choice of parameters that guarantees a maximal period for generator, and equal to m. This in means that all the numbers until m will be emitted by the generator in a random order exactly once.Albeit, the choice of parameters must obey the following conditions [28]:

1. c is relatively prime to m

2. $a - 1$ is a multiple of all prime factors dividing m

3. $a - 1$ is multiple of 4, if m is a multiple of 4

For example, the choice of the parameters $a = 5$, $m = 8$ and $c = 1$, yields a LCG that has maximal period. Remarkably, the choice of m as a power of two has important implications in terms of performance from a computational perspective, allowing for faster modulus operations.

With this in mind, a properly configured LCG can be harnessed for the purpose of random local move enumeration without replacement. This is achieved by associating each local move with a number $0 < i < m$, which, upon generation, is decoded into the corresponding LocalMove object. Notably, the parameter m needs to be adjusted in order to contemplate all possible moves.

Local Move Application

This method is designed to implement a problem-specific perturbation to a given solution during LS. The integer parameter in the method signature provided below is employed to determine the "kick strength", which signifies the magnitude of the perturbation to be applied.

```
def step(self: Solution, m: LocalMove) → None: ...
```

Local Move Objective Increment Calculation

Similar to the approach used for CS, where the `objective_increment_add` method is employed to evaluate the incremental contribution of a particular component to the objective value, a comparable functionality is necessary for local search. However, in this case, the evaluation pertains to local moves. This requirement is addressed through the following method.

```
def objective_increment_local(self: Solution, m: LocalMove) →
                        ↪ Optional[T]: ...
```

From an implementation perspective, similar to all incremental evaluations (including those for CS), this operation should maintain efficiency. Arguably, this could be one of the most frequently used operations within an optimization process, as it enables the assessment of whether a candidate solution has improved or worsened following a particular action.

Perturbation

This method serves the purpose of applying a problem-specific perturbation to a given solution undergoing LS. The integer parameter in the method signature shown bellow is used to identify the "kick strength", *i.e.* the intensity of the perturbation to be applied.

```
def perturb(self: Solution, ks: int) → None: ...
```

4.1.5   Utility Methods

From an implementation perspective, there exists a need for a *copy* operation to manage the preservation of the best solution(s) found in memory. This operation prevents the unintended overwriting of a given solution due to modifications made during the optimization process. In the specific context of this framework, this operation applies to the `Solution` object and is significant for both CS and LS approaches.

Although it might be perceived as an implementation detail, we consider that it deserves its distinct place within the model definition. This stems from the critical aspect that a solution object must consistently access the problem object, which contains problem-specific information. However, during the copying process of a solution, duplicating this information is usually unnecessary, and

referencing it suffices. With this in mind, the following method was, introduced to address this requirement.

```python
def copy(self: Solution) → Solution: ...
```

4.1.6 Outline

In summary, this specification encompasses what we consider as the model within the realm of meta-heuristics. While not formulated mathematically but rather in computational terms, this representation through the described constructs, from our perspective, effectively captures the attributes of a problem. It enables the problem to be addressed in a black-box manner by solvers, which make use of this standardized set of methods, as we will describe next, in Section 4.2. The modeling endeavor then involves implementing each of these methods in accordance with their context within a particular problem.

The Table 4.1 serves as a summary of all the methods related to the model API which were documented in this section and are part of our implementation of the principled modeling framework [40].

4.2 Solvers

This section introduces a potential implementation of solvers using the proposed framework. The aim is to provide an overview of how meta-heuristic algorithms can be developed utilizing the methods provided by the model API. To accomplish this, we will start by describing a basic solver that utilizes heuristic information for constructing solutions. Subsequently, we will demonstrate how solvers for both CS and LS methods can be implemented, using BS and ILS meta-heuristics as illustrative examples.

4.2.1 Heuristic Construction

The Listing 4.1 describes a possible implementation of a MH solver that iteratively constructs a solution using only heuristic information. This solver takes as input a `Problem` instance and returns a `Solution`.

This solver can be summarized as follows:

1. **Initialization**: Initialize a new empty `Solution`. and select a `Component` to be added heuristically (lines 2-3)

2. **Construction**: Build the solution (lines 4-6).

```
1: def heuristic_construction(problem: Problem) → Solution:
2:    solution = problem.empty_solution()
3:    c = solution.heuristic_add_move()
4:    while c is not None:
5:      solution.add(c)
6:      c = solution.heuristic_add_move()
7:    return solution
```

Listing 4.1: Heuristic Construction Solver Implementation

(a) If there is a component (c) available for addition to the current solution, (*i.e.*, c is not the None sentinel value), proceed; otherwise stop the **Construction**. (line 4)

(b) Add the selected component to the solution (line 5).

(c) Heuristically select a new component (line 6).

(d) Repeat the process.

3. **Termination** Return the final Solution (line 7).

In fact, this simple solver can serve as practical tool for testing the implementation of a model. Its simplicity arises from its reliance on only two fundamental methods: the add method, which introduces a component to the solution, and the heuristic_add_move method, which determines the next component to be added using a heuristic criterion. For testing purposes, other uncomplicated MH solvers can be crafted using the same model. For instance, solvers that handle construction in a random or a in greedy manner, based on components that yield the most favorable increments with respect to the objective value and/or bound.

From an implementation perspective, this particular version takes a problem instance as input and uses it to generate an initial empty solution. However, this method could have been designed to accept a partial solution and then complete its construction from that point onwards. Albeit, the option to provide the problem instance rather than a solution as a parameter was chosen primarily for illustrative purposes and to showcase the interaction between the Problem and the Solution data structures.

4.2.2   Beam Search

The Listing 4.2 presents a possible implementation of a BS constructive MH solver. This MH, as discussed earlier in Chapter 2, can be viewed as having two phases. In the first phase, it generates a candidate list of partial solutions

that extend the solutions present in a limited-size archive. Then, in the second phase, it filters and selects the best solutions based on either the upper bound value or a heuristic value. The selected solutions are retained in the archive for the next iteration. Additionally, during this process, if a feasible solution is encountered, the best solution found thus far is updated accordingly.

The execution time of this algorithm is not limited a stopping criterion is required. In this implementation, we define the stopping condition as a time budget, which is represented by the `Timer` data structure. The `finished` method of the `Timer` indicates whether the allocated time budget has been used up. If the time budget is exhausted, the algorithm stops.

Additionally, this solver takes other parameters as input in this implementation. These include a `Problem` instance and the beam width (bw), which specifies the size of the solution archive. The choice for supplying the problem as a parameter instead of a solution follows the same motivation as previously mentioned in Section 4.2.1.

```
1: def beam_search(problem: Problem, timer: Timer, bw: int) → Solution:
2:    solution = problem.empty_solution()
3:    best = solution if solution.feasible() else None
4:    bobj = solution.objective() if solution.feasible() else None
5:    beam = [(solution.upper_bound(), solution)]
6:    while not timer.finished():
7:      cs = []
8:      for ub, s in beam:
9:        for c in s.add_moves():
10:          cs.append((ub + s.upper_bound_increment_add(c), s, c))
11:      if not len(cs):
12:        break
13:      cs.sort(reverse=True, key=lambda x: x[0])
14:      beam = []
15:      for ub, s, c in cs[:bw]:
16:        s = s.copy()
17:        s.add(c)
18:        if s.feasible():
19:          obj = s.objective()
20:          if bobj is None or obj > bobj:
21:            best, bobj = s, obj
22:        beam.append((ub, s))
23:    return best
```

Listing 4.2: Beam Search Solver Implementation

In brief, this BS solver implementation is described by the following steps:

1. **Initialization**: Start by initializing an empty `Solution`. If this initial solution is `feasible`, store it as the best solution found so far. Add a tuple representing this solution to the archive (beam), including its upper

bound value. (lines 2-5)

2. **Construction**: Construct the solution while there is still available time in the budget (lines 6-22).

   (a) Create an empty candidate list (`cs`) (line 7).

   (b) **Branch**: Expand the current solutions in the archive by adding components to create new partial solutions (lines 8-10).

      i. Select a solution from the archive (line 8).

      ii. Enumerate all possible components that can be added to that solution (line 9).

      iii. Add these new solutions to the candidate list `cs` as tuples containing the upper bound value, the partial solution, and the component leading to this contribution (line 10).

      iv. Repeat while possible.

   (c) If the candidate list (`cs`) is not empty, filter the `bw` best candidate solutions based on their upper bound values. (lines 11-13)

   (d) Empty the the `beam` list (line 14).

   (e) **Update**: Build the beam for the next iteration and update best solution (lines 15-22).

      i. Select a candidate solution (make a `copy`) to potentially update the best solution (lines 16).

      ii. Add the respective component to the current candidate solution (line 17).

      iii. If the solution is feasible and its objective value is better than the best solution found so far, update the best solution by replacing it with the current one (lines 18-21).

      iv. Add the candidate solution to the `beam` list (line 22).

      v. Repeat while possible.

3. **Termination** Return the best `Solution` found (line 23).

It is important to note that, this algorithm requires the implementation of certain methods within the model, specifically in `Solution` class. However, the algorithm can still function successfully even if some of the remaining methods are not implemented, as long as these mandatory methods are provided.

4.2.3 Iterated Local Search

The Listing 4.3 presents a possible implementation of an ILS solver which is
a LS MH. As discussed in Chapter 2, This meta-heuristic commences with a
feasible solution and subsequently employs a LS procedure. Upon discovering
a solution that outperforms the current best solution, the algorithm updates
the best solution, and optionally perturbs the solution, before repeating the
entire process. In our implementation, we use FI for as the LS strategy.

Similarly to the previous section's implementation of the BS, this algorithm is
also not limited by execution time and requires a stopping criterion. Here, we
utilize a time budget represented by the Timer object, as previously discussed.
Additionally, this solver accepts several other input parameters. These include
a Solution instance and the *kick strength* (ks), which determines the magnitude
of the perturbation to be applied to the solution.

```python
 1: def ils(solution: Solution, timer: Timer, ks: int) → Solution:
 2:   best = solution.copy()
 3:   bobj = best.objective()
 4:   while not timer.finished():
 5:     for m in solution.random_local_moves_wor():
 6:       increment = solution.objective_increment_local(m)
 7:       if increment > 0:
 8:         solution.step(m)
 9:         break
10:       if timer.finished():
11:         if solution.objective() > bobj:
12:           return solution
13:         else:
14:           return best
15:     else:
16:       if solution.objective() >= bobj:
17:         best = solution.copy()
18:         bobj = solution.objective()
19:       else:
20:         solution = best.copy()
21:       solution.perturb(ks)
22:   if solution.objective() > bobj:
23:     return solution
24:   else:
25:     return best
```

Listing 4.3: Iterated Local Search Solver Implementation

In brief, this ILS solver implementation is described by the following steps:

1. **Initialization**: Begin with the current best solution as the starting
   solution. (lines 2-3)

2. **Local Search**: Conduct local search on the starting solution while there

is available time in the budget. (lines 4-21)

    (a) **First Improvement**: Enhance the solution through the first local move that results in a better objective value. (lines 5-14)

        i. Randomly pick a local move from the neighborhood of the current solution. (line 5)

        ii. Calculate the incremental change in the objective value due to the local move. (line 6)

        iii. Check if applying the current local move results in a better solution. (line 7)

        iv. If improvement is achieved, apply the local move and proceed to the **Update** stage. (lines 8-9)

        v. If the time budget is exhausted during the first improvement stage, halt the **Local Search** and return the best solution found so far. (lines 10-14)

        vi. Repeat this process until all possible local moves have been explored.

    (b) **Update**: Update the current best solution if an improvement was achieved in the first improvement stage. (lines 15-20)

    (c) **Perturb**: Optionally, perturb the best solution with a kick strength of ks, if an improvement was obtained in the first improvement stage. (line 21)

3. **Termination** Return the best Solution found. (lines 22-15)

4.2.4   Outline

This section demonstrated the process of implementing solvers using the provided modeling framework. We provided practical examples for both CS and LS meta-heuristics. However, it is important to note that in this work, all the meta-heuristics described in Chapter 2 were implemented [40]. Furthermore, each of these meta-heuristics requires different methods from the model, which we summarize in Table 4.2.

4.3   Property Tests

As discussed, this framework offers capabilities in both problem modeling and
the implementation of meta-heuristic solvers. This versatility allows it to satisfy
the needs of two distinct user profiles: those primarily focused on problem-
solving, seeking access to a variety of solvers for effective results, and MH
developers who concentrate on MH refinement. The latter group employs the
framework's provided problems as a means to rigorously evaluate and test their
developed meta-heuristics. Albeit, there can also be individuals interested in
both aspects.

A common thread among all users of the framework is the need to test imple-
mentations. From the vantage point of a MH developer, the primary objective
revolves around constructing tests that assess the meta-heuristic's correct im-
plementation, *e.g.*, *unit-tests*. From the perspective of a model developer, these
tests, while valuable for assessing the correctness of the model's implementa-
tion, extend beyond mere verification of whether methods yield the expected
result. Instead, the tests should encompass the evaluation of whether these
outcomes adhere to fundamental properties that should universally held true
for any given input-output pair, irrespective of the specific problem at hand.
This type of testing, is known as *property testing*.

To illustrate, let's take a scenario where a component we intent to add a compo-
nent to a solution. In a conventional *unit-test* approach, one could verify if the
objective value of the solution matches the expected outcome after adding that
particular component. Yet, a broader property to consider is that, irrespective
of the specific objective value, the solution's value following the component
addition should be equivalent to the value prior to the addition, plus the in-
cremental contribution attributable to that specific component. This property
testing goes beyond specific cases and encapsulates the core behavior of the
operation.

```
 1: def objective_increment_add_test(self: Problem) → Solution:
 2:   solution = self.empty_solution()
 3:   c = solution.random_add_move()
 4:   while c is not None:
 5:     before = solution.objective()
 6:     increment = solution.objective_increment_add(c)
 7:     solution.add(c)
 8:     assert before + increment == solution.objective()
 9:     c = solution.random_add_move()
10:   return solution
```

Listing 4.4: Objective Increment Add Property Test

Importantly, the design of the modeling framework offers a user-friendly interface for creating these tests, as exemplified in Listing 4.4. In essence, the implementation of a property test for an objective value increment is follows the subsequent steps:

1. **Initialization**: Start by initializing a random solution and selecting a random component to add to the solution. (lines 2-3)

2. **Property Test**: Execute a property test by constructing a random solution. (lines 4-9)

   (a) If a random component (`c`) is available for addition to the current solution (*i.e.*, `c` is not the `None` sentinel value), proceed; otherwise, conclude the **Property Test**. (line 4)

   (b) Record the objective value of the current solution and calculate the increment achieved by applying the chosen component. (lines 5-6)

   (c) Add the component to the solution. (line 7)

   (d) Assert that the sum of the objective value before adding the component and the calculated increment equals the current objective value. If this assertion fails, the property is not verified. (line 8)

   (e) Choose a new component and repeat the process. (line 9)

3. **Termination** Conclude the property test and return the `Solution`. (line 10)

It is worth noting that this type of testing is inherently randomized, and it should undergo multiple iterations to ensure statistical validation of the implemented model's adherence to the properties. Furthermore, this test is designed to take the `Problem` instance as an input, illustrating the potential for seamlessly integrating (or *injecting*) these tests into any `Problem`. This offers the same possibility as with implementing solvers — allowing others to also develop and contribute with test implementations in a similar manner.

Additionally, this specific example of a property test is just a single instance, with the potential for similar tests to be conducted for various types of operations. For instance, one could examine the enumeration of local moves — whether performed in a random manner (`random_local_moves_wor`) or following a specified order (`local_moves`) — to validate that they consistently yield identical components and maintain a consistent count of components with each iteration. This underscores the versatility and ease-of-use of property tests in evaluating a range of aspects within the proposed framework.

4.4   Concluding Remarks

In this chapter, we a proposed framework for problem modeling that enables the development of solvers capable of addressing problems in a black-box manner. While we used our *Python* implementation to illustrate the framework from a modeling, solver development, and testing perspective, the fundamental constructs of this framework can be extrapolated and applied to any programming language. Our intent was to demonstrate the general principles that underlie the framework. We applied this implementation of the framework to model two specific problems in the Google Hash Code competition, which we will discuss in Chapters 5 and 6.

The selection of *Python* as our programming language, for developing the framework and models, was primarily driven by its ease of use, rapid prototyping capabilities, and rich high-level features. However, the language interpreted nature and relative slowness compared to compiled languages can impact the efficiency of solver execution. To mitigate this, we turned to an alternative *Python* implementation known as *PyPy*. This just-in-time (JIT) compiled version of Python has a substantial speed boost, claiming to be approximately 4.8 times faster than the on found and average interpreter (*CPython*).

The decision to employ *PyPy* proved crucial, as it significantly improved the execution times of our solvers. This was particularly evident when dealing with complex problem instances, where the performance difference was remarkable. Albeit, while the current *Python* implementation served its purpose for prototyping and exploring ideas for problem-solving, a reimplemented version of this framework in a more performant programming language would be necessary for optimal performance. Nonetheless, for brainstorming and rapidly testing new ideas from a modeling standpoint, from our perspective, this framework remains a valuable choice.

Class	Methods	Input	Output
Problem	empty_solution	—	Solution
	copy	Solution	Solution
	feasible	—	bool
	objective	—	Optional[T]
	upper_bound	—	Optional[T]
	components	—	Iterable[Component]
	add_moves	—	Iterable[Component]
	remove_moves	—	Iterable[Component]
	heuristic_add_moves	—	Iterable[Component]
	heuristic_add_move	—	Optional[Component]
	random_add_move	—	Optional[Component]
	random_remove_move	—	Optional[Component]
Solution	local_moves	—	Iterable[LocalMove]
	random_local_moves_wor	—	Iterable[LocalMove]
	random_local_move	—	Optional[LocalMove]
	add	Component	None
	remove	Component	None
	step	LocalMove	None
	perturb	int	None
	heuristic_value	Component	Optional[T]
	objective_increment_local	LocalMove	Optional[T]
	objective_increment_add	Component	Optional[T]
	objective_increment_remove	Component	Optional[T]
	upper_bound_increment_add	Component	Optional[T]
	upper_bound_increment_remove	Component	Optional[T]
Component	id	—	Hashable
LocalMove	—	—	—

Table 4.1: Modeling API Specification

Note: The parameter T denotes a generic comparable type. In the case of the increment functions this type must also support addition and subtraction. The class LocalMove does not expose any methods.

Meta-Heuristic	Methods	
BS	copy	objective
	upper_bound	feasible
	add	add_moves
	upper_bound_increment_add	
GRASP	copy	objective
	add	feasible
	upper_bound_increment_add	add_moves
IG	copy	objective
	add	feasible
	upper_bound	remove
	add_moves	random_remove_move
	upper_bound_increment_add	
ACO	copy	feasible
	objective	add
	add_moves	upper_bound
	upper_bound_increment_add	components
FI	step	objective_increment_local
	random_local_moves_wor	
BI	step	local_moves
	objective_increment_local	
ILS	objective	copy
	step	perturb
	random_local_moves_wor	objective_increment_local
ILS	copy	objective
	step	random_local_moves_wor
	objective_increment_local	
TS	copy	objective
	step	random_local_moves_wor
	objective_increment_local	

Table 4.2: Required Methods from Model API for Multiple Meta-Heuristics

Note: Only the essential methods according to our implementation are presented, although other implementations may utilize different methods. Additionally, for the context of ACO, our implementation adopts the *Max-Min Ant System* variant [10]. Nevertheless, the method prerequisites should generalize to various other variants of this MH.

Chapter 5

Optimize a Data Center Problem

> *"In theory, theory and practice are the same. In practice, they are not."*
>
> — Albert Einstein

This chapter offers an in-depth analysis of the problem presented in the Google Hash Code 2015 qualification round, entitled "Optimize a Data Center". Particularly, it delves into the problem details with focus on the model that was both conceptualized and implemented following the principles of the framework proposed in Chapter 4, and the results achieved through this approach.

5.1 Problem Description

Over the years, Google has been building its own distinctively designed data centers, deploying numerous machines around the world. Within these facilities, arrays of servers operate continuously, fueling essential services that a multitude of people rely on daily. Nevertheless, the process of designing data centers is a complex optimization challenge with multiple factors to consider. While the primary objective is to optimize the utilization of computing capacity offered to users, it is equally imperative to ensure the uninterrupted provision of computing power, even in scenarios where hardware failures are inevitable. Based on this motivation, Google invites participants to address these challenges in a (fictional) Google Data Center modelled after a real-world scenario, which we described next.

The data center, as outlined in the problem statement, is modelled as set of distinct *rows*, each comprising a certain number of available spaces known

as *slots*, where servers can be positioned. It is worth highlighting that rows share resources, including electric power, Consequently, in the event of a hardware failure, an entire row of servers may become inaccessible, impacting the entirety of the servers located within that row. Moreover, the maximum number of slots is consistent across all rows within the data center. However, not all slots may be accessible due to specific installations or requirements that impose restrictions on the use of certain spaces.

In Figure 5.1, an illustration depicts a layout featuring two rows, each consisting of a total of seven slots. Some of these slots are marked as unavailable as indicated by the cross symbol.
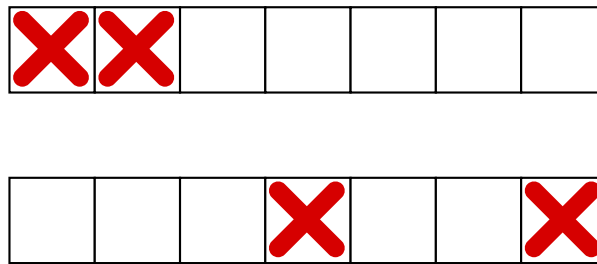
Figure 5.1: Example Data Center Layout

The servers are defined by a tuple that includes two attributes: the *size* of the server, which is measured in terms of the number of consecutive slots occupied by the machine, and the computing *capacity* of the server, represented as an integer value that indicates the machine's CPU resources. The Table 5.1 presents potential values for these parameters, showcasing examples of four distinct servers.

Server	Size	Capacity
1	3	2
2	2	5
3	3	10
4	2	3

Table 5.1: Server Properties

Moreover, when servers are positioned within the data center rows, they are logically associated with resource *pools*, to which they can contribute their individual computing capacities. The capacity of a pool is defined as the collective sum of the capacities of all the servers allocated to it.

For clarification, Figure 5.2 offers a potential assignment of servers based on the data center layout depicted in Figure 5.1, and the server attributes provided

in Table 5.1. In this particular case, servers are allocated to two distinct resource pools.
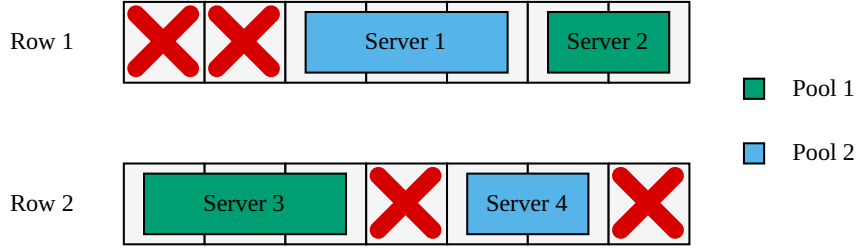

Figure 5.2: Example Server Assignment

Upon closer examination of the this example, we can deduce that ensuring the reliability of a specific resource pool implies the distribution of servers across various rows. This approach ensures that in the event of a row failure, the pool can still operate with diminished capacity, drawing upon the servers located in the unaffected rows. In the context of the problem, this represents the concept of a *guaranteed capacity* for a pool, which can be defined as follows:

Given a resource pool (p), the guaranteed capacity of that (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 unavailable. Formally, this can be described as shown in Equation (5.1), where $\mathcal{M}$ denotes the set of servers (machines) available and c_m the computing capacity of a given server $(m \in \mathcal{M})$.

$$gc_p = \min_{r \in \mathcal{R}} \left(\sum_{m \in \mathcal{M} \wedge m \in p} c_m - \sum_{m \in \mathcal{M} \wedge m \in p \wedge m \in r} c_m \right) \tag{5.1}$$

In essence, the objective of this problem can be succinctly defined as: Given a layout description of a data center, the objective is to determine the optimal arrangement of servers within the data center rows and assign these servers to resource pools, such that, the minimum guaranteed capacity across all pools is maximized. Mathematically, this can be expressed as shown in Equation 5.2, where s represents a candidate solution, $\mathcal{P}$ the set of pools, and gc_p denotes the guaranteed capacity for a specific pool $(p \in \mathcal{P})$.

$$\max f(s) = \min_{p \in \mathcal{P}} \left(gc_p \right) \tag{5.2}$$

Moreover, in a competition context, the value obtained from evaluating a solution represents the score a contestant would attain for solving a particular instance of this problem. For clarification of all the aforementioned concepts,

the Example 5.1.1 illustrates the scoring process of a candidate solution.

Example 5.1.1 (Scoring). *Regarding the data center layout shown in Figure 5.2 and the server properties provided in Table 5.1, we can summarize the capacities assigned to each resource pool, by row, as shown in Table 5.2.*

	Row 1	Row 2	Guaranteed Capacity	**Score**
Pool 1	5	10	5	**2**
Pool 2	2	3	2	

Table 5.2: Guaranteed Capacity & Score

The score obtained for this server placement, as determined by the evaluation of the objective 5.2, can thus be calculated as $\min(5, 2) = 2$.

Regarding the problem instances to be optimized, crucial details are provided concerning the layout of the data center, the available servers for allocation, and specifics about the unavailable slots. These aspects are represented using the following parameters and their corresponding constraints:

$\mathcal{R}$. $(1 \leq \mathcal{R} \leq 1000)$ The number of rows in the data center.

$\mathcal{S}$. $(1 \leq \mathcal{S} \leq 1000)$ The number of slots in each row of the data center.

$\mathcal{U}$. $(0 \leq \mathcal{U} \leq \mathcal{R} \times \mathcal{S})$ The number of unavailable slots.

$\mathcal{P}$. $(1 \leq \mathcal{P} \leq 1000)$ The number of resource pools.

$\mathcal{M}$. $(1 \leq \mathcal{M} \leq \mathcal{R} \times \mathcal{S})$ The number of servers to be allocated.

Notably, in a compettion setting, participants were required to read and output both the problem description details presented, and the solution obtained, in a specific format as outlined in the problem statement. For a description of the these and other considerations regarding mostly implementation and evaluation details, the reader can refer to the official problem statement which can be found on the Google Coding Competitions Archive [37].

5.2   Model

5.2.1   Representation

5.3   Results

5.4   Concluding Remarks

Chapter 6

Book Scanning Problem

"Success is walking from failure to failure with no loss of enthusiasm."

— Winston Churchill

In this chapter, we delve into an extensive analysis of the problem presented in the Google Hash Code 2020 qualification round, entitled "Book Scanning". Similarly to the previous chapter, our primary emphasis lies in elucidating the the modeling process and how it is strategically employed to address the unique challenges posed by this particular problem.

6.1   Problem Description

The primary objective of the Google Books project is to create a digital library with global accessibility. Through collaborations with libraries and publishers worldwide, the project has amassed a collection of over 40 million books in more than 400 languages. The process of adding a book to the digital library involves several key steps, such as library registration, on-site visits by logistics experts, and shipping the books for scanning. Therefore, the development of an efficient scanning pipeline for global libraries requires meticulous planning for optimal effectiveness. To address this challenge, Google invites participants to create an efficient scanning pipeline for books, using a scenario based on real-world circumstances as inspiration.

This problem revolves around two crucial components: libraries and books. The central focus is on the books themselves, which require scanning. The primary objective is to maximize the diversity of scanned books, each contributing a

specific score upon completion. On the other hand, libraries are characterized by the collection of books they house and the rate at which they can dispatch books for scanning. This implies that the number of books a library can scan per day varies among different libraries. However, a library becomes operational for book scanning only after completing a sign-up process that spans a certain number of days.
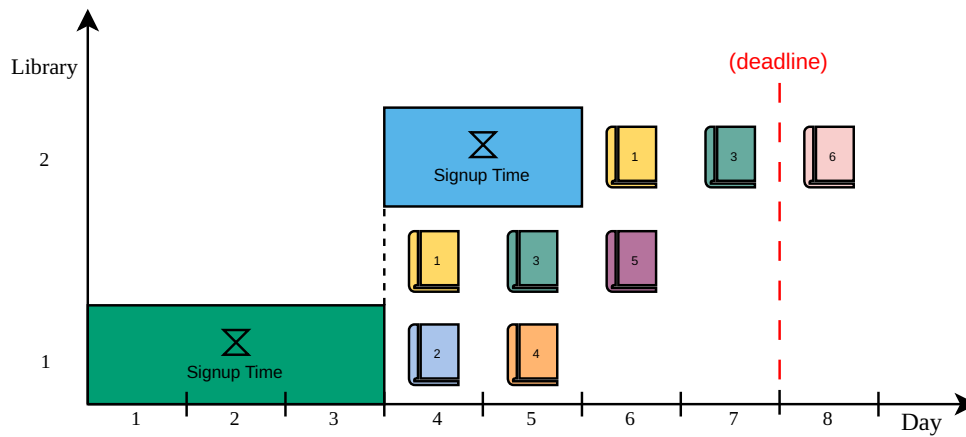

Figure 6.1: Book Scanning Process Example

6.2   Model

6.3   Results

6.4   Concluding Remarks

73

Chapter 7

Conclusion

> *"We can only see a short distance ahead, but we can see plenty there that needs to be done."*
>
> — Alan Turing

Acronyms

ACO Ant Colony Optimization. x, 19–21, 25, 48, 50, 66

API Application Programming Interface. 28–30, 46, 51, 56

BBO Black-Box Optimization. 12, 26, 45

BI Best Improvement. 22, 24, 66

BS Beam Search. x, 17, 25, 56–58, 60, 66

CO Combinatorial Optimization. 2, 3, 5–9, 13, 14, 44

CS Constructive Search. x, 14, 15, 17–19, 25, 27, 29, 30, 48, 49, 53, 55, 56, 61

FI First Improvement. 22, 60, 66

GBO Glass-Box Optimization. 12, 26

GO Global Optimization. 10–12

GRASP Greedy Randomized Adaptive Search Procedure. x, 18, 19, 25, 52, 66

HC Hill Climbing. x, 21, 22, 25

IG Iterated Greedy. x, 19, 20, 25, 51, 66

ILP Integer Linear Programing. 12, 26

ILS Iterated Local Search. x, 22, 23, 25, 56, 60, 66

KP 0/1 Knapsack Problem. 7, 8, 12, 26

LCG Linear Congruential Generator. 53, 54

LO Local Optimization. 10–12

LS Local Search. x, 15, 16, 18, 19, 21–25, 27, 29, 48, 52–56, 60, 61

MH Meta-Heuristic. 2, 3, 6, 8, 9, 14, 16–23, 26, 28, 47, 48, 56, 57, 60, 62, 66

nasf4nio Not Another Framework for nature-inspired optimization. 28–30, 46

nasf4nio-cs Not Another Framework for nature-inspired optimization — Constructive Search. 28–30, 46

POF Python Optimization Framework. 28, 29, 46

SA Simulated Annealing. x, 19, 22–25

TS Tabu Search. x, 23–25, 66

Bibliography

[1]   David S. Johnson. "Approximation Algorithms for Combinatorial Prob-
       lems". 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.

[2]   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.

[3]   Peng Si Ow and Thomas E. Morton. "Filtered Beam Search in Schedul-
       ing†". In: *International Journal of Production Research* 26.1 (Jan. 1, 1988),
       pp. 35–62. ISSN: 0020-7543. DOI: 10.1080/00207548808947840.

[4]   R. W. Eglese. "Simulated Annealing: A Tool for Operational Research". In:
       European Journal of Operational Research 46.3 (June 15, 1990), pp. 271–
       281. ISSN: 0377-2217. DOI: 10.1016/0377-2217(90)90001-R.

[5]   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.

[6]   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.

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

[8]   Jens Clausen. "Branch and Bound Algorithms - Principles and Examples."
       In: *Branch and Bound Algorithms - Principles and Examples.* (1999).

[9]   Fred Glover and Manuel Laguna. *Tabu Search I*. Vol. 1. Jan. 1, 1999. ISBN:
       978-0-7923-9965-0.

[10]   Thomas Stützle and Holger Hoos. "MAX-MIN Ant System". In: 16 (Nov. 30,
       1999).

[11]   Christian Blum. "Metaheuristics in Combinatorial Optimization". In: *ACM Computing Surveys* 35.3 (2003).

[12]   Luca Di Gaspero and Andrea Schaerf. "EasyLocal++: An Object-Oriented Framework for the Flexible Design of Local-Search Algorithms". In: *Software—Practice & Experience* 33.8 (July 10, 2003), pp. 733–765. ISSN: 0038-0644. DOI: 10.1002/spe.524.

[13]   S. Cahon, N. Melab, and E.-G. Talbi. "ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics". In: *Journal of Heuristics* 10.3 (May 2004), pp. 357–380. ISSN: 1381-1231. DOI: 10.1023/B: HEUR.0000026900.92269.ec.

[14]   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.

[15]   Ana Vieira. "Uma plataforma para a avaliação experimental de meta-heurísticas". Doctoral Thesis. University of Algarve, Portugal, 2009.

[16]   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.

[17]   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.

[18]   Walter J. Gutjahr. "Stochastic Search in Metaheuristics". 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. 573–597. ISBN: 978-1-4419-1665-5. DOI: 10.1007/978-1-4419-1665-5_19.

[19]   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.

[20]   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 Sci-

ence. Boston, MA: Springer US, 2010, pp. 1–39. ISBN: 978-1-4419-1665-5. DOI: 10.1007/978-1-4419-1665-5_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] Xinjie Yu and Mitsuo Gen. "Combinatorial Optimization". In: *Introduction to Evolutionary Algorithms*. Ed. by Rajkumar Roy. Decision Engineering. London: Springer, 2010, pp. 263–324. ISBN: 978-1-84996-129-5. DOI: 10.1007/978-1-84996-129-5_7.

[23] Juan J. Durillo and Antonio J. Nebro. "jMetal: A Java Framework for Multi-Objective Optimization". In: *Advances in Engineering Software* 42.10 (Oct. 1, 2011), pp. 760–771. ISSN: 0965-9978. DOI: 10.1016/j.advengsoft.2011.05.014.

[24] 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.

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

[26] 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.

[27] 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.

[28] Donald E. Knuth. "The Art of Computer Programming: Seminumerical Algorithms, Volume 2". In: Addison-Wesley Professional, May 6, 2014, pp. 17–19, 145. ISBN: 978-0-321-63576-1.

[29] 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.

[30] Thomas Stützle and Rubén Ruiz. "Iterated Greedy". In: *Handbook of Heuristics*. Ed. by Rafael Martí, Panos M. Pardalos, and Mauricio G. C.

Resende. Cham: Springer International Publishing, 2018, pp. 547–577. ISBN: 978-3-319-07124-4. DOI: 10.1007/978-3-319-07124-4_10.

[31] Thomas Bartz-Beielstein et al. *Benchmarking in Optimization: Best Practice and Open Issues*. July 7, 2020.

[32] 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.

[33] 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.

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

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

[36] Valentina Cacchiani et al. "Knapsack Problems — An Overview of Recent Advances. Part I: Single Knapsack Problems". In: *Computers & Operations Research* 143 (July 2022), p. 105692. ISSN: 03050548. DOI: 10.1016/j.cor.2021.105692.

[37] Google LLC. *Coding-Competitions-Archive*. Aug. 19, 2023. URL: https://github.com/google/coding-competitions-archive (visited on 08/20/2023).

[38] Zibada. *Google Coding Competitions Unofficial Archive*. 2023. URL: https://zibada.guru/gcj/#hc (visited on 08/20/2023).

[39] Pedro Rodrigues. *Hascode-Models*. Github. URL: https://github.com/pedromig/hashcode-models (visited on 08/31/2023).

[40] Pedro Rodrigues. *Nasf4nio-Py*. GitHub. URL: https://github.com/pedromig/nasf4nio-py (visited on 08/31/2023).