FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

Pedro Miguel Duque Rodrigues

Principled Modeling of the Google Hash Code Problems for Meta-Heuristics

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

September 2023

The work presented in this thesis was carried out in the Algorithms and Optimization Laboratory of the Adaptive Computation group of the Centre for Informatics and Systems of the University of Coimbra and financially supported by the Foundation for Science and Technology under a scholarship with reference UIDP/00326/2020.

Abstract

Acknowledgments

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 usually to find "good" solutions in a time-efficient manner.

Tackling an optimization problem can usually 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 mixed-integer linear optimization software packages which are widely used to solve real-world problems, *e.g.*, *Gurobi* [1], *CPLEX* [2], or *GLPK* [3]. In particular, such packages expect a mathematical formulation (model) describing the problem as a linear objective function and a set of linear constraints involving continuous or discrete variables, and then use one or more algorithms designed to solve

[1] https://www.gurobi.com/

[2] https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer

[3] https://www.gnu.org/software/glpk/

such linear optimization problems. This clear separation of concerns has 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 any currently known exact approach grows faster than polynomially with the problem size. 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 on which high-level strategies depend from the MH problem-independent solving process. In fact, the abundance of MH optimization software that provides specific frameworks for 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 [36], among others, who 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 interested in constructing a collection of benchmark optimization problems that hold both theoretical and practical significance [32]. 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. Still, exact algorithms to solve these problems efficiently are not known, and to the best of 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 should serve as interesting benchmarks for the evaluation of meta-heuristics offering ample research potential. Furthermore, they should allow the feasibility of the aforementioned unifying framework bo be assessed 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 two 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 modeling approach for meta-heuristics that has been partially explored in previous research [15, 35, 36]. The objective is to solidify existing concepts while introducing additional functionality, both in conceptual understanding and practical application.

Furthermore, we aim to construct models of 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 existing ideas explored by previous work on modelling frameworks [15, 35, 36] be formalized and a practical implementation be developed, potentially contributing with new features?

R2. Can 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 are related to the aforementioned research questions, as follows:

C1. Building upon existing research on well-structured modeling for meta-heuristics [15, 35, 36], 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 tools 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

were developed that explore the different properties of each of the problems in an attempt to obtain the best possible solutions. Furthermore, these models provide a practical example on how to model relatively complex problems and also allow us to think critically about the framework capabilities.

1.4 Software

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

S1. Pedro Rodrigues. *Nasf4nio-Py*. GitHub. URL: https://github.com/pedromig/nasf4nio-py [41]

S2. Pedro Rodrigues. *Hashcode-Models*. Github. URL: https://github.com/pedromig/hashcode-models [40]

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, metaheuristics 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 an optimal 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 [38] 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 Knapsack Problem (KP) [37, 22, 28] 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 [36, 28, 27]). *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 to solve these problems. However, finding an optimal solution can be difficult, and exhaustive search strategies may still not be able to efficiently solve CO problems, which are often NP-Hard [22, 28]. 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 estimate the best objective value that can be obtained by adding zero or more components to a partial solution. In particular, the upper bound of a given solution is any value that is greater than or equal to the objective function values of feasible solutions that include the components of that specific solution, and similarly for the lower bound. The same logic applies to the concept of tight upper and lower bounds. However, in the context of CO, the term *tight* is commonly used to describe a bound that is closer to the optimal value but may not necessarily be strictly optimal.

Formally speaking, the upper bound and lower bound [7, 36] 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 any 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 any 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 common in CO algorithms, especially in exact approaches, as will be detailed in Section 2.2.1. Nonetheless, concerning MH methods, bounds can be helpful tools to guide the solution construction process. For example, consider the choice between adding one of two components to a partial solution. We can consider the two partial solutions that resulting from adding either component, compute their upper bound and use it as a way to determine which choice may be more promising. Note that a tight bound is often an important 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 due to the solution being infeasible.

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.

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 a global optimum (Definition 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 those that are in its proximity, 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, for every feasible solution $s \in \mathcal{S}$ there is a set of neighboring solutions $\mathcal{N}(s) \subseteq \mathcal{S}$, namely its neighborhood.

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 s is a local optimum with respect to a given neighborhood structure $\mathcal{N}(s)$ iff:*

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

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

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

As an illustrative example, consider the objective function $f(s)$ shown in Figure 2.1. With respect to the Definitions 2.1.2 and 2.1.11, the solution s^1 is a global optimum and s^2, s^3 are (strict) local optima. It is worth noting that, in this example, the neighborhood is defined based on the adjacency to the S axis.

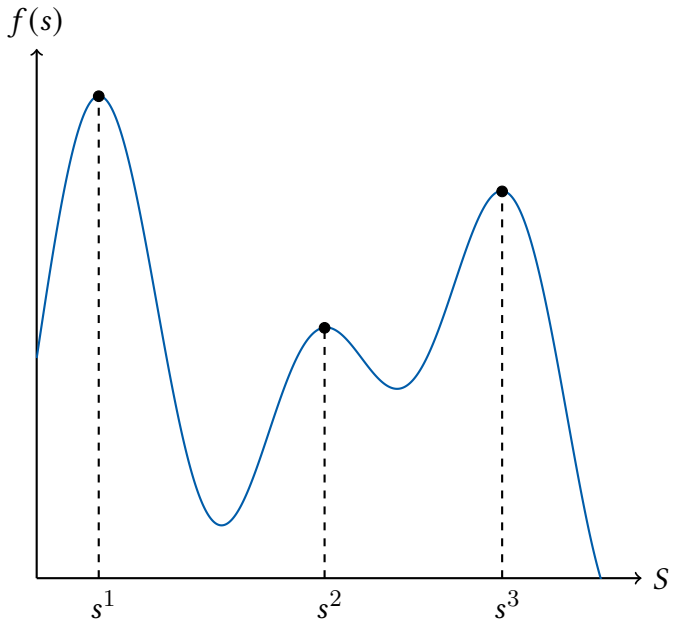

Figure 2.1: Global and Local Optima

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, which 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 [34], or the objective function is too complex to be approached from an analytical perspective. As such, algorithms to solve these problems do so only by interacting with the problem through the evaluation of potential candidate solutions [33, 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 [33]. Hence, the algorithms used may take advantage of more analytical properties of the problem since they are transparent to the solver.

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.*, Integer Linear Programing (ILP) to tackle sub-problems that are simpler to formalize.

2.2 Optimization Strategies

Combinatorial optimization literature extensively documents a series of methods for solving multiple problems [7, 22, 26]. 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, (meta-)heuristic approaches are often described in terms of constructive and local search procedures.

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, 28].

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 the 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 [28], thereby avoiding recomputing repeated 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 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, 28, 26]. 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 selected 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 [27] is shown in Algorithm 2.1.

Algorithm 2.1: Constructive Search Procedure

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

$s \leftarrow \emptyset$
$C \leftarrow \{\, c \in \mathcal{G} \setminus s \mid s \cup \{c\}$ `is feasible` $\}$
while $C \neq \emptyset$ **do**
    $c \leftarrow$ `SelectComponent`(C)
    $s \leftarrow s \cup \{c\}$
    $C \leftarrow \{\, c \in \mathcal{G} \setminus s \mid s \cup \{c\}$ `is feasible` $\}$
end
return s

It is 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 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 a feasible solution to a given problem, and then make modifications by adding, removing, and swapping components in order to improve it. The range of possible modifications defined by the user

define the neighborhood structure for a problem, which LS approaches attempt to exploit. The LS approach terminates when no neighboring solution is better (or equal), *i.e.*, when the current solution is 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 allow worsening a solution in order to explore previously unseen regions of the search space. This action is commonly referred to as a *perturbation* [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  M ← LocalMoves(s)
2  while M ≠ ∅ do
3  |  m ← SelectLocalMove(M)
4  |  s ← Step(s, m)
5  |  s ← Perturb(s)                    ▷ Optional
6  |  M ← LocalMoves(s)
7  end
8  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 few distinctive traits [11] such as:

Search Strategy. This refers to the method used to find 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 is 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, 36] 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 terminates 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. In this context, the notation argmax_w is employed to introduce the concept of selecting the top w elements from a given set.

Algorithm 2.3: Beam Search

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

```
1  s ← ∅
2  bobj ← −∞
3  B ← {∅}
4  if s is feasible then
5  |  bobj ← f(s)
6  end
7  while B ≠ ∅ ∧ stopping criteria not met do
8  |  B' ← ∅
9  |  foreach s' ∈ B do
10 |  |  B' ← B' ∪ Branch(s')
11 |  end
12 |  if B' ≠ ∅ then
13 |  |  B ← argmax_{s' ∈ B', w} Φ_ub(s') ▷ Select the "w" best solutions
14 |  |  foreach s' ∈ B do
15 |  |  |  if s' is feasible ∧ f(s') > bobj then
16 |  |  |  |  s ← s'
17 |  |  |  |  bobj ← f(s')
18 |  |  |  end
19 |  |  end
20 |  end
21 end
22 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 is 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. Likewise, a similar check is carried out for the empty solution, as its feasibility can vary depending on the specific problem being addressed.

It is 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. Futhermore, the effectiveness of this MH is also closely linked to the quality of the upper bound function.

2.3.2 Greedy Randomized Adaptive Search Procedure

Greedy Randomized Adaptive Search Procedure (GRASP) [21, 36, 11] is a stochastic, CS and LS, MH that iteratively builds solutions by sequencing a construction phase with an optional 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 works.

It's important to highlight that this meta-heuristic, provides a means of controlling the balance between randomization and greediness in solution construction (`GreedyRandomizedConstruction`). 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 components with the optimal heuristic or bound value, will be selected at random, rendering this MH more greedy. It's worth noting that the inclusion of the optional local search step (`LocalSearch`) can significantly enhance the quality of the randomized greedy process. However, the crux of fine-tuning lies in determining an appropriate value for α.

2.3.3 Iterated Greedy

Iterated Greedy (IG) [31, 36] 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 [31], implement an acceptance criterion that probabilistically permits the acceptance of solu-

Algorithm 2.4: Greedy Randomized Adaptive Search Procedure

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

```
1  s ← ∅
2  bobj ← −∞
3  if s is feasible then
4  |  bobj ← f(s)
5  end
6  while stopping criteria not met do
7  |  s′ ← GreedyRandomizedConstruction(Φ_ub, α)
8  |  s′ ← LocalSearch(s′)                    ▷ Optional
9  |  if s′ is feasible ∧ f(s′) > bobj then
10 |  |  s ← s′
11 |  |  bobj ← f(s′)
12 |  end
13 end
14 return s
```

tions 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.

Algorithm 2.5: Iterated Greedy

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

```
1  s ← Construct (∅, Φ_ub)
2  while stopping criteria not met do
3  |  s′ ← Destruct(s)
4  |  s′ ← Construct(s′, Φ_ub)
5  |  if s′ is feasible ∧ f(s′) > f(s) then
6  |  |  s ← s′
7  |  end
8  end
```

2.3.4 Ant Colony Optimization

Ant Colony Optimization (ACO) [16, 10, 26, 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 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 Algorithm 2.6 illustrates this MH.

Algorithm 2.6: Ant Colony Optimization

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

$s \leftarrow s'$
$bobj \leftarrow -\infty$
$\mathcal{P} \leftarrow \{\emptyset\}$
if s `is feasible` **then**
  $bobj \leftarrow f(s)$
end
while `not stopping criteria met` **do**
  $\mathcal{P} \leftarrow$ `AntBasedSolutionConstruction`$(\mathcal{P}, \vec{\tau})$
  $\mathcal{P} \leftarrow$ `LocalSearch`$(\mathcal{P})$ ▷ `Optional`
  $s' \leftarrow \underset{s' \in \mathcal{P}}{\operatorname{argmax}} f(s')$ ▷ `Select best solution`
  if $f(s') > bobj$ **then**
    $s \leftarrow s'$
    $bobj \leftarrow f(s')$
  end
  $\mathcal{P} \leftarrow$ `PheromoneUpdate`$(\mathcal{P}, \mathcal{R}, \vec{\tau}, \alpha)$
end
return s^*

In summary, the ACO meta-heuristic can be described as a process that comprises of a solution construction phase (`AntBasedSolutionConstruction`), 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`), and culminating in a pheromone update phase (`PheromoneUpdate`). This process is then repeated for multiple iterations, until certain stopping criteria is met.

2.3.5 Hill-Climbing

Hill Climbing (HC) [26, 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)

```
while stopping criteria not met do
    s' ← Perturb(s)
    if f(s') > f(s) then
        s ← s'
    end
end
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 [26].

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, 26, 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.

Algorithm 2.8: Iterated Local Search

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

```
while stopping criteria not met do
    s' ← LocalSearch(s)
    s' ← Perturb(s', k)
    if f(s') > f(s) then
        s ← s'
    end
end
return s
```

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 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.

Algorithm 2.9: Simulated Annealing

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

$t \leftarrow t_0$
while `stopping criteria not met` **do**
  $s' \leftarrow$ `Perturb`(s)
  $\delta \leftarrow f(s)$ - $f(s')$
  if $\delta < 0 \vee$ `Random`$(0, 1) < e^{-\frac{\delta}{t}}$ **then**
    $s \leftarrow s'$
  end
  $t \leftarrow t \cdot \alpha$
end
return s

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, 26] 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

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.

Algorithm 2.10: Tabu Search

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

$\mathcal{T} \leftarrow \{\emptyset\}$
while `stopping criteria not met` **do**
  $s' \leftarrow \underset{s' \in \mathcal{N}(s) \setminus \mathcal{T}}{\operatorname{argmax}} f(s')$ ▷ `Select best neighbor` $s' \notin \mathcal{T}$
  if $f(s') > f(s)$ **then**
    $s \leftarrow s'$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{s'\}$
  end
  if $|\mathcal{T}| > l_{\max}$ **then**
    $\mathcal{T} \leftarrow \mathcal{T} \setminus$ `Oldest`$(\mathcal{T})$
  end
end
return s

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

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	Trajectory	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 [30].

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.

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

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, 36, 35].

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 [36], 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 [35]. In general, this API follows the same concepts as the nasf4nio-cs, which we will detail next. Regarding CS, the work by Outeiro [36] 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.

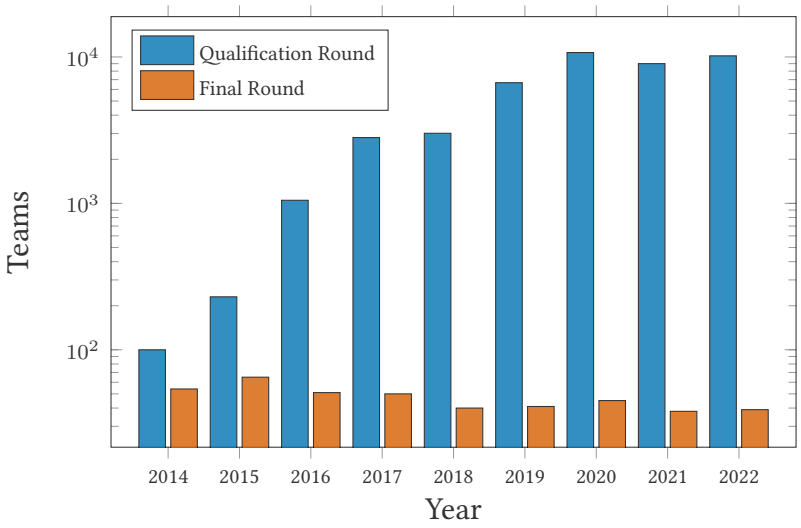

Figure 3.1: Google Hash Code Competition Attendance 2014-2022

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 [38]. 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 [39].

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 [38]

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 BBO [32].

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 [32]. 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 usage of this framework 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 have 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 [35], and Outeiro [36]. 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 using a set of pre-defined functions 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 details of the problem.

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 of a solution. In terms of implementation, these act as *getters* for the properties of the model and should avoid executing resource-intensive computations when possible, 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 ground set of the problem – 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 sequence 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 is 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 is 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.

```
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 a sequence of 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.

```
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.

```
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.

```
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.

```
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.

```
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).

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

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

When it comes to random enumeration of the neighboring moves of a candidate solution 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 [29]). *A linear congruential generator is a pseudo-random number generator described by the following recurrence relation:*

$$\mathcal{X}_{n+1} = (a\mathcal{X}_n + c) \bmod m \tag{4.1}$$

Where,

$$\begin{aligned} m,&\quad 0 < m \text{ — is the modulus} \\ a,&\quad 0 < a < m \text{ — is the multiplier} \\ c,&\quad 0 \leq c < m \text{ — is the increment} \\ \mathcal{X}_0,&\quad 0 \leq \mathcal{X}_0 < m \text{ — is the "seed" or "starting value"} \end{aligned}$$

For a LCG it is known [29] 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 [29]:

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

As it is crucial for CS methods to be able to apply components, the same holds true for LS methods, albeit in the context of local moves. Therefore, this method

is specifically designed to facilitate the application of a `LocalMove` to a given candidate solution.

```
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.

```
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 [41].

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

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

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.

```
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 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 goto **Termination**. (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 or a different solution of the same quality was achieved during the first improvement stage. (lines 15-20)

(c) **Perturb**: Optionally, perturb the current 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 [41]. 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 implementations. From the vantage point of a MH developer, the primary objective revolves around constructing tests that assess the meta-heuristic's correct implementation, *e.g.*, *unit-tests*. From the perspective of a model developer, these tests, while valuable for assessing the correctness of the model's implementation, 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 component 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 incremental 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*[1]. This just-in-time (JIT) compiled version of Python has a substantial speed boost, claiming to be approximately 4.8 times faster than the implementation found and average interpreter (*CPython*).

In fact, this became particularly evident when dealing with complex problem instances, where the performance difference was remarkable. 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, rapidly testing new ideas from a modeling standpoint this framework remains a valuable choice.

Lastly, it's important to acknowledge that the specification of the framework might not be exhaustive. The range of MH methods, as documented in the literature [6], is extensive, and new methods could potentially emerge in the future. While the range of meta-heuristics covered in the framework is substantial, it is possible that certain use-cases could necessitate the addition of new functionalities. Therefore, the framework's completeness and stability will certainly depend on insights from the community.

[1] https://www.pypy.org/

Class	Methods	Input	Output
Problem	empty_solution	—	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]
	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, and was only added for completeness.

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
SA	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 the Implemented 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 "Optimize a Data Center" problem of the Google Hash Code 2015 qualification round. Particularly, in Section 5.1 we provide a formal description of the problem, highlighting details that we considered essential for modeling. In Section 5.2 we delve into the details of the model that was developed for this problem. In Section 5.3 we present an overview of the results obtained, comparing them to those achieved by other participants during the competition. Finally, Section 5.4 offers some concluding remarks on the entire process, regarding both the implementation and problem modeling aspects.

5.1 Problem Description

As outlined in the problem statement, 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. Drawing inspiration from these engineering

challenges, this problem presents a Google Data Center design scenario that mirrors real-world complexities, as we describe next.

The data center, as outlined in the problem statement, is modelled as collection of distinct *rows*. Each row consists of a specific number of "slots", designated for the placement of servers. Notably, this number of slots remains consistent across all rows within the data center. Additionally, certain slots within the data center may be unusable due to other installations that impose restrictions on the utilization of those specific spaces. However, because rows share resources, such as electrical power, a hardware failure in one row can render the entire row of servers inoperable.

In Figure 5.1, an illustration depicts a layout featuring two rows, each consisting of a total of seven slots. Note that some of these slots are marked as unavailable, as indicated by the cross symbol.

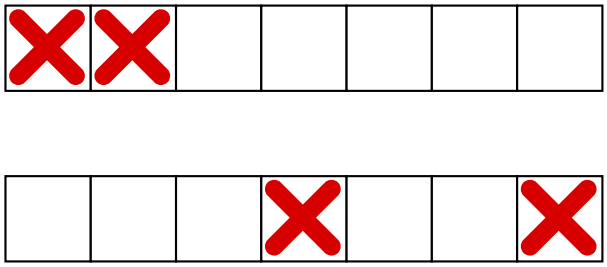

Figure 5.1: Example Data Center Layout

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. Henceforth, we shall denote as $\mathcal{M}$ the total number of servers, and use ℓ_m and c_m to represent the *size* and computing *capacity* of each server ($m = 1, \ldots, \mathcal{M}$), respectively.

For illustration purposes, Table 5.1 presents potential values for the aforementioned 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

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 instance, servers are distributed across two separate resource pools, resulting in a total capacity of 15 and 5 for pools numbered 1 and 2, respectively.

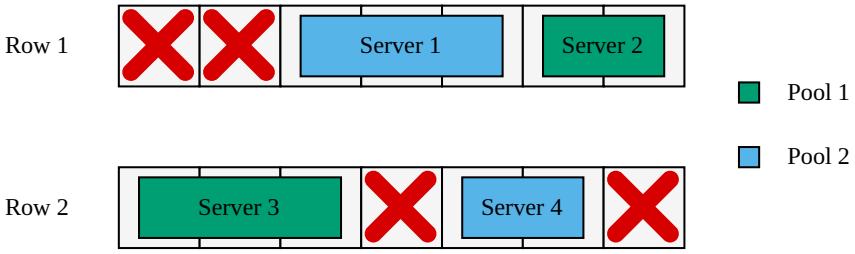

Figure 5.2: Example Server Assignment

By examining Figure 5.2, it becomes evident that the number of available slots for a row does not solely determine the constraint for placing servers. In reality, the presence of unavailable slots can further limit this placement, even if the total number of slots in that row would otherwise allow it. Consequently only the set of contiguous slots within each row, which we refer to as *segments*, can be considered.

Another important aspect is that ensuring the reliability of a specific resource pool implies the distribution of servers across various rows. This strategy 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.

Let, $\mathcal{P}$ be the number of resource pools, $\mathcal{R}$ the number of rows, $\mathcal{I}$ the number of segments, $\mathcal{L}$ the set containing the size (in slots) of each segment ($i = 1, \ldots, \mathcal{I}$), and $\mathcal{I}^r$ the set containing the indices of all segments for a given row ($r = 1, \ldots, \mathcal{R}$). The guaranteed capacity, gc_p, of a pool ($p = 1, \ldots, \mathcal{P}$) is a measure of the remaining computing capacity available in the event that at most one arbitrary row of the data center becomes inoperable. Formally, this can be described as shown in Equation (5.1) where, $x_{m,p,i}$, is a binary variable indicating whether the server, m, is assigned (1) or not (0) to pool, p, and segment i.

$$gc_p(x) = \sum_{m=1}^{\mathcal{M}} \sum_{r=1}^{\mathcal{R}} \sum_{i \in \mathcal{I}^r} c_m \cdot x_{m,p,i} - \max_{r=1}^{\mathcal{R}} \sum_{m=1}^{\mathcal{M}} \sum_{i \in \mathcal{I}^r} c_m \cdot x_{m,p,i} \tag{5.1}$$

In simple terms, the leftmost part of Equation (5.1) represents the capacity assigned to pool p, and the rightmost part indicates the capacity that pool p loses if the row with the highest capacity contribution to that pool becomes unavailable.

The objective of this problem can thus be succinctly described as: given a layout description of a data center, determine the optimal arrangement of servers to data center rows and resource pools, such that, the minimum guaranteed capacity across all pools is maximized. Mathematically, this can be expressed as shown in Equation (5.2).

$$\begin{aligned} \max \; f(x) &= \min_{p=1}^{\mathcal{P}} gc_p(x) \\ \text{s.t.} \; & \sum_{p=1}^{\mathcal{P}} \sum_{r=1}^{\mathcal{R}} \sum_{i \in I^r} x_{m,p,i} \leq 1 \qquad \forall\, m = 1, \ldots, \mathcal{M} \\ & \sum_{m=1}^{\mathcal{M}} \sum_{p=1}^{\mathcal{P}} \ell_m \cdot x_{m,p,i} \leq \mathcal{L}_i \qquad \forall\, i = 1, \ldots, \mathcal{I} \\ & x \in \{0,1\}^{\mathcal{M} \times \mathcal{P} \times \mathcal{R}} \end{aligned} \tag{5.2}$$

Note that, the constraints encapsulate the fundamental conditions that a server, once assigned to a specific row and segment, cannot be reassigned elsewhere, and that the sum of the sizes of of servers allocated to a particular segment must not exceed its predefined size.

To demonstrate the evaluation of the objective, consider the capacities assigned to each resource pool per row as showcased in Table 5.1. The resulting objective value for this server placement is determined as $\min(5, 2) = 2$.

Pool	Row 1	Row 2	Guaranteed Capacity	$f(x)$
1	5	10	5	**2**
2	2	3	2	

Table 5.2: Guaranteed Capacity & Score

Lastly, it is important to note that several instance-specific constraints are applied to various parameters that define the data center design within the context of this problem. These constraints are outlined as follows:

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

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

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

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

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

5.2 Problem Modeling

In the following, we delve into the modeling developed for this problem, and highlight important aspects related to its implementation within the framework proposed in Chapter 4.

Problem

A problem instance is uniquely defined by the set of available servers eligible for allocation, along with their respective sizes and capacities, the number of resource pools to be created, and the set of segments, each of which is characterized by its size.

This corresponds to the information stored within the `Problem` class. From an implementation standpoint, certain pre-processing operations were necessary to obtain this information. In particular, the construction of set of available segments using information related to available rows and the list of unavailable slots. Furthermore, given that this class serves as a repository for immutable problem-related information and is instantiated only once, a server ordering relevant for the construction rules was also included. This enabled us to iterate through and access the server order without the need to recompute it.

Solution

A solution is characterized by a collection of server assignments to pools and segments. Notably, in the context of this problem, any partial solution is feasible.

This data is contained within the `Solution` class, along with other pertinent information that is helpful in describing the state of the solution. This includes details such as the set of unassigned servers, the guaranteed capacities for each of the pools, the total capacities for each pool, and the same capacities data broken down by row. Furthermore, it also encompasses the details of evaluation metrics like the current objective function and upper bound values, in addition to other relevant data that facilitates their incremental calculations.

Component

A component can be represented as a tuple that contains a server, a pool, a segment, effectively representing an assignment. However, it is important to note that a valid decision is to refrain from placing a server in any of the available segments. As such, another valid component can be characterized solely by the server, indicating the action of *forbidding* it from being placed. Notably, this information is encoded in the `Component` class.

$$\mathcal{C} = \{(m, p, i) \mid \forall m = 1, \ldots, \mathcal{M},\ \forall p = 1, \ldots, \mathcal{P},\ \forall i = 1, \ldots, \mathcal{I}\} \tag{5.3}$$

$$\mathcal{F} = \{m \mid \forall m = 1, \ldots, \mathcal{M}\} \tag{5.4}$$

Given the component definitions in Equations (5.3) and (5.4), the ground set can be defined as $\mathcal{G} = \mathcal{C} \cup \mathcal{F}$.

Construction Rules

Since not all subsets of components represent feasible solutions, the construction rules specify which components can be enumerated and the order by which the enumeration is done. As such, the following sections describe the approaches that were considered.

Standard This approach involves enumerating all possible combinations of segments and pools where any of the unassigned servers can be assigned. It also considers the option of *forbidding* a server from being used. Algorithm 5.1 illustrates the pseudocode for this enumeration. Note that, the function `IsUsed` denotes whether a given server is available for assignment. Moreover, the function `Fit` indicates if a server m can be assigned to segment i.

Sequential In this approach, a predetermined order for the servers is followed. The process involves selecting the next server in a sequence and, similar to the standard approach, enumerate all possible combinations of segments and pools where a given server can be assigned. Again, the option for forbidding a server from being used is considered. The pseudocode for this enumeration is demonstrated in Algorithm 5.2.

Notably, the order considered for server enumeration $\phi^{\mathcal{M}}$, where $\phi^{\mathcal{M}}_m$ denotes the m-th server within this order, is inspired by heuristics for the KP, and prioritizes servers based on a non-decreasing order of the ratio between size and capacity, ℓ_m / c_m.

Algorithm 5.1: Standard Component Enumeration

Input : Number of Servers ($\mathcal{M}$), Number of Pools ($\mathcal{P}$), Number of Segments ($\mathcal{I}$)
Output : Enumerated Components

```
for m = 1 to M do
    if IsAssigned(m) then
        continue
    end
    for p = 1 to P do
        for i = 1 to I do
            if Fit(m, i) then
                yield (m, p, i)  ▷ Assign server m to pool p and
                 segment i
            end
        end
    end
    yield m                  ▷ Forbid server m
end
```

Heuristic This approach entails establishing a specific order for servers, pools, and rows, and then enumerating the potential combinations following that predefined order. In terms of pools, the order prioritizes pools with the lowest capacity. Concerning rows, the order gives preference to rows that have the least capacity with respect to the pool undergoing enumeration. Finally, segments are enumerated based on whether the chosen server can fit into the available space. It is worth noting that the option of forbidding server placement is also taken into account in this approach.

In Algorithm 5.3, the symbol $\phi^{\mathcal{M}}$ represents the previously mentioned order for server enumeration. The symbol $\phi^{\mathcal{P}}$ signifies the order for pool enumeration, with $\phi^{\mathcal{P}}_p$ indicating the p-th pool in this order. Finally, the symbol $\phi^{\mathcal{R}}$ signifies the order for row enumeration, where $\phi^{\mathcal{R}}_{p,r}$ specifies the r-th row affected by the choice of the p-th pool in this order.

Objective Function

The objective function for this problem remains the same as defined in the problem description, which involves maximizing the minimum guaranteed capacity across all pools. However, this function is a bottleneck. Therefore, for the purpose of optimization, we used an alternative objective function, $g(x)$, that, for a given assignment, x, uses as the objective value the tuple containing all guaranteed capacities for all the pools sorted in non-decreasing order, as illustrated in Equation (5.5).

Algorithm 5.2: Sequential Component Enumeration

Input : Server Index (m), Server Order ($\phi^{\mathcal{M}}$), Number of Pools ($\mathcal{P}$), Number of Segments ($\mathcal{I}$)
Output: Enumerated Components

```
if ¬ IsAssigned(φ_m^M) then
    for p = 1 to P do
        for i = 1 to I do
            if Fit(φ_m^M, i) then
                yield (φ_m^M, p, i)
            end
        end
    end
    yield φ_m^M
end
```

$$
\begin{aligned}
g(x) &= (gc_{\pi_1},\ gc_{\pi_2},\ \ldots,\ gc_{\pi_{\mathcal{P}}}) \\
&\text{s.t. } gc_{\pi_1} \leq gc_{\pi_2} \leq \ldots \leq gc_{\pi_{\mathcal{P}}}
\end{aligned}
\tag{5.5}
$$

In this equation, π is a permutation of the set $\{1, 2, \ldots, \mathcal{P}\}$.

Upper Bound

The upper bound developed for this problem is grounded in a relaxation of the second constraint presented in Equation (5.2). Rather than directly assigning servers to individual segments, we opt to consider the number of slots available for server placement in each row. This number is determined by summing the count of available slots across all segments within that row. Following this relaxation and for the purpose of computing the upper bound, we refer to the set containing the number of available slots for placement in all rows except row r as $\mathcal{W}_r$. The calculation of the initial values for all elements of this set is displayed in Equation (5.6):

$$
\mathcal{W}_r = \sum_{i \in \mathcal{I}^r} \mathcal{L}_i \quad r \in \{1, \ldots, \mathcal{R}\} \setminus \{r\} \tag{5.6}
$$

The upper bound calculation process is then divided into two steps, with the first step involving the estimation of an optimistic guaranteed capacity, followed by a second step for correction to ensure a tight upper bound.

In the first step, we consider the total size resulting from every combination of rows, excluding one row denoted as $\mathcal{W}_r$. We systematically assign available

Algorithm 5.3: Heuristic Component Enumeration

Input : Number of Servers ($\mathcal{M}$), Number of Pools ($\mathcal{P}$), Number of Segments ($\mathcal{I}$), Server Order ($\phi^{\mathcal{M}}$), Pool Order ($\phi^{\mathcal{P}}$), Row Order ($\phi^{\mathcal{R}}$)

Output : Enumerated Components

```
for m = 1 to M do
    if IsAssigned(φ_m^M) then
        continue
    end
    for p = 1 to P do
        z ← φ_p^P
        for r = 1 to R do
            k ← φ_{z,r}^R
            for i = 1 to I^k do
                if Fit(φ_m^M, i) then
                    yield (φ_m^M, φ_p^P, i)
                end
            end
        end
    end
    yield φ_m^M
end
```

servers to these rows until the cumulative size is fully utilized. The sum of server capacities for each row, except one, is then divided by the total number of pools to derive a quantity referred to as the "row-wise upper bound".

Additionally, the assignment of servers adheres to the previously mentioned heuristic order ($\phi^{\mathcal{M}}$), inspired by the knapsack problem. However, in the context of the upper bound calculation, this order prioritizes servers that are already assigned, followed by the unassigned servers, and excludes those marked as forbidden.

Formally, the row-wise upper bound calculation process is outlined in Equation (5.7):

$$\begin{aligned} \Phi^{r}_{ub} &= \frac{\sum_{m=1}^{\mathcal{M}} c_{\phi^{\mathcal{M}}_m}}{\mathcal{P}} \\ \text{s.t.} \sum_{m=1}^{\mathcal{M}} \ell_{\phi^{\mathcal{M}}_m} &\leq \mathcal{W}_r \end{aligned} \tag{5.7}$$

In the second step of this calculation, we can further refine each row-wise upper bound by removing pools and servers that already possess a guaranteed

capacity exceeding the value of the row-wise upper bound when dropping that row. After removing these elements, we recalculate the bound for the reduced server set and the number of remaining pools. This process is repeated until no further corrections can be applied, meaning that no considered pool has a guaranteed capacity greater than the bound.

The upper bound is determined by selecting the minimum value among all the row-wise upper bounds after applying the correction, which is calculated, as illustrated in Equation (5.8):

$$\Phi_{ub} = \min_{r=1}^{\mathcal{R}} \Phi_{ub}^{r} \tag{5.8}$$

While this bound is valuable, it can be enhanced for greater informativeness. During the computation of each row-wise upper bound, we maintain the smallest value for each pool in a vector denoted as Λ. It is important to note that at the end of this process, the values within Λ correspond to an upper bound on the guaranteed capacity of each pool. Subsequently, this vector is utilized to define an upper bound for the problem, as depicted in Equation (5.9), where π represents a permutation of the set $1, 2, \ldots, \mathcal{P}$. Notably, this upper bound aligns with the choice for the alternative objective function, $g(x)$, mentioned earlier.

$$\begin{aligned} \Phi_{ub} &= (\Lambda^{\pi_1},\ \Lambda^{\pi_2},\ \ldots,\ \Lambda^{\pi_{\mathcal{P}}}) \\ &\text{s.t. } \Lambda^{\pi_1} \leq \Lambda^{\pi_2} \leq \ldots \leq \Lambda^{\pi_{\mathcal{P}}} \end{aligned} \tag{5.9}$$

Local Moves

The local moves considered for this problem are as follows:

1. Assigning a server to a pool and segment if there is an available one.
2. Removing a server from a segment.
3. Changing the segment to which a particular server is allocated, if other segments that can accommodate its size are available.
4. Changing the pool to which a server is allocated, moving it to a different pool.
5. Swapping the pools of two assigned servers, thereby transferring the capacity of the servers between pools.
6. Swapping the segment in which servers are assigned. This is applicable

when there is available space on both sides of the transfer to accommodate the change.

Notably, these operations in the context of the modelling framework implementation are encoded in the `LocalMove` class.

Perturbation

Regarding the perturbation of the solution, for this problem, we did not employ any specific operation apart from randomly applying the described local moves.

5.3 Results

After formally describing the problem model, the information was translated into a practical implementation developed within the modeling framework mentioned in Chapter 4. This implementation enabled the use of various metaheuristic methods for experimentation. Multiple iterations of the model were tested, each involving different types of component enumerations, bounds, heuristics, and various solvers.

The machine used to run the solvers on the single available instance for this problem had the following s specifications:

OS	CPU	RAM
Ubuntu 22.04.2 LTS	Intel i7-12700H (20 cores) — 4.600GHz	64 GB

Table 5.3: Benchmark Machine Specifications

The best results were achieved through a simple heuristic strategy in a constructive search phase. This strategy involved enumerating ten best components at each iteration using the heuristic strategy described in Algorithm 5.3. The selected component to add to the solution was the one that contributed the most increment to the upper bound value. Notably, the upper bound function used was the one described in Equation 5.9, which effectively discriminated the results, allowing for a construction that yielded a final score of 386 points in only 1 second.

Other CS algorithms, such as GRASP, IG and BS proved to be slower in constructing solutions of similar quality compared to the simple heuristic strategy. Multiple *add-hoc* parametrizations were tested for each of these algorithms, but they consistently produced initial results ranging from 280 to 320 points, taking additional time to further improve them as to match the described heuristic.

For LS strategy, the best meta-heuristic was ILS. When run with a total timeout time of 15 seconds and using a perturbation with a kick strength of 2, ILS consistently improved the solution returned by the heuristic construction strategy (regardless of the seed), yielding a score of 410 points. This result was satisfying and led to us move on to the next problem.

Notably, the score of **410** is better than the best score in the competition leaderboard (**407**).

5.4 Concluding Remarks

In this chapter, we provided our analysis and described model for the Google Hash Code problem entitled "Optimize a Data Center". Despite having only a single instance, which seemed simple, this problem posed a several optimization challenges, mainly due to its bottleneck objective function.

The task of optimizing the implementation of the model within the framework was a meticulous process that required extensive thought and many hours of development. It involved the incremental calculation of various values to achieve computational efficiency, enabling solvers to address the problem within a reasonable time frame. This work has provided valuable insights into the effective modeling of problems and the design of robust upper bounds. While progress has been made, there are still ideas that can be explored, ranging from different component enumerations to more advanced bound techniques, which we plan to explore the future.

The process of modeling posed significant challenges; however, it underscored the effectiveness of this modeling approach. It offers a structured method for problem-solving, which, when translated into implementation, enables rapid testing with a variety of algorithms without the need for additional development efforts, as the models can be reused. Furthermore, having these models in place allows for a more comprehensive study of the performance of MH methods in the future. This will enable the assessment of which algorithms work best for this problem.

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, titled "Book Scanning". Similarly to the previous chapter, our primary goal lies in describing the modeling process for this particular problem. As such, in Section 6.1 we present a description of the problem focusing on the details found most relevant. In Section 6.2 we describe the modeling strategy used for this problem. In Section 6.3 we highlight the main results that were obtained in comparison to those achieved by participants during the competition. Lastly, Section 6.4 offers some remarks on problem-solving process.

6.1 Problem Description

As described in the problem statement, 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 process requires careful planning for optimal efficiency. Motivated by these real-world challenges, this problem presents a scenario that resembles the real-world

challenges face by the Google Books team.

In the context of this problem, we are presented with a global library system housing a diverse collection of books, with each library having at most one copy of these books. Libraries are characterized by three key factors: the books they hold, sign-up time, and shipping rate. The sign-up time represents the duration (measured in days) required for a library to complete the sign-up process before it can commence sending books for scanning. The shipping rate indicates how many books a library can send for scanning daily once the sign-up process concludes. Henceforth, given a number of libraries denoted as $\mathcal{L}$, we shall use the notation t_ℓ and r_ℓ to refer to the sign-up time of a library, $\ell = 1, \ldots, \mathcal{L}$, respectively.

There are specific constraints associated with tThe book scanning process, as described, is nevertheless limited by time as there is a fixed global deadline,$\mathcal{D}$. Consequently, it is likely that only a portion of all available libraries will undergo the sign-up process. Moreover, for the libraries that are already signed up, this deadline restricts the number of days during which they are allowed to ship books for scanning. Firstly, only one library can be signed-up at any given time, irrespective of any predefined order. Moreover, once the sign-up process commences for a particular library, it cannot be halted, and as soon as it concludes, that library becomes immediately available for sending books for scanning. Notably, each book scanned contributes a designated score; however, this score is counted only once, regardless of how many times the book is scanned. Given a number of books, $\mathcal{B}$, the notation s_b will be used to denote the score of book $b = 1, \ldots, \mathcal{B}$.

The book scanning process, as described, is nevertheless limited by time as there is a fixed global deadline, $\mathcal{D}$. Consequently, it is likely that only a portion of all available libraries will undergo the sign-up process. Moreover, for the libraries that are already signed up, this deadline restricts the number of days during which they are allowed to ship books for scanning.

As an illustrative example, refer to Figure 6.1, which depicts a possible book scanning process. The characteristics of the libraries in the example are detailed in Table 6.1. It is important to note that, in this specific example, the global deadline has been set at 7 days (inclusive).

In this specific example, two libraries have signed up in a non-decreasing order based on their labels, resulting in a combined sign-up time of 5 days. Additionally, the scanning rates for these libraries are 2 and 1 for library 1 and 2, respectively. This implies that with this library order, library 1 can ship a

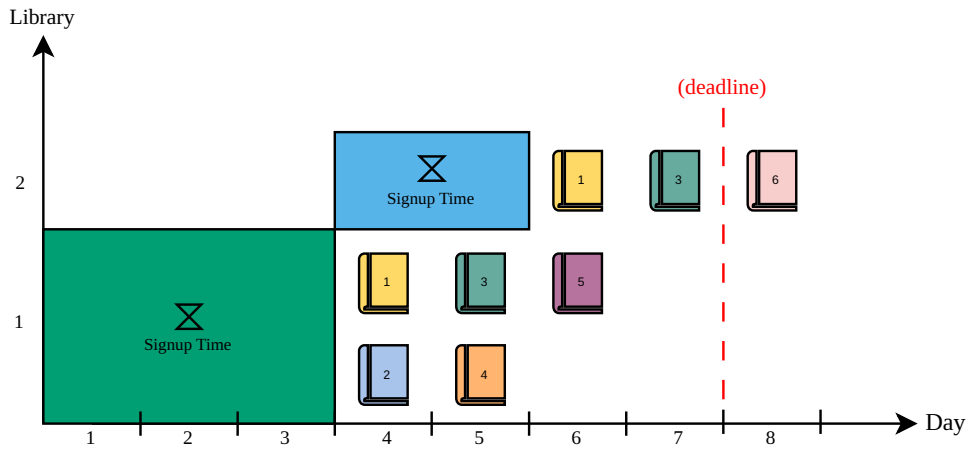

Figure 6.1: Book Scanning Process Example

Library	Sign-up Time	Rate	Books
1	3	2	1, 2, 3, 4, 5
2	2	1	1, 3, 6

Table 6.1: Library Properties

maximum of 8 books, while library 2 can ship 2 books at most. Consequently, the latter ends up not contributing one of its books to the scanning process. Importantly, this example also highlights a scenario where both libraries possess duplicate copies of the same books (books 1 and 3). As mentioned earlier, these duplicate copies will only be scanned once, regardless of the number of duplicates, ensuring that they are counted for scoring only once.

In essence, the objective of this problem is to determine the optimal order for library sign-ups and book selections for scanning, aiming to maximize the sum of the scores of the unique books sent for scanning before the deadline. Mathematically, this can be expressed as shown in Equation (6.1), where, $\mathcal{I}$ the number of libraries signed-up, $\phi^{\mathcal{I}}$ denotes the order for the signed-up libraries $i = 1, \ldots, \mathcal{I}$, and $x_{b,l}$ is a binary variable indicating whether a particular book, b, was shipped by library ℓ for scanning (1) or not (0).

$$
\begin{aligned}
\max f(x) = & \sum_{b=1}^{\mathcal{B}} s_b \cdot \min\left(\sum_{i=1}^{\mathcal{I}} x_{b,\phi_i^{\mathcal{I}}}, 1\right) \\
\text{s.t} & \sum_{b=1}^{\mathcal{B}} x_{b,\phi_i^{\mathcal{I}}} \leq r_i \cdot \left(\mathcal{D} - \sum_{k=1}^{i} t_{\phi_k^{\mathcal{I}}}\right) \quad \forall i = 1, \ldots, \mathcal{I} \\
& \sum_{i=1}^{\mathcal{I}} t_{\phi_i^{\mathcal{I}}} \leq \mathcal{D}
\end{aligned}
\tag{6.1}
$$

Note that in the objective function given by Equation (6.1), the constraints represent the maximum number of books that a library can ship for scanning between its sign-up date and the deadline, as well as the maximum number of libraries that can undergo the sign-up process before the deadline.

For clarification of all the above concepts, consider the books scores presented in Table 6.2.

Book	1	2	3	4	5	6
Score	3	1	5	4	7	1

Table 6.2: Example Book Scores

Given the order for the libraries shown in Figure 6.1 and the properties exposed in Table 6.1 the resulting objective value for this scheduling is $3+1+5+4+7 = 20$.

Lastly, several instance-specific constraints are applied to various parameters that define the data center design within the context of this problem. These constraints are as follows:

- $\mathcal{L}$. The number of libraries ($1 \leq \mathcal{L} \leq 10^5$).
- $\mathcal{T}$. The number of days required to finish a library sign-up ($1 \leq \mathcal{T} \leq 10^5$).
- $\mathcal{B}$. The number of unique books ($1 \leq \mathcal{B} \leq 10^5$).
- $\mathcal{N}$. The number of books per library ($1 \leq \mathcal{N} \leq 10^5$).
- $\mathcal{D}$. The deadline ($0 \leq \mathcal{D} \leq 10^5$).

6.2 Model

In the following, we present the modeling developed for this problem, and highlight aspects that are relevant for its implementation.

Problem

A problem instance is defined by the set of all libraries available, along with their sign-up times, book shipping rate, and list of books that can be shipped, the scores for all the books, and the deadline.

Solution

The solution in this problem is defined by a collection of assignments of books to libraries and the order in which libraries are scanned. It is important to note that, in the context of this problem, any partial solution is considered feasible.

Component

One type of component is a tuple containing a book and library, representing an assignment. Moreover, given that libraries need to be signed-up in order be possible to scan books, another type of component is a pair of libraries (i, j) where, $i = 0, \ldots, \mathcal{L}$ and $j = 1, \ldots, \mathcal{L}$; denoting that library j is signed-up after library i. To denote the first library that is signed up we consider the component $(0, j)$.

Construction Rules

For this problem, we considered two construction rules which are described as follows:

Standard This approach encompasses enumerating all possible combinations of already signed-up libraries and not yet scanned books. Additionally, it also enumerates all libraries that can be signed-up next before the deadline.

Sequential This approach is a slight variation of the previous one. Instead of enumerating all possible books for all signed-up libraries simultaneously, it only generates assignments for the last signed-up library and enumerates the books strictly for that library. Note that, the selection of which library to sign-up next is also an enumerated component. As a result, we reduce the complexity of the enumeration of all components by a factor of $\mathcal{L}$.

Objective Function

The objective function for this problem is the same as the one defined in the problem description.

Upper Bound

A possible bound for this problem is to consider each library as a knapsack problem where the capacity is the number of books that can still be signed-up by a library until the deadline, and each book represents an item with weight of 1 and profit equal to its score. Note that, for libraries already signed-up the time until the deadline is known, whereas for libraries that have not yet been signed up that time is not known. However, we can consider that each not signed up library will be the next to open, thus relaxing the constraint that dictates that two libraries can not be signed-up simultaneously. The bound for this problem is then the sum of a knapsack bound for all libraries considering only the books that have not yet been signed up and that are available for each

library. Note that, we are also relaxing the fact that each book can only score once by possibly counting it for the bound of multiple libraries.

Another bound for this problem, is to consider the number of books that can be shipped until the deadline for all libraries as the knapsack. Thus, we relax the constraint set on the number of books that can be scanned for a particular library imposed by the objective function. Additionally, we consider that a book can be placed in the knapsack regardless of which library it belongs to. The association with a specific library is taken into account once the book is actually scanned. The value of this upper bound is determined by the sum of the scores of the unique books that can fit within the knapsack.

Local Moves

The local moves considered for this problem include:

1. Adding a book to the set of books to be scanned by a given library.
2. Removing a book from the set of books that were going to be scanned by a library.
3. Swapping books between libraries. This move ensures that both libraries have copies of the books being swapped.
4. Removing a book from the list of books to be scanned by a library and adding that book to another library. If possible, replace the removed book in the first library with another one that is available there.

Perturbation

Regarding the perturbation of the solution, for this problem, we did not employ any specific operation apart from randomly applying the described local moves.

6.2.1 Two-Phase Approach

The previous model, although valid, was not competitive in practice. Therefore, an alternative strategy for modeling this problem was developed. This strategy involved breaking down the optimization tasks into two distinct stages. The first stage focused on finding an good order for the libraries, and the second stage employed an algorithm to assign books to the libraries, effectively solving the problem.

Construction Rules

The library selection strategy employed was based on a heuristic value calculated at each step for each library not yet signed-up. The objective was to rank the libraries according to their potential contribution given by the books they contained. There are numerous possible criteria for ranking libraries, such as prioritizing those with the shortest sign-up time, libraries containing the rarest books, or libraries with the highest score, among others. However, the most effective heuristic value for ranking libraries in practice was the one that takes into account libraries with the highest cumulative score of books not present in any of the assigned libraries and could be shipped before the deadline, divided by the sign-up time.

After iteratively selecting libraries based on their heuristic values until it is no longer possible and defining the library order, we can then solve the book assignment optimally using an exact method. There are various algorithms in the literature to solve assignment problems, such as the *Hungarian Method* [25]. However, these algorithms require, in general, the use of an assignment matrix, which for the instances of this problem becomes too large and impractical due to memory constraints.

The alternative approach considered involved modeling the problem as a bipartite graph and using the *max-flow min-cost* algorithm [25] to determine the assignment. In this approach, books and libraries were represented as nodes, and the presence of each book in a library was represented as an edge. Additionally, *source* and *sink* nodes were added to the graph as required by the algorithm. The bipartite graph is illustrated in Figure 6.2, where there are 3 books and libraries, and book 2 is present in two libraries simultaneously.

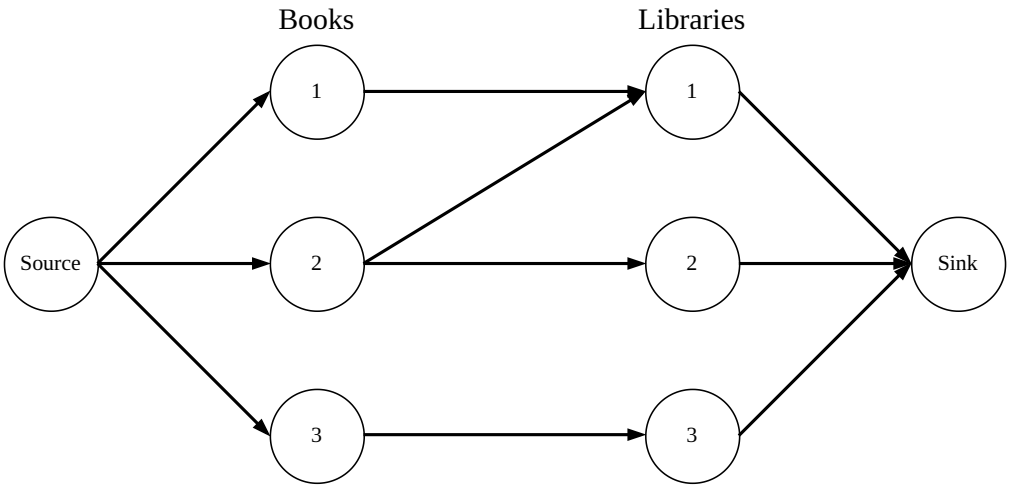

Figure 6.2: Assignment Problem Modeled as a Bipartite Graph

The edges of the graph are then adjusted in the following way:

From Source to Books. The cost of the edge was set to 0 since we do not wish to count the contribution of this edge, and the capacity was set to 1, symbolizing that all books can be considered for assignment.

From Books to Libraries. The cost was set to -s_b to formulate the problem for maximization, and the capacity was set to 1, symbolizing that each book can be assigned once to that particular library, thereby contributing with its score.

From Libraries to Sink. The cost was set to 0 since libraries do not contribute to the score, and the capacity was given a value equal to the number books can be placed in that library, until the deadline $\mathcal{D}$.

It is important to note that, the assignment will correspond to edges in the graph that maximize the flow, and secondarily minimize the cost.

Local Moves

To further improve the objective value, we can experiment with different library orderings by applying the following local moves to the existing order,$\phi^{\mathcal{I}}$.

- Reverse the sign-up order between two libraries in $\phi^{\mathcal{I}}$
- Change the positions of two libraries in the order $\phi^{\mathcal{I}}$, adjusting the sign-up times of every library in between.
- Select one library to remove and add another library that is not currently considered in that position, if possible.

6.3 Results

This chapter, described our model and analysis of the google hash code problem entitled "Book Scanning". This problem presented instances, each with different challenges and relatively large sizes.

We began by approaching the problem from a constructive search perspective, and in doing so, we made several choices related to the enumeration of components and the calculation of the upper bound. Initially, we attempted to use the *standard* enumeration method for components, but it quickly became apparent that this approach was impractical due to the size of the instances. Consequently, we switched to the *sequential* enumeration method, which was better suited for handling the large instances and ultimately became our default

choice. Concerning the upper bound, the first one presented in the problem model proved to be too computationally demanding and was not used in practice. However, the second bound, which further relaxed the constraints of the problem, proved to be more performant and is the one we utilized in practice.

These choices enabled us to employ a greedy constructive MH that selected the solution with the best upper bound value at each step with a 15 minute timeout, leading to the results presented in Table 6.3. Other experiments were conducted with various constructive meta-heuristics we developed in the context of the modeling framework. However, these approaches proved to be slower in optimizing the solution within the aforementioned time frame.

Instance	Score
"Read On"	5 822 900
"Incunabula"	185 017
"Tough Choices"	2665
"So many books"	140 189
"Libraries of the world"	275 446

Table 6.3: Book Scanning Simple Constructive Search Results

From a local search perspective, we encountered challenges in achieving improved results using the implemented meta-heuristics. Despite having promising local moves in theory, none of them led to improvements in the solution obtained after constructive search. Consequently, we considered an alternative approach that focused on optimizing the ordering of libraries and then solved the book assignment problem exactly.

For this purpose, we employed the two-phase approach which yielded the best results we were able to achieve. Remarkably, the usage of an exact solver for the *max-flow min-cost* problem provided in the *Python* package *networkx*[1] achieved surprisingly good performance, taking at most 2 minutes to fully construct a solution after an ordering for the libraries was established, even in the largest instance. Furthermore, the local search following this construction, using a HC technique, was able to further improve the scores for some of the instances within the 30-minute time limit we set for this procedure.

This approach indeed yielded the best results we were able to achieve, which are presented in detail in Table 6.4alongside the best results obtained by other contestants during the competition.

Remarkably, this placed us in **33rd** place on the competition leaderboard.

[1] https://networkx.org/

Instance	Score	Best Known Score
“Read On”	5 822 900	5 822 900
“Incunabula”	5 689 822	5 690 888
“Tough Choices”	5 028 530	5 107 113
“So many books”	5 208 455	5 237 345
“Libraries of the world”	5 328 034	5 348 248

Table 6.4: Book Scanning Best Results

6.4 Concluding Remarks

In this chapter, we presented our analysis and described the model for the Google Hash Code problem entitled “Book Scanning” This problem posed significant challenges, both in terms of conceptualization and implementation using the modeling framework.

From a conceptual perspective, developing the model for this problem involved a great deal of thought about possible approaches to address the issues arising from the large instance sizes. Many ideas were tested, and most of them yielded poor results due to either slowness or worse performance on specific problem instances with more complex characteristics. Nevertheless, this problem underscored the importance of creating a highly descriptive model and demonstrated that a well-crafted model can significantly impact the effectiveness of problem-solving algorithms.

From an implementation perspective, our experience showed that the framework is flexible enough to accommodate various problem-solving approaches. It provides a versatile API that gives users the freedom to employ different strategies for solving problems while adhering to the framework’s specifications. This flexibility allows users to leverage existing solvers and experiment with new algorithms easily.

Chapter 7

Conclusion

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

— Alan Turing

7.1 Future Work

Acronyms

ACO Ant Colony Optimization. x, 19, 20, 25, 48, 50, 66

API Application Programming Interface. 27–30, 46, 51, 56, 88

BBO Black-Box Optimization. 12, 26, 44, 45

BI Best Improvement. 21, 24, 66

BS Beam Search. x, 16, 17, 25, 56–58, 60, 66, 77

CO Combinatorial Optimization. 2, 3, 5–9, 12, 13, 44

CS Constructive Search. x, 14–16, 18, 19, 25, 26, 29, 30, 48, 49, 53–56, 61, 77

FI First Improvement. 21, 22, 60, 66

GBO Glass-Box Optimization. 12, 25

GO Global Optimization. 10, 11

GRASP Greedy Randomized Adaptive Search Procedure. x, 18, 19, 25, 52, 66, 77

HC Hill Climbing. x, 21, 22, 25, 87

IG Iterated Greedy. x, 18, 19, 25, 51, 66, 77

ILP Integer Linear Programing. 12, 25

ILS Iterated Local Search. x, 22, 25, 56, 60, 66, 78

KP Knapsack Problem. 7, 8, 25, 26, 72

LCG Linear Congruential Generator. 53, 54

LO Local Optimization. 10, 11

LS Local Search. x, 14, 15, 18, 19, 21–26, 29, 48, 52–56, 60, 61, 78

MH Meta-Heuristic. 2, 3, 6, 8, 9, 13, 16–23, 26, 28, 47, 48, 56, 57, 60, 62, 64, 66, 78, 87

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

nasf4nio-cs Not Another Framework for nature-inspired optimization — Constructive Search. 28, 29, 46

POF Python Optimization Framework. 28, 29, 46

SA Simulated Annealing. x, 19, 22, 23, 25, 66

TS Tabu Search. x, 23–25, 66

Bibliography

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

[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 Scheduling†”. 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 Optimization: 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 metaheurísticas”. 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] Lyle Ramshaw and Robert E. Tarjan. "On Minimum-Cost Assignments in Unbalanced Bipartite Graphs". In: *HP Labs, Palo Alto, CA, USA, Tech. Rep. HPL-2012-40R1* 20 (2012).

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

[27] 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.

[28] 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.

[29] 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.

[30] Thomas Witelski and Mark Bowen. *Methods of Mathematical Modelling*. Springer Undergraduate Mathematics Series. Cham: Springer Interna-

tional Publishing, 2015. ISBN: 978-3-319-23041-2. DOI: 10.1007/978-3-319-23042-9.

[31] 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.

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

[33] 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.

[34] 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.

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

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

[37] 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.

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

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

[40] Pedro Rodrigues. *Hashcode-Models*. Github. URL: https://github.com/pedromig/hashcode-models.

[41] Pedro Rodrigues. *Nasf4nio-Py*. GitHub. URL: https://github.com/pedromig/nasf4nio-py.