UNIVERSIDADE D
COIMBRA

Samuel Barroca do Outeiro

# An Application Programming Interface for Constructive Search

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

October 2021

Samuel Barroca do Outeiro
An Application Programming Interface for Constructive Search
UNIVERSIDADE D COIMBRA
1 2 9 0

Faculty of Sciences and Technology
Department of Informatics Engineering

# An Application Programming Interface for Constructive Search

Samuel Barroca do Outeiro

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

October 2021

UNIVERSIDADE D
COIMBRA

# Acknowledgements

Firstly, I would like to express my gratitude to my advisor Professor Carlos Fonseca for his patient guidance and continuous support throughout this research work. Since I started to collaborate with him at the beginning of my master's degree, I have learned so much from him and deepened my knowledge in the field of optimization, which I am very enthusiastic about and expect to continue to work on in the future. Additionally, he has granted me various opportunities to obtain a scholarship, for which I am very thankful.

I would also like to thank all members of the ALGorithms and Optimization (ALGO) Laboratory for their participation in a coding event, where they got hands-on experience on the work developed in this dissertation.

I wish to thank all my friends for their companionship and the great moments spent with them. Their presence will always be appreciated, especially in moments when work performance is at its lowest, and I need to unwind from it. I am also grateful for their support in times where I needed them.

And special thanks to my family, who always showed me love and care during my best and worst times and always kept me motivated.

Finally, I wish to acknowledge that this work was financed by national funds through the Foundation for Science and Technology (FCT), in the scope of project CISUC — UIDB/00326/2020 under research grant L.726690-Ref 1, and was partially carried out in the scope of project MobiWise: From Mobile Sensing to Mobility Advising (P2020 SAICTPAC/0011/2015), co-financed by COMPETE 2020, Portugal 2020 – Operational Program for Competitiveness and Internationalization (POCI), European Union's European Regional Development Fund (ERDF), and the Foundation for Science and Technology (FCT).

# Abstract

Optimization practice is intimately related to the availability of software tools to support it. Current approaches to combinatorial optimization typically fall into one of two broad classes: glass-box mathematical programming formulations and solvers such as Integer Linear Programming (ILP) and Constraint Programming (CP), and black/gray-box problem models and (usually heuristic) algorithms implemented directly in software. The latter may be more flexible and easier to integrate into existing workflows. However, the lack of a *de-facto standard* for modeling/implementing optimization problems in software hinders the adoption of such algorithms in practice. In fact, different optimization software frameworks usually require problems to be implemented in that framework. In addition, most frameworks strongly emphasize local search algorithms over constructive search. Thus, an opportunity arises for the development of an Application Programming Interface (API) for constructive-search problems and algorithms.

This API separates the problem formulation from the algorithm that solves it by specifying a number of abstract elementary operations that problems must implement and solvers can use in a problem-independent way. Both exact and (meta)heuristic algorithms are supported, including Branch & Bound, Beam Search, GRASP, Ant Colony Optimization algorithms, among others.

The implications of the proposed abstraction for the development of novel constructive-search algorithms are also discussed in this work. Notably, a study is conducted on the effect of the problem model on solver performance.



# Keywords

Constructive Search, Combinatorial Optimization, Exact Algorithms, Metaheuristics, Optimization Software.

## Resumo

A utilização prática de otimização está intimamente relacionada com a disponibilidade de ferramentas de *software* para a suportar. As abordagens atuais para otimização combinatória caem em duas categorias: modelos de caixa branca (ou transparente) tais como Programação Linear Inteira (ILP) e Programação por Restrições (CP), e modelos de problemas de caixa preta/cinzenta e algoritmos (tipicamente heurísticos) que são diretamente implementados em *software*. Estes últimos poderão ser mais flexíveis e fáceis de integrar em fluxos de trabalho existentes. Contudo, a falta de *convenções* para a modelação/implementação de problemas de otimização em *software* dificultam a adoção de tais *algoritmos* na prática. De facto, distintos *frameworks* de *software* de otimização tipicamente obrigam a que os problemas estejam implementados nessa *framework*. Para além disso, a maioria das *frameworks* focam-se principalmente em algoritmos de procura local ao invés de procura construtiva. Assim, surge uma oportunidade para o desenvolvimento de uma Interface de Programação de Aplicações (API) para problemas e algoritmos de procura construtiva.

Esta API separa o (modelo do) problema do algoritmo que o resolve através da especificação de várias operações elementares e abstratas que os problemas precisam de implementar e que os algoritmos podem usar independentemente do problema. Tanto algoritmos exatos, como (meta)heurísticos são suportados, dos quais se destacam o *Branch & Bound*, o *Beam Search*, o GRASP, os algoritmos de *Ant Colony Optimization*, entre outros.

As consequências da abstração proposta para o desenvolvimento de novos algoritmos de procura construtiva são também discutidos neste trabalho. Em particular, é conduzido um estudo sobre o efeito dos modelos de problemas no desempenho de algoritmos.

## Palavras-Chave

Procura Construtiva, Otimização Combinatória, Algoritmos Exatos, Meta-heurísticas, Software de Otimização.

# Contents

**Contents** ix

**List of Figures** xiii

**List of Tables** xv

**List of Algorithms** xvii

**1 Introduction** 1

**2 Background** 3
- 2.1 Combinatorial Optimization . . . 3
- 2.2 Categories of Optimization Techniques . . . 4
  - 2.2.1 Local and Constructive Search . . . 4
  - 2.2.2 Exact, Approximation, and Heuristic Approaches . . . 7
- 2.3 Constructive-Search Algorithms . . . 8
  - 2.3.1 Branch & Bound . . . 8
  - 2.3.2 Iterated Greedy Algorithms . . . 9
  - 2.3.3 Greedy Randomized Adaptive Search Procedures . . . 10
  - 2.3.4 Ant Colony Optimization . . . 10
  - 2.3.5 Beam Search . . . 12
- 2.4 Concluding Remarks . . . 13

**3 State-of-the-Art Optimization Software for Constructive Search** 15
- 3.1 Description of Optimization Software for Constructive Search . . . 16
- 3.2 Constructive-Search Algorithms Supported by Optimization Software . . . 17
- 3.3 Concluding Remarks . . . 18

**4 An API for Constructive Search** 21
- 4.1 Fundamental Concepts . . . 21
- 4.2 Requirements . . . 25
- 4.3 Proposed API for Constructive Search . . . 26
- 4.4 Concluding Remarks . . . 32

**5 Evaluation of Constructive Search API** 33
- 5.1 Formulation of Optimization Problems . . . 33
  - 5.1.1 Knapsack Problem . . . 34
  - 5.1.2 Travelling Salesman Problem . . . 36
  - 5.1.3 Cable Trench Problem . . . 41
- 5.2 Implementation of Constructive-Search Algorithms . . . 44
  - 5.2.1 Branch & Bound . . . 44
  - 5.2.2 Iterated Greedy Algorithms . . . 46

5.2.3 Greedy Randomized Adaptive Search Procedures . . . . . . . . . . . 47
5.2.4 Ant Colony Optimization . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2.5 Beam Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.3 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
5.3.1 Computational Overhead Introduced by the API . . . . . . . . . . . . 53
5.3.2 Solver Performance Evaluation . . . . . . . . . . . . . . . . . . . . . 56
5.3.3 Effect of the Problem Model on Solver Performance . . . . . . . . . . 59
5.3.4 Computational Overhead of the Second Level of the API . . . . . . . 61
5.4 User Feedback . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.5 Concluding Remarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

**6 Conclusion and Future Work** **69**

**References** **73**

# Acronyms

**ACO** Ant Colony Optimization.

**ACS** Ant Colony System.

**ANTS** Approximate Nondeterministic Tree Search.

**API** Application Programming Interface.

**AS** Ant System.

**AS**$_{rank}$ Ranked-Based Ant System.

**B&B** Branch & Bound.

**BS** Beam Search.

**CO** Combinatorial Optimization.

**CTP** Cable Trench Problem.

**DP** Dynamic Programming.

**EA** Evolutionary Algorithm.

**EAS** Elitist Ant System.

**GRASP** Greedy Randomized Adaptive Search Procedures.

**HC-ACO** Hyper-Cube Framework for Ant Colony Optimization.

**IG** Iterated Greedy.

**ILS** Iterated Local Search.

**KP** Knapsack Problem.

**MKP** Multidimensional Knapsack Problem.

$\mathcal{MM}$**AS** $\mathcal{MAX}-\mathcal{MIN}$ Ant System.

**MOF** Metaheuristic Optimization Framework.

**QAP** Quadratic Assignment Problem.

**QMKP** Quadratic Multidimensional Knapsack Problem.

**SA** Simulated Annealing.

**TS** Tabu Search.

**TSP** Travelling Salesman Problem.

**TSPLIB** Travelling Salesman Problem Library.

**VND** Variable Neighborhood Descent.

# List of Figures


2.1 Illustration of basins of attraction for a search space with one dimension . . 5
2.2 Example of dominance between two partial tours . . . . . . . . . . . . . . . . 7

4.1 Example of a construction graph . . . . . . . . . . . . . . . . . . . . . . . . . 24

5.1 Illustration of an empty solution for the Travelling Salesman . . . . . . . . . 38
5.2 Example of construction with multiple-fragment heuristic . . . . . . . . . . . 39
5.3 Example of intermediate lower bound for the Travelling Salesman . . . . . . 40
5.4 Example of strong lower bound for the Travelling Salesman . . . . . . . . . 40
5.5 Illustration of an empty solution for the Cable Trench . . . . . . . . . . . . 42
5.6 Example of construction with greedy heuristic for the Cable Trench . . . . . 43
5.7 Example of strong lower bound for the Cable Trench . . . . . . . . . . . . . 44
5.8 Run times of "handcrafted B&B" and "B&B using API" . . . . . . . . . . . 54
5.9 Distribution of run time ratio of "B&B using API" to "handcrafted B&B" . 54
5.10 Solver performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.11 Effect of lower bound on solver performance . . . . . . . . . . . . . . . . . . 60
5.12 Run times of first-level and second-level solver . . . . . . . . . . . . . . . . 63

# List of Tables


3.1 Supported constructive-search algorithms . . . . . . . . . . . . . . . . . . . . . 18

5.1 Flat profile of the "Branch & Bound (B&B) using API" . . . . . . . . . . . . 55
5.2 Parameter values fixed for each solver . . . . . . . . . . . . . . . . . . . . . . 57
5.3 Flat profile of GRASP applied to the *kroA100* instance . . . . . . . . . . . . 64
5.4 Flat profile of Beam Search applied to the *kroA100* instance . . . . . . . . . 65

# List of Algorithms


2.1 Branch & Bound . . . . . 9
2.2 Iterated Greedy Algorithm . . . . . 10
2.3 Greedy Randomized Adaptive Search Procedures (GRASP) . . . . . 10
2.4 Ant Colony Optimization . . . . . 12
2.5 Beam Search . . . . . 12

5.1 Greedy heuristic for the Knapsack . . . . . 35
5.2 Upper bound for the Knapsack . . . . . 35
5.3 Depth-first Branch & Bound based on the API . . . . . 45
5.4 Iterated Greedy algorithm based on the API . . . . . 47
5.5 Greedy Randomized Adaptive Search Procedures (GRASP) based on the API 48
5.6 Ant System based on the API . . . . . 50
5.7 Beam Search based on the first level of the API . . . . . 51
5.8 Beam Search based on the second level of the API . . . . . 52

# Chapter 1

# Introduction

Many important problems of practical interest can be expressed as Combinatorial Optimization (CO) problems. Such problems involve finding the best solution in a potentially very large set of solutions, where exhaustive enumeration would take an unacceptable amount of time. Thus, it is essential to adopt efficient strategies to find the best solution in a suitable amount of time.

A variety of optimization techniques is available to solve CO problems. These techniques vary in terms of what quality guarantee they provide for the results as well as the time they take to produce those results. Some find solutions for a specific set of problems, while others offer an abstraction adjustable to many problems. Several are inspired by natural phenomena such as natural selection and collective behavior, while others are based on physical and other phenomena. The point is that there is such a panoply of optimization strategies that it is a challenge to mention them all. Among the most well-known are Backtracking, Branch & Bound (B&B) [20; 49; 51; 52], Hill Climbing, Evolutionary Algorithms (EAs) [7], Ant Colony Optimization (ACO) [28; 32], Simulated Annealing (SA) [47].

Due to the panoply of available optimization techniques, it is generally advised to decide on an appropriate for each given CO problem. The *No Free Lunch* theorems states that all algorithms have the same average performance over the entire set of optimization problems. However, that does not prevent some algorithms from being better than others on specific classes of problems [16]. Therefore, it is usually advised that various optimization strategies are applied to a given problem and subsequently assessed to decide on the best one. Consequently, the availability of general-purpose optimization software is beneficial since it eases the implementation, testing, and comparison of different algorithms. Furthermore, it is also favorable to test a given algorithm for several problems [75].

This work explores the domain of CO by focusing on some well-known problems and related optimization strategies. Specifically, it focuses on constructive optimization strategies. Constructive algorithms build a solution from scratch by adding components one at a time until a complete and feasible solution is reached. Examples of such algorithms are B&B [20; 49; 51; 52], ACO [28; 32], the first stage of Greedy Randomized Adaptive Search Procedures (GRASP) [35; 36], and Dynamic Programming (DP) [9; 10].

An Application Programming Interface (API) for constructive search is proposed in this work. The API defines the set of operations suitable for the implementation of a variety of constructive algorithms. These operations were derived by studying an extensive selection of algorithms that are constructive in nature. Additionally, a survey of optimization software and conceptual models for constructive search was performed, in order to create an

API that is robust and adaptable for various algorithms. This API enables various algorithms to be implemented in a problem-independent way, and CO problems to be modeled and implemented independently of particular algorithms.

The proposed API also has the potential of exposing problem features that may be exploited by constructive algorithms. These features would enable constructive algorithms to leverage additional information provided by the problem. As a result, the search for good solutions for the problem would be more efficient. One example is how much a lower bound is affected by adding a specific component to a partially constructed solution. That value would then serve as an alternative to the static heuristic information associated with each component commonly used in the literature.

Overall, the main contribution provided by this work is the definition of an API that establishes a common foundation for constructive algorithms. The proposed API is developed by analyzing constructive-search algorithms in the literature in light of a conceptual model developed by others [57], and allows constructive-search algorithms to be implemented independently of any problem-specific details. It also allows problems to be modeled and implemented independently of the algorithms that will be used to solve them. This aspect provides for the fair comparison of different strategies on a specific problem, as long as those strategies share the same common foundation. Other contributions include:

- The implementation of a set of constructive-search algorithms (*solvers*) based on the proposed API. These solvers can be applied to any constructive-search problem model implementing the API specification.
- A set of problem models implemented according to the proposed API. These models serve as documented examples of how the API can be used in practice and, together with the implemented solvers, provide evidence of the expressiveness of the API.
- A set of experiments aimed at evaluating some computational aspects of the proposed API. These experiments allow the computational overhead introduced by the use of the API to be measured in a concrete scenario, and shed light onto how different models of the same underlying combinatorial optimization problem may affect the optimization process.

Furthermore, this API promotes the development of improved models in order to make the search more efficient. In particular, making given model operations faster directly benefits all solvers that rely on such operations.

The remainder of this document is structured as follows. In Chapter 2, some background concepts are presented and the goals surrounding this work are further contextualized. Chapter 3 provides an overview of optimization software for constructive search and the corresponding limitations. Chapter 4 describes the process of developing the proposed API for constructive search, while Chapter 5 presents the results of its evaluation. Finally, the main conclusions from this work are presented Chapter 6, where possible research directions in future work are also identified.

# Chapter 2

# Background

This chapter provides an overview of the concepts that better contextualize the development of an Application Programming Interface (API) for constructive search. Specifically, it goes into more detail on the theory and literature closely related to constructive search while briefly referring to other ideas within the same domain. It also provides a background for topics discussed in subsequent chapters. Section 2.1 presents the field of combinatorial optimization, where this work belongs. Section 2.2 distinguishes, describes, and compares different categories of optimization techniques. In particular, a distinction is made between local and constructive search, and also between exact, approximation, and heuristic approaches. Then, as this work mainly focuses on constructive search, Section 2.3 describes various well-known algorithms, and identifies operations that are common to those algorithms. Finally, Section 2.4 provides some concluding remarks on the concepts that were reviewed.

## 2.1 Combinatorial Optimization

As defined by Papadimitriou and Steiglitz [63], optimization is the act of finding the best configuration or set of parameters for a problem, according to a given objective. Such problems constitute a hierarchy that classifies them according to their specification and the techniques to solve them. They may be divided into two categories: those with *continuous* variables and those with *discrete* variables. Continuous problems are defined on a continuous subset of the real numbers or real vectors for a set of real numbers or a function. Meanwhile, discrete problems are defined on a finite, or countably infinite, set. In general, an optimization problem can be defined as follows:

**Definition 2.1 (Optimization problem [63])** *An instance $\mathcal{I}$ of an optimization problem is a pair $(\mathcal{S}, f)$, where $\mathcal{S}$ is the set of feasible solutions for the problem, and $f$ is the objective function, which is a mapping such that*

$$f : \mathcal{S} \mapsto \mathbb{R}$$

*That is, each solution $s \in \mathcal{S}$ is assigned a numeric value indicating the quality of that solution. The optimization problem consists in finding a solution $s^* \in \mathcal{S}$ for which*

$$f(s^*) \leq f(s), \text{ for all } s \in \mathcal{S}$$

*Such a solution $s^*$ is called a (globally) optimal solution.*

This definition applies only to minimization problems. However, it is possible to reformulate maximization problems for minimization. Thus, without loss of generality, only minimization problems are considered in this work.

Discrete problems include a subset of problems that are called *combinatorial*. The corresponding solutions are expressed using concepts in combinatorics, such as sets, permutations, graph structures, etc. In regard to the previous definition, a Combinatorial Optimization (CO) problem can be further defined as follows.

**Definition 2.2 (Combinatorial optimization problem [22; 63, p. 1055])** *An instance $\mathcal{I}$ of a combinatorial optimization problem is an instance of an optimization problem where the set of feasible solutions $\mathcal{S}$ is a finite set. These solutions can be encoded as binary strings by combining the representation of their constituent parts, i.e., an instance $\mathcal{I}$ has an associated (finite) set of components, called the ground set, $\mathcal{G} := \{e_1, ..., e_n\}$, and these components can be present* (1) *or absent* (0) *in a solution $s \in \mathcal{S} \subseteq 2^{\mathcal{G}}$.*

As an illustrative example for the previous definition, let us consider two CO problems: the Knapsack Problem (KP) [78, p. 270–271] and the Travelling Salesman Problem (TSP) [78, p. 276–278]. For the first one, $\mathcal{G}$ is the set of items that may be put into the knapsack with capacity $W$. Each solution $s \in \mathcal{S}$ is a binary string, where $s_i = 1$ means item $i$ is put into the knapsack and $s_i = 0$ means that item $i$ is discarded. Hence, each solution $s \in \mathcal{S}$ is a subset of items (in $\mathcal{G}$) such that the capacity of does not exceed $W$. As for the second problem, identifying $\mathcal{G}$ is not as straightforward. $\mathcal{S}$ is a set of Hamiltonian cycles, typically represented by permutations that indicate the order in which cities are visited. By projecting this problem into a graph, it becomes apparent that solutions differ in the edges that constitute a Hamiltonian cycle. Thus, $\mathcal{G}$ is the set of edges that connect any two distinct vertices in the graph, and $s \in \mathcal{S}$ is a binary string, where $s_{ij} = 1$ means edge $(v_i, v_j)$ is in the Hamiltonian cycle and $s_{ij} = 0$ means edge $(v_i, v_j)$ is not in the Hamiltonian cycle.

## 2.2 Categories of Optimization Techniques

In the literature there is a vast number of techniques for solving optimization problems. Those techniques are sufficiently diverse such that it is possible to structure them in a hierarchy that categorizes them appropriately. However, such categorizations are slightly inconsistent across the literature. Regardless, there are a few categories of techniques that should be clearly defined to contextualize this work. Specifically, a distinction is made between local and constructive search, and another between exact, approximation, and heuristic algorithms.

### 2.2.1 Local and Constructive Search

Various algorithms can be used to solve CO problems. Such algorithms can be classified according to different properties, and one of those properties is how solutions are manipulated throughout the optimization process. Namely, most algorithms may be categorized as local search approaches or constructive search approaches.

Local search approaches [13; 56] start from some feasible solution to the problem, which is improved throughout the optimization process by modifying the current solution into a *tentatively* better one, selected from a defined neighborhood. In other words, each

solution has a neighborhood containing a set of feasible solutions, known as *neighbors*, which result from modifying that solution. As a general rule, a neighbor with better quality is selected to replace the current solution (where possible), but controlled degradation is also accepted by some approaches. Furthermore, the neighborhood of a solution is specified by a *neighborhood structure* (or *function*), which can be defined as follows:

**Definition 2.3 (Neighborhood structure [13])** *A neighborhood structure* $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ *assigns to each solution* $s \in \mathcal{S}$ *a set of neighbors* $\mathcal{N}(s) \subseteq \mathcal{S}$*, called the neighborhood of* $s$*.*

It is usually considered that the $s \in \mathcal{N}(s)$. A neighborhood structure can also be defined by the set of rules that must be applied to a solution in order to generate all of its neighbors. Furthermore, a solution is called a local minimum if there is no neighbor with better quality than that solution. Formally, a local minimum can be defined as:

**Definition 2.4 (Local minimum [13])** *A local minimum with respect to a neighborhood structure* $\mathcal{N}$ *is a solution* $\hat{s}$ *such that* $\forall s \in \mathcal{N}(\hat{s}) : f(\hat{s}) \leq f(s)$*. Moreover, a solution* $\hat{s}$ *is called a strict local solution if* $\forall s \in \mathcal{N}(\hat{s}) \setminus \{\hat{s}\} : f(\hat{s}) < f(s)$*.*

Local search approaches solve a local formulation of a global optimization problem, and converge towards a local minimum. Furthermore, the absolute quality of the locally optimal solutions found with these approaches strongly depends on the rules that define the neighborhood structure. Thus, a local search algorithm for a given neighborhood structure partitions the search space into so-called basins of attraction of local minima, as shown by Figure 2.1. In this figure, $s_i$ are the local minima, where $i = 1, 2, \ldots, 4$.

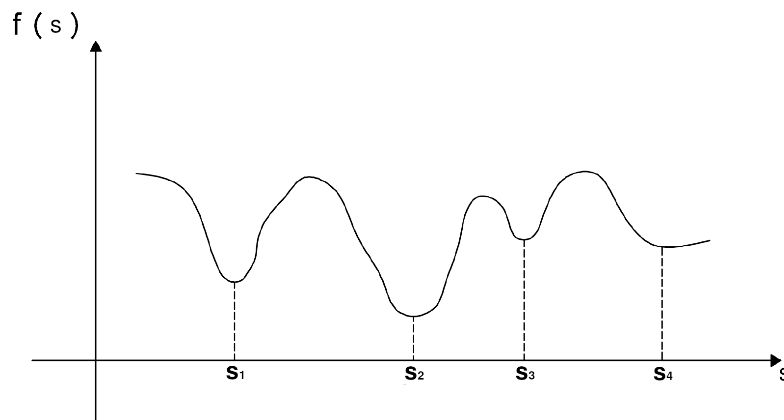



Figure 2.1: Illustration of basins of attraction for a search space with one dimension

In this case, the basin of attraction of a local minimum is the set of solutions that are guaranteed to cause the search to terminate in that same local minimum, given a specific neighborhood structure and local search approach.

Constructive search [13; 56] approaches, in contrast, work differently than local search approaches. Here, solutions are built from scratch, seeing that these approaches generally start with an empty solution that is iteratively constructed by adding new components. This addition follows a construction mechanism, which specifies at each iteration the possible extensions for a partially constructed solution. In other words, at each construction step, there is a set of components that can be added to a partial solution, from which one

is selected and subsequently added. The construction process terminates when no further extension can be made for a partial solution, which means that the solution is complete.

Two concepts that are usually associated to the constructive search paradigm are those of lower bound and dominance relation, especially in exact algorithms such as Branch & Bound [20; 49; 51; 52] and Dynamic Programming [9; 10]. In particular, in the context of constructive search, a *lower bound* may be defined as follows:

**Definition 2.5 (Lower bound [63])** *A lower bound of a (partial) solution $s^p$ is a numeric value $\Phi_{lb}(s^p)$ such that $\forall s \supseteq s^p : \Phi_{lb}(s^p) \leq f(s)$.*

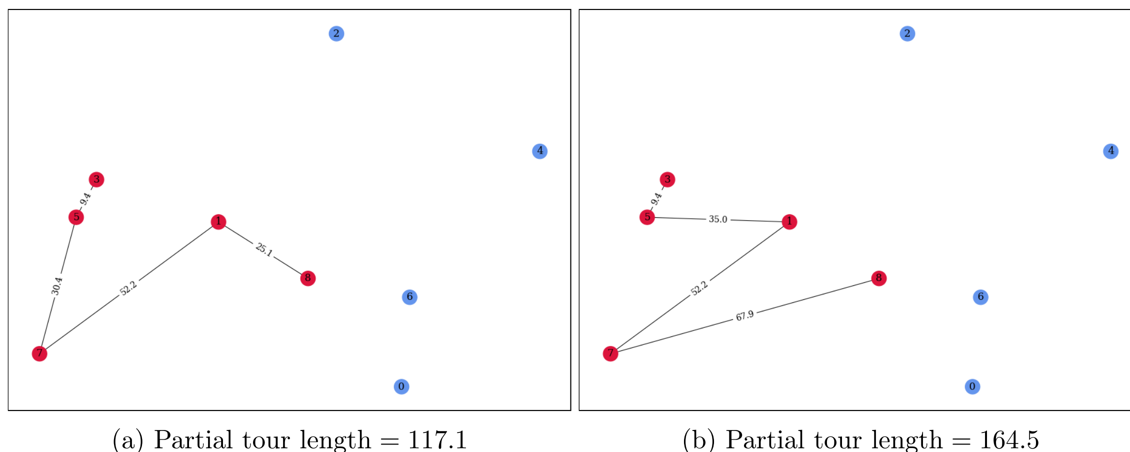
This definition shows that the lower bound of a partial solution is less than or equal to the objective value of each and every feasible solution that extends that partial solution. A lower bound is usually used to construct a proof of the optimality of a given (partial) solution. In other words, it is used to estimate the objective value of the best feasible solution that extends a given (partial) solution (and such an estimate must not exceed that value) without exhaustively enumerating all solutions. Furthermore, if the lower bound of a partial solution is greater than the objective value of a feasible solution that does not extend the latter, then that feasible solution may "*prune*" all solutions which extend the partial solution.

In the context of constructive search, a *dominance relation* can be defined as:

**Definition 2.6 (Dominance relation [63])** *If at any point the best extension of a partial solution $y$ is at least as good as the best extension of another partial solution $x$, then it is said that $y$ dominates $x$.*

Dominance relations are a crucial concept in optimization since they aid the search by avoiding the exploration of subspaces that do not contain high-quality solutions (better than the best-so-far solution). As an illustrative example of a dominance relation, let us consider the KP and the TSP. In the KP, a (partial) knapsack solution dominates another (partial) knapsack solution if the former has an overall profit greater than or equal to the latter, and it has an overall weight less than or equal to the latter. This relation implies that the best extension of the dominated solution will not be as good as the best extension of the non-dominated solution, thus suggesting that the former does not need to be extended further. In the case of the TSP, a partially constructed tour dominates another partially constructed tour if both visit the same cities, begin and end in the same cities, and the length of the former is less than or equal to the length of the latter. Figure 2.2 illustrates such a relation, where partial tour 2.2a dominates partial tour 2.2b. Furthermore, the dominated partial tour does not need to be extended further since it will not result in a better (feasible) tour.

Constructive search approaches are suitable to solve optimization problems that contain useful information associated with the internal structure of solutions. Such information may correspond to how a component or a combination of components affect the overall solution quality. The availability of this information is required for these approaches, as they directly exploit it throughout the optimization process. In particular, using this knowledge in a constructive approach is valuable since it allows partial solutions to be extended towards high-quality solutions. Thus, it is said that the construction process is guided by context information, where the relation between the internal solution structure and overall solution quality is well understood.




(a) Partial tour length = 117.1

(b) Partial tour length = 164.5

Figure 2.2: Example of dominance between two partial tours

Local search approaches, in contrast, are suited to optimization problems that primarily contain information related to the external neighborhood structure. In other words, they are suitable for solving problems that merely provide a measurement for the overall quality of each solution. Furthermore, specifying a neighborhood is crucial for these approaches, as it determines the search space exploration. The neighborhood structure should also allow for any solution to be reached from any other solution by a sequence of moves between neighbors. The evaluation of a neighborhood is thus an essential aspect for a local search approach, since it enables the discrimination of potentially good neighbors, which may lead the exploration towards an optimum. Thus, the local search process is guided by the stipulated neighborhood and the overall quality of solutions.

It is also possible to interpret constructive search as a local search procedure acting on an extended search space that includes partial and complete solutions. In such a case, the neighborhood is defined by the applicable construction mechanisms. Furthermore, local search approaches may also explore this extended search space if the appropriate construction mechanisms are defined as the neighborhood structure. This topic will be further detailed in Section 4.1 of Chapter 4. However, it will not be implemented in this work since it is not the main focus.

Lastly, some well-known algorithms for local and constructive search can be identified. An sample of local-search algorithms includes Iterated Local Search [53], Tabu Search [41], Simulated Annealing [47], Variable Neighborhood Descent [43], and Evolutionary Algorithms [7]. Moreover, constructive-search algorithms include Branch & Bound [20; 49; 51; 52], GRASP [35; 36], and Ant Colony Optimization [28; 32], among others. These are further explained in Section 2.3, along with other constructive-search algorithms.

### 2.2.2 Exact, Approximation, and Heuristic Approaches

Two other properties used to classify CO algorithms are the time complexity and the guarantee to find an optimal solution. With respect to the latter, an algorithm may be categorized as exact, approximation, or heuristic. Furthermore, metaheuristics are an important family of heuristic algorithms.

*Exact* approaches [37] guarantee that a CO problem is always solved to optimality. However, using these methods often becomes practically impossible as the size of those instances increases. Since these methods usually operate by enumerating all the possible solutions

(implicitly or explicitly) and choosing the best one, and the number of solutions typically grows exponentially (or even faster!) with the size of the problem, so does the computation time in many cases. In fact, although some CO problems can be solved in polynomial time (e.g. the Shortest Path Problem), no exact polynomial-time approach is known for many CO problems, which are therefore considered computationally intractable. Some examples of exact approaches are Branch & Bound [20; 49; 51; 52] and Dynamic Programming [9; 10].

Alternatively, *approximation* approaches [37; 77] find suboptimal solutions whose objective value is guaranteed to be within a certain proximity of the optimum value. For example, consider an $\alpha$-approximation algorithm that solves a minimization problem, where $\alpha$ is known as the approximation factor, and let $\alpha = 2$. For all instances of the problem, this algorithm returns a solution whose value is at most twice as large as the optimum value. An advantage of these algorithms is that they often take polynomial time to find a solution. However, there are cases where no approximation algorithm is available to solve an optimization problem, as those algorithms are generally problem-specific, an others where a given algorithm has a high approximation factor, implying that it *may* find lower-quality solutions in contrast to an algorithm that has a tighter guarantee.

Finally, *heuristic* approaches [14; 37; 54] find near-optimal solutions but do not provide a guarantee about their quality in regard to the optimum. Heuristics are usually problem-specific. Regardless, if a heuristic is appropriately designed, it generally produces good solutions in a reasonable computation time. Furthermore, metaheuristics are an important subclass of heuristic algorithms. Metaheuristics are high-level strategies that “guide” the search process in the solution space. They present abstract level descriptions, which allows them to be adapted to a variety of problems. A crucial aspect of metaheuristics is the dynamic balance between exploration and exploitation. Exploration ensures that a large region of the search space is searched such that it does not focus on a specific area. Exploitation identifies areas in the search space with high-quality solutions. With this balance, not much time is taken in searching regions that are either already explored or do not provide high-quality solutions. Some examples of heuristics are Greedy Algorithms. Some examples of metaheuristics are Iterated Local Search [53], Simulated Annealing [47], Tabu Search [41], GRASP [35; 36], Ant Colony Optimization [28; 32], and Evolutionary Algorithms [7].

## 2.3 Constructive-Search Algorithms

A concise review of CO algorithms is presented following the previous definition of their categories. As this work only focuses on constructive search, a suitable selection of constructive-search algorithms is presented. Specifically, an exact algorithm such as Branch & Bound is discussed, along with the following metaheuristics: Iterated Greedy Algorithms, GRASP, Ant Colony Optimization, and Beam Search.

### 2.3.1 Branch & Bound

The *Branch & Bound* (B&B) [20; 49; 51; 52; 63] method is based on the idea of implicitly enumerating all the feasible solutions of CO problem. Such a method works by producing a proof that a solution is optimal, based on successive partitioning of the solution search space. The *branch* part refers to the partitioning process, while the *bound* part refers to the (lower) bounds used to produce a proof of optimality without exhaustive search. Moreover, the Branch & Bound (B&B) process can be visualized as a tree, where the root represents

**Algorithm 2.1:** Branch & Bound

```
1  𝓛 ← {∅} // set of partial solutions, initialized with empty solution
2  U ← ∞ // global upper bound
3  while 𝓛 ≠ {} do
4      select partial solution s^p ∈ 𝓛; 𝓛 ← 𝓛 \ {s^p}
5      // bounding:
6      Φ_lb(s^p) ← determine lower bound for s^p
7      if Φ_lb(s^p) < U then
8          s ← try to determine heuristic solution s ⊇ s^p
9          if ∃s ∧ f(s) < U then
10             s* ← s; U ← f(s) // new best solution
11         end
12     end
13     if Φ_lb(s^p) < U then
14         // branching:
15         produce partial solutions s'^p_1, ..., s'^p_k that represent search space partitions
16         // should cover disjoint nonempty regions of the search space
17         𝓛 ← 𝓛 ∪ {s'^p_1, ..., s'^p_k}
18     end
19 end
```

the original solution search space, and each node of the tree represents a subspace of that region.

The pseudocode of a B&B algorithm is shown in Algorithm 2.1, which provides a detailed overview of how this algorithm works.

### 2.3.2 Iterated Greedy Algorithms

*Iterated Greedy* (IG) algorithms [70] are metaheuristics based on the following idea: at each iteration, the incumbent solution $s$ is partially destroyed and subsequently reconstructed, resulting in a candidate solution $s'$. The destruction process is usually done through a stochastic approach, where a probability distribution is defined over the solution components. In particular, components that offer low usefulness have a higher chance of being *removed* from $s$. After the destruction, the resulting partial solution $s^p$ is subject to a reconstruction process, which produces a new complete solution $s'$ that contains $s^p$. This process can be the same as the one used to construct the initial solution, where components are *added* according to specific criteria. This reconstruction is followed by the application of local search to the new solution $s'$. However, the latter is optional since most of the IG algorithms published in the literature do not use this feature. Finally, the candidate solution $s'$ may or may not be accepted as the new current solution $s$. Concretely, some acceptance criteria that may be applied are: accepting only in the case of improvement, accepting also low deterioration, or a combination of both (similarly to Simulated Annealing [47]).

An advantage of this optimization strategy is its relative simplicity and high computational performance. Besides, as stated by Hoos and Stützle [46], constructing a solution from a partial one allows for a faster construction process and the direct exploitation of desirable components of the solution.

The pseudocode of an IG algorithm is shown in Algorithm 2.2.

**Algorithm 2.2:** Iterated Greedy Algorithm

```
1 s ← ConstructInitialSolution()
2 while stopping criterion not met do
3     s^p ← DestroyPartially(s)
4     s' ← ReconstructSolution(s^p)
5     s' ← LocalSearch(s') // optional
6     s* ← UpdateBestSolution(s')
7     s ← ApplyAcceptanceCriteria(s, s')
8 end
```

### 2.3.3 Greedy Randomized Adaptive Search Procedures

*Greedy Randomized Adaptive Search Procedures* (GRASP) [35; 36] are based on the following idea: at each iteration, a greedy randomized heuristic constructs a solution which subsequently undergoes local search. This constructive heuristic begins with an empty solution $s_0^p = \emptyset$ and *adds* a new component $\ell_j$ at each construction step $i$, selected uniformly at random from a restricted candidate list $\ell$. This list contains the best components available to extend the partial solution $s_i^p$, according to a greedy function. In other words, given the set of components $\mathcal{N}(s_i^p)$ that may lengthen the partial solution $s_i^p$, the best $\alpha$ components according to a function are chosen to belong to a restricted list $\ell$, in a greedy fashion. The list length $\alpha$ determines the strength of the bias introduced in the search strategy. The extreme case of $\alpha = 1$ is equivalent to a greedy heuristic where the best component is added to the partial solution $s_i^p$ at each step $i$. On the other hand, choosing $\alpha = |\mathcal{N}(s_i^p)|$ corresponds to a random solution construction without any heuristic bias. Thus, $\alpha$ is a critical parameter in GRASP that controls the diversity of the solution creation.

After the greedy randomized construction, a local search strategy is applied to the constructed solution $s$. An option for this strategy is typically a basic Hill Climbing.

GRASP is a relatively simple optimization strategy that produces high-quality solutions in a short computation time. For its maximum effectiveness, the construction mechanism should sample promising regions of the search space to obtain good starting points for local search. Fine-tuning the length $\alpha$ of the restricted candidate list $\ell$ is the main contributor to this effectiveness. Aside from that, other mechanisms can be used to randomly select a component that will extend the partial solution $s_i^p$ at a construction step $i$, such as Roulette Wheel Selection or Tournament Selection [54].

The pseudocode of GRASP is shown in Algorithm 2.3.

**Algorithm 2.3:** Greedy Randomized Adaptive Search Procedures (GRASP)

```
1 while stopping criterion not met do
2     s ← ConstructGreedyRandomizedSolution()
3     s ← LocalSearch(s)
4     s* ← UpdateBestSolution(s)
5 end
```

### 2.3.4 Ant Colony Optimization

*Ant Colony Optimization* (ACO) [28; 32] is inspired by the path-finding behavior of natural ant colonies and essentially works as follows. Given the ground set $\mathcal{G}$ of a problem instance,

a range of pheromone values $\vec{\tau}$ is defined by mapping each component $e_j$ in the ground set $\mathcal{G}$ to a pheromone value $\tau_j$. This range of values is commonly known as the *pheromone model* and is essentially a parameterized probabilistic model. In other words, this model is used throughout the construction process of a solution as it affects the stochastic selection of components that will constitute a solution. Moreover, the pheromone model is a pivotal part of any ACO algorithm.

Similarly to GRASP, the construction process in ACO initializes an empty solution $s_0^p = \emptyset$ and *adds* a new component $e_j$ at each construction step $i$, selected randomly from a given set $\mathcal{N}(s_i^p)$. However, there are some significant differences between these two algorithms, enough to distinguish them into two different optimization techniques. One of these differences is that a group of solutions influences the construction of other subsequent solutions in ACO algorithms. The set of these solutions is usually called a *population* $\mathcal{P}$ of "ant trails." These "ant trails" are built independently from one another in the current iteration, i.e. they do not influence each other's construction process. However, they will affect the construction of the "ant trails" in the next iteration. Another difference between GRASP and ACO algorithms is how a component $e_j$ is selected to be added to a partial solution $s_i^p$. In the case of ACO, this selection is randomly made amongst a set of feasible components $\mathcal{N}(s_i^p)$, but those that have better heuristic information $\eta_j$ or pheromone values $\tau_j$ are more likely to be chosen. In other words, a new component $e_j$ is selected based on the linear combination of these two variables, where $\epsilon$ and $\delta$ are coefficients that respectively define the weight of the heuristic information $\vec{\eta}$ and the pheromones $\vec{\tau}$. Thus, the construction process in an ACO algorithm can be summarized as follows. A component $e_j$ is selected at random from a set of feasible components $\mathcal{N}(s_i^p)$ using a probability distribution $P$ based on the linear combination of the heuristic information $\vec{\eta}$ and the pheromone values $\vec{\tau}$. This component is then added to the partial solution $s_i^p$. This process terminates when no further extension can be made to a feasible solution.

After constructing the population $\mathcal{P}$ of "ant trails," some daemon actions may be applied to them. These are usually problem-specific actions that cannot be performed during the ant-based construction. Such an action may be, for example, the application of local search to the constructed solutions. This step is usually optional in ACO algorithms.

Finally, at the end of an iteration, the pheromone values are updated based on the population of "ant trails." Specifically, the pheromone values corresponding to the components that constitute "ant trails" with higher quality will be updated by a higher amount. This process ensures that components in high-quality solutions will have a higher chance of being sampled in the next iterations. Therefore, it is clear that a population of "ant trails" will influence the following construction process by gradually concentrating the search towards high-quality regions of the solution search space. However, pheromones are also evaporated by decreasing their corresponding value at a specific rate. The evaporation process ensures that good components are not selected all the time, promoting the exploration of other parts of the search space.

An advantage of this optimization strategy is that it is guided not only by the heuristic information in each component but also their "historical quality," represented by the pheromones. Moreover, this strategy is robust for various problems and is reported to be capable of achieving near-optimal solutions. Yet, one shortcoming of ACO is that it disregards the linkage among components. In other words, their selection is based on how good a single component is, instead of a how good combination of multiple components may be.

The class of ACO algorithms comprises of several variants. Among the most popular ones are the Ant System (AS) [31], the Ant Colony System (ACS) [29; 30], and the

$\mathcal{MAX} - \mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) [71; 72; 73].

The pseudocode of a generic ACO algorithm is shown in Algorithm 2.4.

**Algorithm 2.4:** Ant Colony Optimization

```
1 while stopping criterion not met do
2     P ← AntBasedSolutionConstruction()
3     P ← DaemonActions(P) // optional
4     s* ← UpdateBestSolution(P)
5     PheromoneUpdate(P)
6 end
```

### 2.3.5 Beam Search

**Algorithm 2.5:** Beam Search

```
1  B ← {∅} // set of partial solutions, initialized with empty solution
2  U ← ∞ // global upper bound
3  while B ≠ {} do
4      B' ← {}
5      foreach s^p ∈ B do
6          if s^p is feasible ∧ f(s^p) < U then
7              s* ← s^p; U ← f(s^p) // new best solution
8          end
9          // branching:
10         E ← select components that may extend s^p; i ← 0
11         while E ≠ {} and i < k_ext do
12             c* ← argmax{g(c | s^p) | c ∈ E}
13             produce (partial) solution s'^p by extending s^p with c*
14             // bounding:
15             Φ_lb(s'^p) ← determine lower bound for s'^p
16             if Φ_lb(s'^p) < U then
17                 B' ← B' ∪ {s'^p}
18             end
19             E ← E \ {c*}; i ← i + 1
20         end
21     end
22     set B as the best min{|B'|, b_width} partial solutions from B' w.r.t. the LB
23 end
```

*Beam Search* (BS) [61] is based on a breath-first search, where a limited number of nodes is expanded at each level of the search tree. In particular, nodes are filtered according to a problem-specific heuristic at each level of the tree. Then, those nodes with better bound values are selected to be expanded at the next level. This algorithm terminates when no nodes can be further expanded. The pseudocode of BS is shown in Algorithm 2.5, which provides a detailed overview of how this algorithm works.

## 2.4 Concluding Remarks

This chapter provided an overview of the concepts related to CO and constructive search. A selection of constructive-search algorithms that solve CO problems was also discussed. Such algorithms need to be implemented in software so they can be applied to concrete optimization problems. Consequently, optimization software is fundamental for the adoption of these algorithms. Thus, it is crucial to survey current optimization software for constructive search to understand what algorithms are supported and how they are implemented.

# Chapter 3

# State-of-the-Art Optimization Software for Constructive Search

Optimization practice is intimately related to the availability of software tools to support it. New problems emerge from a multitude of application areas, which require immediate response. At the time of writing, a broad selection of tools is available to ease the implementation of optimization techniques and modeling problems. Such tools may be frameworks, libraries, modules, or programming languages, among other types. Moreover, it is possible to categorize them into various subclasses following the sort of techniques that can be implemented or the problems that can be modeled. Within one of those subclasses, it is usual that the tools that constitute it are similar to each other, with differences that are not immediately clear to a user. Comparative studies are beneficial for making such a distinction and providing a clear description of the studied tools.

Within the literature, there are various surveys and comparatives studies that catalog and distinguish existing optimization software tools. Some of these tools are used for formulating Combinatorial Optimization (CO) problems or implementing search strategies that find the corresponding optimum solution. Furthermore, the literature focusing on surveying or comparing tools for constructive search may be split into two arbitrary categories. One category mainly focuses on reviewing and comparing Metaheuristic Optimization Frameworks (MOFs), while the other focuses on frameworks for exact approaches, especially Branch & Bound (B&B). This aspect suggests that a common foundation between exact and heuristic constructive-search algorithms is not evident in the literature, such that a tool can implement both approaches.

A number of surveys and comparative studies [19; 25; 39; 65; 75; 76] discuss various MOFs, including some that incorporate a few constructive-search algorithms. In particular, Parejo et al. [65] perform a comprehensive comparative study between ten MOFs based on the features that each one implements. Such an analysis was made by first establishing a set with 271 expected features, which were then evaluated for each framework. The MOFs worth highlighting from this article are FOM [64], MALLBA [3; 4], and OAT [17] since they support constructive-search heuristics such as GRASP and Ant Colony Optimization (ACO). Furthermore, Vieira [75] conducts a survey on MOFs in the literature and also summarizes their features. Besides the ones that were already mentioned, it highlights frameworks such as *Discropt* [66; 67] and MDF [45], which also support constructive-search algorithms (although *Discropt* only supports basic greedy heuristics). Other works [19; 25; 39; 76] present surveys and comparative studies on optimization software frameworks and libraries, but do not mention software that supports constructive search, focusing mainly

on local search.

Furthermore, other surveys and comparative studies [23; 33; 44] discuss frameworks for exact approaches, with a special emphasis on parallel B&B. In particular, Crainic et al. [23] survey various aspects that are pertinent for B&B parallelization. One such aspect is the review of frameworks that help the user in implementing different B&B strategies. Moreover, Herrera et al. [44] perform a comparative study between three implementations of the B&B with varying abstraction levels, where one implementation is based on a B&B framework. Dorta et al. [33] conduct a performance analysis on two B&B skeletons (sequential and parallel) while briefly mentioning other frameworks for B&B. One framework that should be highlighted from these studies is MALLBA, which was previously noted in the context of a MOF. MALLBA also proposes a skeleton for Dynamic Programming (DP) [5]. This framework allows for the implementation of exact and heuristic approaches, along with their hybridization. Another framework for parallel B&B is Bob++ [27; 40], which also features DP. This framework is heavily mentioned in related literature, where it was chosen as a representative of B&B frameworks due to its high use in solving CO problems [6; 26; 58; 59; 60].

Another important survey for this work is the one presented by Basseur et al. [8]. Notably, it discusses the hybridization between exact and heuristic algorithms and their potential for parallelization. On this account, it also surveys some frameworks that would allow for such hybridization. Most of these frameworks were already mentioned, while others fall outside the scope of this work. Furthermore, this article concludes that the state of current frameworks for the support of hybrid algorithms is still underdeveloped. Such a conclusion is also reached by Parejo et al. [65] in regard to MOFs. Lastly, the article states that MALLBA is the only framework that provides skeletons that allow for the hybridization of exact and heuristic algorithms.

The remainder of this chapter, is structured as follows. Section 3.1 reviews a selection of optimization software tools in order to gain insight on their paradigm and limitations. In particular, those that revealed to be more interesting were the following: FOM, OAT, MDF, MALLBA, and Bob++. Then, Section 3.2 provides an overview of the constructive-search algorithms supported by these tools in order to understand which algorithms are prioritized, and which ones are neglected. Lastly, Section 3.3 provides some concluding remarks on the state of current optimization software for constructive search and its implications.

## 3.1 Description of Optimization Software for Constructive Search

### FOM – Framework for Metaheuristic Optimization

FOM [64] is a *framework* implemented in Java. This framework mainly focuses on local search, but it also (is one of the few that) supports GRASP. It attempts to separate the concepts of problem, solution, and neighborhood (but this separation is unclear and conceptually unusual). There is also a lack of documentation (only one article), and the source code is no longer available online. In addition, it is not clear how this framework supports the construction of solutions for GRASP. In particular, there is a class called `GRASPConstructor` that specifies a method named `getInitialSolution()`, which returns an object of class `GRASPSolution`. No specification is given on how a solution is constructed in that method (perhaps the user needs to implement it from scratch).

### OAT – Optimization Algorithm Toolkit

OAT [17] is a *framework* implemented in Java. It implements a variety of metaheuristics, including ACO algorithms. It is unclear how this framework separates concepts such as problem and solution since architecture specifications are not provided by Brownlee et al. [17] (or anywhere else). Additionally, there is a lack of documentation for this framework (only one article), and also the source code is no longer available online. Thus, no pertinent aspects could be drawn about this framework.

### MDF – Meta-heuristics Development Framework

MDF [45; 50] is a *framework* implemented in C++. It supports a few metaheuristics, including ACO (in particular, Ant Colony System (ACS)). This framework makes a separation between objective function, solution representation, and move. However, there is no clear explanation on how solutions are constructed in ACS. Furthermore, not much documentation is available on this framework, and the source code could not also be found.

### MALLBA – MAlaga + La Laguna + BArcelona

MALLBA [3; 4] is a *framework* implemented in C++. It supports both exact and heuristic algorithms, including B&B, DP, and ACO. In addition, it provides a library of skeletons that are independent of the problem itself. It makes a clear separation of concepts such as problem and solution, which are classes that the user implements for a particular problem. These two classes can also be used across all implemented algorithms in the framework. However, those algorithms also require specific classes to be implemented. For example, DP requires that the user implements classes `Stage`, `State`, and `Decision`; B&B requires class `SubProblem`. This reveals a lack of abstraction from the framework. Moreover, the level of documentation is relatively high, and despite not being currently active, it is one of the few constructive-search frameworks whose source code is still accessible [1] (however, this source code does not include B&B and DP).

### Bob++

Bob++ [26; 40] is a *framework* implemented in C++. This framework is mainly directed towards the support of parallel B&B. However, it also claims to support DP. The separation of concepts such as problem, solution, and move is implied in this framework. The level of documentation is also satisfactory, but it does not discuss in detail how a B&B algorithm works. Instead, it focuses more on the parallelization of B&B. In addition, the source code of this framework is inaccessible. A portion of this framework was recovered, but it was poorly documented and unclear how it operated.

## 3.2 Constructive-Search Algorithms Supported by Optimization Software

This section provides a detailed overview of the constructive-search algorithms that are (and are not) supported by the frameworks described in Section 3.1. Such a summary is crucial to understand which of the algorithms presented in Section 2.3 are more popular

and which lack any adoption. It should be noted that the support of a high number of algorithms does not illustrate the (high or low) quality of a framework. In fact, it is possible that algorithms supported by a framework closely resemble each other. The algorithms analyzed in this section have distinguishable features from one another (ACO is one type of algorithm).

| **Algorithm** | **Variant** | **FOM** | **OAT** | **MDF** | **MALLBA** | **Bob++** | **Sum** |
|---|---|---|---|---|---|---|---|
| **B&B** | | | | | ✔ | ✔ | 2 |
| **DP** | | | | | ✔ | ✔ | 2 |
| **IG** | | | | | | | 0 |
| **GRASP** | | ✔ | | | | | 1 |
| **ACO** | AS | ✔ | ✔ | | ✔ | | 3 |
| | EAS | | | | | | 0 |
| | ACS | ✔ | ✔ | ✔ | ✔ | | 4 |
| | $\mathcal{MM}$AS | ✔ | ✔ | | | | 2 |
| | $AS_{rank}$ | | ✔ | | | | 1 |
| | ANTS | | | | | | 0 |
| **BS** | | | | | | | 0 |

Table 3.1: Supported constructive-search algorithms

Table 3.1 shows the constructive-search algorithms that are supported by the previously reviewed frameworks. ACO algorithms are by far those which frameworks offer more support. This is unsurprising since they are relatively popular in the literature. B&B and DP are also two algorithms generally supported in frameworks that implement exact algorithms (B&B mostly, as seen by the amount of optimization software for their parallel implementation). In case of GRASP, this algorithm is only supported by one framework (and it is not clear how the framework supports solution construction). This is also unsurprising since it is often overshadowed by ACO algorithms which typically produce better results. However, the lack of support for Iterated Greedy (IG) algorithms was unusual. These algorithms have a low computational cost as they do not construct each solution from scratch, and they also provide good results. A reason for this lack of support may be that these algorithms only recently started to get adopted in the literature (in contrast to ACO and GRASP, which were already established decades ago). In addition, the frameworks that support constructive search are also quite outdated. Therefore, they were probably no longer active when the wave of popularity of IG algorithms came. Concerning Beam Search (BS), this algorithm is also not supported by any framework. However, since it is a tree search algorithm, it may be possible to extend a framework for B&B to support this algorithm (if the proper tools are provided).

## 3.3 Concluding Remarks

This chapter provided an overview of the available optimization software for constructive search. Most of the available gray/black box optimization software focuses on local search or a combination of local and constructive search. Additionally, those that in-

clude constructive-search algorithms are no longer active and consequently outdated with the current literature. Furthermore, there was no evidence that existing software with constructive-search algorithms had an underlying abstraction for solution construction. The closest was MALLBA, which supports both exact and heuristic approaches. It also allows for the use of the same `Problem` and `Solution` classes across all algorithms (skeletons) such that they are implemented in a problem-independent way. However, a big caveat of this framework is that it often requires other specific classes to be implemented for particular algorithms. Besides decreasing the generalization of a problem model, it also aggravates the implementation effort for a user who intends to use various algorithms.

Therefore, an opportunity arises for the development of an Application Programming Interface (API) for constructive-search. Such an API should separate the problem formulation from the algorithm that solves it by specifying a number of abstract elementary operations that problems must implement and solvers can use in a problem-independent way. Furthermore, it should support both exact and heuristic constructive-search algorithms.

# Chapter 4

# An API for Constructive Search

After reviewing a concise list of optimization software and noting the lack of support for constructive-search algorithms, an Application Programming Interface (API) for constructive search is proposed in this chapter. The development of this API is described to contextualize the decisions made throughout the process. Section 4.1 reviews a conceptual model for constructive search already formalized by other authors. The API is heavily based on this conceptual model. Section 4.2 establishes the requirements that must be met by the API to enable the implementation of well-known constructive-search algorithms. Then, Section 4.3 formally proposes the API for constructive search. Lastly, Section 4.4 discusses to the implications following the proposal of this API for the development and deployment of new and current constructive-search algorithms.

## 4.1 Fundamental Concepts

Some theoretical foundations should be presented and explained before proposing an API for constructive search, particularly the fundamental concepts that characterize the paradigm. This section introduces a conceptual model for this API by reviewing what was already proposed by Vieira [75] and further formalized by Martins and Fonseca [57]. This model supports both exact and heuristic constructive approaches since it grants a way to formulate operations common to both types of approaches. Furthermore, it should be possible to extend this model to include support for local search approaches.

In Section 2.1, it was discussed that a Combinatorial Optimization (CO) problem instance refers to a finite set of components $\mathcal{G}$, which may be present or absent in a feasible solution $s \in \mathcal{S}$ of that problem instance. Such a solution can be tweaked by modifying the presence or absence of *specific* components, resulting in another feasible solution. This idea is fundamentally used in local search procedures. However, this notion can be further extended to constructive search procedures. In the following text, concepts such as construction set, construction graph, and constructive neighborhood are described.

### Construction Set

As mentioned in Section 2.2, a constructive approach works by selecting a component from a set and adding it to a partial solution at each construction step. Such an action can be formulated as deciding which components within the available extensions are present or excluded in an unknown target solution. In particular, the selection of a component *may*

restrict other components from being selected in a subsequent step. Additionally, this idea suggests that the components are initially in an absent state. Thus, one can describe a partial solution by the components it contains, those it cannot contain, and those that are yet to be decided.

The set of elements (solutions) that can be visited during the construction process is defined as the construction set [57]. Such a set includes both partial and complete solutions, feasible and unfeasible. Since every component can be in one of three states, an element in the construction set can be represented by indicator vectors $\mathbf{u} \in \{-1, 0, 1\}^n$. In this case, $n$ is the size of the component set $\mathcal{G}$, and $-1$, $0$, and $1$ correspond to the component states forbidden (or excluded), absent, and present, respectively. Furthermore, such a representation implies that the size of the construction set is at most $3^{|\mathcal{G}|}$.

It is possible to map every indicator vector $\mathbf{u}$ into a more compressed representation. This representation is defined by a set $\mathcal{C}$, which is the union of two disjoint sets $\mathcal{C}^+$ and $\mathcal{C}^-$. Each element $\bar{c_i}$ in set $\mathcal{C}^-$ represents a forbidden component $e_i \in \mathcal{G}$, while each element $c_i$ in set $\mathcal{C}^+$ represents a present component $e_i \in \mathcal{G}$. Based on Martins and Fonseca [57], the general construction set can be defined as follows:

**Definition 4.1 (General Construction Set)** *The general construction set*

$$\mathcal{U} \subseteq \{u \in 2^{\mathcal{C}} : \{c_i, \bar{c_i}\} \not\subseteq u, i = 1, ..., n\}$$

*is the set containing all elements that can be visited during the construction of a feasible solution. Given an element $u \in \mathcal{U}$, a component $e_i \in \mathcal{G}$ is either i. present if $c_i \in u$, ii. forbidden if $\bar{c_i} \in u$, iii. or absent if $c_i \notin u$ and $\bar{c_i} \notin u$. Moreover, a component $e_i \in \mathcal{G}$ cannot be simultaneously present and forbidden, i.e. $\{c_i, \bar{c_i}\} \not\subseteq u$.*

As an illustrative example, let us consider a simple CO problem with three components so that $\mathcal{G} = \{e_1, e_2, e_3\}$. The number of elements that compose the general construction set is $|\mathcal{U}| \leq 3^{|\mathcal{G}|} = 27$. Suppose that the construction set can be represented by the following set of indicator vectors

$$\{(0, 0, 0), (1, 0, -1), (1, 0, 0), (0, -1, 0), (1, -1, 0), (1, 1, -1)\}$$

This set can be mapped into a more compressed representation, resulting in

$$\mathcal{U} = \{\{\}, \{c_1, \bar{c_3}\}, \{c_1\}, \{\bar{c_2}\}, \{c_1, \bar{c_2}\}, \{c_1, c_2, \bar{c_3}\}\}$$

## Construction Graph

As previously mentioned, the general construction set includes all elements that represent partial and complete solutions, feasible and unfeasible. However, no explicit structure is defined among them such that a constructive algorithm may exploit it. Yet, it is possible to derive this by observing the differences between elements in the general construction set, which results in a component-wise structure.

As an illustrative example, let us consider two elements $u = \{\bar{c_2}\}$ and $v = \{c_1, \bar{c_2}\}$, in the previously mentioned construction set $\mathcal{U}$. The difference between these elements is $v - u = c_1 \in \mathcal{C}^+$ and $u - v = \emptyset$. In other words, $u$ and $v$ differ by $c_1$, so that it is possible to transform $u$ into $v$ by *adding* $c_1$. This demonstrates the first binary relation that can

be established over the elements of the general construction set. This relation is denoted by as $A$ and is defined as

$$\forall u, v \in \mathcal{U}, (u, v) \in A \iff (u \subset v) \wedge (|v| = |u| + 1) \wedge (v - u \in \mathcal{C}^{+})$$

Simply put, two elements $u$ and $v$ are in relation $A$ if and only if they differ by one element of $\mathcal{C}^{+}$, which represents the present components.

Now let us consider another two elements $u = \{c_1\}$ and $v = \{c_1, \bar{c_3}\}$ in $\mathcal{U}$. The difference between these elements is $v - u = \bar{c_3} \in \mathcal{C}^{-}$ and $u - v = \emptyset$. This case highlights another binary relation that can be established over the elements of the general construction set. This relation is denoted by $F$ and is defined as

$$\forall u, v \in \mathcal{U}, (u, v) \in F \iff (u \subset v) \wedge (|v| = |u| + 1) \wedge (v - u \in \mathcal{C}^{-})$$

Simply put, two elements $u$ and $v$ are in relation $F$ if and only if they differ by one element of $\mathcal{C}^{-}$, which represents the forbidden components.

It is also possible to derive the inverse of these binary relations $A^{-1}$ and $F^{-1}$, which are defined as follows

$$\forall u, v \in \mathcal{U}, (u, v) \in A^{-1} \iff (u \supset v) \wedge (|v| = |u| - 1) \wedge (u - v \in \mathcal{C}^{+})$$

$$\forall u, v \in \mathcal{U}, (u, v) \in F^{-1} \iff (u \supset v) \wedge (|v| = |u| - 1) \wedge (u - v \in \mathcal{C}^{-})$$

As an example, $u = \{c_1, \bar{c_2}\}$ and $v = \{\bar{c_2}\}$ are in relation $A^{-1}$, while $u = \{c_1, \bar{c_3}\}$ and $v = \{c_1\}$ are in relation $F^{-1}$.

Thus, the elements of the general construction set are structured in a way that they constitute a digraph $G = (\mathcal{U}, E)$. Here, $E$ is the union of all binary relations such that $E = A \cup F \cup A^{-1} \cup F^{-1}$. Each arc $(u, v) \in E$ represents the element $\mathbf{c} \in \mathcal{C}$ by which the two elements $u$ and $v$ of the general construction set differ. Additionally, an arc weight may correspond to the heuristic information $\eta_i$ of a component $e_i \in \mathcal{G}$ that, in turn, is related to an element $\mathbf{c} \in \mathcal{C}$. According to Martins and Fonseca [57], the general construction graph can be defined as follows:

**Definition 4.2 (General Construction Graph)** *"Given a general construction set $\mathcal{U}$ and a set of arcs $E$, $G = (\mathcal{U}, E)$ is a general construction graph."*

Figure 4.1 illustrates the construction graph $G = (\mathcal{U}, E)$ of the previously mentioned subset of the construction set $\mathcal{U}$. It is assumed that each component $e_i \in \mathcal{G}$ of the problem instance possesses heuristic information $\eta_i$, such that $\eta = \{7.0, 8.0, 2.0\}$.

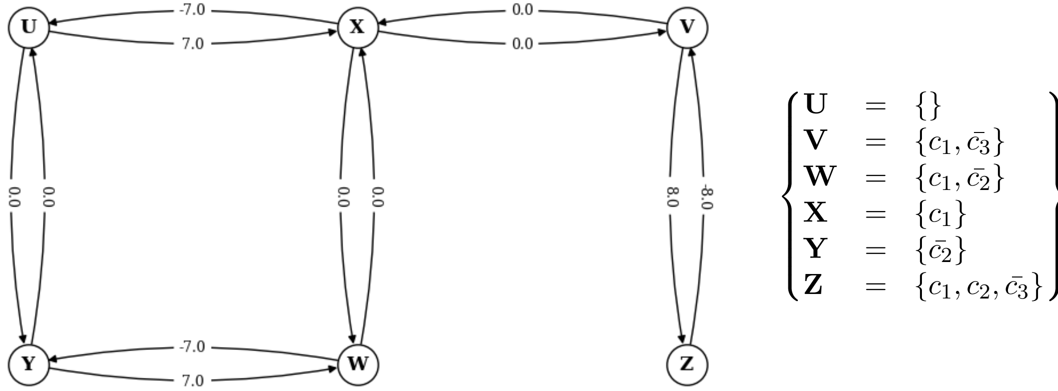



Figure 4.1: Example of a construction graph

## Constructive Neighborhood

As stated earlier, a general construction graph $G = (\mathcal{U}, E)$ imposes a structure over the construction set. It is possible to derive a neighborhood structure from this graph such that two vertices $u, v \in \mathcal{U}$ are neighbors if there is an arc connecting them, in a way that $(u, v) \in E$. The general constructive neighborhood $N(u)$ for an element $u \in \mathcal{U}$ can be defined as follows [57]:

**Definition 4.3 (General Constructive Neighborhood)** *Let $G = (\mathcal{U}, E)$ be the general construction graph $G = (\mathcal{U}, E)$ and $u \in \mathcal{U}$. The general constructive neighborhood $N(u)$ contains all elements $v \in \mathcal{U}$ within a geodesic distance[1] $d(u, v) \leq 1$. Therefore, for $\forall u \in \mathcal{U}$, there is a constructive neighborhood $N(u)$ such that*

$$N(u) \coloneqq u \cup N_A(u) \cup N_F(u) \cup N_{A^{-1}}(u) \cup N_{F^{-1}}(u)$$

In other words, the general constructive neighborhood $N(u)$ is the union of element $u$ itself and the constructive subneighborhoods $N_R(u)$ imposed by the binary relations $R = A, F, A^{-1}, F^{-1}$, such that $N_R(u) \coloneqq \{v \in \mathcal{U} : (u, v) \in R\}$. Furthermore, each subneighborhood is imposed by exactly one binary relation. A nomenclature may be attributed to each subneighborhood: $N_A$ will be called `ADD`, $N_F$ will be called `FORBID`, $N_{A^{-1}}$ will be called `REMOVE`, and $N_{F^{-1}}$ will be called `PERMIT`.

As an example of the previous concepts, let us consider vertex $\mathbf{X}$ in the digraph of Figure 4.1. In this case, the constructive neighborhood $N_R(\mathbf{X})$ is defined as $N_R(\mathbf{X}) = \{\mathbf{U}, \mathbf{V}, \mathbf{W}\}$. Furthermore, each subneighborhood can be defined such that $N_A = \{\}$, $N_F = \{\mathbf{V}, \mathbf{W}\}$, $N_{A^{-1}} = \{\mathbf{U}\}$, and $N_{F^{-1}} = \{\}$.

According to the structure imposed by the construction graph $G = (\mathcal{U}, E)$, two vertices $u, v \in \mathcal{U}$ in which a component $e_i \in \mathcal{G}$ is respectively present and forbidden cannot be neighbors. That is

$$\forall u, v \in \mathcal{U}, c_i \in u \wedge \bar{c_i} \in v \Rightarrow v \notin N(u) \wedge u \notin N(v)$$

Thus, the state of a component $e_i \in \mathcal{G}$ must be reset through its inverse operation ($A^{-1}$ or $F^{-1}$) before being set again.

[1] The number of edges in the shortest path that connects two vertices

Another aspect related to the general construction graph $G = (\mathcal{U}, E)$ is that it is strongly connected, meaning that every vertex $v_j \in \mathcal{U}$ is reachable from every other vertex $v_i \in \mathcal{U}$. This implies that there is always a directed path $v_i \rightarrow v_j$ connecting any two vertices $v_i, v_j \in \mathcal{U}$, such that

$$v_k \in N(v_{k-1}),\ k = i{+}1, ..., j$$

Such a path can be characterized by the arcs it contains. For example, if one consists of only arcs of $A$, then it is said that $v_j$ is $A$-reachable from $v_i$.

Ultimately, the general construction graph can represent any instance of any CO problem. However, it may be exponentially large (size up to $3^{|\mathcal{G}|}$). Thus, the constructive graph can only implicitly be defined by the operators that implement the constructive search. The same issue arises in local search as well.

## 4.2 Requirements

The following set of functional requirements for an API for constructive search were identified through the analysis of the constructive-search algorithms reviewed in Section 2.3 in the light of the conceptual model presented in Section 4.1.

### Functional Requirements

The functional requirements for the API for constructive search can be identified.

**FR1** Support the separation of the concepts of problem, solution, and component.

**FR2** Support the instantiation of a CO problem.

**FR3** Support the definition of solutions as subsets from a ground set of a problem instance.

**FR4** Allow components to be in one of three states during the construction process: *present*, *absent*, and *forbidden*.

**FR5** Support the initialization of an empty solution as a starting point for constructive search.

**FR6** Support the generation of a feasible solution from a (partial) solution with respect to a given problem-specific heuristic. This operation is used, for example, in Iterated Greedy Algorithms to obtain an initial solution and reconstruct those partially destroyed.

**FR7** Support the enumeration of (all) components that may be added/removed to/from a (partial) solution. This operation allows for those components to be efficiently enumerated from a computational perspective.

**FR8** Support the selection of components that may be added to a partial solution according to a *branching rule*. This operation is used in Tree Search Algorithms such as Branch & Bound.

**FR9** Support the heuristic enumeration of components that may be added/removed to/from a (partial) solution for cases where there is a large number of such components, and it is possible to sort them with respect to available heuristic information. This operation allows for those components to be efficiently enumerated in an appropriate order.

**FR10** Support the uniform random sampling with replacement of components that may be added/removed to/from a (partial) solution for cases where there is a large number of such components, and it is not efficiently possible to enumerate them according to a given heuristic. Furthermore, uniform random sampling without replacement should also be supported if all components are to be potentially sampled.

**FR11** Support the addition/removal of a component to/from a (partial) solution.

**FR12** Support the forbidding/permitting of a component in a (partial) solution. This operation is used in Tree Search Algorithms such as Branch & Bound.

**FR13** Check whether a solution is partial or complete in order to determine when to stop the construction process. In other words, the API must support operations that check whether or not a solution can be further extended.

**FR14** Check whether a solution is feasible or unfeasible.

**FR15** Check whether a solution dominates another with respect to a problem-specific dominance relation.

**FR16** Support full and/or incremental (feasible) solution evaluation.

**FR17** Support full and/or incremental (lower and upper) bound evaluation of a (partial) solution.

**FR18** Support heuristic information access given a component.

**FR19** Support pheromone access and/or update with respect to a component, which may or may not belong to a solution. This operation is used in Ant Colony Optimization.

### Non-Functional Requirements

In addition, the following non-functional requirements for the API for constructive search are identified.

**NFR1** The API for constructive search should allow efficient implementation of problems and algorithms, and introduce low overhead.

**NFR2** A user who is familiar with the principles and ideas for the formulation of a problem as a constructive-search problem should be able to easily understand and use the API.

## 4.3 Proposed API for Constructive Search

This API will extend an already existing local-search API called *nasf4nio* [38]. The design of that API focuses on separating the problem formulation from the algorithm that solves it (commonly referred to as *solver*) by specifying a number of abstract elementary operations that problems must implement and solvers can use in a problem-independent way. In particular, solvers can interact with the problem *exclusively* by calling a number of prespecified functions implemented by the model of the problem. Currently, the *nasf4nio* code base includes Stochastic Hill Climbing, Iterated Local Search (ILS), and Genetic Algorithms. However, other local-search algorithms, such as Simulated Annealing (SA), are also supported by this API. The new API proposed in this work aims to provide support for constructive search approaches by adopting some of the principles behind *nasf4nio*.

Furthermore, two levels are specified for the proposed API. The first level contains a subset of operations that are required by (most) metaheuristics to solve an instance of a problem. The second contains all operations specified by the API. The difference between these two levels is that the second level can perform operations such as forbidding and permitting a component in a solution. These operations are required for the partitioning of the search space in exact algorithms such as Branch & Bound (B&B), but may impose a large overhead. This overhead is propagated to all solvers that use the model implemented at the second level, including those that can use first-level models. Thus, a model should be implemented on the first level unless B&B or other algorithms that partition the search space are applied. Another reason for the definition of these two levels is that the second level may add complexity to the implementation of a model, which may only be justified if solvers require it.

The definition of the data structures and functions provided by the API for constructive search are presented as follows.

### Enumerations

- `enum ComponentState { PRESENT, ABSENT, FORBIDDEN }`
- `enum SubNeighbourhood { ADD, REMOVE, FORBID, PERMIT }`

### Data Structures

- `struct problem {...}`
  Store data that fully characterizes a particular instance of a problem. This data must be available in advance, and is not changed by the solver in any way.
- `struct solution {...}`
  Store data that fully characterizes a (possibly partial) solution to a given problem instance.
- `struct component {...}`
  Store data that characterizes a component from the ground set of a given problem instance.

### Problem Instantiation

- `struct problem *newProblem() {...}`
  Allocate a problem structure and initialize it. Being problem-specific, the function arguments are deliberately left unspecified. The function returns `NULL` if instantiation fails.

### Problem Inspection

- `long getNumComponents(struct problem *p) {...}`
  Return the size of the ground set of a problem instance.
- `long getMaxSolutionSize(struct problem *p) {...}`
  Return the largest number of components that a solution can potentially have present.

- `long getMaxNeighbourhoodSize(struct problem *p, const enum SubNeighbourhood nh) {...}`
  Return the largest possible number of direct neighbors in a given subneighborhood.

### Memory Management

- `void freeProblem(struct problem *p) {...}`
  Free all memory used by a problem structure.
- `struct solution *allocSolution(const struct problem *p) {...}`
  Allocate memory for a solution structure. The function returns `NULL` if allocation fails.
- `void freeSolution(struct solution *s) {...}`
  Free all memory used by a solution structure.
- `struct component *allocComponent(const struct problem *p) {...}`
  Allocate memory for a component structure. The function returns `NULL` if allocation fails.
- `void freeComponent(struct component *c) {...}`
  Free all memory used by a component structure.

### Reporting

- `void printProblem(const struct problem *p) {...}`
  Print a user-formatted representation of a problem instance.
- `void printSolution(const struct solution *s) {...}`
  Print a user-formatted representation of a solution.
- `void printComponent(const struct component *c) {...}`
  Print a user-formatted representation of a component.

### Solution Generation

- `struct solution *emptySolution(struct solution *s) {...}`
  Initialize a solution structure as an empty solution.
- `struct solution *heuristicSolution(struct solution *s) {...}`
  Heuristically construct a feasible solution, preserving all present and forbidden components in a given solution and modifying the solution in place. Calling this function with the same given solution multiple times may generate different heuristic solutions. The function returns `NULL` if no feasible solution is found, in which case the original input is lost.

**Remark:** `heuristicSolution(...)` preserves all present and forbidden components in a given (partial) solution so that it is possible to compute the upper bound of that (partial) solution. That is, the objective value of a heuristic solution is always greater than or equal to the objective value of the best solution that the given (partial) solution can reach (in minimization problems).
Furthermore, this function may generate different solutions from the same input

if a randomized heuristic is used to construct a heuristic solution.
Lastly, heuristically constructing a solution may result in not finding a feasible solution. However, that does not imply that a given (partial) solution cannot reach a feasible solution; it only indicates that the heuristic was unable reach one.

### Solution Inspection

- `double *getObjectiveLB(double *objLB, struct solution *s) {...}`
  Single or multiple objective full and/or incremental lower bound evaluation. The lower bound of a solution must be less than or equal to the lower bound of another solution if the sets of present and forbidden components of the former are subsets of the corresponding sets of the latter. The function returns its first input argument, which may have been modified (e.g. to store the computed bound values).

**Remark:** Section 2.2 mentions that the lower bound of a partial solution is less than or equal to the objective value of each feasible solution which extends that partial solution. Without loss of generality, it can also be said that the lower bound of a partial solution should be less than or equal to the lower bound of a (partial) solution which extends the latter. If the sets of present and forbidden components of a solution are subsets of the corresponding sets of another solution, then it can be said that the latter is an extension of the former. Thus, the lower bound of the former should be less than or equal to the lower bound of the latter.

- `double *getObjectiveVector(double *objv, struct solution *s) {...}`
  Single or multiple objective full and/or incremental solution evaluation. The function updates and returns its first input argument if a given solution is feasible or `NULL` if it is unfeasible, in which case the content of the first argument is unspecified (in particular, it may have been modified to contain temporary values).

**Remark:** Section 2.1 defines that an instance of an optimization problem is characterized by the set of *feasible* solutions $\mathcal{S}$ and the objective function $f$ that measures their quality. `getObjectiveVector(...)` supports solution evaluation based on an objective function. Since the objective function *exclusively* measures the quality of feasible solutions, `getObjectiveVector(...)` *only* evaluates feasible solutions.

- `int isFeasible(struct solution *s) {...}`
  Return `1` (true) if a given solution is feasible or `0` (false) if it is unfeasible.

- `long getNumSolutionComponents(struct solution *s, const enum ComponentState st) {...}`
  Return the number of components of a solution that are in a given state.

- `long getNeighbourhoodSize(struct solution *s, const enum SubNeighbourhood nh) {...}`
  Return the number of direct neighbours in a given subneighborhood of a solution (not counting the solution itself).

- `long enumSolutionComponents(struct solution *s, const enum ComponentState st) {...}`
  Enumerate the components of a solution that are in a given state, in unspecified order. The function returns a unique component identifier in the range $0..|\mathcal{G}|-1$ if a new component is enumerated or `-1` if there are no components left.

- `struct solution *resetEnumSolutionComponents(struct solution *s, const enum ComponentState st) {...}`
  Reset the enumeration of the components of a solution that are in a given state, so that the next call to `enumSolutionComponents()` will start the enumeration of the components in that state from the beginning.

**Remark:** `enumSolutionComponents(...)` is intended to support pheromone access in Ant Colony Optimization (ACO) and related algorithms.

### Operations on Solutions

- `struct solution *copySolution(struct solution *dest, const struct solution *src) {...}`
  Copy the contents of the second argument to the first argument. The copied solution is functionally indistinguishable from the original solution.

- `struct solution *applyMove(struct solution *s, const struct component *c, const enum SubNeighbourhood nh) {...}`
  Modify a solution in place by applying a move to it. It is assumed that the move was generated for, and possibly evaluated with respect to, that particular solution and the given subneighborhood. In addition, once a move is applied to a solution, it can be reverted by applying it again with the opposite subneighborhood. For example, after an `ADD` move generated for a given solution is applied to that solution, it may be applied again as a `REMOVE` move to the resulting solution in order to recover the original solution. The result of applying a move to a solution in any other circumstances is undefined.

### Component/Move Generation

- `struct component *enumMove(struct component *c, struct solution *s, const enum SubNeighbourhood nh) {...}`
  Enumeration of a given subneighborhood of a solution, in an unspecified order. This is intended to support fast neighborhood exploration and evaluation with `getObjectiveLBIncrement()`, particularly when a large part of the neighborhood is to be visited. The function returns `NULL` if there are no moves left to enumerate.

- `struct solution *resetEnumMove(struct solution *s, const enum SubNeighbourhood nh) {...}`
  Reset the enumeration of a given subneighborhood of a solution, so that the next call to `enumMove()` will start the enumeration of that subneighborhood from the beginning.

**Remark:** The order of move generation in `enumMove(...)` is unspecified. Thus, this function should be implemented so that moves are generated and possibly evaluated, in the most favorable order from a computational perspective.

- `struct component *heuristicMove(struct component *c, struct solution *s, const enum SubNeighbourhood nh) {...}`
  Generate a heuristic move from a given subneighborhood of a solution. This may be an empty set for some subneighborhoods, in which case the function returns `NULL`. Calling this function with the same arguments multiple times may generate different moves. Furthermore, heuristic moves may or may not be greedy with respect to how they affect the lower bound.

- `struct component *heuristicMoveWOR(struct component *c, struct solution *s, const enum SubNeighbourhood nh) {...}`
  Heuristic enumeration of a given subneighborhood of a solution, without replacement. Heuristic moves may or may not be generated in any particular order with respect to how they affect the lower bound. The function returns `NULL` if there are no moves left to enumerate.

- `struct solution *resetHeuristicMoveWOR(struct solution *s, const enum SubNeighbourhood nh) {...}`
  Reset the heuristic enumeration without replacement of a given subneighborhood of a solution, so that the next call to `heuristicMoveWOR()` will start the heuristic enumeration from the beginning. The order of move generation may not be preserved by such a reset.

**Remark:** `heuristicMove(...)` may generate different moves for the same input if a randomized heuristic is used to sample a heuristic move. A similar argument is applied to `resetHeuristicMoveWOR(...)`, which may not preserve the order of move generation after resetting the heuristic enumeration without replacement. Furthermore, heuristic move enumeration may or may not be greedy with respect to how moves affect the lower bound of a (partial) solution (if the move is applied) due to how costly the computation of a lower bound can be, in contrast to a heuristic that is simpler and more efficient (therefore allowing for faster move enumeration).

- `struct component *randomMove(struct component *c, struct solution *s, const enum SubNeighbourhood nh) {...}`
  Uniform random sampling of a given subneighborhood of a solution, with replacement. This may be an empty set for some subneighborhoods, in which case the function returns `NULL`.

- `struct component *randomMoveWOR(struct component *c, struct solution *s, const enum SubNeighbourhood nh) {...}`
  Uniform random sampling of a given subneighborhood of a solution, without replacement. The function returns `NULL` if there are no moves left to sample.

- `struct solution *resetRandomMoveWOR(struct solution *s, const enum SubNeighbourhood nh)`
  Reset the uniform random sampling without replacement of a given subneighborhood of a solution, so that any move corresponding to that subneighborhood can be generated by the next call to `randomMoveWOR()`.

**Remark:** In component/move generation functions, results may be cached in the `solution` structure itself in order to speed up the generation of future moves. In particular, a `solution` may store data related to invalid moves (for example,

a forbidden component) to avoid them from being sampled again in future function calls.

### Component/Move Inspection

- `long getComponentID(const struct component *c) {...}`
  Return the unique identifier in the range $0..|\mathcal{G}|-1$ with respect to a given component.

**Remark:** `getComponentFromID(...)` is intended to support pheromone access in ACO and related algorithms.

- `double *getObjectiveLBIncrement(double *obji, struct component *c, struct solution *s, const enum SubNeighbourhood nh) {...}`
  Single or multiple objective move evaluation with respect to the solution and subneighborhood for which the move was generated, before it is actually applied to that solution (if it ever is). The result of evaluating a move with respect to a solution and subneighborhood other than those for which it was generated (or to a pristine copy of that solution and the same neighborhood) is undefined. The function returns its first input argument, which was modified to contain the lower bound increments.

### Operations on Components/Moves

- `struct component *copyComponent(struct component *dest, const struct component *src) {...}`
  Copy the contents of the second argument to the first argument. The copied component is functionally indistinguishable from the original component.

## 4.4 Concluding Remarks

An API for constructive search was proposed in this chapter. This API will allow for the modeling and implementation of optimization problems as constructive-search problems and the implementation of constructive-search algorithms. The non-functional requirements identified for this API also need to be measured in order to understand to what extent they are satisfied.

Furthermore, the proposed API also allows for the development of new constructive-search algorithms. In particular, it enables the implementation effort to be focused on improving problem models based on the API instead of a particular algorithm.

# Chapter 5

# Evaluation of Constructive Search API

This chapter demonstrates the expressiveness of the Application Programming Interface (API) for constructive search proposed in Chapter 4. It also discusses the computational overhead introduced by the use of such an API (efficiency) and reports on the feedback provided by users who were unfamiliar with it (usability). Furthermore, the implications of the underlying abstraction for the development of novel constructive-search algorithms are also discussed in this chapter.

Section 5.1 describes how Combinatorial Optimization (CO) problems can be modeled as constructive-search problems based on the API. Section 5.2 describes the implementation of constructive-search algorithms based on the API. Section 5.3 reports on the results produced by various experiments concerning the measurement of the computational overhead introduced by the use of the API and the study of the performance of solvers that use improved problem models. Section 5.4 presents the feedback provided by users after interacting with the API. Lastly, Section 5.5 provides some concluding remarks.

## 5.1 Formulation of Optimization Problems

As mentioned in Section 4.1, it is possible to apply the conceptual model to any CO problem. Since the API is heavily based on this model, it is presumed that it can also be used to formulate any CO problem as a constructive-search problem. However, concrete examples need to be implemented to validate this criterion. This section describes the modeling of the Knapsack Problem (KP), the Travelling Salesman Problem (TSP), and the Cable Trench Problem (CTP) as constructive-search problems, and provides an overview of their implementations based on the API. In addition, other CO problems such as the Multidimensional Knapsack Problem (MKP), the Quadratic Assignment Problem (QAP), the Quadratic Multidimensional Knapsack Problem (QMKP), and the 2019 Hash Code Photo Slideshow Problem [2] were formulated and implemented according to the API for constructive search.

It should be mentioned that the ** symbol denotes the cases and operations which only occur on the second level of the API for constructive search.

### 5.1.1 Knapsack Problem

The KP [78, p. 270–271] can be described as follows: "A thief takes a knapsack with limited capacity to a department store with various items and intends to profit from the items that (still) fit in the knapsack. What is the optimal selection of items such that the profit is maximum?" Formally, this problem can be formulated as an integer linear program as follows:

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

where $p_i$ and $w_i$ denote the profit and weight of item $i$, respectively, and $W$ denotes the capacity of the knapsack. $x_i = 1$ means that item $i$ was selected and $x_i = 0$ means that item $i$ was not selected.

The relevant aspects to be considered when implementing the KP according to the proposed API are presented next.

**Problem Instance** An instance of this problem is characterized by the profit and weight values of each item, and by the capacity of the knapsack, $W$. This information is usually stored in two arrays of size $N$, where $N$ denotes the number of items of the instance of interest, and a scalar variable.

**Solution Representation** A solution to an instance of this problem is characterized by a list of items that should be selected. Furthermore, a (possibly partial) solution can be represented by a list of $k$ distinct integers in the range $0..N-1$, representing the indices of the items to be selected, provided that the sum of their weights is not greater than $W$. A solution is always feasible if the former constraints are met, regardless of whether it is partial or complete.

**Component Representation** In this problem, an item represents a component from the ground set of an instance. This information is stored as an integer that uniquely identifies an item. Furthermore, the size of ground set is equal to the number of items of an instance.

**Empty Solution Generation** An empty solution is initialized with an empty list (of items).

**Heuristic Solution Generation** A heuristic solution can be constructed with a greedy heuristic that iteratively adds the item with the highest profit density (if it fits in the knapsack) until no items can be added. The density of an item $i$ is the ratio between the profit $p_i$ and weight $w_i$ of that item. Algorithm 5.1 shows the pseudocode of such a greedy heuristic, which also handles situations where a component might be present or forbidden**.

**Algorithm 5.1:** Greedy heuristic for the Knapsack

```
   input: partial solution s^p
1  E ← select all items of the instance
2  while E ≠ {} do
3      c* ← argmax{ p_c / w_c | c ∈ E}
4      if (Σ_{i=1}^{k} w_i) + w_{c*} ≤ W and c* is not present and c* is not forbidden** then
5          s^p ← s^p ∪ {c*}; k ← k + 1
6      end
7      E ← E \ {c*}
8  end
9  return s^p
```

**Solution Evaluation** In this problem, the performance of a feasible solution is measured by the sum of the profit of all selected items. This value is to be maximized.

**Bound Evaluation** The (upper) bound of a (partial) solution can be computed according to different methods, each of which has its relative strength (how close it is to the optimum value):

**Weak** The sum of the profit of all selected items and all the non-selected items that fit in the knapsack.

**Strong** The sum of the profit of all selected items, the (non-selected) items with the highest density (iteratively add them until the next does not fit), and a portion of the next item that does not fit in the knapsack (such that the partial profit is proportionate to the remaining capacity). Algorithm 5.2 shows the pseudocode of this (greedy) upper bound, which also handles situations where a component might be present or forbidden**.

It should be noted that forbidden** components cannot be used in the upper bound computation.

**Algorithm 5.2:** Upper bound for the Knapsack

```
   input: partial solution s^p
1  E ← select all items of the instance
2  U ← Σ_{i=1}^{k} p_i // upper bound
3  B ← W − (Σ_{i=1}^{k} w_i) // remaining capacity of the knapsack
4  while E ≠ {} and B > 0 do
5      c* ← argmax{ p_c / w_c | c ∈ E}
6      if B ≥ w_{c*} and c* is not present and c* is not forbidden** then
7          U ← U + p_{c*}; B ← B − w_{c*}
8      else if B < w_{c*} and c* is not present and c* is not forbidden** then
9          U ← U + (B / w_{c*}) × p_{c*}; B ← 0
10     end
11     E ← E \ {c*}
12 end
13 return U
```

**Solution Modification** A valid move that can be applied to a solution relies on what type of action that move performs. In other words, it depends for what type of

subneighborhood a component (represented in the move) was sampled. In the context of this problem, the valid moves for each subneighborhood are presented as follows.

`ADD` Any item that fits in the knapsack (as long as it is not forbidden**). This item is added to the list of those that are selected.

`REMOVE` Any selected item. This item is removed from the list of those that are selected.

`FORBID`** Any non-selected item. This item is added to a list of those that are excluded.

`PERMIT`** Any forbidden item. This item is removed from a list of those that are excluded.

It should be noted that the list of items that are excluded may grow to the size of the ground set. In this problem, this matter is not an issue since it is common to store the data of all items of an instance. However, in other problems, this can be an issue.

**Move Enumeration** Iteratively enumerate the items in a given (constructive) subneighborhood in the same order that those items are stored in the component list (or some other list).

**Heuristic Move Generation** Heuristically select the item in a given (constructive) subneighborhood in a way that favors those that have a higher density, if they are to be added, or those that have a lower density, if they are to be removed.

**Random Move Generation** Sample an item uniformly at random from the list of items that belong to the given (constructive) subneighborhood.

**Move Evaluation** A move is evaluated with respect to how it affects the upper bound of a (partial) solution. If an item was (entirely) used in the upper bound computation of the original (partial) solution, then adding it will not affect the upper bound. In general, the upper bound may need to be recomputed for the case where the move is applied. Fortunately, it is possible to efficiently recompute it by focusing only on the incremental changes done in the bound computation.

### 5.1.2 Travelling Salesman Problem

The TSP [78, p. 276–278] can be described as follows: "A salesman, who lives in a city, intends to visit other cities to peddle some of his goods. He wishes to visit each city only once and then return to his home city at the end of the tour. What is the optimal route such that the traveled distance is minimum?" The Symmetric TSP variant is considered in the remainder of this chapter. It assumes that the distance between two cities is the same, independently of direction. Formally, this problem can be formulated as an integer linear program as follows [24]:

$$
\begin{aligned}
& f(x) = \min \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{i=1, i \neq j}^{n} x_{ij} = 1 && j = 1, 2, \ldots, n \quad \texttt{// come from one city} \\
& \sum_{j=1, j \neq i}^{n} x_{ij} = 1 && i = 1, 2, \ldots, n \quad \texttt{// go to one city} \\
& \sum_{i \in S}^{n} \sum_{j \in S, j \neq i}^{n} x_{ij} \leq |S| - 1 && \forall S \subsetneq V, 2 \leq |S| \leq n - 2 \quad \texttt{// no subtours} \\
& x_{ij} \in \{0, 1\}, \forall (i, j) \in E
\end{aligned}
$$

where $c_{ij}$ denotes the distance from city $i$ to city $j$, $V$ denotes the set of vertices (cities), and $E$ denotes the set of arcs. $x_{ij} = 1$ means that city $j$ is visited immediately after city $i$, and $x_{ij} = 0$ means otherwise.

The formulation of the TSP as a constructive-search problem, and its implementation, should consider the following:

**Problem Instance** An instance of this problem is characterized by the distances from each city to every other city. This information is usually stored in an $N \times N$ distance matrix, where $N$ denotes the number of cities of an instance.

**Solution Representation** A solution to an instance of this problem is characterized by a list of cities in the order that they should be visited. Furthermore, a (possibly partial) solution can be represented by a sequence of $k$ distinct integers in the range $0..N-1$, such that $0 \leq k \leq N$. A solution is feasible (and complete) if all cities are visited exactly *once*, and the tour ends at the home city.

**Component Representation** In this problem, an arc connecting two cities represents a component from the ground set of an instance. This information is usually stored as a pair of integers, where each integer represents a city. Furthermore, the size of the ground set is equal to the number of arcs of an instance, which is

$$\frac{N \times (N-1)}{2}$$

**Empty Solution Generation** An empty solution is initialized with only the home city in the sequence (of cities). Figure 5.1 illustrates an empty solution of an instance of this problem.

**Heuristic Solution Generation** A heuristic solution can be constructed using a greedy heuristic such as the Multiple-Fragment [11]. This heuristic iteratively adds the arc with minimal cost (if it does not produce a subcycle, nor does it increase the degree of any city beyond two) until a (feasible) tour is obtained. Figure 5.2 illustrates the heuristic construction of a (feasible) solution from a given (partial) solution with present and forbidden** components.

**Solution Evaluation** In this problem, the performance of a feasible solution is measured by the length of a (feasible) tour (which is the sum of the cost of the arcs that constitute it). This value is to be minimized.

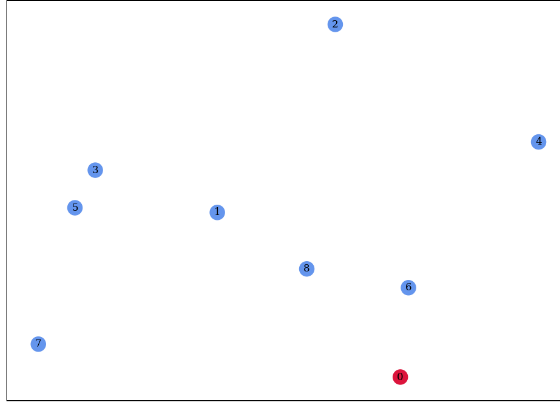



Figure 5.1: Illustration of an empty solution for the Travelling Salesman

**Bound Evaluation** The (lower) bound of a (partial) solution can be computed according to different methods, each of which has its relative strength. The following are well known in the literature.

**Weak** The length of the (partially constructed) tour (which is the sum of the cost of the arcs that contain in it).

**Intermediate** The length of the (partially constructed) tour and the $N-k$ shortest arcs that are not included that tour, where $N$ denotes the number of cities in an instance, and $k$ denotes the number of arcs in the (partial) tour. Figure 5.3 illustrates the (intermediate) lower bound evaluation of a given (partial) solution with present and forbidden** components, where its value is 337.0.

**Strong** A lower bound [42] based on the 1-tree bound [21]: the length of the (partially constructed) tour, the Minimum Spanning Tree for the graph consisting of not visited cities, and the two shortest arcs that connect the home city and the last (visited) city to not visited cities (which must be different). Figure 5.4 illustrates the (strong) lower bound evaluation of a given (partial) solution with present and forbidden** components, where its value is 357.9.

It should be noted that forbidden** components cannot be used in the lower bound computation.

**Solution Modification** A valid move that can be applied to a solution depends on what type of action that move performs. In other words, it depends for what type of subneighborhood a component (represented in the move) was sampled. In the context of this problem, the valid moves for each subneighborhood are as follows.

`ADD` Any arc that connects the last (visited) city to a not visited city. A new city is appended to the sequence of cities to be visited.

`REMOVE` The last arc added to the (partially constructed) tour. The last (visited) city is dropped from the sequence of cities to be visited.

`FORBID`** Any arc not included in the (partially constructed) tour. This arc is added to a list of arcs that are excluded.

`PERMIT`** Any forbidden arc. This arc is removed from the list of arcs that are excluded.

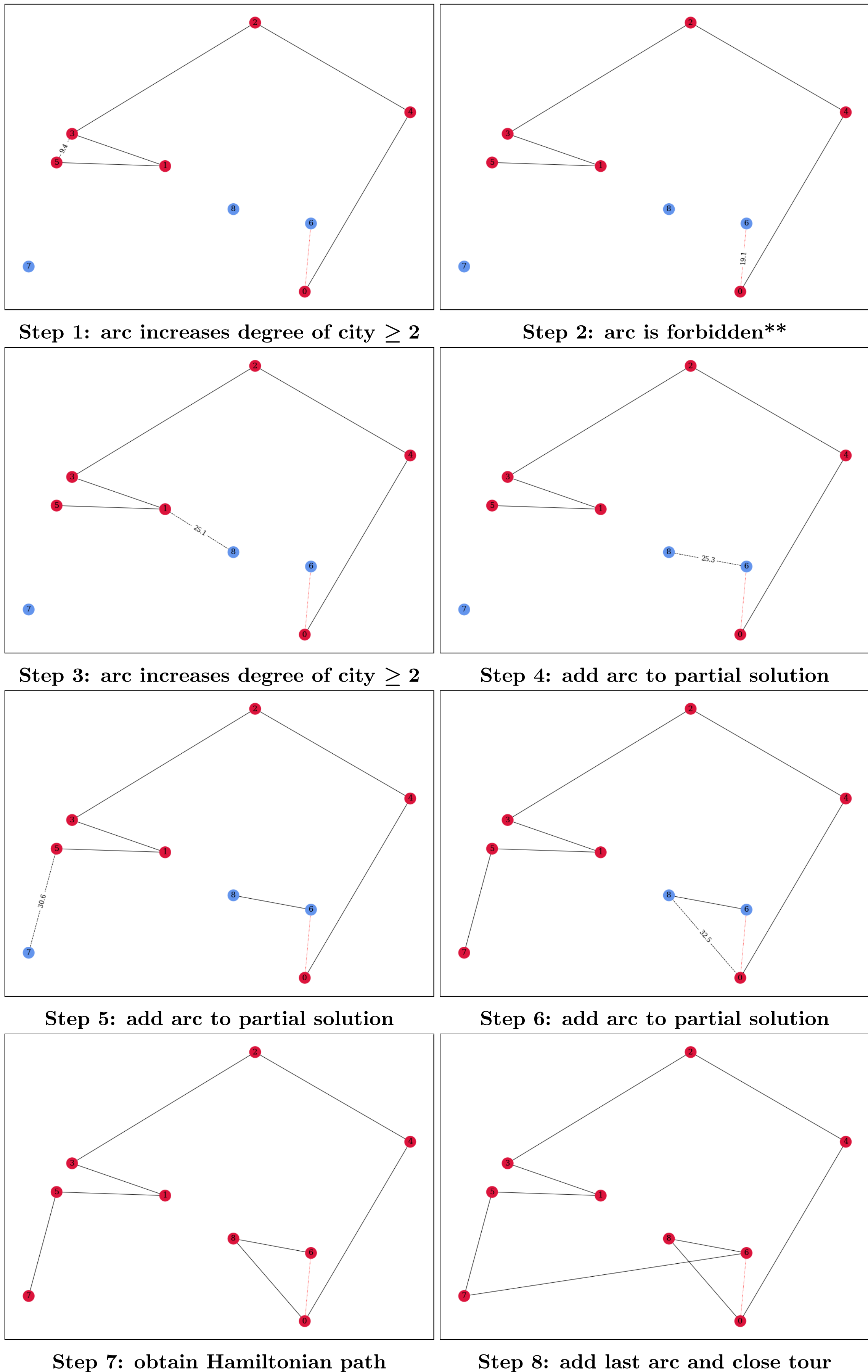



Figure 5.2: Example of construction with multiple-fragment heuristic

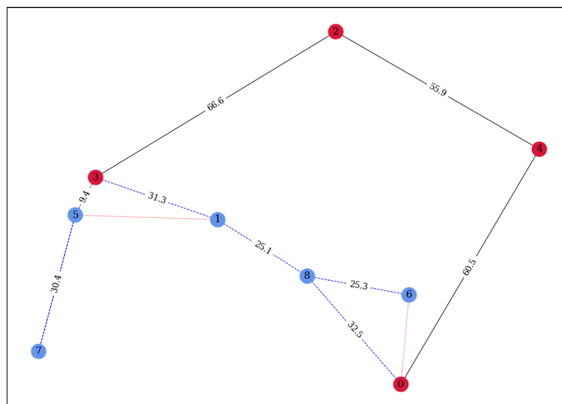



Figure 5.3: Example of intermediate lower bound for the Travelling Salesman

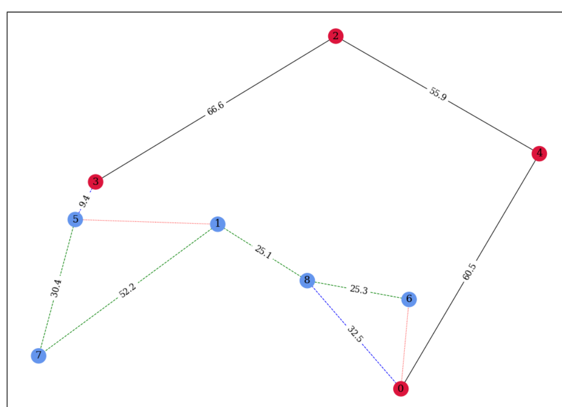



Figure 5.4: Example of strong lower bound for the Travelling Salesman

It should be noted that the list of arcs that are excluded may grow to the size of the ground set. This is an issue for this problem since the ground set has a quadratic size with respect to the size of the instance. Consequently, this can cause high memory usage for large instances. This is one of the reasons why the API for constructive search should support two levels, where one does not require this list to be implemented.

**Move Enumeration** Iteratively enumerate the arcs in a given (constructive) subneighborhood in the same order that those arcs are (explicitly or implicitly[1]) stored.

**Heuristic Move Generation** Heuristically select the arcs in a given (constructive) subneighborhood in a way that favors those that are shorter, if they can be added, or those that are longer, if they can be removed. However, there is only one arc that can be removed due to the imposed construction rules.

**Random Move Generation** An arc can be sampled uniformly at random among a list of those that belong in the same (constructive) subneighborhood.

**Move Evaluation** A move is evaluated with respect to how it affects the lower bound of a (partially constructed) tour. However, the lower bound may need to be recomputed for that given move. Fortunately, it is possible to efficiently recompute it by focusing only on the incremental changes done in the bound computation. For example, Panangadan and Korf [62] describes how to incrementally update a Minimum

[1] In the case of the `ADD` subneighborhood, arcs are sampled by enumerating the not visited cities with respect to the last (visited) city.

Spanning Tree. This incremental computation would be used by the **Strong** bound to speed up its calculation.

### 5.1.3 Cable Trench Problem

The CTP [74] can be described as follows: "Buildings on a University campus are to be connected to the main computer building by dedicated high-speed cable. Trenches will be dug in order to allow cables to be laid from one building to another. In addition, dug trenches may be used by more than one cable in order to connect other buildings farther away from the main computer building. The cost of digging a trench is proportional to its length, and so is the cost of a cable. What is the optimal layout of the trenches such that the total (trench and cable) set-up cost is minimum?" Vasko [74] provides a mixed integer linear formulation for this problem.

A constructive-search model of this problem would take into account the following aspects:

**Problem Instance** An instance of this problem is characterized by the distances from each building to every other building, the cost of the cable, and the cost of digging a trench. The distances are usually stored in an $N \times N$ distance matrix, where $N$ denotes the number of buildings of an instance, while the cost of the cable and the cost of digging a trench are stored as a scalar variables.

**Solution Representation** A solution to an instance of this problem is characterized by a set of arcs that form a tree (any two buildings in that tree are connected by exactly *one* path). Furthermore, a (possibly partial) solution can be represented by a list of $k$ pairs of integers, such that $0 \leq k \leq N - 1$, where $N$ denotes the number of buildings of an instance. A solution is feasible (and complete) if all buildings of an instance constitute the (spanning) tree.

**Component Representation** In this problem, an arc connecting two buildings represents a component from the ground set of an instance. This information is usually stored in a pair of integers, where each integer represents a building. Furthermore, the size of the ground set is equal to the number of arcs of an instance, which is

$$\frac{N \times (N - 1)}{2}$$

**Empty Solution Generation** An empty solution is initialized with an empty list (of arcs) and a tree containing only the main computer building. Figure 5.5 illustrates an empty solution of an instance of this problem.

**Heuristic Solution Generation** A heuristic solution can be constructed using a greedy heuristic that iteratively adds the arc that minimizes the cost of the next tree until a feasible (spanning) tree is obtained, much like how Prim's and Dijkstra's algorithms operate. Figure 5.6 illustrates the heuristic construction of a (feasible) solution from a given (partial) solution with present and forbidden** components. It should be noted that in this particular example, the cost of the cable per unit is 0.3 and the cost of digging a trench per unit is 1.7.

**Solution Evaluation** In this problem, the performance of a feasible solution is measured by the sum of two parts: **1)** the length of the arcs that constitute the (spanning) tree (multiplied by the cost of digging a trench), **2)** and the length of the paths from each building to the main computer (multiplied by the cost of the cable). This value is to be minimized.

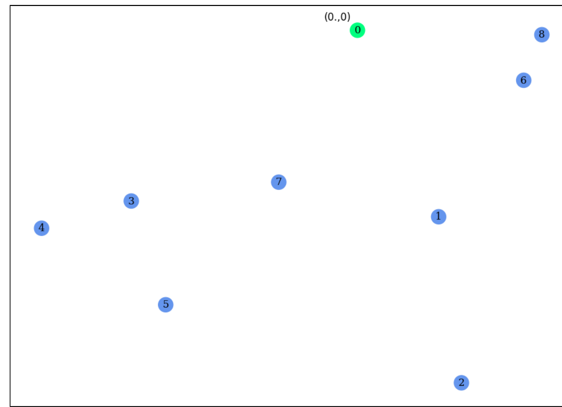



Figure 5.5: Illustration of an empty solution for the Cable Trench

**Bound Evaluation** The (lower) bound of a (partial) solution can be computed according to different methods, where each one has its relative strength (how close it is to the optimum value):

**Weak** The sum of two parts: **1)** the length of the arcs that constitute the tree (multiplied by the cost of digging a trench), **2)** and the length of the paths from each building in the tree to the main computer (multiplied by the cost of the cable).

**Strong** The sum of two parts: **1)** the length of the Minimum Spanning Tree that includes the arcs in the (partial) solution (multiplied by the cost of digging a trench), **2)** and the length of each path in the Shortest Path Tree, which includes the arcs in the (partial) solution (multiplied by the cost of the cable). Figure 5.7 illustrates the (strong) lower bound evaluation of a given (partial) solution with present and forbidden** components. It should be noted that in this particular example, the cost of the cable per unit is 0.3 and the cost of digging a trench per unit is 1.7. Furthermore, the lower bound value is $1.7 \times 2662.3 + 0.3 \times 4497.1 = 5875.04$.

**Solution Modification** A valid move that can be applied to a solution relies on what type of action that move performs. In other words, it depends for what type of subneighborhood a component (represented in the move) was sampled. In the context of this problem, the valid moves for each subneighborhood are presented as follows.

`ADD` Any arc that connects a building in the tree to another not included in the tree. This arc is added to the set of those that form the tree.

`REMOVE` Any arc in the tree that does not disconnect that tree if removed. This arc is removed from the set of those that form the tree.

`FORBID**` Any arc not included in the tree. This arc is added to a list of those that are excluded.

`PERMIT**` Any forbidden arc. This arc is removed from a list of those that are excluded.

It should be noted that the list of arcs that are excluded may grow to the size of the ground set, as in the case of the TSP.

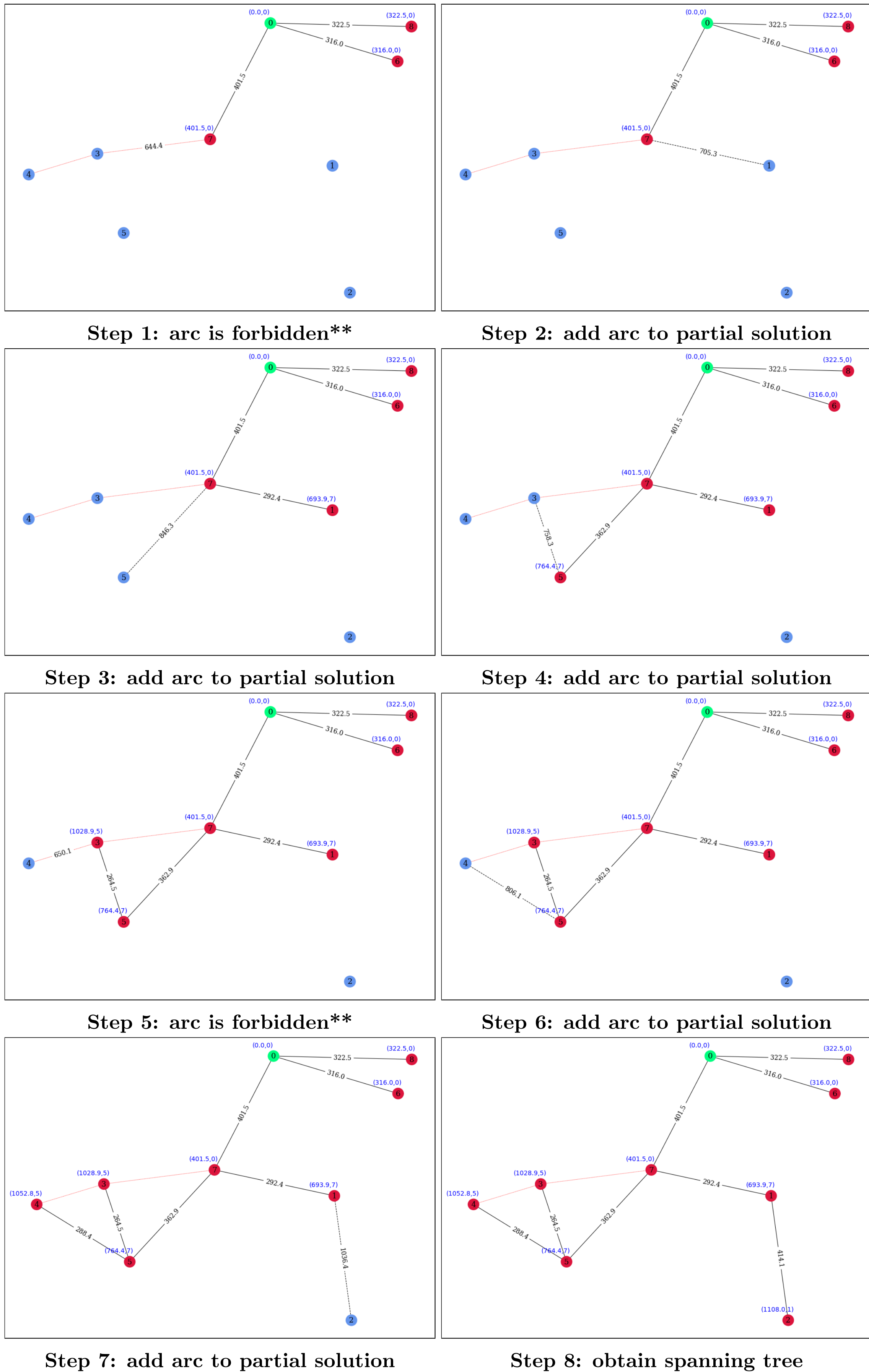
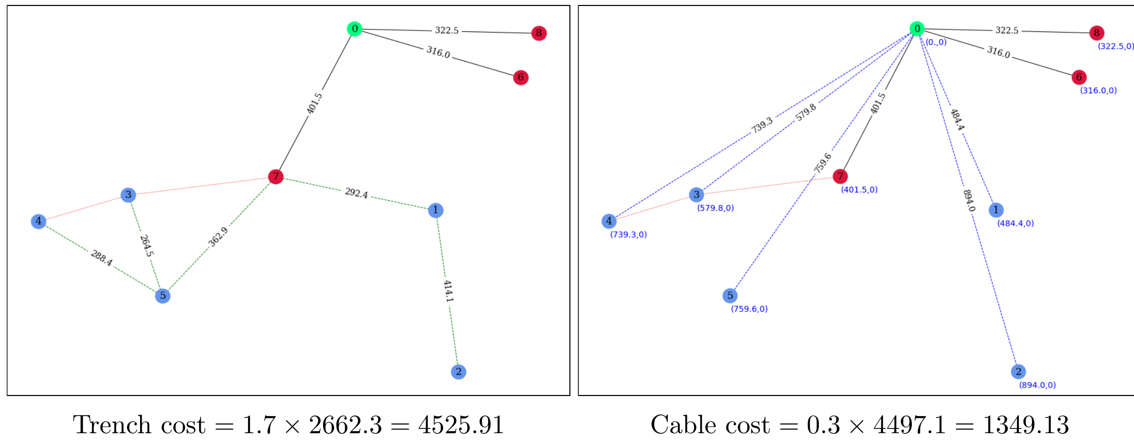



Figure 5.6: Example of construction with greedy heuristic for the Cable Trench




Trench cost $= 1.7 \times 2662.3 = 4525.91$ Cable cost $= 0.3 \times 4497.1 = 1349.13$

Figure 5.7: Example of strong lower bound for the Cable Trench

**Move Enumeration** Iteratively enumerate the arcs in a given (constructive) subneighborhood in the same order that those arcs are (explicitly or implicitly[2]) stored.

**Heuristic Move Generation** Heuristically select arcs in a given (constructive) subneighborhood in a way that favors those that minimize the cost of the next tree (if it is applied to the current solution).

**Random Move Generation** An arc can be sampled uniformly at random among a list of those that belong in the same (constructive) subneighborhood.

**Move Evaluation** A move is evaluated with respect to how it affects the lower bound of a (partial) solution. If an arc was used in one of the terms of the lower bound of the original (partial) solution, then adding it will not affect the value of that term. In general, both terms may need to be recomputed.

## 5.2 Implementation of Constructive-Search Algorithms

This section describes various implementations of constructive-search algorithms based on the API for constructive search. These implementations use the functions presented in Section 4.3, exposing that they use more or less the same underlying operations, despite having different search strategies. Furthermore, a good sample of algorithms that is representative of a larger portion available in the literature is Branch & Bound (B&B), Iterated Greedy (IG) algorithms, Greedy Randomized Adaptive Search Procedures (GRASP), Ant Colony Optimization (ACO), and Beam Search (BS). These algorithms were implemented according to the API in order to validate its coverage of a wide variety of constructive-search algorithms. In other words, this section corroborates that the API contains the necessary operations for the implementation of a panoply of constructive-search algorithms.

### 5.2.1 Branch & Bound

Algorithm 5.3 shows the pseudocode of B&B based on the API for constructive search. This algorithm can be described as follows:

[2] In the case of the `ADD` subneighborhood, arcs are sampled by enumerating the non-included buildings with respect to another building (in the tree)

**Algorithm 5.3:** Depth-first Branch & Bound based on the API

```
1  function ExpandSolution(U, s^p)
2      // bounding:
3      Φ_lb(s^p) ← getObjectiveLB(s^p) // compute lower bound
4      if Φ_lb(s^p) ≥ U then
5          return
6      end
7      if s ← heuristicSolution(s^p) ≠ ∅ then // find heuristic solution
8          // heuristic solution exists
9          f_s ← getObjectiveVector(s) // objective value of solution
10         if f_s < U then
11             s* ← copySolution(s*, s); U ← f_s // new best solution
12         end
13     end
14     // branching:
15     if (c ← heuristicMove(s^p, ADD)) ≠ ∅ then // get best component to add
16         s^p ← applyMove(s^p, c, ADD) // add component c
17         s^p ← ExpandSolution(U, s^p) // expand partial solution s^p further
18         // remove and then forbid component c
19         s^p ← applyMove(s^p, c, REMOVE)
20         s^p ← applyMove(s^p, c, FORBID)
21         // explores a disjoint nonempty region of the search space
22         // where component c is forbidden
23         s^p ← ExpandSolution(U, s^p)
24         s^p ← applyMove(s^p, c, PERMIT) // permit component c
25     end
26 end
27 begin
28     U ← ∞ // global upper bound
29     s^p ← emptySolution() // empty solution
30     s* ← ExpandSolution(U, s^p) // recursive function
31 end
```

**Initialization** Initialize an empty solution by calling `emptySolution(...)`. This solution represents all *feasible* solutions in the search space of a problem instance.

**Branching** Select a component to expand a *partial* solution by calling `heuristicMove-(ADD)`. The search space is partitioned into two disjoint nonempty subspaces using `applyMove(ADD)` and `applyMove(FORBID)`: one contains the selected component (`PRESENT`), and the other does not (`FORBIDDEN`). Since this algorithm is a depth-first implementation of a B&B, only one of those subspaces can be explored at a time. Thus, `applyMove(REMOVE)` and `applyMove(PERMIT)` are used to revert a solution to the previous (partial) solution so that the other subspace can also be explored.

**Update Global Upper Bound** Try to heuristically construct a feasible solution from a (partial) solution by calling `heuristicSolution(...)`. Update the global upper bound (and consequently the best-so-far solution) if the heuristic construction was successful and the objective value of the resulting solution is less than the global upper bound value. The objective value is obtained by calling `getObjectiveVector(...)`. In addition, update the best-so-far solution by using `copySolution(...)`.
It should be noted that `heuristicSolution(...)` returns its input (solution) if a

complete solution is specified. Thus, this section (in the pseudocode) also handles cases where a complete solution is reached in the search.

**Bounding** Compute the lower bound of a (partial) solution using `getObjectiveLB(...)`. This (partial) solution represents the subspace of all *feasible* solutions that contain it. If the lower bound value is greater than or equal to the global upper bound value, then that subspace is excluded from being further expanded.

This process ends when the entire search space is (implicitly) explored.

This constructive-search algorithm can only be applied to problem models implemented on the second level of the API since it contains operations such as *forbidding* and *permitting* components in a (partial) solution.

### 5.2.2 Iterated Greedy Algorithms

Algorithm 5.4 shows the pseudocode of an IG algorithm based on the API for constructive search. This algorithm can be described as follows:

**1.** Initialization of the current solution as a feasible solution that was heuristically constructed in `heuristicSolution(...)`.

**2.** Partial destruction of the current solution, which is done as follows:

   (a) Enumerate all components that can be removed from the solution by using `enumMove(..., REMOVE)`.

   (b) With `getObjectiveLBIncrement(..., REMOVE)`, evaluate each enumerated component for how much it affects the lower bound if it is removed from the solution.

   (c) Select a component at random among those that were enumerated, where those with a high contribution to the lower bound have a higher chance of being selected.

   (d) Remove the selected component from the solution by using `applyMove(..., REMOVE)`.

   This process is repeated at most $d$ times, resulting in a partially destroyed solution or an empty solution, in which case the destruction process may have terminated earlier as there were no components left.

**3.** Reconstruction of the partially destroyed solution by calling `heuristicSolution(...)`, which preserves all components present in that solution.

**4.** Update of the best solution, where the objective value of the reconstructed solution is obtained by calling `getObjectiveVector(...)` and then compared to the objective value of the best-so-far solution; if a better solution is found, the best-so-far solution is replaced by a new one by using `copySolution(...)`.

**5.** Application of the acceptance criteria to replace the current solution: if the reconstructed solution has a better objective value than the current solution, the latter is replaced by the new one by using `copySolution(...)`; otherwise there is a probability of replacing the current solution with a worse solution.

**Algorithm 5.4:** Iterated Greedy algorithm based on the API

```
1  U ← ∞
2  // construct initial solution
3  s ← heuristicSolution(emptySolution())
4  while stop criteria not met do
5     // destroy solution partially
6     for i ← 1 to d do // remove at most d components
7        𝒩 ← {}; H ← {} // empty feasible component list
8        while (c ← enumMove(s, REMOVE)) ≠ ∅ do // components for removing
9           η ← getObjectiveLBIncrement(c, REMOVE) // cost measure
10          𝒩 ← 𝒩 ∪ {c}; H ← H ∪ {η}
11       end
12       if 𝒩 = {} then
13          break // no components left to remove - stop destruction
14       else
15          // select component ċ based on a probability distribution
16          ċ ← StochasticSampling(𝒩, H)
17          s ← applyMove(s, ċ, REMOVE) // remove component ċ
18       end
19    end
20    // reconstruct solution
21    s' ← heuristicSolution(s^p)
22    // update best solution
23    f_s' ← getObjectiveVector(s)
24    if f_s' < U then
25       s* ← copySolution(s*, s'); U ← f_s'
26    end
27    // apply acceptance criteria
28    if f_s' < f_s then
29       s ← copySolution(s, s'); f_s ← f_s' // accept as current solution
30    else if random(0,1) ≤ exp(f_s − f_s')/T then
31       s ← copySolution(s, s'); f_s ← f_s' // accept worse solution
32    end
33 end
```

This process is repeated a given number of times (except for the first step).

Another variant of an IG algorithm was implemented according to the API. That variant has a different destruction process, where a component is selected uniformly at random among those that can be removed by using `randomMove(...)`.

### 5.2.3 Greedy Randomized Adaptive Search Procedures

Algorithm 5.5 shows the pseudocode of GRASP based on the API for constructive search. This algorithm can be described as follows:

1. Construction of a candidate solution, which is done in the following way:
   (a) Initialize an empty solution by calling `emptySolution(...)`.
   (b) Compute the the size of the restricted candidate list $|\ell|$, which is a fraction of the number of components that can be added to the partial solution, whose

value is returned by `getNeighbourhoodSize(..., ADD)`.

(c) With `heuristicMoveWOR(..., ADD)`, heuristically enumerate $|\ell|$ components at most, in an order that may or may not be greedy with respect to how each component affects the lower bound if it is added to the partial solution. These components constitute the restricted candidate list.

(d) Select a component uniformly at random from the restricted candidate list.

(e) Add the component to the partial solution by using `applyMove(..., ADD)`.

This process is repeated until the solution is complete. That is, it stops when the solution is feasible, which is verified by calling `isFeasible(...)`, and there are no components that can be added.

**Algorithm 5.5:** Greedy Randomized Adaptive Search Procedures (GRASP) based on the API

```
1  U ← ∞
2  while stop criteria not met do
3      // construct greedy randomized solution
4      s ← emptySolution()
5      while true do
6          ℓ ← {} // restricted candidate list
7          |ℓ| ← ⌈α×getNeighbourhoodSize(s, ADD)⌉
8          for i ← 1 to |ℓ| do // sample at most |ℓ| components
9              if (c ← heuristicMoveWOR(s, ADD)) = ∅ then
10                 break // no moves left to add - stop move sampling
11             end
12             ℓ ← ℓ ∪ {c}
13         end
14         if ℓ = {} and isFeasible(s) then
15             break // solution is feasible - stop construction
16         else if ℓ = {} then
17             s ← emptySolution() // solution is unfeasible - try again
18         else
19             // select component ċ uniformly at random
20             ċ ← UniformRandomSampling(ℓ)
21             s ← applyMove(s, ċ, ADD) // add component ċ
22         end
23     end
24     // update best solution
25     f_s ← getObjectiveVector(s)
26     if f_s < U then
27         s* ← copySolution(s*, s); U ← f_s
28     end
29 end
```

**2.** Update of the best solution, where the objective value of the candidate solution is obtained by calling `getObjectiveVector(...)` and then compared to the objective value of the best-so-far solution; if a better solution is found, the best-so-far solution is replaced by a new one by using `copySolution(...)`.

This process is repeated a given number of times.

Other variants of GRASP with alternate construction mechanisms were implemented ac-

cording to the API. They are distinguished by how the restricted candidate list is constructed. One variant enumerates all components that can be added to the partial solution by using `enumMove(..., ADD)`; then, it evaluates these components with respect to how they affect the lower bound by calling `getObjectiveLBIncrement(..., ADD)`; the components are sorted according to this value, and only the best $|\ell|$ ones are selected to constitute the restricted candidate list. The other variant also enumerates all components that can be added to the partial solution and evaluates them according to how they affect the lower bound. However, components are only selected to enter the restricted candidate list if their value is lower than a threshold, such that $\{c \in \mathcal{N} \mid \eta_v < \eta_{min} + \alpha \cdot (\eta_{max} - \eta_{min})\}$, where .

### 5.2.4 Ant Colony Optimization

Algorithm 5.6 shows the pseudocode of the Ant System (AS) [31] based on the API. Other ACO variants were also implemented according to the API, namely the Ant Colony System (ACS) [29; 30], the Approximate Nondeterministic Tree Search (ANTS) [55], the Elitist Ant System (EAS) [31], the Hyper-Cube Framework for Ant Colony Optimization (HC-ACO) [15], the $\mathcal{MAX}-\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) [71; 72; 73], and the Ranked-Based Ant System (AS$_{rank}$) [18]. For the sake of conciseness, only the implementation of the AS is described here, as most of the previous variants derive from it by including additional features or making slight changes that improve the overall performance of the algorithm.

The AS based on the API for constructive search can be described as follows:

1. Initialization of the pheromone model, which has the same size as the ground set of the problem instance, whose value is returned by `getNumComponents(...)`.

2. Construction of the population of "ant trails," where each "ant trail" is constructed as follows:

   (a) Initialize the "ant trail" as an empty solution by calling `emptySolution(...)`.

   (b) Enumerate all components that can be added to the partially constructed "ant trail" by using `enumMove(..., ADD)`.

   (c) With `getObjectiveLBIncrement(..., ADD)`, evaluate each enumerated component. This value represents the heuristic information that is commonly mentioned in the literature.

   (d) Obtain the pheromone value "deposited" in each enumerated component by calling `getComponentID(...)`, which returns a unique identifier that is used to access the corresponding pheromone in the pheromone model.

   (e) Select a component at random among those that were enumerated, where the probability of selecting each component is based on the linear combination of the heuristic information and the pheromone value.

   (f) Add the selected component to the partially constructed "ant trail" by using `applyMove(..., ADD)`.

   This process is repeated until the "ant trail" is complete. That is, it stops when the "ant trail" is a feasible solution, which is verified by calling `isFeasible(...)`, and there are no components that can be added.

**Algorithm 5.6:** Ant System based on the API

```
1  U ← ∞
2  // pheromone initialization
3  τ⃗ ← PheromoneInitialization()
4  while stop criteria not met do
5      foreach s ∈ P do
6          // ant based solution construction
7          s ← emptySolution()
8          while true do
9              N ← {}; H ← {}; T ← {}
10             while (c ← enumMove(s, ADD)) ≠ ∅ do // components for adding
11                 η ← getObjectiveLBIncrement(c, ADD) // heuristic information
12                 i ← getComponentID(c); τ_i ← τ⃗[i] // pheromone value
13                 N ← N ∪ {c}; H ← H ∪ {η}; T ← T ∪ {τ_i}
14             end
15             if N = {} and isFeasible(s) then
16                 break
17             else if N = {} then
18                 s ← emptySolution()
19             else
20                 ċ ← StochasticSampling(N, H, T)
21                 s ← applyMove(s, ċ, ADD)
22             end
23         end
24     end
25     foreach s ∈ P do
26         // update best solution
27         f_s ← getObjectiveVector(s)
28         if f_s < U then
29             s* ← copySolution(s*, s); U ← f_s
30         end
31         // pheromone update
32         while (i ← enumSolutionComponents(s, PRESENT)) ≠ −1 do
33             τ⃗[i] ← (1 − ρ) × τ⃗[i] + 1/f_s
34         end
35     end
36 end
```

3. Update of the best solution, where the objective value of each “ant trail” is obtained by calling `getObjectiveVector(...)` and then compared to the objective value of the best-so-far solution; if a better solution is found, the best-so-far solution is replaced by a new one by using `copySolution(...)`.

4. Update of the pheromone values according to the quality of each “ant trail,” whose components are enumerated by calling `enumSolutionComponents(...)` and then used to update their respective pheromones.

This process is repeated for a given amount of times (except for the first step).

### 5.2.5 Beam Search

**Algorithm 5.7:** Beam Search based on the first level of the API

```
1  U ← ∞
2  s^p ← emptySolution(); B ← {s^p} // initialize beam with empty solution
3  while B ≠ {} do
4      B′ ← {}
5      foreach s^p ∈ B do
6          if isFeasible(s^p) and (f_{s^p} ← getObjectiveVector(s^p)) < U then
7              s* ← copySolution(s*, s^p); U ← f_{s^p} // new best solution
8          end
9          // branching:
10         i ← 0
11         // get next best component to add
12         while (c ← heuristicMoveWOR(s^p, ADD)) ≠ ∅ and i < k_ext do
13             // add component c to a copy of partial solution s^p
14             s′^p ← copySolution(s′^p, s^p); s′^p ← applyMove(s′^p, c, ADD)
15             // bounding:
16             Φ_lb(s′^p) ← getObjectiveLB(s′^p) // compute lower bound
17             if Φ_lb(s′^p) < U then
18                 // (partial) solution s′^p may or may not be added to beam
19                 B′ ← UpdateBeam(B′, b_width, s′^p)
20             end
21             i ← i + 1
22         end
23     end
24     B ← B′ // new current beam
25 end
```

Algorithm 5.7 shows the pseudocode of BS based on the first level of the API for constructive search. This algorithm can be described as follows:

1. Initialize an empty solution with `emptySolution(...)` and add it to the beam.
2. Expand the (partial) solutions in the beam, where each (partial) solution is expanded as follows:
   (a) Heuristically enumerate a maximum of $k_{ext}$ components that can be added to a partial solution by calling `heuristicMoveWOR(ADD)`. No components are sampled if the solution is complete.
   (b) Add each component to an *exclusive* copy of the partial solution by using `copySolution(...)` followed by `applyMove(ADD)`. This process results in a maximum of $k_{ext}$ (partial) solutions.
   (c) Compute the lower bound of each (partial) solution using `getObjective-LB(...)`. If the lower bound of a (partial) solution is less than the global upper bound value, that (partial) solution is added to the beam of expanded solutions.

   This process is repeated until there are no solutions left in the beam. Then, those in the beam of expanded solutions are transferred to the other beam. However, only at most $b_{width}$ (partial) solutions are moved such that those with a smaller lower bound value are favored.
3. Update the best solution: **1)** verify which solutions are feasible with respect to the

problem, **2)** evaluate the feasible solutions by calling `getObjectiveVector(...)`, **3)** and compare their objective values to the global upper bound value; if the objective value is less than the global upper bound value, update the best-so-far solution by using `copySolution(...)`.

This process is repeated until there are no solutions left in the beam.

It is also possible to implement BS on the second level of the API for constructive search. Algorithm 5.8 shows the pseudocode of BS based on the second level of the API. This algorithm partitions the subspace into disjoint nonempty subspaces such that each expanded (partial) solution explores a partition of the search space. This exploration is more efficient since a solution can only be found (at most) once throughout optimization process. Thus, the second-level BS may produce better quality solutions than BS on the first-level BS (depending on the problem).

**Algorithm 5.8:** Beam Search based on the second level of the API

1 $U \leftarrow \infty$
2 $s^p \leftarrow \mathit{emptySolution}()$; $B \leftarrow \{s^p\}$
3 **while** $B \neq \{\}$ **do**
4     $B' \leftarrow \{\}$
5     **foreach** $s^p \in B$ **do**
6         **if** $\mathit{isFeasible}(s^p)$ **and** $(f_{s^p} \leftarrow \mathit{getObjectiveVector}(s^p)) < U$ **then**
7             $s^* \leftarrow \mathit{copySolution}(s^*, s^p)$; $U \leftarrow f_{s^p}$
8         **end**
9         `// branching:`
10         $i \leftarrow 0$
11         `// get best component to add`
12         **while** $(c \leftarrow \mathit{heuristicMove}(s^p, \texttt{ADD})) \neq \varnothing$ **and** $i < k_{ext}$ **do**
13             `// add component` $c$ `to a copy of partial solution` $s^p$
14             $s'^p \leftarrow \mathit{copySolution}(s'^p, s^p)$; $s'^p \leftarrow \mathit{applyMove}(s'^p, c, \texttt{ADD})$
15             `// forbid component` $c$ `in partial solution` $s^p$
16             $s^p \leftarrow \mathit{applyMove}(s^p, c, \texttt{FORBID})$
17             `// explores two disjoint nonempty regions of the search`
18             `// space where either component` $c$ `is present or forbidden`
19             `// bounding:`
20             $\Phi_{lb}(s'^p) \leftarrow \mathit{getObjectiveLB}(s'^p)$
21             **if** $\Phi_{lb}(s'^p) < U$ **then**
22                 $B' \leftarrow \text{UpdateBeam}(B', b_{width}, s'^p)$
23             **end**
24             $i \leftarrow i + 1$
25         **end**
26     **end**
27     $B \leftarrow B'$
28 **end**

## 5.3 Experimental Evaluation

### 5.3.1 Computational Overhead Introduced by the API

The adoption of an API by a software application usually introduces additional overhead besides the normal computational cost used of that application. The API for constructive search is no different, and may introduce an extra cost in addition to the one utilized by a constructive algorithm throughout its execution. It is crucial to measure this extra cost in order to know the limitations of the use of the API, as well as understanding how it can be improved. Thus, an experimental evaluation is performed to assess the computational overhead introduced by the API, and the results are discussed to understand how that overhead may be reduced.

In order to measure the computational overhead introduced by the API, two implementations of a depth-first dichotomic B&B for the CTP were compared: one B&B was a previously available[3] dedicated implementation in C that was specifically designed (and optimized) for the CTP, while the other is a generalized B&B that is applied to a computational model of the CTP, both based on the API. Hereafter, the former shall be referred as "handcrafted B&B," while the latter shall be referred as "B&B using API." Furthermore, this optimization problem and algorithm were selected to measure the computational overhead associated with the API because the implementation from scratch in C was already available, requiring only to the development of a computational model based on the API, which could then be used by the generalized B&B. However, not all of the optimizations were implemented in the API model. This decision had an impact on the results, which are later discussed in this section.

In the interest of a fair comparison, both implementations of the B&B need to meet certain criteria: *a.* report the same optimum solution, *b.* explore the same nodes in the search tree, in the same order (verifying if the pruning and branching rules are equal). The first criterion was validated by comparing the objective values reported by each algorithm at the end of their execution (or comparing both solutions, for some cases). The second criterion was validated by printing the nodes explored by each algorithm and verifying if they are the same and in the same order.

Besides using different implementations of the B&B, the size of the CTP instances was also varied in this experimental evaluation since it is important to assess how the overhead evolves as instances grow larger. Based on the empirical analysis of various runs, this parameter was fixed at the following levels: $n = \{5, 10, 15, \ldots, 40\}$. Furthermore, two other parameters related to the CTP needed to be fixed: the cost of the cable and the cost of digging a trench. Various runs were performed to decide on ideal values. However, after an empirical analysis, these parameters were fixed at only one: the cable cost per unit length was set to 0.05 and the trench cost per unit length was set to 0.95. For other different sets of values, the run time of both B&Bs was too long for larger instances.

Thirteen instances were randomly generated for each size, where the coordinates of each building were both random real values in the range $[0., 1000.)$, resulting in a total of 104 instances. Then, each implementation of the B&B was executed once for all instances of the CTP, and the run time was measured for each execution.

[3] Thanks to Professor Carlos Fonseca for providing the Branch & Bound implementation

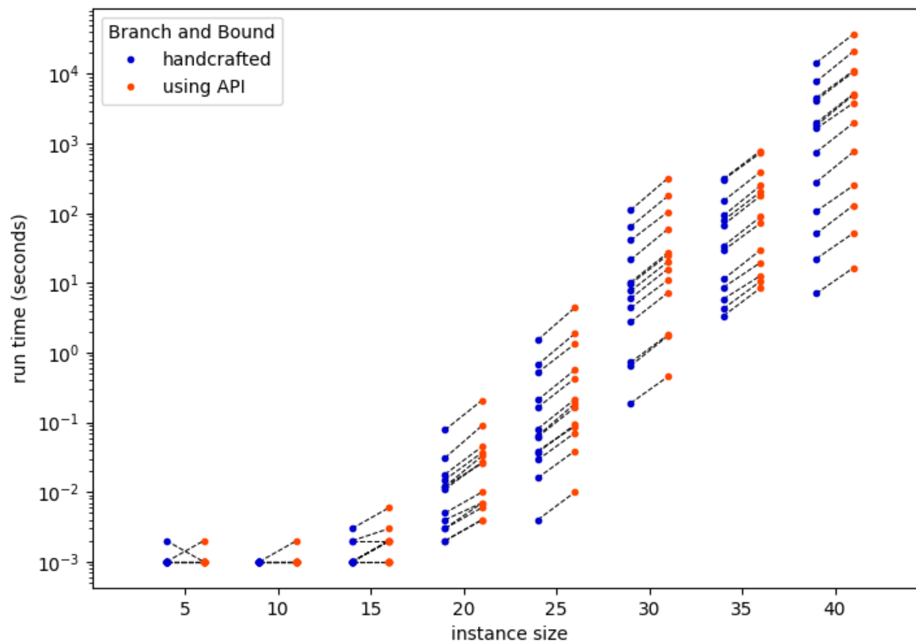



Figure 5.8: Run times of “handcrafted B&B” and “B&B using API”

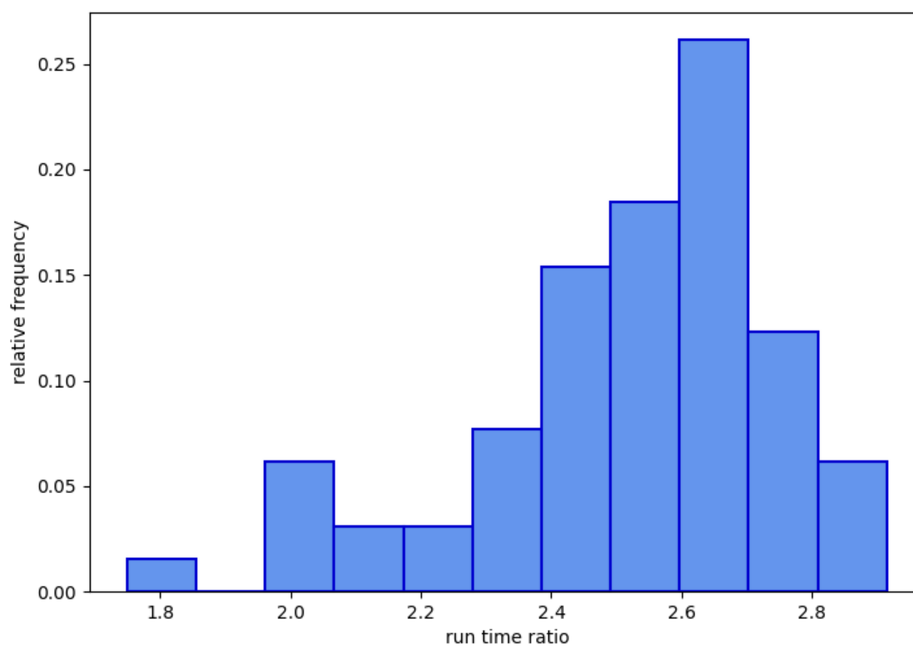



Figure 5.9: Distribution of run time ratio of “B&B using API” to “handcrafted B&B”

Figure 5.8 shows the differences of the run time for both implementations of the B&B, in seconds, concerning the 104 CTP instances with different sizes. While it is difficult to take discernible conclusions about the overhead introduced in smaller instances (due

to precision limitations in measuring the run time), it is clear that the API introduces a slowdown in larger instances. Such a slowdown seems to remain at a constant factor as the size of an instance grows, as evidenced by the lines that pair both B&Bs applied to the same instance, where those lines are nearly parallel to each other. Consequently, it is crucial to measure the constant factor, which is done by calculating the run time ratio of the "B&B using API" to the "handcrafted B&B."

Figure 5.9 shows the distribution of the run time ratio of the "B&B using API" to the "handcrafted B&B," concerning the instances of size $\{20, 25, \ldots, 40\}$. It is clear that the slowdown introduced by the API is lower than *three* times.

Furthermore, to check if it would be possible to reduce this slowdown further, the *Gprof* profiling tool was used to diagnose the execution of the "B&B using API."

| **% time** | **cumulative seconds** | **self seconds** | **calls** | **self ms/call** | **name** |
|---|---|---|---|---|---|
| 31.41 | 0.92 | 0.92 | 608 445 | 0.00 ($1.51 \times 10^{-3}$) | `lb` |
| 29.53 | 1.79 | 0.87 | 523 695 915 | 0.00 ($1.66 \times 10^{-6}$) | `pairing` |
| 18.09 | 2.32 | 0.53 | 304 222 | 0.00 ($1.74 \times 10^{-3}$) | `removeMove` |
| 9.90 | 2.61 | 0.29 | 304 222 | 0.00 ($9.53 \times 10^{-4}$) | `addMove` |
| 3.41 | 2.77 | 0.10 | 21 275 437 | 0.00 ($4.70 \times 10^{-6}$) | `swap` |

Table 5.1: Flat profile of the "B&B using API"

Table 5.1 shows the flat profile of the top 5 function calls with larger run times, concerning the execution of the "B&B using API" for an instance of size 30. It appears that `lb(...)` and `removeMove(...)` are the functions that take on average more time whenever they are called (as evident by the **self ms/call** column, which represents the average number of milliseconds spent in a function per call). In this case, `lb(...)` computes the lower bound of a given (partial) solution, and `removeMove(...)` removes a component (an arc) from a (partial) solution, whereas `pairing(...)` is a small utility function. While it may be possible to optimize `lb(...)` further, this function is unlikely to be the main cause of overhead because both B&Bs implement the same incremental computations in the lower bound calculation. However, `removeMove(...)` is likely to be introducing a considerable slowdown: while the "handcrafted B&B" stores the data of the previous partial solution, the "B&B using API" needs to recompute all of its information (mainly a list of arcs with minimal cost that may be added to that partial solution) before backtracking. Thus, it may be possible to reduce the slowdown further by optimizing the implementation for the case where only the last added component is ever removed. There are other optimizations implemented in the "handcrafted B&B." However, these are problem-specific optimizations that slightly alter how the B&B itself works (and the computational model cannot affect the behavior of the B&B which is applied to it).

### 5.3.2 Solver Performance Evaluation

The API for constructive search supports exact and metaheuristic algorithms such as B&B, IG algorithms, GRASP, ACO, and BS. Exact algorithms such as B&B have a guarantee that an instance of a problem is always solved to optimality. However, these approaches are computationally expensive for large instances of a problem since their run time grows exponentially as the size of an instance increases. In contrast, metaheuristics take a reasonable run time, but they do not guarantee that an instance of a problem will be solved to optimality. Furthermore, these approaches are significantly better than random search, given that they tend to find near-optimal solutions. When a large panoply of metaheuristics is available to be applied to an optimization problem, there is bound to be some that are better than others. The API for constructive search allows for problem models to be used directly by various solvers (constructive-search algorithms). Consequently, it is possible to assess which solvers outperform others when applied to the same problem (model). Thus, an experimental evaluation is done to measure the performance of various solvers (which only include metaheuristics) when applied to the Symmetric Travelling Salesman Problem.

The solvers that were selected for this experimental evaluation are the following: the Ant Colony System (ACS), the Approximate Nondeterministic Tree Search (ANTS), the Ant System (AS), the $\mathcal{MAX}-\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS), and GRASP. Note that most of these solvers are variants in Ant Colony Optimization (ACO) (and also GRASP). The remaining metaheuristics were not selected for this experimental evaluation due to the following reasons:

- The Elitist Ant System and the Hyper-Cube Framework for Ant Colony Optimization only vary slightly from the Ant System, and using these ACO variants would not be as interesting as using other variants that are considerably different.
- The Iterated Greedy algorithm was not selected due to the construction rules implemented in the problem model of the TSP, which specify that only the last arc added in the (partial) tour can be removed. This rule would (indirectly) result in the reconstruction of same solution (by `heuristicSolution(...)`) after it was partially destroyed. Thus, the same solution would be produced at each iteration, and no optimization would be performed.
- Beam Search is a Tree Search Algorithm based on a breath-first search that limits the number of nodes produced at each level of the search tree. Unlike other (constructive) metaheuristics, this algorithm only completes the construction of solutions at the end of its execution (instead of one by one). Consequently, in order to match the same number of tours produced by each solver, Beam Search would need to allocate a long array of solutions, which would cause high memory usage for large instances.

Furthermore, as mentioned in Section 5.2, there are various mechanisms to construct the restricted candidate list in GRASP; in this experimental evaluation, only components whose value is less than a threshold are selected.

The parameters of the selected solvers need to fixed before applying them to instances of the Symmetric TSP. In the case of ACO algorithms such as ACS, AS, and $\mathcal{MM}$AS, their parameters were set according to the values suggested by Dorigo and Stützle [32], and Stützle and Hoos [73] (which are recommended for the TSP). In the case of ANTS, the parameters were set in line with those reported by Maniezzo [55] (despite being used for the QAP) and those used by the AS. As for GRASP, its single parameter was set as

reported by Resende and Ribeiro [69]. Table 5.2 shows all of the parameter values that were fixed for each solver. It should be noted that the heuristic information weight was set to a value slightly larger than the standard (such increase is justified in the next experimental evaluation).

| **ACO Algorithm** | $\alpha$ | $\beta$ | $\rho$ | $m$ | $\tau_0$ |
|---|---|---|---|---|---|
| AS | 1 | 3 | 0.5 | $n$ | $m/C^{nn}$ |
| $\mathcal{MM}$AS | 1 | 3 | 0.02 | $n$ | $1/\rho C^{nn}$ |
| ACS | 1 | 3 | 0.1 | 10 | $1/nC^{nn}$ |
| ANTS | – | – | 0.5 | $n$ | $m/C^{nn}$ |

Here, $n$ denotes the number of cities in an instance, $\alpha$ denotes the pheromone weight, $\beta$ denotes the heuristic information weight, $\rho$ denotes the evaporation rate, $m$ denotes the population size, $\tau_0$ denotes the initial pheromone value, and $C^{nn}$ denotes the objective value of the solution obtained by the Multi-Fragment heuristic.
$\mathcal{MM}$AS: The probability of constructing the best solution is 0.05. The frequency at which the global best solution updates the pheromones is at each $10^{th}$ iteration. The number of iterations until stagnation is 50. The smoothing rate value is 0.5.
ACS: In the local pheromone trail update rule: $\xi = 0.1$. In the pseudorandom proportional action choice rule: $q_0 = 0.9$
ANTS: The relative importance of the pheromone trail is 0.3. The width of the moving average window is $4 \times n$.
GRASP: In the construction of the restricted candidate list: $\alpha = 0.2$.

Table 5.2: Parameter values fixed for each solver

Furthermore, to perform a fair comparison between the selected solvers, the number of tours constructed in each instance of the TSP needs to be the same across all algorithms. AS, $\mathcal{MM}$AS, and ANTS build more solutions in each iteration than ACS and GRASP since their population size is larger. Thus, the number of iterations needs to be adjusted so that each algorithm produces the same number of solutions. Furthermore, this number should be at least 1000 as most solvers are reported to start to converge around this number of iteration [32; 69]. Ultimately, the number of iterations performed by each algorithm was set to the following: *a.* ACS performed $\frac{1000 \times n}{10}$ iterations, *b.* AS performed 1000 iterations, *c.* $\mathcal{MM}$AS performed 1000 iterations, *d.* ANTS performed 1000 iterations, *e.* and GRASP performed $1000 \times n$ iterations.

The solvers were applied to instances of the Symmetric TSP from TSPLIB [68]. In particular, they were applied to instances with less than 300 cities (this value is justified in the next experimental evaluated). They all used the **Weak** lower bound mentioned in Section 5.1.2 as heuristic information since this value is commonly used in the literature (the length of an arc). Each solver was executed once for all considered instances. At each execution, the objective value of each constructed tour was saved to an output file. However, only the value of the best solution is considered in the analysis of the results. In addition, the seed value that initializes the random number generator was stored in case the results need to be reproduced in the future.

Figure 5.10 shows the approximation ratio of the best solution reported by the solver concerning the instances of the TSP whose number of cities is less than 300. The results indicate that the ACS, the AS, and the $\mathcal{MM}$AS had approximately the same performance, with the $\mathcal{MM}$AS performing slightly better. In contrast, ANTS and GRASP had significantly worse performance.

To draw more accurate conclusions on the performance differences between each solver, the Friedman test was applied to the results with a significance level of 0.05. This test is used to detect differences in the ranking of solvers with respect to the various instances that were solved. For example, if a solver consistently reported the best solution, its ranking would be significantly different than the others, such that the Friedman test would be able

to detect it. Thus, when this test was applied to the results above, it reported that the null hypothesis was *rejected*. In other words, there are solvers whose ranking is significantly different than those of other solvers.

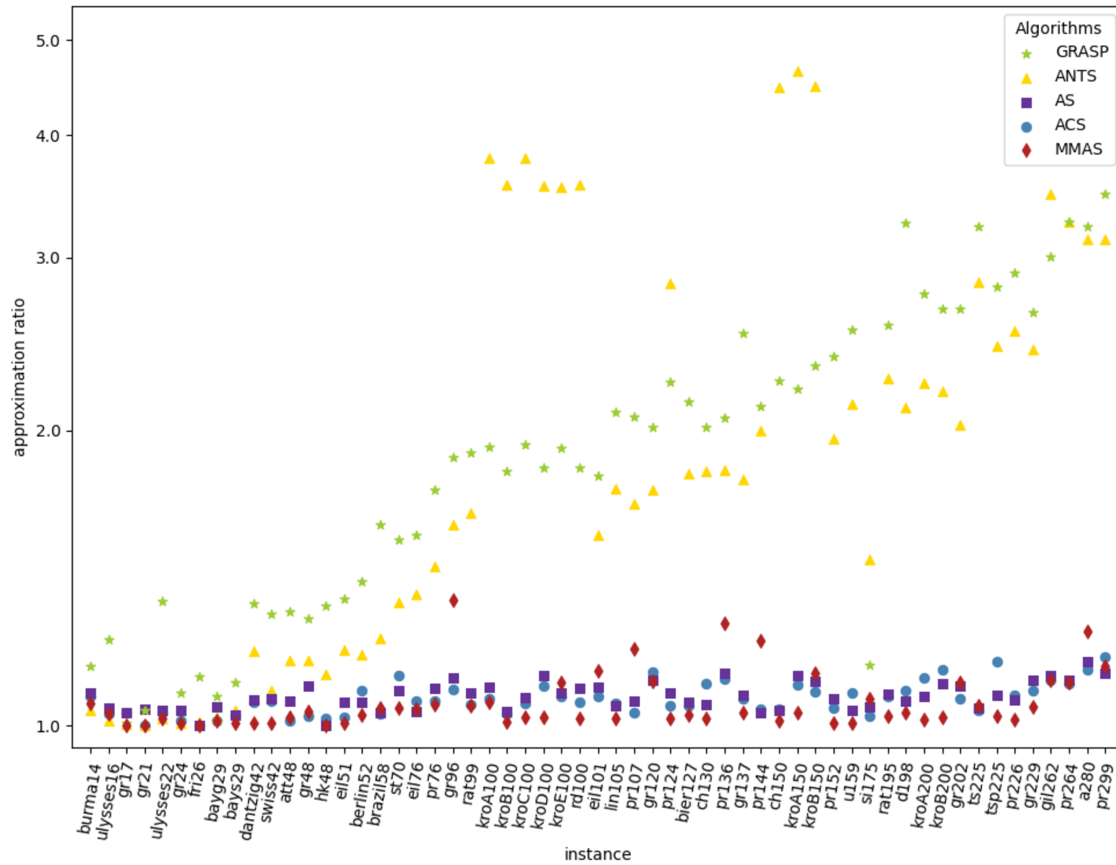



Figure 5.10: Solver performance

In order to understand which pairs of solvers have a significantly different ranking, the Nemenyi *post hoc* test for unreplicated blocked data was applied with a significance level of 0.05. These differences are seen in the following order *from best to worst* solver concerning their ranking values:

$$\overline{\mathcal{MM}\text{AS}\ \underline{\text{ACS}}}\underline{\ \text{AS}}\ \overline{\text{ANTS GRASP}}$$

Here, two solvers are not significantly different if the same line is drawn over (or under) them, and they are significantly different if otherwise. From this analysis, it is possible to conclude that the $\mathcal{MM}$AS slightly outperformed the other solvers, with the ACS being the only algorithm that was not significantly different. This result was not surprising since the $\mathcal{MM}$AS is reported in the literature as one of the best ACO algorithms. However, the result of ANTS was surprising as this ACO variant did not show the same performance as the other variants. In fact, it is shown that its performance is *significantly* different. A reason for this might be that its parameters were not adjusted to solve the TSP. However, another plausible reason for its poor performance is that this algorithm did not use a stronger bound for the ant trails construction. ANTS is described in the literature as an ACO algorithm that computes lower bounds on the completion of a partial solution [32].

### 5.3.3 Effect of the Problem Model on Solver Performance

The proposed abstraction of the API allows for any problem model to be used with any solver (if the required functions are implemented). Furthermore, there are different ways to formulate a problem as a constructive-search problem. This section conducts an experimental evaluation of the effect of the problem model on solver performance. In particular, a study is done on how the three lower bounds mentioned in Section 5.1.2 affect the performance of metaheuristics that solve the Symmetric TSP.

Both the **Weak** and **Intermediate** lower bounds implemented incremental evaluation. In contrast, the **Strong** lower bound did not implement incremental evaluation. As a result, the full lower bound computation occurs whenever a component/move is evaluated with respect to a given (partial) solution. A consequence of this implementation is that solvers take a large run time when using this lower bound. Therefore, only instances of the TSP with less than 300 cities were selected for the experimental evaluations. In fact, some of the selected instances did not run to completion as they exceeded the allotted time (∼48h). Ultimately, the implementation of the **Strong** lower bound needs to be optimized in the future to justify its application in concrete scenarios.

Furthermore, due to the computation time of the **Strong** bound, only two solvers were selected to study the effect of lower bounds on solver performance. These algorithms are $\mathcal{MM}$AS and ANTS. $\mathcal{MM}$AS was chosen since there was an interest in observing whether a stronger bound would further improve the performance of the best solver (with respect to the previous experimental evaluation). ANTS was selected because this ACO algorithm is described in the literature as an "Ant System that uses lower bounds as replacement of the heuristic information." As such, there was an interest in observing whether this solver would actually benefit from stronger lower bounds.

The parameters used by these solvers are the same as those used in the previous experimental evaluation. Table 5.2 shows all of the parameter values that were set for these solvers. It should be noted that the heuristic information weight was set to a value slightly larger than the standard since this experiment focuses on the effect of lower bounds as a replacement for the heuristic information (and less on the weight of pheromone values). Moreover, the same number of constructed tours was ensured across all selected solvers that use each bound, concerning each instance of the TSP. In this case, both algorithms performed 1000 iterations.

As already mentioned, these solvers were applied to instances of the Symmetric TSP from TSPLIB [68] with less than 300 cities. Each combination of solver and lower bound was executed once for all considered instances. Only the value of the best solution is considered in the analysis of the results despite saving the objective values of all constructed tours. Additionally, the seed value that initializes the random number generator was stored in case the results need to be reproduced in the future.

Figure 5.11 shows the approximation ratio of the best solution reported by the solver (using a given lower bound) concerning the instances of the TSP with less than 300 cities. The results indicate that a stronger lower bound does not *significantly* improve the performance of the $\mathcal{MM}$AS. In fact, the performance of the $\mathcal{MM}$AS using the **Intermediate** lower bound seems to be worse than the $\mathcal{MM}$AS using the **Weak** lower bound. In contrast, results suggest that a stronger bound does improve the performance of the ANTS.

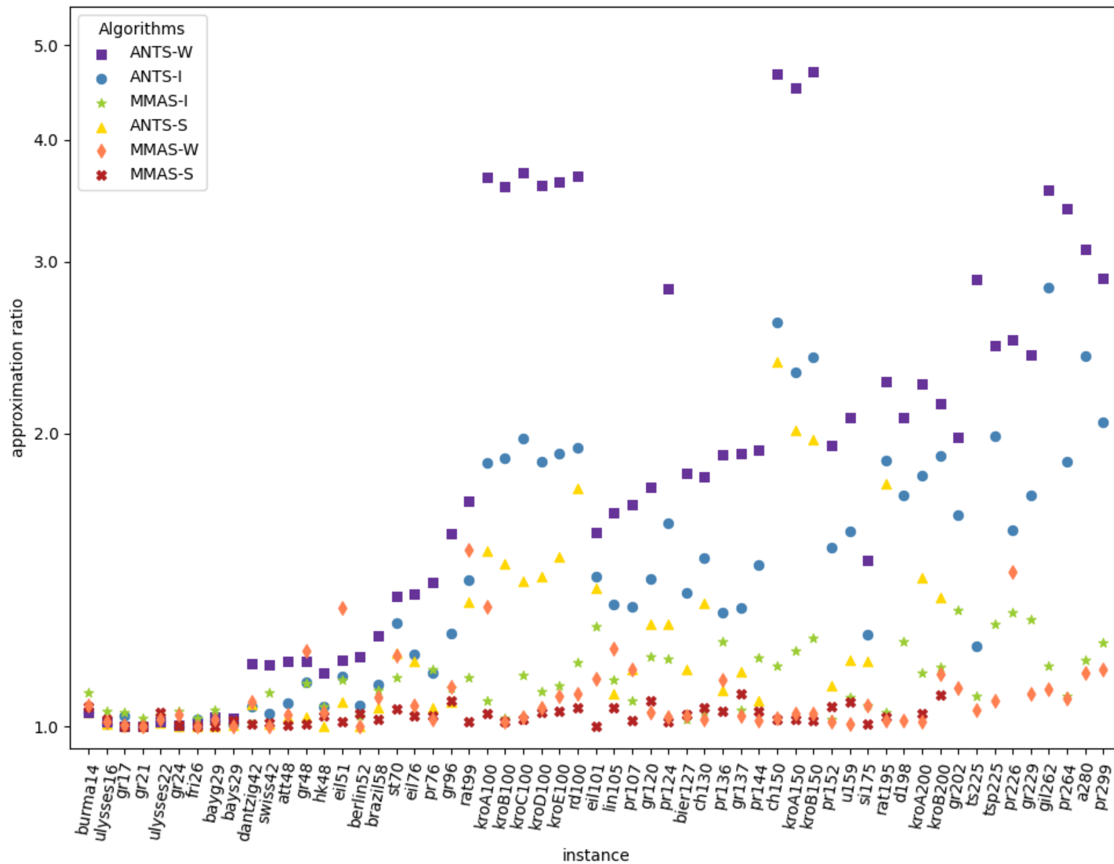



Figure 5.11: Effect of lower bound on solver performance

To draw more accurate conclusions on the performance (i.e. ranking) differences between each combination of solver/bound, the Friedman test was applied to the results with a significance level of 0.05. This test reports that the null hypothesis was *rejected*. That is, there are combinations whose ranking is significantly different than those of other combinations.

In order to understand which combinations of solver/bound have a significantly different ranking, the Nemenyi *post hoc* test for unreplicated blocked data was applied with a significance level of 0.05. These differences are seen in the following order *from best to worst* combination concerning their ranking values:

$$\overline{\mathcal{MM}\text{AS-}\mathbf{S}\ \underline{\mathcal{MM}\text{AS-}\mathbf{W}\ \text{ANTS-}\mathbf{S}\ \mathcal{MM}\text{AS-}\mathbf{I}}}\ \overline{\text{ANTS-}\mathbf{I}\ \text{ANTS-}\mathbf{W}}$$

Here, two combinations are not significantly different if the same line is drawn over (or under) them, or they are significantly different if otherwise. From this analysis, it is possible to conclude that ANTS benefits from a stronger lower bound since the ranking of the **Strong** one is significantly different from those of the **Weak** and **Intermediate** bounds. However, it did not have a significant improvement over the performance of $\mathcal{MM}$AS despite using a stronger lower bound over the traditional heuristic information. This performance may have been caused by its parameters not being adjusted to solve the TSP (unlike $\mathcal{MM}$AS). Furthermore, the **Intermediate** bound was not significantly different from the **Weak** bound despite looking worse for the $\mathcal{MM}$AS in Figure 5.11. This result suggests that its strength is also not significantly different from that of the **Weak** bound. Lastly, no conclusive evidence could be taken from the effect of a stronger bound on the performance of the $\mathcal{MM}$AS. Perhaps this solver already showed a favorable performance that was difficult

to be further improved. An extended experimental evaluation should be conducted for large instances in order to draw a more decisive conclusion. The implementation of the **Strong** bound should be optimized to perform incremental evaluation.

### 5.3.4 Computational Overhead of the Second Level of the API

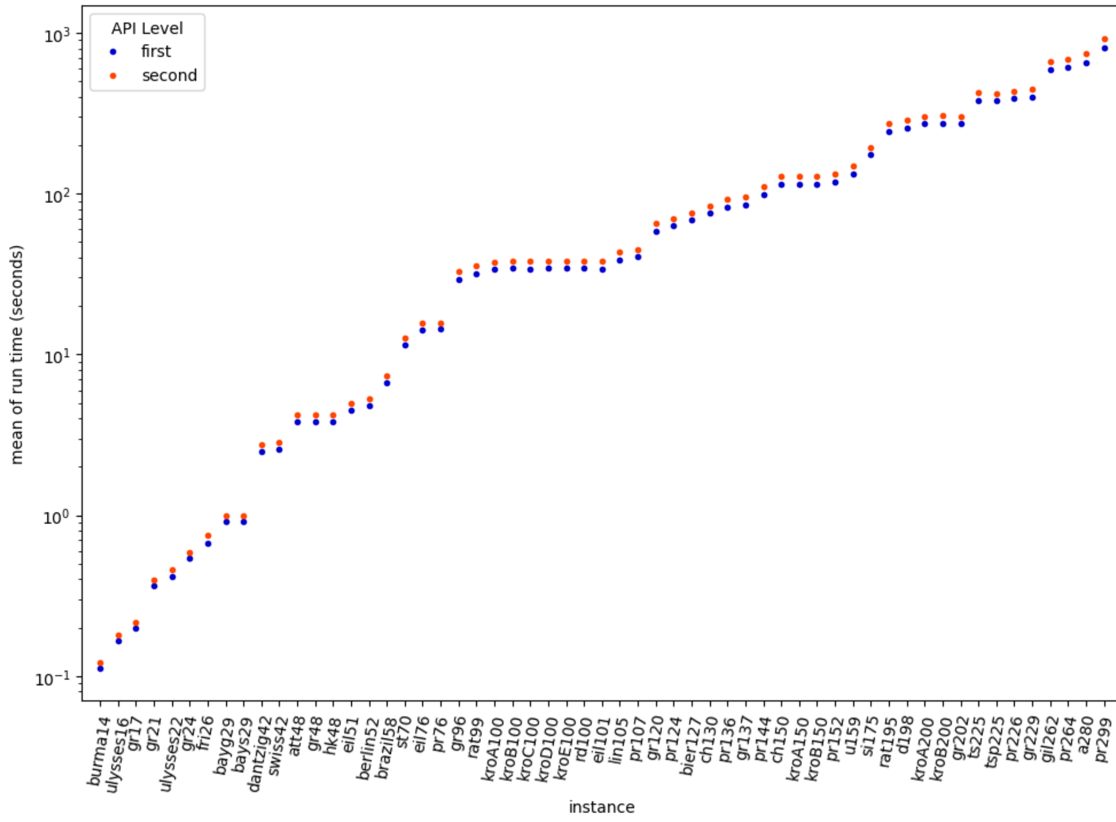
As mentioned in Section 4.3, two levels are defined for the API for constructive search. Models that implement the first level can be used with (most) metaheuristics. Models that implement the second can be used with metaheuristics and other solvers that partition the search space, such as B&B. Despite its generalizability, modeling problems on the second level may incur additional computational expense. These costs propagate to all solvers, including those that can use first-level models. Thus, this section performs an experimental evaluation that measures the overhead introduced by a second-level model when applied to metaheuristics that those on the first level can use.

In order to measure this overhead, two models that implement the TSP at each level were compared by applying them to a selection of (metaheuristic) solvers. Those solvers include the AS, the $\mathcal{MM}$AS, GRASP, and BS based on the first level. Other ACO variants were not selected since those implement approximately the same API operations as AS. Additionally, IG was not considered since it does not perform any optimization with the current models of the TSP. It should also be noted that the mechanism used by GRASP to construct the restricted candidate list is the one reported in Algorithm 5.5, where at most $|\ell|$ components are heuristically sampled to constitute the list. This implementation was selected in order to measure the overhead introduced by an API function (`getNeighbourhoodSize(ADD)`) that the other solvers do not use.

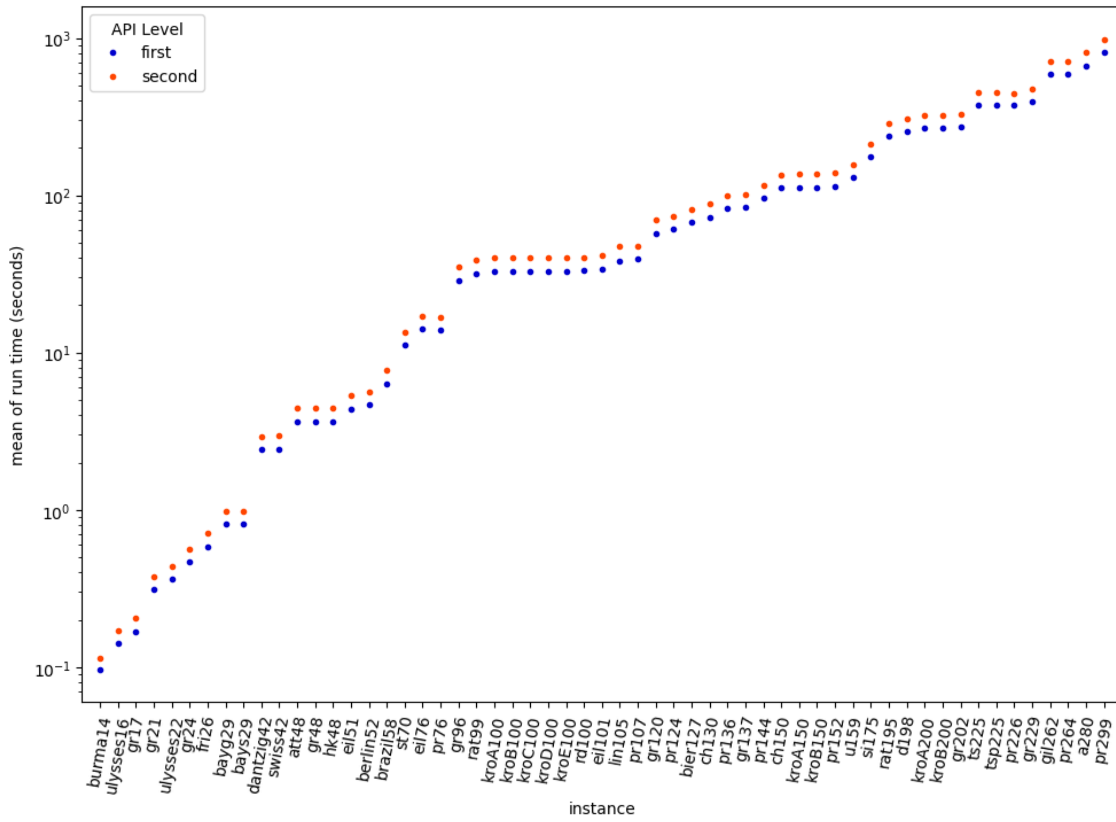
In the interest of a fair comparison, both models of the TSP should produce the same results despite being implemented at different levels. In other words, it is expected that the functions implemented by both models have the same characteristics. Furthermore, if a solver uses a random number generator, the same seed should be provided to both models to produce the same results.

The parameters used by AS, $\mathcal{MM}$AS, and GRASP are the same as those used in the previous experimental evaluation. Table 5.2 shows the parameter values that were set for these solvers. However, the heuristic information weight was an exception, and it was set to the standard value ($\beta = 2$) since it does not have any relevance in this experiment. Moreover, BS parameters such as $k_{ext}$ and $b_{width}$ were set to $\frac{n}{2}$ and 2750, respectively. Here, $n$ denotes the number of cities. There are no recommended values in the literature to how these parameters should be set. Regardless, setting these values ensure that the number of times that a component is added to a solution is the same as in other solvers. Within the same logic, the number of iterations set for the AS, the $\mathcal{MM}$AS, and GRASP were 1000, 1000, and $n \times 1000$, respectively. This decision was made so that all solvers perform about the same scale of operations such that it is easier to compare them to each other.

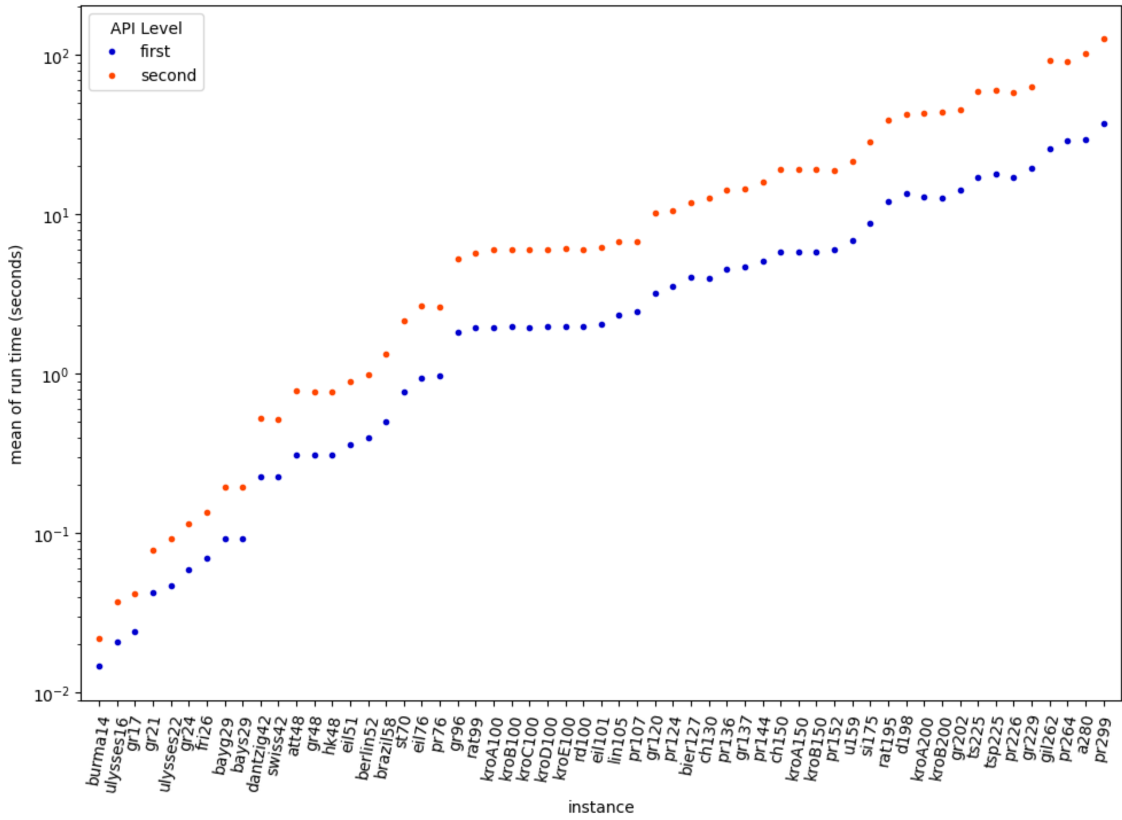
The instances of the TSP considered in the previous experiments were also selected in this experimental evaluation. In addition, all solvers used the **Weak** lower bound to reduce additional computational cost. Each selected solver was executed 13 times for all considered instances, and the run time was measured for each execution.



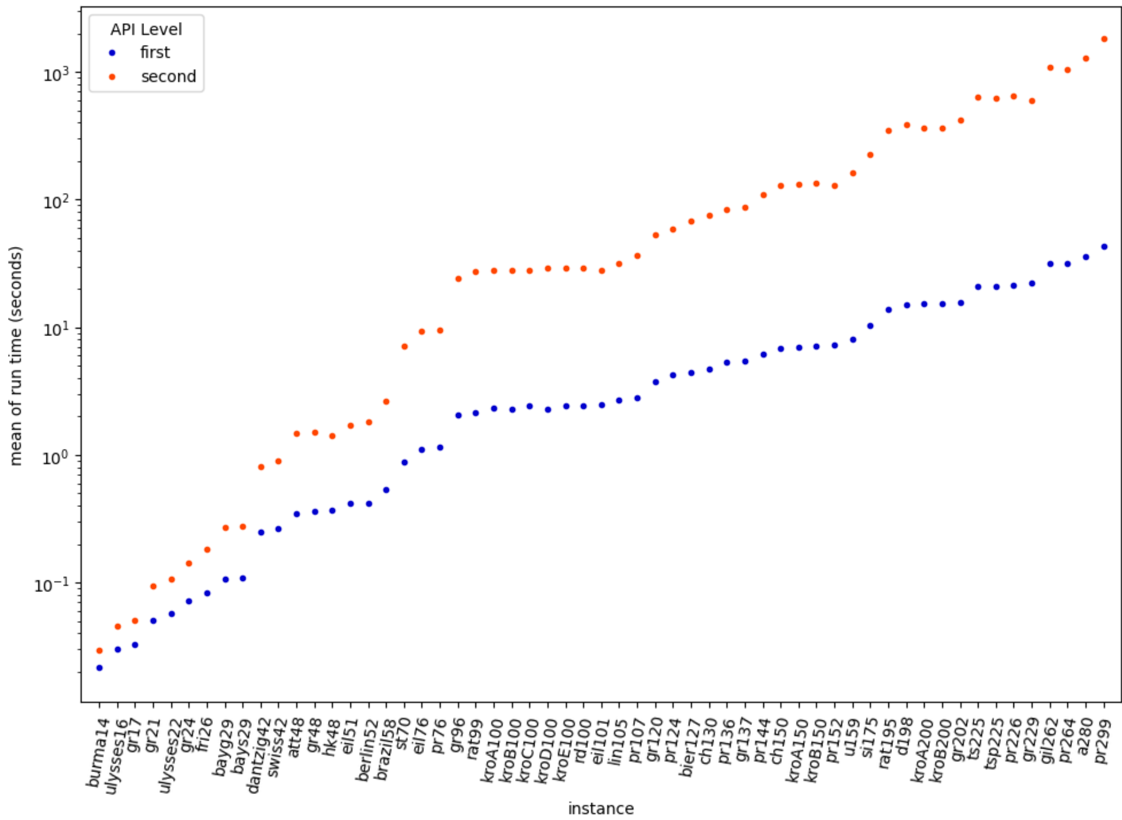

(a) Ant System




(b) $\mathcal{MAX} - \mathcal{MIN}$ Ant System




(c) GRASP




(d) Beam Search

Figure 5.12: Run times of first-level and second-level solver

Figure 5.12 shows the differences in the run time, in seconds, for solvers that use the models implemented at each level concerning the selected TSP instances. While the second-level model does not seem to have a visible overhead in AS and $\mathcal{MM}$AS, this model introduces a considerable computational cost in GRASP and BS. In fact, this cost seems to increase as the size of an instance grows. In order to analyze the cause of this overhead, a profiling tool was used to diagnose the execution of GRASP and BS applied to the first-level and second-level models.

| API level | % time | cumulative seconds | self seconds | calls | self ms/call | name |
|---|---|---|---|---|---|---|
| First | 76.64 | 14.94 | 14.94 | 418 518 432 | 0.00 ($3.57 \times 10^{-8}$) | `heuristicMoveWOR` |
| | 6.62 | 16.23 | 1.29 | 418 338 332 | 0.00 ($3.08 \times 10^{-9}$) | `pairing` |
| | 2.10 | 16.64 | 0.41 | 40 200 000 | 0.00 ($1.02 \times 10^{-8}$) | `getObjectiveVector` |
| | 1.74 | 16.98 | 0.34 | 40 000 000 | 0.00 ($8.50 \times 10^{-9}$) | `applyMove` |
| | 1.59 | 17.29 | 0.31 | 40 200 000 | 0.00 ($7.71 \times 10^{-9}$) | `nh_size` |
| Second | 32.83 | 18.96 | 18.96 | 40 200 000 | 0.00 ($4.72 \times 10^{-7}$) | `getNeighbourhoodSize` |
| | 27.88 | 35.07 | 16.10 | 418 518 432 | 0.00 ($3.85 \times 10^{-8}$) | `heuristicMoveWOR` |
| | 17.75 | 45.32 | 10.25 | 200 000 | 0.00 ($5.13 \times 10^{-5}$) | `emptySolution` |
| | 0.88 | 54.01 | 8.69 | 4 953 909 854 | 0.00 ($1.75 \times 10^{-9}$) | `pairing` |
| | 0.70 | 54.52 | 0.51 | 157 236 482 | 0.00 ($3.24 \times 10^{-9}$) | `swap` |

Table 5.3: Flat profile of GRASP applied to the *kroA100* instance

Table 5.3 shows the flat profile of the top 5 function calls with larger run times concerning the execution of GRASP using the first-level and second-level models for the *kroA100* instance. It appears that `getNeighbourhoodSize(ADD)` and `emptySolution(...)` introduce a large overhead when presented to the second-level model (as evident by the **self ms/call** column, which represents the average number of milliseconds spent in a function per call). In particular, `getNeighbourhoodSize(ADD)` on the second level traverses the entire subneighborhood to count which (valid) components are *not* forbidden, in contrast to the first level, which performs a direct calculation. Such an operation introduces a high computational cost! Furthermore, `emptySolution(...)` on the second level initializes an additional array of integers, which maintains the list of forbidden components, in contrast to the first level, which only initializes the sequence of cities. This array has the same size as the ground set, which is quadratic to the number of cities. Thus, besides introducing an overhead on computational time, second-level models also introduce overhead on the size of allocated memory. In fact, while the size of allocated memory on the first level is 236 880 bytes, on the second level is 355 792 bytes, concerning the *kroA100* instance.

| API Level | % time | cumulative seconds | self seconds | calls | self ms/call | name |
|---|---|---|---|---|---|---|
| First | 82.74 | 22.51 | 22.51 | 40 575 350 | 0.00 ($5.55 \times 10^{-7}$) | `lb` |
| | 5.86 | 24.11 | 1.60 | 41 122 701 | 0.00 ($3.89 \times 10^{-8}$) | `heuristicMoveWOR` |
| | 1.34 | 24.48 | 0.37 | 40 575 350 | 0.00 ($9.12 \times 10^{-9}$) | `applyMove` |
| | 1.27 | 25.16 | 0.35 | 40 575 357 | 0.00 ($8.63 \times 10^{-9}$) | `copySolution` |
| | 1.25 | 25.43 | 0.34 | 40 575 351 | 0.00 ($8.38 \times 10^{-9}$) | `nh_size` |
| Second | 56.77 | 11.75 | 11.75 | 40 575 350 | 0.00 ($2.90 \times 10^{-7}$) | `lb` |
| | 11.62 | 14.16 | 2.41 | 40 575 357 | 0.00 ($5.94 \times 10^{-8}$) | `copySolution` |
| | 9.66 | 16.16 | 2.00 | 41 122 701 | 0.00 ($4.86 \times 10^{-8}$) | `heuristicMoveWOR` |
| | 3.50 | 16.89 | 0.73 | 40 575 350 | 0.00 ($1.80 \times 10^{-8}$) | `applyMove` |
| | 2.32 | 17.32 | 0.48 | 161 359 006 | 0.00 ($2.97 \times 10^{-9}$) | `swap` |

Table 5.4: Flat profile of Beam Search applied to the *kroA100* instance

Table 5.4 shows the flat profile of the top 5 function calls with larger run times concerning the execution of the BS using the first-level and second-level models for the *kroA100* instance. It appears that `copySolution(...)` introduces a large overhead when implemented on the second level. In particular, this function (entirely) copies the array that represents the list of forbidden components every time it is called. Furthermore, BS duplicates a solution whenever it wants to add a component (because this solver also needs to maintain the original). Since this operation is repeated frequently throughout the execution of BS, it gradually leads to a very *high* computational time. BS also allocates a beam of solutions whose size depends on parameter $b_{width}$. As a result of solutions maintaining their list of forbidden components, a large overhead is also introduced on the size of allocated memory, which grows as the size of an instance and parameter $b_{width}$ increase. In fact, while the size of allocated memory on the first level is 14 248 396 bytes, on the second level is an impressive 232 303 676 bytes, concerning the *kroA100* instance and a $b_{width} = 2750$.

To mitigate the amount of computational overhead introduced by second-level models of the API, more sophisticated data structures and resource-management techniques could be implemented in these models. Those would include persistent data structures, and copy-on-write operations, among others.

## 5.4 User Feedback

An internal coding event called "ALGO Code Fest"[4] was organized[5] among undergraduate, Master's, and PhD students of the ALGorithms and Optimization (ALGO) Laboratory in order to understand how users who are unfamiliar with the API for constructive search would interact with it. The program of this event consisted of three parts:

**Opening Presentations** A presentation was made on how to formulate an optimization problem as a constructive-search problem. Then, another presentation was made on a subset of operations from the API for constructive search (first-level only).

**Main Activity** Participants organized themselves into various groups, and a task was assigned to each group. This task consisted in modeling a given CO problem as a constructive-search problem and implementing the corresponding model according to a subset of functions from the API (first-level only). These functions could then be used by provided solvers to optimize instances of the problem, and experimental results would be collected. The problems considered for this coding event were the following: Campus Network (Cable Trench Problem [74]), Community Detection (Clique-Partitioning Problem [12]), Hypervolume Subset Selection [48], and the 2018 Hash Code Self-Riding Cars Problem [2]. Moreover, the amount of time that groups had to complete the assigned task was only (approximately) 8 hours.

**Closing Presentations** Each group made a small presentation on the modeling and implementation decisions taken for their corresponding problem. Additionally, (preliminary) experimental results could also be presented if it was possible to apply the solvers to their model.

Among the four groups, three were able to formulate their optimization problem as a constructive-search problem, and one group was able to implement all functions of the provided subset. However, this group did not succeed in using the available solvers due to some faults in their implementation, which they were not able to fix in time.

Overall, most participants seemed to understand the design behind (a subset of) the API for constructive search, and (most importantly) they understood how to model an optimization problem as a constructive-search problem. However, some were unable to grasp concepts related to constructive search (e.g. constructive subneighborhood) and the behavior of each function. They suggested that a more detailed definition of each function should have been provided (and also what should be avoided), as well as an example of how to model and implement a simple optimization problem (e.g. the KP). Furthermore, a testing environment to validate the properties defined by each function should also have been provided, as well as another environment where users can examine the problem-specific outcome of each function. The latter would also enable users to observe how functions interact with each other. Lastly, further documentation of the API should be produced (e.g. elaborate training materials), and other opportunities for users to interact with the API should be made possible (e.g. organize workshops and events). In order to maintain any (optimization) software "alive", there needs to be users and the means for teaching them how to use it.

[4] ALGO Code Fest #0
`https://github.com/jerbubias/algo-code-fest-0`

[5] Thanks again to Professor Carlos Fonseca for helping organize the coding event

## 5.5 Concluding Remarks

This chapter presented the implementations of optimization problems and constructive-search algorithms based on the API. The underlying abstraction allows for the modeling and implementation of any CO problem and the implementation of several distinct constructive-search algorithms. Additionally, the computational overhead associated with the use of the API was measured for a specific scenario, determining that the API introduces a slowdown lower than three times (which may be further improved) on the test scenario considered. Furthermore, it was observed that the computational overhead of the second level of the API may be considerable in regard to the first level. However, it is possible to reduce such overhead by implementing sophisticated data structures and memory management techniques.

A study on the effect of models on the performance of solvers was also conducted in this chapter. In particular, it was seen that using stronger lower bounds may potentially benefit the solvers to which a model can be applied. This reveals an unusual paradigm in the development of optimization algorithms, where an emphasis is placed on improving a model instead of developing "brand new" algorithms (which in reality may be very similar to other existing algorithms).

Finally, a coding event was organized in order to observe how unfamiliar users would interact with the API. In general, almost all participants were able to understand how it is possible to model an optimization problem as a constructive-search problem. Some useful feedback was also provided by them. Notably, the production of teaching materials and the development of testing environments were among the most requested suggestions.

This concludes the evaluation of the API for constructive search. All of the recommendations reported throughout this chapter are possible research directions in future work.

# Chapter 6

# Conclusion and Future Work

This chapter summarizes the main conclusions taken from the completion of this work and emphasizes its main contributions. Furthermore, a reflection is made on the limitations presented throughout this work and subsequently provides future research directions that can be made to improve it.

The main focus of this dissertation was to develop an Application Programming Interface (API) for constructive-search problems and algorithms. This API separates the problem formulation from the algorithm that solves it by specifying a number of abstract elementary operations that problems must implement and solvers can use in a problem-independent way. Such operations were derived from the specifications presented by the conceptual model in Section 4.1 and the constructive-search algorithms in Section 2.3. The proposed API allowed for the modeling and implementation of Combinatorial Optimization (CO) problems as constructive-search problems, as well as the implementation of a selection of algorithms. It also promotes the development and improvement of problem models instead of developing "novel" algorithms. In fact, improving a model may potentially benefit all solvers, in contrast to only focusing on improving a single solver.

In a concrete scenario, it was observed that using the API resulted in a slowdown by a factor of less than *three* (which may still be improved) in comparison to a handcrafted implementation. In addition, two levels were defined for the API, where one introduces less computational overhead than the other (but it is restricted to a subset of solvers). Despite the fact that users unfamiliar with the API were not able to produce fully-working problem models in only *eight* hours, they seemed to understand how a problem can be modeled as a constructive-search problem in the context of the API.

Overall, the main contribution of this work was the proposal of an API for constructive-search problems and algorithms. The API was developed based on the analysis of constructive-search algorithms in the light of a conceptual model. Additionally, the following contributions were made:

- A number of constructive-search solvers were implemented based on the API, which can be applied to any constructive-search problem model implementing the API specification.
- A number of problem models were also implemented based on the API, which, together with the implemented solvers, demonstrate the expressiveness of the API.
- The API was evaluated through a set of computational experiments, and informal feedback was collected from users who were exposed to it for the first time.

Lastly, possible research directions in future work are presented as follows:

**Dynamic Programming**
Dynamic Programming is not *yet* supported by the API for constructive search. Although some operations were identified and partially defined (`checkDominance(...)` and `getDominanceClass(...)`) in regard to problem-specific dominance relations between solutions, no definite consensus was reached, and none of these operations was implemented in a problem model.

**Improved Problem Models**
Current implementations of problem models are incomplete/non-optimized. For example, a good sample of models lack incremental solution/bound evaluation (e.g. incremental evaluation for the **Strong** lower bound of the Travelling Salesman Problem (TSP)), others only have very weak (lower) bounds implemented, and some do not implement heuristic solution generation and heuristic move selection/enumeration. Improving these models will allow for their application in a larger scope of solvers and may potentially enhance the performance of these solvers.

**Small Set of Solvers**
A small set of solvers is supported by the API so far. Other algorithms can also be implemented, including (the already mentioned) Dynamic Programming, Dijkstra's algorithm, best-first and breadth-first Branch & Bound, Beam-ACO, Pilot and Rollout Method, "Squeaky Wheel" Optimization, Ruin-and-Recreate, among others. An issue that should also be taken into account is that some algorithms may be similar to others. Viewing these algorithms through the lenses of the underlying abstraction of the API may allow for these similarities to be easily detected.

**Second-Level Overhead**
The second level of the API introduces a substantial overhead in solvers. Sophisticated data structures and memory management techniques should be implemented in the problem models in order to mitigate this overhead.

**Multi-Objective Optimization**
Simplistic semantics are used for the computation of the lower bound in multi-objective cases. In particular, the API does not support the calculation of (lower) bound sets [34], and it is only restricted to one point. Consequently, a dedicated can be proposed for the API, which supports the computation of lower bound sets.

**Testing Environment**
User-friendly testing environments are not *yet* developed for the API. In particular, a property-based test suite should be developed for the API for users to understand if the implemented functions are behaving according to their definition. Despite a testing environment being already established for property-based testing, the tests themselves still need to be defined. In addition, a testing environment where users can interact with functions and observe their problem-specific behavior should also be provided.

**Maintaining the API**
Currently, there is not much documentation on the API (apart from this document). Developing teaching materials and proper documentation maintains an (optimization) software active since new users who are initially unfamiliar with it need to be captivated. In addition, users might be turned off from using the API if the implementation effort far exceeds their expectations. Thus, developing a library that provides generic implementations to ease this effort should also be considered in future work.

**Stable Release**
A stable version of the API should be prepared so that it can be released to the public.

This version should include proper testing environments, easy to understand documentation, complete and fully functional problem models, and a variety of solvers that the models can use.

**Experiments**

The experiments made so far did not draw *definite* conclusions for general cases. Thus, conducting other experiments with API is encouraged. Applying the TSP model to larger instance is an example of a future experiment (after optimizing the **Strong** bound). Additionally, trying to replicate the results reported by other authors is a good way to check if the implemented solvers are working as intended in the literature.

**Bridge Local and Constructive Search**

Based on the conceptual model reviewed in Section 4.1, it is possible to extend the current underlying abstraction of the API to support local search by using construction mechanisms. This subject may potentially have a lot of research directions that are not immediately clear. However, a possible starting point would be an attempt in merging both local (in *nasf4nio* [38]) and constructive search APIs.

# References


[1] MALLBA Library v2.0. `https://neo.lcc.uma.es/mallba/easy-mallba/`. Last Modified: 2013-12-17, Accessed: 2021-01-17.

[2] Hash Code Archive. `https://codingcompetitions.withgoogle.com/hashcode/archive`. Accessed: 2021-01-18.

[3] E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, et al. MALLBA: A Library of Skeletons for Combinatorial Optimisation. In *European Conference on Parallel Processing*, pages 927–932. Springer, 2002.

[4] E. Alba, G. Luque, J. Garcia-Nieto, G. Ordonez, and G. Leguizamon. MALLBA: A Software Library to Design Efficient Optimisation Algorithms. *International Journal of Innovative Computing and Applications*, 1(1):74–85, 2007.

[5] F. Almeida, D. Gonzalez, and I. Pelaez. Parallel Dynamic Programming. *Parallel Combinatorial Optimization*, 58:29, 2006.

[6] P. Amar, M. Baillieul, D. Barth, B. LeCun, F. Quessette, and S. Vial. Parallel Biological in Silico Simulation. In *Information Sciences and Systems 2014*, pages 387–394. Springer, 2014.

[7] T. Bäck, D. B. Fogel, and Z. Michalewicz. Handbook of Evolutionary Computation. *Release*, 97(1):B1, 1997.

[8] M. Basseur, L. Jourdan, and E. Talbi. Toward Parallel Design of Hybrids Between Metaheuristics and Exact Methods. *Parallel Combinatorial Optimization*, 58:163, 2006.

[9] R. Bellman. Dynamic Programming. *Science*, 153(3731):34–37, 1966.

[10] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 2015.

[11] J. J. Bentley. Fast Algorithms for Geometric Traveling Salesman Problems. *ORSA Journal on Computing*, 4(4):387–411, 1992.

[12] J. Bhasker and T. Samad. The Clique-Partitioning Problem. *Computers & Mathematics with Applications*, 22(6):1–11, 1991.

[13] C. Blum and G. R. Raidl. *Hybrid Metaheuristics: Powerful Tools for Optimization*. Springer, 2016.

[14] C. Blum and A. Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[15] C. Blum, A. Roli, and M. Dorigo. HC-ACO: The Hyper-Cube Framework for Ant Colony Optimization. In *Proceedings of MIC*, volume 2, pages 399–403, 2001.

[16] I. Boussaïd, J. Lepagnot, and P. Siarry. A Survey on Optimization Metaheuristics. *Information Sciences*, 237:82–117, 2013.

[17] J. Brownlee et al. OAT: The Optimization Algorithm Toolkit. *Complex Intelligent Systems Laboratory (CIS), Centre for Information Technology Research (CITR), Faculty of Information and Communication Technologies (ICT), Swinburne University of Technology, Victoria, Australia, Technical Report A*, 20071220, 2007.

[18] B. Bullnheimer, R. F. Hartl, and C. Strauss. A New Rank Based Version of the Ant System – A Computational Study. *Working Papers SFB "Adaptive Information Systems and Modelling in Economics and Management Science"*, 1997.

[19] S. Cahon, N. Melab, and E.-G. Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[20] J. Clausen. Branch and Bound Algorithms – Principles and Examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[21] W. J. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver. *The Traveling Salesman Problem*, chapter 7, pages 241–271. John Wiley & Sons, Ltd, 1997.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[23] T. G. Crainic, B. Le Cun, and C. Roucairol. Parallel Branch-and-Bound Algorithms. *Parallel Combinatorial Optimization*, 1:1–28, 2006.

[24] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a Large-Scale Traveling-Salesman Problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.

[25] L. Di Gaspero and A. Schaerf. EasyLocal++: An Object-Oriented Framework for the Flexible Design of Local-Search Algorithms. *Software: Practice and Experience*, 33(8):733–765, 2003.

[26] A. Djerrah, S. Jafar, V.-D. Cung, and P. Hahn. Solving Quadratic Assignment Problem on Cluster With a Bound of Reformulation Linearization Techniques. 2005.

[27] A. Djerrah, B. Le Cun, V.-D. Cung, and C. Roucairol. Bob++: Framework for Solving Optimization Problems With Branch-and-Bound Methods. In *2006 15th IEEE International Conference on High Performance Distributed Computing*, pages 369–370. IEEE, 2006.

[28] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[29] M. Dorigo and L. M. Gambardella. Ant Colonies for the Travelling Salesman Problem. *Biosystems*, 43(2):73–81, 1997.

[30] M. Dorigo and L. M. Gambardella. Ant Colony System: A Cooperative Learning Approach to the Travelling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[31] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[32] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, 2004.

[33] I. Dorta, C. León, and C. Rodríguez. Performance Analysis of Branch-and-Bound Skeletons. In *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 8–pp. IEEE, 2006.

[34] M. Ehrgott and X. Gandibleux. Bound Sets for Biobjective Combinatorial Optimization Problems. *Computers & Operations Research*, 34(9):2674–2694, 2007.

[35] T. A. Feo and M. G. Resende. A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem. *Operations Research Letters*, 8(2):67–71, 1989.

[36] T. A. Feo and M. G. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6(2):109–133, 1995.

[37] 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)*, pages 1–20. IEEE, 2014.

[38] C. M. Fonseca et al. Not Another Software Framework for Nature-Inspired Optimisation (nasf4nio). `https://github.com/cmfonseca/nasf4nio`, 2019. Accessed: 2021-10-31.

[39] C. Gagné and M. Parizeau. Genericity in Evolutionary Computation Software Tools: Principles and Case-Study. *International Journal on Artificial Intelligence Tools*, 15(02):173–194, 2006.

[40] F. Galea and B. Le Cun. Bob++: A Framework for Exact Combinatorial Optimization Methods on Parallel Machines. In *International Conference High Performance Computing & Simulation*, pages 779–785, 2007.

[41] F. Glover and M. Laguna. Tabu Search. In *Handbook of Combinatorial Optimization*, pages 2093–2229. Springer, 1998.

[42] R. Grymin and S. Jagiełło. Fast Branch and Bound Algorithm for the Travelling Salesman Problem. In *IFIP International Conference on Computer Information Systems and Industrial Management*, pages 206–217. Springer, 2016.

[43] P. Hansen and N. Mladenović. Variable Neighborhood Search: Principles and Applications. *European Journal of Operational Research*, 130(3):449–467, 2001.

[44] J. F. Herrera, J. M. Salmerón, E. M. Hendrix, R. Asenjo, and L. G. Casado. On Parallel Branch and Bound Frameworks for Global Optimization. *Journal of Global Optimization*, 69(3):547–560, 2017.

[45] L. Hoong Chuin, W. Wee Chong, and L. Min Kwang. A Development Framework for Rapid Meta-heuristics Hybridization. In *Proceedings of the 28th Annual International Computer Software and Applications Conference: COMPSAC 2004: 28-30 September, 2004, Hong Kong*, page 362. Citeseer, 2004.

[46] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier, 2004.

[47] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[48] T. Kuhn, C. M. Fonseca, L. Paquete, S. Ruzika, M. M. Duarte, and J. R. Figueira. Hypervolume Subset Selection in Two Dimensions: Formulations and Algorithms. *Evolutionary Computation*, 24(3):411–425, 2016.

[49] A. H. Land and A. G. Doig. An Automatic Method for Solving Discrete Programming Problems. In *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer, 2010.

[50] H. C. Lau, W. C. Wan, S. Halim, and K. Toh. A Software Framework for Fast Prototyping of Meta-Heuristics Hybridization. *International Transactions in Operational Research*, 14(2):123–141, 2007.

[51] E. L. Lawler and D. E. Wood. Branch-and-Bound Methods: A Survey. *Operations research*, 14(4):699–719, 1966.

[52] J. D. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An Algorithm for the Travelling Salesman Problem. *Operations Research*, 11(6):972–989, 1963.

[53] H. R. Lourenço, O. C. Martin, and T. Stützle. Iterated Local Search. In *Handbook of Metaheuristics*, pages 320–353. Springer, 2003.

[54] S. Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at `http://cs.gmu.edu/~sean/book/metaheuristics/`.

[55] V. Maniezzo. Exact and Approximate Nondeterministic Tree-Search Procedures for the Quadratic Assignment Problem. *INFORMS Journal on Computing*, 11(4):358–369, 1999.

[56] R. Martí and G. Reinelt. *The Linear Ordering Problem: Exact and Heuristic Methods in Combinatorial Optimization*, volume 175. Springer Science & Business Media, 2011.

[57] J. P. Martins and C. M. Fonseca. Constructive Neighborhoods as Practical Models for Combinatorial Optimization Problems. Unpublished Manuscript, 10 pages, 2016.

[58] T. Menouer and B. Le Cun. A Parallelization Mixing OR-Tools/Gecode Solvers on Top of the Bobpp Framework. In *2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 242–246. IEEE, 2013.

[59] T. Menouer and B. Le Cun. Anticipated Dynamic Load Balancing Strategy to Parallelize Constraint Programming Search. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*, pages 1771–1777. IEEE, 2013.

[60] T. Menouer and B. Le Cun. Adaptive N to P Portfolio for Solving Constraint Programming Problems on Top of the Parallel Bobpp Framework. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 1531–1540. IEEE, 2014.

[61] P. S. Ow and T. E. Morton. Filtered Beam Search in Scheduling. *The International Journal Of Production Research*, 26(1):35–62, 1988.

[62] A. Panangadan and R. Korf. Incremental Evaluation of Minimum Spanning Trees for the Traveling Salesman Problem.

[63] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Courier Corporation, 1998.

[64] J. A. Parejo, J. Racero, F. Guerrero, T. Kwok, and K. A. Smith. FOM: A Framework for Metaheuristic Optimization. In *International Conference on Computational Science*, pages 886–895. Springer, 2003.

[65] J. A. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernandez. Metaheuristic Optimization Frameworks: A Survey and Benchmarking. *Soft Computing*, 16(3):527–561, 2012.

[66] V. Phan and S. Skiena. An Improved Time-Sensitive Metaheuristic Framework for Combinatorial Optimization. In *International Workshop on Experimental and Efficient Algorithms*, pages 432–445. Springer, 2004.

[67] V. Phan, P. Sumazin, and S. Skiena. A Time-Sensitive System for Black-Box Combinatorial Optimization. In *Workshop on Algorithm Engineering and Experimentation*, pages 16–28. Springer, 2002.

[68] G. Reinelt. TSPLIB – A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[69] M. G. Resende and C. C. Ribeiro. Greedy Randomized Adaptive Search Procedures: Advances and Extensions. In *Handbook of Metaheuristics*, pages 169–220. Springer, 2019.

[70] R. Ruiz and T. Stützle. A Simple and Effective Iterated Greedy Algorithm for the Permutation Flowshop Scheduling Problem. *European Journal of Operational Research*, 177(3):2033–2049, 2007.

[71] T. Stützle. Local Search Algorithms for Combinatorial Problems. *Darmstadt University of Technology PhD Thesis*, 20, 1998.

[72] T. Stützle and H. H. Hoos. MAX-MIN Ant System and Local Search for the Travelling Salesman Problem. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)*, pages 309–314. IEEE, 1997.

[73] T. Stützle and H. H. Hoos. MAX-MIN Ant System. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[74] F. J. Vasko, R. S. Barbieri, B. Q. Rieksts, K. L. Reitmeyer, and K. L. Stott Jr. The Cable Trench Problem: Combining the Shortest Path and Minimum Spanning Tree Problems. *Computers & Operations Research*, 29(5):441–458, 2002.

[75] A. C. Vieira. *Uma Plataforma para a Avaliação Experimental de Meta-heurísticas*. PhD thesis, Faculdade de Ciências e Tecnologia da Universidade do Algarve, 2009.

[76] S. Voß and D. L. Woodruff. Optimization Software Class Libraries. In *Optimization Software Class Libraries*, pages 1–24. Springer, 2003.

[77] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[78] X. Yu and M. Gen. *Introduction to Evolutionary Algorithms*. Springer Science & Business Media, 2010.