Tommaso Urli

Hybrid Meta-Heuristics for Combinatorial Optimization

Ph.D. Thesis

Università degli Studi di Udine
Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica
Dottorato di Ricerca in Ingegneria Industriale e dell'Informazione

Typeset in LaTeX.
Self-published in March 2014.
The front page platypus illustration is by Tommaso Urli.

author
Tommaso Urli

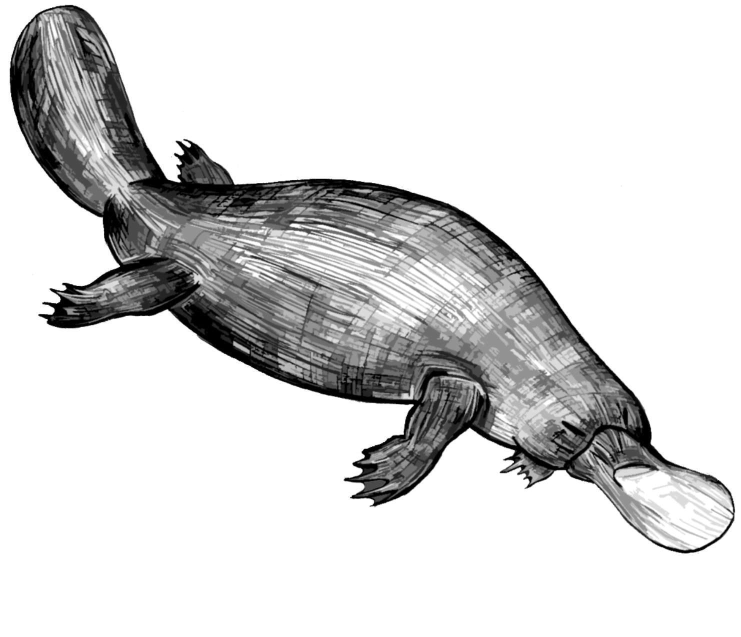
address
Via delle Scienze, 208
33100 – Udine (UD)
Italy

web http://www.diegm.uniud.it/urli
mail tommaso.urli@uniud.it

Tommaso Urli

Hybrid Meta-Heuristics for Combinatorial Optimization

Ph.D. Thesis



Advisor
Prof. Luca Di Gaspero

To my family.

Acknowledgements

Before getting to the central topic of this thesis, I'd like to thank a bunch of people for sharing with me this three-years-long journey.

First of all, I wish to thank my advisor, Luca, one of the most knowledgeable persons I know, for the many hours of fruitful discussion, the countless pair programming sessions, and for always promoting my personal initiative. I also wish to thank my other advisor, Andrea, for the many research advice, for conveying me the value of perseverance, and for always helping me to put difficulties in the right perspective. Particular thanks go also to my friend, colleague, and office mate Sara, for pushing me to apply for a position at what ended up to be a great research unit. I wish to thank you all for being such nice and fun people to work with.

Special thanks go to Andrea Roli and Nysret Musliu for accepting to review this thesis, and for providing useful insight on how to improve it.

I wish to thank Amos, for the endless and passionate discussions about what can, and cannot be, considered proper music, and for being a good friend. I also wish to thank Roberto, which initially talked me into starting a doctoral career, and Ruggero, for sharing with me his statistics knowledge.

I wish to thank Markus Wagner, one of the hardest working person I have ever met, for being a great co-author and friend, and Frank Neumann, for hosting me in his wonderful research unit in Adelaide. I wish to thank Andrea, for being one of the best co-authors I have worked with. I am also obrigado to Tiago de Oliveira Januario, for being an early adopter (and beta tester) of JSON2RUN, and for supporting me in my quest to learn Portuguese.

I wish to thank Martina, for teaching me the value of always aiming high, and Paolo, for being the best friend that I have ever had. I got from the two of you more than you, and I, could imagine.

Finally, I wish to thank my family. My parents, Erme and Norberto, who never denied me an opportunity to make an experience, and my sister, Valentina, for always backing me. You are truly extraordinary people, and I love you.

Contents

Introduction

Combinatorial optimization problems arise, in many forms, in various aspects of everyday life. Nowadays, a lot of services are driven by optimization algorithms, enabling us to make the best use of the available resources while guaranteeing a level of service. Examples of such services are public transportation, goods delivery, university time-tabling, and patient scheduling.

Thanks also to the *open data* movement, a lot of usage data about public and private services is accessible today, sometimes in aggregate form, to everyone. Examples of such data are traffic information (Google), bike sharing systems usage (CitiBike NYC), location services, etc. The availability of all this body of data allows us to better understand how people interacts with these services. However, in order for this information to be useful, it is necessary to develop tools to extract knowledge from it and to drive better decisions. In this context, optimization is a powerful tool, which can be used to improve the way the available resources are used, avoid squandering, and improve the sustainability of services.

The fields of meta-heuristics, artificial intelligence, and operations research, have been tackling many of these problems for years, without much interaction. However, in the last few years, such communities have started looking at each other's advancements, in order to develop optimization techniques that are faster, more robust, and easier to maintain. This effort gave birth to the fertile field of hybrid meta-heuristics.

In this thesis, we analyze some of the most common hybrid meta-heuristics approaches, and show how these can be used to solve hard real-world combinatorial optimization problems, and what are the major advantages of using such techniques.

This thesis is based on results obtained by working together with many local and international researchers, and published in a number of peer-reviewed papers. In the following, I summarize the major findings, and refer to the relative publications.

Hyper-heuristics. Chapter 6, describes the development of an adaptive optimization algorithm, which is able to automatically drive itself through the search space, by using the collected knowledge to change its behavior in time. The chapter is based on the following two papers about reinforcement learning-based hyper-heuristics

- Luca Di Gaspero and Urli Tommaso. **A reinforcement learning approach to the cross-domain heuristic search challenge**. In *Proceedings of the 9th Metaheuristics International Conference (MIC 2011)*. 2011.
- Luca Di Gaspero and Urli Tommaso. **Evaluation of a family of reinforcement learning cross-domain optimization heuristics**. In *Learning and Intelligent Optimization (LION 6)*. 2012.

Balancing bike sharing systems. Chapter 8 describes the mathematical models that we developed to optimize the re-balancing operations for bike sharing systems, and Chapter 7 describes two propagation-based meta-heuristics that we developed to operate on such models. These two chapters are based on the following papers

- Luca Di Gaspero, Andrea Rendl, and Tommaso Urli. **A Hybrid ACO+CP for Balancing Bicycle Sharing Systems**. In *Proceedings of HM'13: 8th International Workshop on Hybrid Metaheuristics*. 2013.
- Luca Di Gaspero, Andrea Rendl, and Tommaso Urli. **Constraint based approaches for Balancing Bike Sharing Systems**. In *Proceedings of CP'13: the 19th International Conference on Principles and Practice of Constraint Programming*. 2013.

Moreover, a follow-up journal paper [87] about our latest work on this problem has been recently submitted for acceptance to the *Constraints* journal.

Curriculum-based course timetabling. Chapter 9 describes models and algorithms developed to tackle the problem of generating university timetables, and is based on the following papers

- Ruggero Bellio, Sara Ceschia, Luca Di Gaspero, Andrea Schaerf, and Tommaso Urli. **A simulated annealing approach to the curriculum -based course timetabling problem**. In *Proceedings of MISTA '13: the 6th Multidisciplinary International Scheduling Conference: Theory and Applications.* 2013.
- Tommaso Urli. **Hybrid CP+LNS for the Curriculum-Based Course Timetabling Problem.**. In *Doctoral Program of CP'13: the 19th International Conference on Principles and Practice of Constraint Programming.* 2013.

Other contributions. Chapter 10 describes a number of other research projects I have contributed to in my doctoral career. In particular, it describes some of the results obtained with the Evolutionary Computation research unit at the University *of* Adelaide, Australia, in the field of theoretical runtime analysis of Genetic Programming (machine learning, multi-objective optimization), and in the field of virtual camera control (3D graphics, continuous optimization). The chapter is based on the following papers

- Tommaso Urli, Markus Wagner, and Frank Neumann. **Experimental Supplements to the Computational Complexity Analysis of Genetic Programming for Problems Modelling Isolated Program Semantics**. In *Proceedings of PPSN 2012 - 12th International Conference on Parallel Problem Solving From Nature.* 2012.
- Anh Quang Nguyen, Tommaso Urli, and Markus Wagner. **Single- and Multi-Objective Genetic Programming: New Bounds for Weighted Order and Majority**. In *Post-proceedings of FOGA 2013 - Foundation of Genetic Algorithms.* 2013.

- Roberto Ranon, Marc Christie, and Tommaso Urli. **Accurately Measuring the Satisfaction of Visual Properties in Virtual Camera Control**. In *Proceedings of Smart Graphics 2010.* 2010.
- Roberto Ranon and Tommaso Urli. **Improving the efficiency of Viewpoint Composition**. *IEEE Transactions on Visualization and Computer Graphics.* 2014.

Additionally, a number of software tools have been developed and made public, as a fundamental element of my research. All these tools are distributed under the permissive MIT license.

- Gecode-LNS, a search engine for the Gecode constraint system, to perform large neighborhood search (LNS) based on a CP model. Available at https://bitbucket.org/tunnuz/gecode-lns.
- Gecode-ACO, a search engine for the Gecode constraint system, to perform an ant colony optimization (ACO)-driven CP sarch. Available at https://bitbucket.org/tunnuz/gecode-aco.
- json2run [105], a tool that allows to automate the design, running, and analysis of computational experiments, as well as perform parameter tuning. Available at https://bitbucket.org/tunnuz/json2run.
- CP-CTT [104], a hybrid CP-based large neighborhood search (LNS) solver for the Curriculum-Based Course Timetabling (CB-CTT). Available at https://bitbucket.org/tunnuz/cpctt.
- GPFramework, an extensible Java framework built to study the runtime analysis of tree-based Genetic Programming (GP). Available at https://bitbucket.org/tunnuz/gpframework.

The thesis is organized in three major parts, which cover different aspects of our research. Part I describes the needed background in order to address the central topic of this thesis. In particular, Chapter 1 describes the domain of combinatorial optimization, Chapter 2 is a brief introduction to one of the most popular exact approaches to combinatorial optimization problems, namely Constraint Programming, Chapter 3 describes the main ideas and the most popular algorithms in Neighborhood Search, a family of practical techniques to tackle large optimization problems, finally Chapter 4 presents Swarm Intelligence algorithms, a class of nature-inspired and population-based algorithms. Part II introduces the central topic of this thesis, namely hybrid Meta-Heuristics. Specifically, Chapter 5 gives a brief overview of the existing families of hybrid approaches, Chapter 6 describes the research we carried out in the field of learning Hyper-Heuristics, Chapter 7 describes the research carried out at the intersection between neighborhood search and constraint programming, and neighborhood search and swarm intelligence. Part III is devoted to the discussion of the application of the investigated approaches to real-world combinatorial optimization problems, namely the rebalancing of bike sharing systems (Chapter 8), and the optimization of university time-tables (Chapter 9). Chapter 10 describes some additional contributions in the field of optimization that are not directly related to the main topic of the thesis. Additionally, the thesis features two Appedices that address some important aspects of the research we carried out, namely reinforcement learning and parameter tuning.

Part I

Background

Chapter 1

Combinatorial Optimization

In this chapter, we outline the domain of combinatorial optimization, and present its core concepts. Moreover, we introduce a representative example problem that will be used throughout the Part I of this thesis.

1.1 Terminology

Before proceeding to the rest of this chapter, we must agree on some terminology. In particular, we need to define, at least informally, the fundamental notions of problem, problem instance, decision variable, constraint, solution, and combinatorial problem.

1.1.1 Problems, variables and constraints

In optimization, a *problem* Π (also called *problem formulation*) is a mathematical representation of a decision process, built so that it is possible for a software *solver* to reason about it, and come up with one or more possible decisions. Problems are stated in terms of *decision variables* (or simply *variables*), representing the decisions to be taken, and *constraints*, restrictions on the values that can be given to the variables. Coming up with an effective mathematical representation of a problem, i.e. *modeling* the problem, is one of the fundamental tasks in optimization.

Example 1.1 (Sudoku). Sudoku is a popular puzzle game played on a 9-by-9 board splitted into nine 3-by-3 subregions. Each match starts with a board in which $k < 9^2$ cells have already been pre-filled with digits in $\{1, \ldots, 9\}$. The goal of the game is to fill in each remaining cell, according to the following rules

1. on each row a digit may only be used once,
2. on each column, a digit may only be used once,
3. on each subregion, a digit may only be used once.

A Sudoku puzzle can be represented with $n = 9^2 - k$ decision variables, where k is the number of cells that have been pre-filled. Every time a digit is chosen for a cell, the set of possible values for the other cells on the same row, column, or subregion, is reduced.

1.1.2 Instances

While a problem formulation is a general description of a decision process, e.g., the set of rules that define the game of Sudoku, a *problem instance* (or simply *instance*) $\pi \in \mathbf{I}_\Pi$ is a specific occurrence of the problem. With reference to Example 1.1, a Sudoku instance specifies the values of k cells (see Figure 1.1).

Sometimes it is convenient to use the term *problem* to refer to a problem instance. In the following, the meaning will be clear from the context.

1.1.3 Solutions and search space

A *solution* s to an instance π is an assignment of all n decision variables (i.e., a *n-tuple*). For example, a solution to a Sudoku instance is one that assigns a digit to every cell of the board. The *search space* $\mathbf{S}_\pi$ of an instance is the set of all possible assignments to its decision variables, i.e., the Cartesian product of its variables domains, which are normally bounded. A distinction is usually made between the set $\mathbf{F}_\pi \subseteq \mathbf{S}_\pi$ of solutions that satisfy all the imposed constraints

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 1.1: A Sudoku instance with $k = 30$.

(*feasible* solutions), and the solutions which violate some of the constraints (*infeasible* solutions). When not otherwise specified, we will use the term *solution* to indicate a generic solution (i.e., $s \in \mathbf{S}_\pi$); if $s \in \mathbf{F}_\pi$, it will be clear from the context, or explicitly stated.

Whenever an instance of a problem has at least one feasible solution, we call it *feasible*. Similarly, an instance without feasible solutions is called *infeasible*.

1.1.4 Combinatorial problems

A *combinatorial* problem is one whose decision variables have discrete and finite domains. As a consequence, a combinatorial problem has a finite number of solutions, although typically exponential in the number of variables. Once again, Sudoku is an example of a combinatorial problem, with 9^{9^2-k} possible solutions, most of which are, however, infeasible.

Note in the following, we will use the symbols $\mathbf{S}, \mathbf{F}, \mathbf{I}, \ldots$, instead of $\mathbf{S}_\pi, \mathbf{F}_\pi, \mathbf{I}_\Pi, \ldots$, whenever the problem, or problem instance, under discussion is generic, or clear from the context.

1.2 Decision and search problems

Given a problem instance, there are various questions that can be answered using a solver. At the most basic level, one may be interested in finding out whether an instance has at least one feasible solution. Answering to such question is commonly referred to as the *decision problem* associated with the instance. With respect to a decision problem, instances can be classified into *yes* instances ($\mathbf{I}_{yes} \subseteq \mathbf{I}$), instances for which the decision problem has a positive answer, and *no* instances ($\mathbf{I}_{no} = \mathbf{I} \setminus \mathbf{I}_{yes}$), instances for which the decision problem has a negative answer. Note that some decision problems can be *undecidable*, i.e., it is not possible to construct an algorithm that halts on all instances giving a positive or negative answer. Among such problems are *semi-decidable* problems, i.e., problems for which it is possible to write an algorithm that halts on *yes* instances and might run forever on *no* instances.

On a pragmatic level, a more useful task is to compute one or more feasible solutions for the instance. We call this task the *search problem* associated with the instance. Of course, finding a feasible solution for an instance also implies solving the associated decision problem. The converse is not necessarily true, as, in principle, it might be possible to produce a proof of feasibility for a problem, without providing a solution to it.

1.3 Complexity

Complexity theory deals with classifying problems, by analyzing the amount of resources (time and space) needed to solve them using well defined computational models. In this brief section, we present two of these models, namely (deterministic) Turing machines (TM) and Nondeterministic Turing machines (NTM), which are at the basis of the characterization of the $\mathcal{P}$ and $\mathcal{NP}$ problem classes.

1.3.1 Languages and strings

An **alphabet** $\mathbf{\Sigma}$ is a finite set of symbols with at least two elements. A **string** w of length k over the alphabet $\mathbf{\Sigma}$ is an element $\langle w_1, \ldots, w_k \rangle$ of the Cartesian product $\mathbf{\Sigma}^k$, in which commas and brackets are usually dropped. The set of strings of any length over the alphabet $\mathbf{\Sigma}$, including the empty string ϵ, is called the **Kleene's closure** of $\mathbf{\Sigma}$, and denoted by $\mathbf{\Sigma}^*$. A **language** $\mathbf{L}$ over $\mathbf{\Sigma}$ is a subset of the Kleene's closure of $\mathbf{\Sigma}$. $\mathbb{B}$ is the set $\{0, 1\}$.

1.3.2 Problems and languages

In the following, we give a formal definition of a decision problem, extend it to include arbitrary problems, and clarify how a problem can be represented by a language[1].

Definition 1.1 (Decision problem [91]). *Let $\mathbf{\Sigma}$ be an arbitrary finite alphabet. A* ***decision problem*** *Π is defined by a set of instances $\mathbf{I}_\Pi \subseteq \mathbf{\Sigma}^*$ of the problem, and a condition $\phi_\Pi : \mathbf{I}_\Pi \mapsto \mathcal{B}$ that has value 1 on* yes *instances and 0 on* no *instances. Then, $\mathbf{I}_{\Pi_{yes}} = \{\pi \in \mathbf{I}_\Pi \,|\, \psi_\Pi(\pi) = 1\}$ are the* yes *instances, and $\mathbf{I}_\Pi \setminus \mathbf{I}_{\Pi_{yes}}$ are the* no *instances.*

Definition 1.2 (Language associated with a decision problem [91]). *The* yes *intances of a decision problem Π are encoded as binary strings by an* ***encoding function*** *$\sigma : \mathbf{\Sigma}^* \mapsto \mathcal{B}^*$, that assigns to each $\pi \in \mathbf{I}_{\Pi_{yes}}$ a string $\sigma(\pi) \in \mathcal{B}^*$. With respect to σ, the* ***language*** *$\mathbf{L}(\Pi)$* ***associated with a decision problem*** *Π is the set $\mathbf{L}(\Pi) = \{\sigma(\pi) \,|\, \pi \in \mathbf{I}_{\Pi_{yes}}\}$.*

Definition 1.3 (Problem, language [91]). *A decision problem can be generalized to a* ***problem*** *Π characterized by a function $f : \mathcal{B}^* \mapsto \mathcal{B}^*$ described by a set of ordered pairs $(w, f(w))$ where each string $w \in \mathcal{B}^*$ appears once as the left-hand side of the pair. Thus, a* ***language*** *is defined by problems $f : \mathcal{B}^* \mapsto \mathcal{B}$ and consists of the strings w on which $f(w) = 1$.*

[1]most of the definitions in this section are adapted from [91].

Note that the following results on languages hold for languages associated with problems as well. As a consequence, they can be used to classify problems based on the amount of resources needed to process their associated languages.

1.3.3 Turing machines

The Turing machine is a classical computation model (an hypothetical device), designed to study the limits of computing. To date no other computational model has been found, that computes functions that a Turing machine cannot compute.

A Turing machine is composed by a (possibly unlimited) **tape** of m-bits cells, and by a **control unit** (a finite states machine) capable of reading from the cell under its head, writing to the cell under its head, and moving left or right on the tape. At each unit of time, the control unit reads a word from the cell under its head, updates its state, write a word to the cell under its head, and moves left or right, by at most one position, on the tape. In the following we formally define the model.

Definition 1.4 (Turing machine [92]). *A **Turing machine (TM)** is a six-tuple $M = \langle \mathbf{\Gamma}, \beta, \mathbf{Q}, \delta, s, h \rangle)$, where Γ is the **tape alphabet** not containing the blank symbol β, $\mathbf{Q}$ is a finite set of* states *in which the control unit can be, $\delta : \mathbf{Q} \times (\mathbf{\Gamma} \cup \{\beta\}) \mapsto (\mathbf{Q} \cup \{h\}) \times \mathbf{\Gamma} \cup \{\beta\} \times \mathbf{L}, \mathbf{N}, \mathbf{R}$ is the **next-state function**, s is the **initial state**, and $h \notin \mathbf{Q}$ is the **accepting halt state**. If M is in state q with a symbol a under the head and $\delta(q, a) = (q\prime, a\prime, \mathbf{C})$, then the control unit enters state $q\prime$, writes $a\prime$ in the cell under its head and moves the head left, right or not at all, if $\mathbf{C}$ is, respectively, $\mathbf{L}$, $\mathbf{R}$, or $\mathbf{N}$.*

A Turing machine M **accepts the input string** $w \in \mathbf{\Gamma}^*$ if, when started in state s with w placed left-adjusted on its otherwise blank tape, and the head at the leftmost tape cell, the last state entered by M is h. Every other halting state is a **rejecting** state. A Turing machine M **accepts the language** $\mathbf{L}(M) \subseteq \mathbf{\Gamma}^*$ consisting of all strings accepted by M. If a Turing machine halts on all inputs, we say that it **recognizes the language**.

Nondeterministic Turing machines (NTM) are identical to standard Turing machines, except for the fact that the control unit accepts an external choice input to determine the next state.

Definition 1.5 (Nondeterministic Turing machine [92]). *A **nondeterministic Turing machine (NTM)** is the extension of the TM model by the addition of a choice input to its control unit. Thus a NTM is seven-tuple $M = \langle \Sigma, \Gamma, \beta, Q, \delta, s, h \rangle)$, where $\mathbf{\Sigma}$ is the **choice input alphabet**, and $\delta : \mathbf{Q} \times \mathbf{\Sigma} \times (\mathbf{\Gamma} \cup \{\beta\}) \mapsto (\mathbf{Q} \cup \{h\}) \times (\mathbf{\Gamma} \cup \{\beta\}) \times \mathbf{L}, \mathbf{N}, \mathbf{R} \cup \{\perp\}$ is its **next-state function**. If $\delta(q, c, a) = \perp$ then there is no successor for the current state with input choice c. If M is in state q with a symbol a under the head, reading choice input c, and $\delta(q, c, a) = (q\prime, a\prime, \mathbf{C})$, then the control unit enters state $q\prime$, writes $a\prime$ in the cell under its head and moves the head left, right or not at all, if $\mathbf{C}$ is, respectively, $\mathbf{L}$, $\mathbf{R}$, or $\mathbf{N}$. At each step, the NTM M reads one symbol of its **choice input string** $c \in \mathbf{\Sigma}^*$.*

A nondeterministic Turing machine M **accepts the input string** $w \in \mathbf{\Gamma}^*$ if **there is** a $c \in \mathbf{\Sigma}^*$ such that the last state entered by M is h when M is started in a state s with w left-adjusted on its otherwise blank tape and the head at the leftmost tape cell. A nondeterministic Turing machine M **accepts the language** $\mathbf{L}(M) \subseteq \mathbf{\Gamma}^*$ consisting of those strings w that it accepts. Therefore, if $w \notin \mathbf{L}(M)$, there is no choice input for which M accepts w.

1.3.4 $\mathcal{P}$ and $\mathcal{NP}$ classes

Turing machines allow us to formally define the class of $\mathcal{P}$ languages (and thus problems).

Definition 1.6 ($\mathcal{P}$ [92]). *A language $\mathbf{L} \subseteq \mathbf{\Gamma}^*$ is in $\mathcal{P}$ if there is a Turing machine M with tape alphabet $\mathbf{\Gamma}$ and a polynomial $p(n)$ such that, for every $w \in \mathbf{\Gamma}^*$, a) M halts in $p(|w|)$ steps, and b) M accepts w if and only if $w \in \mathbf{L}$.*

Similarly, Nondeterministic Turing machines allow us to define the class of $\mathcal{NP}$ languages (and thus problems).

Definition 1.7 ($\mathcal{NP}$ [92]). *A language $\mathbf{L} \subseteq \mathbf{\Gamma}^*$ is in $\mathcal{NP}$ if there* *is a nondeterministic Turing machine M and a polynomial $p(n)$ such that M accepts $\mathbf{L}$, and for each $w \in \mathbf{L}$ there is a choice input $c \in \mathbf{\Sigma}^*$ such that M on input w with this choice input halts in $p(|w|)$ steps. The choice input c is said to **verify** the membership of the string in a language.*

The membership of a string in $\mathcal{NP}$ can thus be verified in polynomial time, with a choice input string of polynomial length.

1.3.5 $\mathcal{P}$-complete and $\mathcal{NP}$-complete classes

While it is obvious that $\mathcal{P} \subseteq \mathcal{NP}$ (because it is possible to simulate a deterministic Turing machine on a Nondeterministic Turing Machine by fixing a choice input string), it is not known whether the opposite stands, which would imply that $\mathcal{P}$ and $\mathcal{NP}$ are the same class. This question, denoted $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$, has been open for decades, and it is possibly the most important unsolved question in computer science. The approach taken to answer this question, is to identify problems that are hardest in $\mathcal{NP}$, and then try to prove that they are, or are not, in $\mathcal{P}$.

Definition 1.8 (Complete language). *A language $\mathbf{L_0}$ is **hardest** in its class, if i) $\mathbf{L_0}$ is itself in the class, and ii) for every other language $\mathbf{L}$ in the class, a test for the membership of a string w in $\mathbf{L}$ can be constructed by translating w with an algorithm to a string v and then testing for membership o v in $\mathbf{L_0}$. If the class is $\mathcal{P}$, the algorithm must use at most a space logarithmic in the size of w, if the class is $\mathcal{NP}$, the algorithm must use at most time polynomial in the length of w. If these conditions hold, $\mathbf{L_0}$ is said to be a **complete language** for its class.*

We now give the formal definitions for $\mathcal{P}$-complete and $\mathcal{NP}$-complete languages (and thus problems).

Definition 1.9 ($\mathcal{P}$-complete language [92]). *A language $\mathbf{L_0} \subseteq \mathbb{B}^*$ is $\mathcal{P}$**-complete** if it is in $\mathcal{P}$ and, for every language $\mathbf{L} \subseteq \mathbb{B}^*$ in $\mathcal{P}$ there is a log-space deterministic program that translates each $w \in \mathbb{B}^*$ into a string $v \in \mathbb{B}^*$ so that $w \in L$ if and only if $v \in \mathbf{L_0}$.*

Definition 1.10 ($\mathcal{NP}$-complete language [92]). *A language* $\mathbf{L_0} \subseteq \mathbb{B}^*$ *is* $\mathcal{NP}$***-complete*** *if it is in* $\mathcal{NP}$ *and, for every language* $\mathbf{L} \subseteq \mathbb{B}^*$ *in* $\mathcal{NP}$ *there is a polynomial-time deterministic program that translates each* $w \in \mathbb{B}^*$ *into a string* $v \in \mathbb{B}^*$ *so that* $w \in \mathbf{L}$ *if and only if* $v \in \mathbf{L_0}$.

If a $\mathcal{NP}$-complete problem were found to be in $\mathcal{P}$, this would mean that every problem in $\mathcal{NP}$ would be in $\mathcal{P}$, and thus the question $\mathcal{P} \overset{?}{=} \mathcal{NP}$ would have a positive answer. Since many decades of research have failed to show this, the classic approach when faced with a complex problem, is to prove that it is $\mathcal{NP}$-complete. This is a *testimonial*, but *not a proof*, of its complexity. In particular, to date, the only general algorithm to solve instances of such problems, is the exhaustive enumeration of solutions, which has a worst-case run time exponential in the size of the input.

1.4 Optimization problems

In most combinatorial problems, an instance may have more than one feasible solution. While for search and decision problems is enough to find *any* solution that satisfies the constraints, optimization problem require to find a solution that is better than the others according to some measure.

1.4.1 Objective function

The natural way to evaluate the goodness of a solution with respect to the others is to define an *objective function* to measure the utility of the solution in the original decision process.

Definition 1.11 (Objective function). *An* objective function *is a function* $f : \mathbf{F} \mapsto \mathbb{R}$, *that associates a value to each solution* $s \in \mathbf{F}$. *Such function can encode a quality measure for* s *that must be maximized (in this case we call it* ***maximization function****), or a penalty that must be minimized (in this case we call it* ***minimization function****). We refer to* $f(s)$ *as the* objective value *of a solution* $s \in \mathbf{F}$.

Note that, in some communities, the objective function can assume different names. For instance, in the field of evolutionary computation, the term *fitness function* is broadly used to indicate the underlying maximization function, because of its parallel with natural sciences, e.g., survival of the *fittest*. In other communities the term *cost function* is used, to denote a “defect” of the solution, which should be minimized through the optimization process. In such cases, we refer to the objective value, respectively, as to the *fitness* or the *cost* value.

It can be shown that solving minimization or maximization problems is, in fact, equivalent, since it is trivial to translate a problem of one class into a problem of the other. As such, without loss of generality, we follow the convention used in most optimization communities, and treat every objective function as a cost function, and every problem as minimization problem.

Note, also, that every optimization problem can be treated as a decision problem where the goal is to tell whether there is a feasible solution so that $f(s) < K$ where K is a given threshold.

Definition 1.12 (Combinatorial optimization problem [36]). *An instance $\pi \in \mathbf{I}_\Pi$ of a combinatorial optimization problem Π, is a triple $\langle \mathbf{S}, \mathbf{F}, f \rangle$, where $\mathbf{S}$ is the (finite) search space of π, $\mathbf{F} \subseteq \mathbf{S}$ is the set of all feasible solutions of π, and f is an objective function associated with the problem.*

Solving the *optimization problem* associated with an instance π, consists in finding a *global optimum* for π, i.e., a solution $s^* \in \mathbf{F}$, so that $f(s^*) \leq f(s), \forall s \in \mathbf{F}$ (note that there might still exist an infeasible $u^* \in \mathbf{S} \setminus \mathbf{F}$ $s.t.$ $f(u^*) \leq f(s^*)$).

1.4.2 Hard and soft constraints

In many problem formulations, constraints are split in *hard constraints* and *soft constraints*. Hard constraints represent restrictions of the original decision process, that cannot be violated for any reason, e.g., laws of physics, regulations. Their violation prevents a solution from being feasible at all, and thus they define the boundaries of the search space.

Soft constraints, on the other hand, *may* be violated, as their violation does not hinder feasibility, but at a price of a higher solution cost. As such, their violations must be minimized by the optimization process. Soft constraints are by far the most common way to define an objective function. For this reason they are sometimes called *cost components*.

1.4.3 Multi-objective optimization

In multi-objective optimization (MOO), instead of just one objective function f, we have a *m-tuple* $\langle f_1, \dots, f_m \rangle$ of objectives. As a consequence, each solution s has an associated *objective vector* $\langle f_1(s), \dots, f_m(s) \rangle$ measuring its cost in each objective. A solution $s_1 \in \mathbf{F}$ is said to *Pareto-dominate* another solution $s_2 \in \mathbf{F}$, if $f_i(s_1) \leq f_i(s_2), \forall i \in \{1, \dots, m\} \wedge \exists j \in \{1, \dots, m\}$ $s.t.$ $f_j(s_1) < f_j(s_2)$, i.e., if it is at least as good as s_2 in all objectives, and better than s_2 in at least one. Note that *Pareto dominance* defines a partial order on the solutions, therefore some of the solutions may be incomparable.

The goal, in multi-objective optimization, is to find a set $\mathbf{F}^* \subseteq \mathbf{F}$ of solutions that are *non Pareto-dominated*. This set of solutions is commonly denoted *Pareto optimal set*, and the respective set of objective vectors is called the *Pareto front*. Solutions in the Pareto optimal set are incomparable with each other. Moreover, they exhibit *Pareto efficiency*, i.e., it is impossible to improve one objective without making another worse.

1.4.4 Scalarization

Most optimization problems arising in real-world domains have multiple objectives. Solving a multi-objective optimization problem, however, requires more effort with respect to a single-objective one. To deal with this issue, a common practice is to *scalarize* the problem, i.e., to combine together the objectives, so to turn the multi-objective problem to a single-objective one.

A popular (but not the only) way to scalarize a multi-objective problem is to use m *weights* $\{w_1, \dots, w_m\}$ to linearly combine the various objective values in a single value

$$f(s) = \sum_{i=0}^{m} w_i \cdot f_i(s).$$

The introduction of weights as a means to transform hard constraints into components of the objective function is often called *Lagrangian relaxation*. Note that, by construction, when performing a linear scalarization, some points int the Pareto optimal set will not be optima anymore. For instance, a solution $s^* \in \mathbf{F}^*$ of the multi-objective problem that dominates all the others in one single objective, might not be an optimum of the single-single objective problem because of the choice of weights.

1.4.5 Example: Traveling Salesperson Problem

The Traveling Salesperson Problem (TSP) is a well-known combinatorial optimization problem, which deals with finding the shortest round trip to serve n customers, so that each customer is visited exactly once. In the TSP, the (hard) constraints are that the route be a round trip, and that it visits each customer exactly once; the objective is that the route be minimal, i.e., the length of the route is the cost of a solution.

TSP can be modeled with a weighted, undirected, and (usually) complete graph $G = \langle \mathbf{V}, \mathbf{E} \rangle$, where each node $v \in \mathbf{V}$ represents a customer, each edge $e = (v_1, v_2) \in \mathbf{E}, v_1, v_2 \in \mathbf{V}$ represents the *leg* from v_1 to v_2 and the weight w_e of each edge represents the resources (time or distance) needed to get from one customer to another. Given G, the TSP consists in finding the Hamiltonian cycle of minimal length. The decision problem associated with an instance seeks whether it is possible to find an Hamiltoninan cycle s, subject to the hard constraint that

$$\sum_{e \in s} w_e < K$$

where K is a positive integer.

The TSP is an interesting problem for a number of reasons. First, it stands in the notable class of NP-hard problems, and its associated decision problem is NP-complete. Second, the TSP, sometimes even in its standard formulation, appears as a subproblem in many other combinatorial optimization domains, such as logistics, DNA sequencing, planning and chip design.

In the next chapters, we will provide various models for the TSP, which will allow us to solve it using different methods.

1.5 Search methods

There are many ways of classifying search methods for optimization. For instance, one may make a distinction between population-based algorithms and algorithms that only keep track of one solution, or discriminate algorithms that can explore discrete search spaces from algorithms that operate in the continuous domain.

In this section, we present two useful criteria for classifying search methods, respectively capturing *i*) what is known about the found solutions, and *ii*) how the solutions are built.

1.5.1 Complete and incomplete methods

When solving an optimization problem, an important aspect in the choice of the search method is the kind of guarantees it provides about the found solutions. *Complete methods* explore the search space exhaustively, keeping track of the best solutions they find. There are three main advantages in using such methods. First, since the search is exhaustive, if a solution exists, a complete method is guaranteed to find it. Second, the best solution found, when the search stops, is also the guaranteed optimum, i.e. the best solution in the search space. Third, it is always possible to know when to stop search. Unfortunately, since most interesting problems in combinatorial optimization are $\mathcal{NP}$-complete, the exhaustive exploration of the search space can have an exponential worst-case run time, with respect to the size of the input, which is considered infeasible for practical applications. Moreover, in many real-world domains, it is often unnecessary to cerficiate that a solution is optimal, because a good one is enough. Among complete methods are, Linear Programming (LP) [35], Logic Programming and Constraint Programming [89] (see Chapter 2).

A possible solution to the performance issue is the use of *incomplete methods*. Such methods explore the search space in a non-exhaustive (and, possibly, stochastic) way, usually improving an *incumbent solution* iteratively. In general, incomplete methods can not offer any guarantee about the quality of the solutions found, because they cannot detect when the search space has been completely explored, which might never happen. On the other hand, they have two important advantages. First, since they do not have to be exhaustive, they can explore promising regions of the search space much earlier than complete methods. Second, since they can move freely through the search space, without sticking to a fixed systematic exploration rule, they usually exhibit better *anytime properties*, i.e., the more time is given to a search method, the better is the returned solution. Examples of incomplete methods are domain-specific greedy heuristics, and stochastic neighborhood search [57] (see Chapter 3).

In most complete approaches, the exploration of the search space is *exhaustive*, but not a mere *enumeration* of all the solutions. A common technique to implement this is to keep an incumbent solution, and try to prove that specific regions of the search space cannot contain better solutions, in order to avoid them (*pruning*). However, if the *search strategy* is unfortunate for the problem being solved, exhaustive or almost exhaustive search can occur.

Figure 1.2 shows a brief classification of the most popular optimization techniques.

1.5.2 Perturbative and constructive methods

Another possible way of classifying search methods is based on how they build solutions. *Constructive methods* start from an completely or partially unassigned solution, and iteratively set all of the variables according to some policy. Among constructive methods are Constraint Programming (Chapter 2) and Ant Colony Optimization (ACO) [45] (see Section 4.2). On the other hand, *perturbative methods* start from a complete solution, obtained randomly or using a greedy heuristic, and generate new ones by changing the values of some variables. The initialization step can either generate

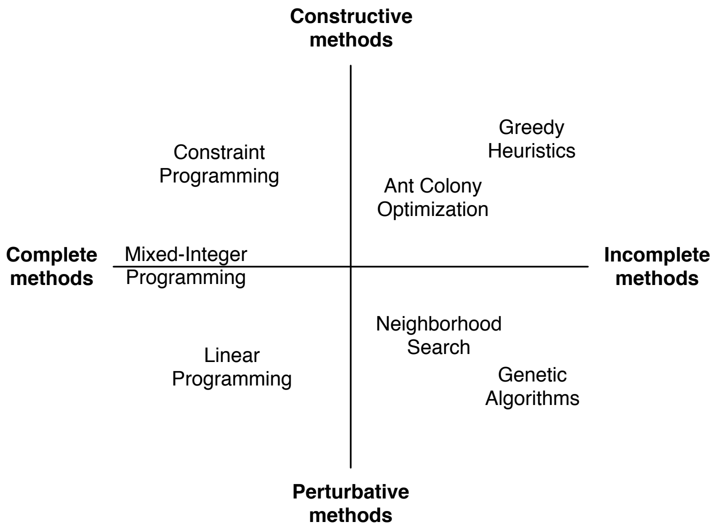


Figure 1.2: Classification of some search methods.

a feasible solution, or an infeasible one. All neighborhood search methods (see Chapter 3) are of this kind. The advantage of perturbative methods is that, no matter when they are stopped, they can always provide a complete solution.

Conclusions

We have described the domain of combinatorial optimization, which is where our research stands. We have the origin of the implicit complexity of the optimization problems that belong to this class. We have described a relevant example of combinatorial optimization problem, namely the TSP, that will be used in the following sections to show how the modeling and resolution is carried out in different search paradigms. Moreover, we provided a brief but useful taxonomy of search methods for combinatorial optimization, pointing out their advantages and disadvantages.

Chapter 2

Constraint Programming

Constraint Programming (CP) [89] is a *declarative* search paradigm for combinatorial optimization. A problem is modeled in CP by declaring *which* are the decision variables, what are their domains, and which combinations of values are forbidden.

Unlike other methods, such as neighborhood search (see Chapter 3), in which modeling and search are tightly intertwined, CP embraces the principle of *separation of concerns* to decouple modeling from search. Once a problem is modeled, (virtually) no information about *how* the solutions are to be found is needed in order to solve it. As such, the search for a solution in CP relies mostly on general techniques for reasoning about constraints (propagation, see Section 2.2.1), in conjunction with backtracking (see Section 2.2.2).

Referring to the classification in Section 1.5, CP is traditionally a complete and constructive approach. However, by virtue of the decoupling between modeling and search, many alternative search methods, not necessarily complete or constructive, have been devised.

2.1 Modeling

In this section, we give some necessary definitions[1], and describe the available facilities for modeling problems.

2.1.1 Variables, domains, and constraints

In the following, we assume that the domains of variables are finite and discrete (as per Definition 1.12), and thus, without loss of generality, we represent them as finite subsets of $\mathbb{Z}$. Given a set of variables $X = \{x_1, \ldots, x_n\}$, we denote the domain of each x_i by $D(x_i)$ (D_i, for short), and the Cartesian product of their domains by $\bigtimes_{x_i \in X} D_i \subseteq \mathbb{Z}^n$.

Definition 2.1 (Constraint). *A* constraint c *is a relation defined on a sequence of variables* $X(c) = \langle x_{i_1}, \ldots, x_{i_{|X(c)|}} \rangle$, *called the* scope *of* c. c *is the subset of* $\mathbb{Z}^{|X(c)|}$ *that contains the combination of values (or* tuples*)* τ *that satisfy* c. $|X(c)|$ *is called the* arity *of* c. *Testing whether a tuple* τ *satisfies a constraint is called a* constraint check.

A constraint may be specified *extensionally*, by listing all the tuples that satisfy it, or *intensionally*, i.e., through a formula, e.g., $x_1 < x_2 + 5$. We will sometimes write $c(x_1, \ldots, x_n)$ for a constraint c with scope $X(c) = \langle x_1, \ldots, x_n \rangle$. A constraint whose scope consists of two variables is called *binary*. A constraint defined over one variable is called *unary*. Note that posting a unary constraint on a variable is equivalent to modifying its domain, however domains are traditionally given in extension.

2.1.2 Problems and solutions

Now we proceed defining the concepts of *constraint satisfaction problem* (CSP), where the goal is to find a solution that satisfies all the constraints, and of *constraint optimization problem* (COP), in which, additionally, an objective function must be optimized.

A convenient way to represent a CSP is as a constraint network, i.e., a *hyper-graph* (see Figure 2.1) where nodes represent the vari-

[1]Many of the definitions in this section are adapted from [11].

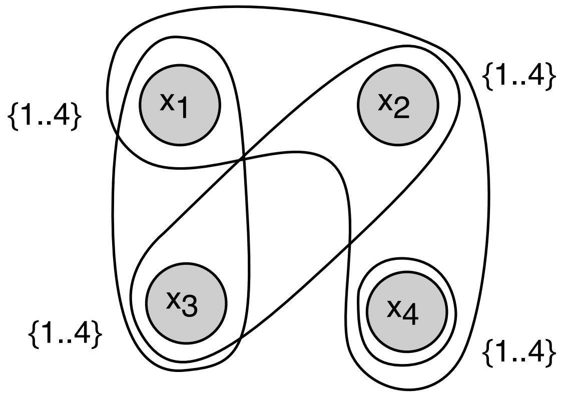


Figure 2.1: A hyper-graph representing a constraint network with variables $X = \{x_1, \ldots, x_4\}$ and constraints $C = \{(x_4 = 2), (x_1 = x_3), (x_3 \geq x_2), (x_1 + x_2 + x_4 \leq 5)\}$.

ables, and *hyper-edges* (edges possibly connecting more than two nodes) represent constraints between them. In the following, we will often use the terms *CSP* and *constraint network* (or *network*) interchangeably.

Definition 2.2 (Constraint satisfaction problem (CSP)). *A* ***constraint satisfaction problem*** *(or* ***constraint network****, or simply* ***network****) N is a triple $\langle X, D, C\rangle$, where X is a n-tuple of variables $X = \langle x_1, \ldots, x_n\rangle$, D is a corresponding n-tuple of domains $D = \langle D_1, \ldots, D_n\rangle$ such that $x_i \in D_i$, and C is a t-tuple of constraints $C = \langle C_1, \ldots, C_t\rangle$, that specify what values or combination of values are allowed for the variables.*

Whenever the ordering of variables, domains, or constraints is not important, we will use sets, e.g., $\{x_1, \ldots, x_n\}$ instead of tuples, e.g., $\langle x_1, \ldots, x_n\rangle$, to simplify the notation. Moreover, we will often refer to the variables, domains, and constraints in a network N with, respectively, X_N, D_N, and C_N.

We now define some operators on variables, that will be used throughout this chapter.

Definition 2.3 (Restriction, projection, intersection, union, and join). *Given a tuple τ on a sequence X of variables, and given $Y \subseteq X$, $\tau[Y]$ denotes the* ***restriction*** *of τ to the variables in Y (modulo reordering). Given a constraint c and a sequence $Y \subseteq X(c)$, $\pi_Y(c)$ denotes the* ***projection*** *of c on Y, i.e., the relation with scheme Y that contains tuples that can be extended to a tuple on $X(c)$ satisfying c. Given two constraints c_1 and c_2 sharing the same scheme $X(c_1) = X(c_2)$, $c_1 \cap c_2$ (resp. $c_1 \cup c_2$) denotes the* ***intersection*** *(resp.* ***union****) of c_1 and c_2, i.e., the relation with scheme $X(c_1)$ that contains tuples τ satisfying both c_1 and c_2 (resp. satisfying c_1 or c_2). Given a set of constraints $\{c_1, \ldots, c_k\}$, $\bowtie_{j=1}^{k} c_j$ (or $\bowtie \{c_1, \ldots, c_k\}$) denotes the* ***join*** *of the constraints, i.e., the relation with scheme $\bigcup_{j=1}^{k} X(c_j)$ that contains the tuples τ such that $\tau[X(c_j)] \in c_j$ for all $j, 1 \leq j \leq k$.*

We can now formally define a solution to a CSP problem.

Definition 2.4 (Instantiation, solution). *Given a constraint network $N = \langle X, D, C \rangle$, an* ***instantiation*** *I on $Y = \langle x_1, \ldots, x_k \rangle \subseteq X$ is an assignment of values $\langle v_1, \ldots, v_k \rangle$ to the variables $x_i \in Y$. An instantiation I on Y is* ***valid*** *if, $\forall x_i \in Y, I[x_i] \in D_i$. An instantiation I on Y is* ***locally consistent*** *if 1) is valid, and 2) for all $c \in C$ s.t. $X(c) \subseteq Y$, $I[X(c)]$ satisfies c . A* ***solution*** *to a network N is an instantiation I on X which is locally consistent. The set of all solutions of N is denoted by $sol(N)$. An instantiation I on Y is* ***globally consistent*** *(or just* ***consistent****) if it can be extended to a solution, i.e., $\exists s \in sol(N)$ with $I = s[Y]$.*

A CSP is a search problem, where the feasible solutions are the locally consistent assignments to its variables. As such, we can define the associated optimization problem by considering an additional objective function.

Definition 2.5 (Constraint optimization problem (COP)). *A* ***constraint optimization problem*** *is a network $N = \langle X, D, C \rangle$, where $\langle X, D, C \rangle$ is a CSP, and $f : \bigtimes_{x_i \in X} D_i \mapsto \mathbb{R}$ is an* objective function, *defined on the variables in X, that assigns an* objective value *to* *every solution of N. An* ***optimal solution*** *to a minimization (resp. maximization) COP is a solution $s^* \in sol(N)$ that minimizes (resp. maximizes) the value of f.*

Referring to the terminology introduced in Section 1.1, the search space of a constraint optimization problem $N = \langle X, D, C, f \rangle$ is the Cartesian product $\bigtimes_{x_i \in X} D_i$ of the domains of its variables, the hard constraints are represented by the constraints $c_i \in C$, and the soft constraints are embedded in the cost function f.

2.1.3 Global constraints

Particularly useful, when modeling a constraint programming problem, are *global constraints*, i.e., constraints that can involve an arbitrary number of variables, and that capture common subproblems. The importance of such constraints is twofold. First, they make modeling more concise, because they describe complex relations between variables, which, alternatively, would require the use of many elementary constraints. Second, they bear a performance advantage, because they come with specific propagation algorithms that allow to eliminate inconsistent values from the variables domains much faster than with general constraint propagation.

Most constraint programming systems, e.g., GECODE [96], MINIZINC, or CHOCO, implement a broad set of global constraints[2], which should be used as much as possible in modeling.

Examples of very common global constraints are the alldifferent $(x_1, \ldots, x_k)$ constraint, which states that all the specific variables take different values, and the gcc (global cardinality constraint, or counting constraint), which ensures that, given a set of variables and a set of values, the number of variables that instantiate to a certain value is between some specified lower and upper bounds (possibly specific to each value). Another common global constraint is the element$(\texttt{a}, \texttt{i}, \texttt{v})$ constraint, which takes an array of variables a, an index variable i, and a free variable v, and enforces the equality $\texttt{v} = \texttt{a}[\texttt{i}]$.

[2] A complete and up-to-date catalog of global constraints can be found at http://www.emn.fr/z-info/sdemasse/gccat.

2.1.4 Example: Traveling Salesperson Problem

We show how the TSP, introduced in Section 1.4.5, can be modeled in a constraint programming system. We present two different models, one built upon elementary constraints, the other fully exploiting the global constraints available in Gecode. At the end of the section, we report a brief comparison of performance.

In both models, a solution involving n customers, is represented by an array (next) of n decision variables with domains $D_i = \{1, \ldots, n\}, i \in \{1, \ldots, n\}$, each one representing the *next* customer to visit after the current one. For instance, if $\texttt{next}_4 = 9$, then customer 9 will be visited right after customer 4 in the tour. Moreover, as customary in Gecode, we use one auxiliary variable c to represent the solution cost, and an array (cost) of n auxiliary variables, to represent the costs of the *legs* going from each i to $\texttt{next}_i$. A global constraint

$$\texttt{c} = \texttt{sum}(\texttt{cost})$$

connects (in both models) the local cost of each leg to the global cost of the soution.

Model with elementary constraints

Since all customers must be visited, and we have n decision variables, the first step is ensuring that all next are assigned different values. We achieve this by posting $(n-1)^2/2$ non-equality constraints.

$$\texttt{next}_i \neq \texttt{next}_j, \forall i, j \in \{1, \ldots, n\}, i < j \tag{2.1}$$

The second step is to ensure that, for each customer i, the next customer $\texttt{next}_i$ is different from itself. This can be done by posting n more non-equality constraints.

$$\texttt{next}_i \neq i, \forall i \in \{1, \ldots, n\} \tag{2.2}$$

The matrix of distances between customers is represented as a one dimensional array (*distance*) of size n^2, where the cost of the leg $(i, \texttt{next}_i)$ is at the index $i \cdot n + \texttt{next}_i$ (Gecode convention).

Since the chosen legs, and thus the indices, depend on the solution being evaluated, we need n linear constraints, to store this index in n auxiliary variables $\mathtt{e}_{(i,\mathtt{next}_i)}$.

$$\mathtt{e}_{(i,\mathtt{next}_i)} = n \cdot i + \mathtt{next}_i, \forall i \in \{1, \ldots, n\} \tag{2.3}$$

Once the $\mathtt{e}$ variables are constrained, it is possible to use n element constraints to collect the local costs of each leg in the cost variables.

$$\mathsf{element}(\mathit{distance}, \mathtt{e}_{(i,\mathtt{next}_i)}, \mathtt{cost}_i), \forall i \in \{1, \ldots, n\} \tag{2.4}$$

Model with global constraints

This model replaces all the constraints posted in Equations 2.1, 2.2, 2.3, and 2.4, with a single circuit global constraint

$$\mathsf{circuit}(\mathtt{next}, \mathit{distance}, \mathtt{cost}, \mathtt{c})$$

which ensures, at the same time, that the next variables form a round trip, and that the costs of the legs are stored and accumulated, respectively, in the cost and c variables. Moreover, our second model features a redundant alldifferent constraint (dubbed distinct in Gecode), to help the propagation.

$$\mathsf{distinct}(\mathtt{next})$$

The use of global constraints makes the model much more readable, conveying the actual meaning of the constraints, and improving the performances of the model, thanks to the specific propagation algorithms.

As an example, on a randomly generated problem with 10 customers, and distances $d_{i,j} \in \{5, \ldots, 20\}, i, j \in \{1, \ldots, 10\}, i \neq j$, the elementary constraints model generates 110 propagators, and finds the best solution with 8000+ constraint propagations. On the same problem, the global constraints model generates 14 propagators, and finds the optimal solution in less than 3000 propagations.

2.2 Search

A common way to solve constraint optimization problems is through the interleaved cooperation of *constraint propagation*, whose aim is to reduced the size of the search space, and to detect subregions of the search space that cannot provably contain feasible solutions, and *backtracking* algorithms, which explore the reduced search space. In this section we will outline both techniques.

2.2.1 Constraint propagation

The role of constraint propagation is, given a network N, to generate a *tightening* N' of N, i.e., a network “smaller” than N so that $sol(N) = sol(N')$. Intuitively, a smaller network can be explored in shorter time by a backtracking algorithm.

In the following, we give a formal definition of these concepts.

Definition 2.6 (Preorder $\preceq$ on networks). *Given two networks N and N', we say that $N' \preceq N$ if and only if $X_{N'} = X_N$ and any instantiation I on $Y \subseteq X_N$ locally inconsistent in N is locally inconsistent in N' as well.*

As a consequence, given the two networks in Definition 2.6, $N' \preceq N$ if and only if $X_{N'} = X_N$, $D_{N'} \subseteq D_N$, and for any constraint $c \in C_N$, for any tuple τ on $X(c)$ that does not satisfy c, either τ is invalid in $D_{N'}$ or there exists a constraint $c' \in C_{N'}$, $X(c') \subseteq X(c)$, such that $\tau[X(c')] \notin c'$. That is, given an inconsistent instantiation τ on N, either τ is inconsistent in N' because its domains are more restrictive, or because there is a constraint making it inconsistent.

Definition 2.7 (Tightenings of a network). *The space $\mathcal{P}_N$ of all possible* ***tightenings*** *of a network $N = \langle X, D, C\rangle$ is the set of networks $N' = \langle X, D', C'\rangle$ such that $D' \subseteq D$ and for all $c \in C, \exists c' \in C'$ with $X(c') = X(c)$ and $c' \subseteq c$. We denote by $\mathcal{P}_N^{sol}$ the set of tightenings of N that preserve $sol(N)$.*

The set of networks $\mathcal{P}_N$, together with $\preceq$, forms a preordered set. The top element of $\mathcal{P}_N$ according to $\preceq$ is N itself, while the bottom elements are the networks with empty domains.

Definition 2.8 (Globally consistent network). *Let $N = \langle X, D, C\rangle$ be a network, and $G_N = \langle X, D_G, C_G\rangle$ be a network in $\mathcal{P}_N^{sol}$ so that $G_N \preceq N', N' \in \mathcal{P}_N^{sol}$, then any instantiation I on $Y \subseteq X$ which is locally consistent in G_N can be extended to a solution of N. G_N is called a* ***globally consistent network****.*

Unfortunately, a globally consistent network is in general exponential both in size and in time to compute. The constraint propagation approach to this problem, is to approximate G_N with another element of $\mathcal{P}_N^{sol}$ that can be computed at a more reasonable cost.

This is accomplished by defining a local consistency property ϕ, and enforcing ϕ-consistency on the network through *propagators* (or *filtering algorithms*). While, in general, tightenings of a network can be generated either by restricting its domains, by restricting its constraints, or by adding new constraints, most constraint propagation techniques are domain-based.

Definition 2.9 (Domain-based tightenings). *The space $\mathcal{P}_{ND}$ of* ***domain-based tightenings*** *of a network $N = \langle X, D, C\rangle$ is the set of networks in $\mathcal{P}_N$ with the same constraints as N. That is, $N' \in \mathcal{P}_{ND}$ if and only if $X_{N'} = X$, $D_{N'} \subseteq D$, and $C_{N'} = C$. We denote by $\mathcal{P}_{ND}^{sol}$ the set of domain-based tightenings of N that preserve $sol(N)$.*

Definition 2.10 (Partial order $\leq$ on networks). *Given a network N, the relation $\preceq$ restricted to the set $\mathcal{P}_{ND}$ is a partial order (denoted by $\leq$).*

In principle, any property ϕ can be used as a domain-based local consistency notion. However, properties that are *stable under union* have particularly useful properties.

Definition 2.11 (Stability under union). *A domain-based property ϕ is* ***stable under union*** *if and only if, for any ϕ-consistent* *networks $N_1 = \langle X, D_1, C\rangle$ and $N_2 = \langle X, D_2, C\rangle$, the network $N' = \langle X, D_1 \cup D_2, C\rangle$ is ϕ-consistent.*

Definition 2.12 (ϕ-closure). *Let $N = \langle X, D, C\rangle$, be a network and ϕ be a domain-based local consistency. Let $\phi(N)$ be the network $\langle X, D_\phi, C\rangle$ where $D_\phi = \bigcup\{D' \subseteq D \mid \langle X, D', C\rangle \; is \; \phi\text{-consistent}\}$. If ϕ is stable under union, then $\phi(N)$ is ϕ-consistent and is the unique network in $\mathcal{P}_{ND}$ such that for any ϕ-consistent network $N' \in \mathcal{P}_{ND}$, $N' \leq \phi(N)$. $\phi(N)$ is called the **ϕ-closure** of N.*

Two properties of $\phi(N)$ make it particularly interesting. First, it preserves $sol(N)$. Second, it can be computed by a **fixpoint procedure**, by iteratively removing values that do not satisfy ϕ until no such value exists. A network N in which all constraints are ϕ-consistent is ϕ-consistent. Enforcing ϕ-consistency on a network N is equivalent to finding $\phi(N)$.

We now present some of the most important local consistency properties, which are implemented in many constraint systems.

Node consistency

The most basic form of local consistency is *node consistency*, which is defined on *unary* constraints, i.e., constraints on the individual domains of variables.

Definition 2.13 (Node consistency (N)). *A unary constraint c on the variable x_k with domain D_k is **node consistent** if, $\forall d_i \in D_k$, $d_i \in c$.*

Node consistency can be enforced using a propagator that sets $D_k = D_k \cap c$, and removes constraint c from the network (this is possible because c is now *embedded* in the domain). The complexity of this propagation on a network $N = \langle X, D, C\rangle$ is $\mathcal{O}(nd)$, where $d = max_{x_i \in X}(|D_i|)$ and $n = |X|$.

Arc consistency

Arc consistency is possibly the better-known form of local consistency. It is defined on normalized binary networks.

Definition 2.14 (Normalized, binary network). *A network $N = \langle X, D, C\rangle$ is **normalized** if $\forall c \in C, |X(c)| = 2$. A binary network is **normalized** if every pair of variables $(x_i, x_j), i \neq j$ appears in at most one constraint. As a consequence, in a normalized binary network we denote a constraint $c \in C$ s.t. $X(c) = \{x_i, x_j\}$ by c_{ij}. Also, c_{ij} and c_{ji} represent the same constraint.*

Definition 2.15 (Arc consistency (AC)). *A binary constraint c on the variabls x_1 and x_2 with respective domains D_1 and D_2 is **arc consistent** if, for all values $d_1 \in D_1, \exists d_2 \in D_2$ such that $(d_1, d_2) \in c$, and vice versa.*

AC can be generalized to constraints with arbitrary arity, in this case it is called generalized arc consistency (GAC).

Definition 2.16 (Generalized arc consistency (GAC)). *A constraint c on the variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots,$ D_n is called **generalized arc consistent** if, for each variable x_i and each value $d_i \in D_i$ there exists compatible values in the domains of all the other variables of c, that is, there exists a tuple $\tau \in c$ (also called a **support** for c) such that $\tau[x_i] = d_i$.*

In time, several algorithms for enforcing AC (and GAC) have been proposed. Most of them are based on Mackworth's AC3 [76]. Algorithm 1 describes the generalized AC3 algorithm (GAC3). Note that modern constraint systems may use slightly more complex propagators, that trade some time complexity for space complexity.

The time complexity of GAC3 is $\mathcal{O}(er^3d^{r+1})$, where $e = |C|$, $r = max_{c \in C}|X(c)|$, and $d = max_{x_i \in X}(|D_i|)$. Note that every network N can be transformed in an equivalent network composed only of binary constraints, on such network, the complexity of GAC3 (AC3) is $\mathcal{O}(ed^3)$.

Weaker local consistencies

Often, enforcing GAC on a network is not computationally feasible. As a consequence, a number of weaker local consistency notions have been proposed, that can be enforced at a lower computational

Algorithm 1 GAC3

```
procedure GAC3($N$)
    $Q \leftarrow \{(x_i, c) \mid c \in C_N, x_i \in X(c)\}$
    repeat
        $(x_i, c) \leftarrow$ pop($Q$)
        if Revise ($x_i$, $c$) then
            if $D_i = \emptyset$ then
                return false
            else
                $Q \leftarrow Q \cup \{(x_j, c') \mid c' \in C_N \wedge x_i, x_j \in X(c') \wedge j \neq i\}$
            end if
        end if
    until $Q \neq \emptyset$
    return true
end procedure

function Revise($x_i$, $c$)
    deleted $\leftarrow$ false
    for all $d \in D_i$ do
        if $\nexists \tau \in c \cap \pi_{X(c)}(D) \mid \tau[x_i] = d$ then
            $D_i \leftarrow D_i \{d\}$
            deleted $\leftarrow$ true;
        end if
    end for
    return deleted
end function
```

cost.

Definition 2.17 (Bound consistency (BC)). *A constraint* c *on the variables* $x_1, \ldots, x_n$ *with respective domains* $D_1, \ldots, D_n$ *is called* ***bound consistent*** *if, for each variable* x_i *and each value* $d_i \in \{min(D_i), \ldots, max(D_i)\}$ *there exist compatible values between the* min *and the* max *domain of all the other variables of c, i.e., there exists a value* $d_j \in \{min(D_j), \ldots, max(D_j)\}, \forall j \neq i$ *such that* $\langle d_1, \ldots, d_i, \ldots, d_n \rangle \in c$.

Definition 2.18 (Range consistency (RC)). *A constraint* c *on the variables* $x_1, \ldots, x_n$ *with respective domains* $D_1, \ldots, D_n$ *is called* ***range consistent*** *if, for each variable* x_i *and each value* $d_i \in D_i$ *there exist compatible values between the* min *and the* max *domain* *of all the other variables of c, i.e., there exists a value* $d_j \in \{min(D_j), \ldots, max(D_j)\}, \forall j \neq i$ *such that* $\langle d_1, \ldots, d_i, \ldots, d_n \rangle \in c$.

Stronger local consistencies

Similarly to the ones presented in the previous section, some stronger forms of local consistency have been defined. While (G)AC, BC, RC, and node consistency sort out values from single variables domanins, the following local consistency notions can be used to prune combinations (pairs, triples, etc.) of values, for this reason they are sometimes called *higher order* local consistencies.

Path consistency (PC) says that, if for a given pair of values $\langle d_i, d_j \rangle$ on a pair of variables $\langle x_i, x_j \rangle$, there exists a sequence of variables from x_i to x_j such that we cannot find a sequence of values for these variables starting at v_i and finishing at v_j, and satisfying all binary constraints along the sequence, then $\langle v_i, v_j \rangle$ is inconsistent. The following is the formal definition of PC.

Definition 2.19 (Path consistency). *Let* $N = \langle X, D, C \rangle$ *be a normalized network. Given two variables* x_i *and* x_j *in* X*, the pair of values* $\langle d_i, d_j \rangle \in D_i \times D_j$ *is* ***path consistent*** *if and only if, for any sequence of variables* $Y = \langle x_i = x_{k_1}, \ldots, x_{k_p} = x_j \rangle$ *such that for all* $q \in \{1, \ldots, p-1\}, c_{k_q,k_{q+1}} \in C$ *there exists a tuple of values* $\langle d_i = d_{k_1}, \ldots, d_{k_p} = d_j \rangle \in \pi_Y(D)$ *such that for all* $q \in \{1, \ldots, p-1\}, (d_{k_q}, d_{k_{q+1}}) \in c_{k_q,k_{q+1}}$.

It can be shown that it is sufficient to enforce PC on paths of length 2 (composed by 3 variables) in order to obtain the same level of local consistency as full PC. PC can be enforced in $\mathcal{O}(d^2n^3)$ time.

K-consistency ensures that each time we have a locally consistent instantiation of size $k-1$, we can consistently extend it to any kth variable.

Definition 2.20 (K-consistency). *Let* $N = \langle X, D, C \rangle$ *be a network. Given a set of variables* $Y \subseteq X$ *with* $|Y| = k-1$*,* I *on* Y *is* k***-consistent*** *if and only if, for any* k*th variable* $x_{i_k} \in X \setminus Y$ *there exists a value* $d_{i_k} \in D_{i_k}$ *such that* $I \cup \{(x_i, v_i)\}$ *is locally consistent. The network* N *is* k*-consistent if and only if, for any set* Y *of* $k-1$

variables, any locally consistent instantiation on Y is k-consistent.

Enforcing k-consistency has time complexity $\mathcal{O}(n^k d^k)$.

Partial order on domain-based local consistencies

It is possible to define a partial order on domain-based local consistency notions, that express how much they manage to go down the partially ordered set $(\mathcal{P}^{sol}_{ND}, \leq)$.

Definition 2.21 (Partial order $\preceq$ on local consistencies)**.** *A domain-based local consistency ϕ_1 is **at least as strong as** another local consistency ϕ_2 if and only if, for any network N, $\phi_1(N) \leq \phi_2(N)$. If, in addition, $\exists N'$ so that $\phi_1(N') < \phi_2(N')$, then ϕ_1 is **strictly stronger** than ϕ_2.*

According to this definition, GAC $\preceq$ RC $\preceq$ BC $\preceq$ N.

2.2.2 Backtracking

A constraint propagation step has three possible outcomes

1. propagation yielded *empty domains* for some of the variables, i.e., the instantiation being propagated is inconsistent with some constraint, and cannot be extended to a complete solution,
2. the propagation reduced the domain of each variable to one single element (*singleton*), i.e., a solution has been found, or
3. there are still multiple values in the domains of some of the variables.

The last situation is the most general, and it means that propagation alone is not sufficient to generate a complete solution, or to prove that none exists. In this case, *search* is needed.

Backtracking

Of course, the naïve search strategy, i.e., generating all the possible solutions and testing them for consistency, is not feasible in general, as it exhibit a $\mathcal{O}(d^n)$ worst-case time complexity. A better strategy consists in instantiating one variable at a time, performing constraint propagation (*pruning*) after each instantiation, and going back (*backtracking*) to the last consistent partial instantiation if empty domains are detected by propagation.

Such a search strategy can be seen as the depth-first exploration of a *search tree* (see Figure 2.2), where each inner node is a partial instantiation of the variables, each branch is an assignment of a value to one of the unassigned variables at the above node, i.e., a *branching constraint*, and leaves are either complete solutions or (complete or incomplete) inconsistent instantiation of the variables.

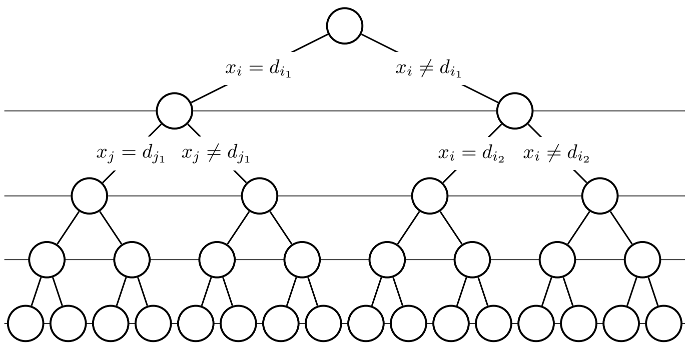


Figure 2.2: Example of a search tree

The role of propagation is both to reduce the domains of the variables, and thus the number of branches that must be explored, and to detect *dead ends*, i.e., nodes which cannot possibly lead to solutions. Moreover, since the cost of a solution is typically modeled as an auxiliary variable, propagation allows to refine the *upper* and *lower bounds* on the cost, enabling the use of cost-based pruning techniques, e.g., *branch & bound* (see Section 2.2.2).

The simplest backtracking algorithm is *cronological backtracking* (BT), which we outline in Algorithm 2. The algorithm is initially called with parameters $(U = X, L = \emptyset, C)$, where U is the set of unassigned variables (all in the beginning), L (called a *labeling*) is the set of assigned variables, and C is the set of constraints.

At each step, a variable x_i is selected using a *variable selection heuristic* (SelectVar), then its possible assignments are tried in order according to a *value selection heuristic* (SelectVal), consistency is tested through propagation (ConstraintPropagation), and then the search procedure is called recursively on the propagated network. If constraint propagation detects inconsistencies (in form of empty domains for some of the variables), the recursive call stack ensures that the search restarts from the last consistent instantiation.

The shape of the generated search tree depends on the *branching strategy* used to extend incomplete instantiations of the variables. Moreover, the performance of the search depend on the chosen variable and value selection heuristics.

Branching strategies

Traditionally, each branch in the search tree corresponds to posting a unary constraint on a chosen branching variable (albeit strategies involving non-unary constraints have been devised). There are several possible ways to to this

Enumeration the chosen variable x_i is assigned, in turn, each value in its domain. A branch is generated for each value, thus if after the previous propagation $k = |D_i|$, there will be k branches out of the node (this is the approach taken in Algorithm 2).

Binary choice points the chosen variable x_i is assigned a value $d_j \in D_i$, each node has two outgoing branches, one representing $(x_i = d_j)$, the other representing $(x_i \neq d_j)$.

Domain splitting the domain of variable x_i is reduced by selecting a value $d_j \in D_i$ and generating two branches $(x_i \leq d_j)$ and $(x_i > d_j)$.

Since enumeration can be simulated with binary choice points, and the latter has better theoretical properties, branching with binary choice points is the most common method in CP with finite domains (even though most constraint systems allow the definition of custom branching strategies). Domain splitting is used in

CP with real-valued variables [9], were assigning specific values to variables would not be feasible.

Variable and value selection heuristics

Variable and value selection heuristics play an important role in determining the performance of a CP solver on a problem. Unfortunately, the choice of the right heuristic depends greatly on the problem instance being solved. Indeed, if the problem has solutions, the selection heuristics should explore *promising values* of the variables first, so that the solutions be located about the left-most branch of the search tree, which is the first to be explored by the search. On the other hand, if the problem does not have any solution, selection heuristics should explore *unpromising variables*, so to prove, as soon as possible, that the subregion of the search space being explored (and, ultimately, the whole search space) cannot contain any solution.

The task of chosing the right heuristics is further complicated by the fact that, by the very nature of tree search, choices made near the root of the search tree have a great impact on the whole search process. In both cases, however, the chosen heuristic should try to minimize the overall size of the serch tree. Many selection heuristics are based on this principle. Here, we report some of the most popular ones.

Variable selection. *Static* heuristics are the less informed, and choose the variables according to orderings defined beforehand, e.g., the order in which they are declared, or a random permutation of it.

Dynamic heuristics, on the other hand, exploit information about the current or the past state of the search, and try to guess a way to obtain a smaller search tree. Among these we have the *first-fail* heuristic, that always chooses the variable with the smallest current domain. The rationale behind this heuristic, is that it is preferable to fail close to the root of the search tree, in order to prune larger subtrees. Similarly, the *max degree* heuristic chooses the variables based on their degree, i.e., number of constraints pending on them. Intuitively, the more constrained is a variable, the more likely is to fail. *Weighted* heuristics keep track of the number of failures generated by variables, and choose the most failing variables earlier. Various combinations of these heuristics exist.

Value selection. Similarly, static value selection heuristics always choose the value to assign according to fixed rules, e.g., smallest value, greatest value, random value, etc.

Among the dynamic heuristics are the ones based on the *reduced cost* that can be achieved by choosing a specific value.

Often, constraint systems allow to define *application specific* variable and value selection heuristics, that exploit some domain knowledge, in order to implement smarter strategies.

Nogood recording and explanations

One of the most effective techniques to improve propagation is by adding *implied* (or *redundant*) constraints. A constraint c is implied, for a network $N = \langle X, D, C \rangle$, if $sol(\langle X, D, C \rangle) = sol(\langle X, D, C \cup \{c\} \rangle)$.

Implied constraints can either be added to a network during the modeling phase (as we did for the alldifferent constraint in Section 2.1.4), or generated automatically after inconsistencies are found during the search [100].

Definition 2.22 (Nogood). *A **nogood** is a set of assignments and branching constraints that is not consistent with any solution.*

Definition 2.23 (Eliminating explanation). *Let $p = \{b_1, \ldots, b_j\}$ be a node in the search tree, and let $d \in D_i$ be a value that is removed from the domain of a variable x_i by constraint propagation at node p. An **eliminating explanation** (or simply **explanation**) for d, denoted expl$(x_i \neq d)$, is a subset (not necessarily proper) of p such that expl$(x_i \neq d) \cup \{x_i = d\}$ is a nogood.*

During the search, each dead end encountered corresponds to a nogood. It is therefore possible to add its negation to the network without changing the set of solutions. Of course, the smaller the nogood, the more effective the propagation. More useful (although not necessarily minimal) nogoods can be generated recursively by inspecting the dead end node.

Definition 2.24 (Jumpback nogood). *Let $p = \{b_1, \ldots, b_j\}$ be a dead end node in the search tree, where $b_i, 1 \leq i \leq j$ is the branching constraint posted at level i. The* ***jumpback nogood*** *for p, denoted by $J(p)$, is defined recursively as follows.*

- *p is a leaf node. Let x_k be a variable whose domain has become empty, and D_k being the initial domain of x_k*

$$J(p) = \bigcup_{d \in D_k} \mathit{expl}(x_k \neq d)$$

- *p is not a leaf node. Let $\{b^1_{j+1}, \ldots, b^k_{j+1}\}$ be all the possible extensions of p attempted by the branching strategy, each of which has failed*

$$J(p) = \bigcup_{i=1}^{k} J(p \cup \{b^i_{j+1}\}) - \{b^i_{j+1}\}$$

Nogoods can be generated during constraint propagation, by making propagators nogood-aware. Generating nogoods from global constraints is a bit more complicated, because if the constraints are used as black-boxes, many nogoods can be generated that sort out all, or almost all, the variables, which are not very useful. A more refined solution is to take into account the semantics of global constraint, and generate nogoods that capture it.

Advanced backtracking techniques

Cronological backtraking is a starting point for more sophisticated ways to explore a search tree.

Branch & bound (B&B). Branch & bound is a search technique that uses information about the objective function f of a COP to speed-up the search, by proving that some subregions of the search space cannot provably contain solutions better solutions than the current one. Recall that the inner nodes of a search tree represent incomplete instantiations of the decision variables, i.e., instantiations in which the domains of some of the variables have not yet been reduced to a single value. As a consequence, also the cost $c = f(s)$ of an incomplete solution s, can take a range of values D_c. We call $min(D_c)$ the *lower bound* of c and $max(D_c)$ the *upper bound* of c. In branch & bound, the solver keeps track of the best solution found during the search. Every time a new best solution is found, the solver dynamically post a constraint on the network, requiring the next solutions to have a lower cost with respect to the best one. Because of constraint propagation, subtrees in which the lower bound of the cost is higher than the cost of the current best solution can be safely pruned, because they cannot provably contain better solutions. Branch & bound thus effectively reduces the size of the search tree.

Backjumping (BJ). Backjumping is a backtracking techniques that exploits jumpback nogoods. Upon discovering a dead end, a backjumping procedure backtracks to the closest branching constraint that is responsible for dead end, where responsibility is discovered using nogoods.

For more details about advanced backtracking, see [107].

Conclusions

We have introduced constraint programming, a popular search paradigm belonging to the category of exact approaches. In particular, we have showed how to use the high-level constraints language to model a combinatorial optimization problem, and how the problem can be solved starting from this model. Finally, we tried to cover briefly the main aspects related to solving combinatorial problems with constraint programming, e.g., the choice of branching heuristics and the use of global constraints.

Algorithm 2 BT (chronological)

```
procedure BT($U, L, C$)
    if $U = \emptyset$ then
        return L
    end if
    $x_i \leftarrow$ SelectVar($U$)
    for all $d \leftarrow$ SelectVal($D_i$) do
        if Consistent($U \setminus \{x_i\}, \{x_i = d\} \cup L, C$) then
            $R \leftarrow$ BT($U \setminus \{x_i\}, \{x_i = d\} \cup L, C$)
            if $R \neq fail$ then
                return R
            end if
        end if
    end for
    return fail
end procedure

function Consistent($U, L, C$)
    ConstraintPropagation($U \cup L, C$)
    if $\exists x_i \in U \ s.t. \ D_i = \emptyset$ then
        return false
    end if
    for all $c \in C$ do
        if AllAssigned($L$) $\wedge L \notin c$ then
            return false
        end if
    end for
    return true
end function
```

Chapter 3

Neighborhood search

Neighborhood search (NS) is a family of incomplete and perturbative search methods. Such methods start from a candidate solution s, and iteratively look for a better candidate to replace it, among the ones in its *neighborhood.* The neighborhood $\mathcal{N}(s)$ of a candidate solution s is the set of solutions that can be generated by making some change to s. Such a change corresponds to a *local* movement in the search space, for this reason neighborhood search is often referred to as *local search* (LS).

Although they cannot guarantee that a solution is optimal, or that a solution exists at all, NS methods are extremely useful in practice, for several reasons. First, they are quite memory-efficient, since they only need to store a constant number of solutions (often just one or two). Second, they typically exhibit good anytime properties, reaching good solutions very fast, compared to most complete methods. Finally, they can be defined at a general (*meta*) level, so that they can be applied to many different problems, by just changing the neighborhood definition.

NS methods have been applied successfully to many combinatorial optimization problems, such as scheduling [28], timetabling [6], bin-packing [29], and vehicle routing [27].

3.1 Modeling

In addition to the the definition of a search space, NS modeling involves defining *neighborhood relations* in terms of *moves*.

Definition 3.1 (Neighborhood relation [36]). *Given a problem* Π*, an instance* $\pi \in \mathbf{I}_\Pi$*, and a search space* $\mathbf{S}$ *for it, we assign to each element* $s \in \mathbf{S}$ *a set* $\mathcal{N}(s) \subseteq \mathbf{S}$ *of neighboring solutions of* s*. The set* $\mathcal{N}(s)$ *is called the* ***neighborhood of*** s*, and each member* $s' \in \mathcal{N}(s)$ *is a* ***neighbor of*** s*.*

Typically, the neighborhood $\mathcal{N}(s)$ of s is not given explicitly, but rather defined implicitly in terms of *moves*, i.e., changes to s that generate all the solutions in $\mathcal{N}(s)$. For this reason, we will sometimes abuse of the notation and say $m \in \mathcal{N}(s)$, where m is a move transforming s in one of its neighbors.

During modeling, domain knowledge is introduced in the model by defining neighborhood relations that reflect the structure of the problem. When feasibility is enforced by the NS method, i.e., only feasible solutions can be generated, a move is commonly called *feasible* if it yields a feasible solution, and *infeasible* otherwise.

3.1.1 Delta costs

The primary metric employed by a NS method to choose whether to keep working on the current solution s, or to move to one of its neighbors $s' \in \mathcal{N}(s)$, is the difference

$$\Delta_f = f(s') - f(s)$$

between their cost values. Such value, often called the *delta cost*, is calculated for several thousands of candidates and incumbent solutions before the end of the optimization run. As such, calculating it efficiently is fundamental in order to obtain good performances.

Not surprisingly, computing the f-values for the two solutions and then subtracting them is not the most efficient way to compute the delta cost. A better technique involves considering some knowledge about the move m that transforms s in s', possibly in conjunction with data structures that store redundant data about s. We will use the following notation to denote the delta cost relative to a move m with respect to a solution s (regardless of how it is computed)

$$\Delta_f(s, m) = f(s \oplus m) - f(s).$$

where $\oplus$ denotes the application of a move to a solution, specifically $s \oplus m$ denotes the solution obtained by applying the move m to a solution s.

3.1.2 Example: Traveling Salesperson Problem

One possible NS model for the TSP involves representing a solution as a n-tuple of customers to be visited in order. Note that this solution semantic is different from the one of `next`, used in the CP model of Section 2.1.4; this facilitates the computation of delta costs. The search space is therefore the space of all permutations of the customers, and feasible moves are the ones that produce valid permutations.

Modeling with swap moves

One possible neighborhood, for a solution s, is the one defined by swap moves, i.e., the set of all solutions that can be obtained by choosing a customer in s and swapping it with another. For instance

$$\langle \mathbf{1}, 3, \mathbf{5}, 2, 4\rangle \overset{\text{swap}}{\mapsto} \langle \mathbf{5}, 3, \mathbf{1}, 2, 4\rangle$$

With such neighborhood relation, given the indices of two customers $i, j \in \{0, \ldots, n-1\}$, the cost function f, and let $d_{h,k}$ denote the distance between node h and node k, then the delta cost relative to the swap of the customers at indices i and j can be computed as

$$\begin{aligned}\Delta_f(s, \text{swap}(i,j)) = &- [d_{i-1,i} + d_{i,i+1} + d_{j-1,j} + d_{j,j+1}] \\ &+ [d_{i-1,j} + d_{j,i+1} + d_{j-1,i} + d_{i,j+1}]\end{aligned}$$

where $i-1$, $i+1$, $j-1$, and $j+1$ are considered modulo n.

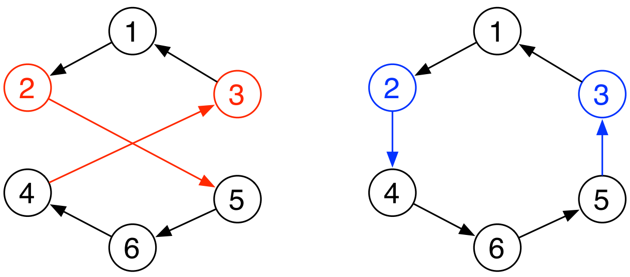


Figure 3.1: Example of 2-opt move.

Modeling with 2-opt moves

Another popular neighborhood is the so-called *2-opt*, where, given the indices of two customers $i, j \in \{0, \ldots, n-1\}$, the order of the customers served between them is inverted

$$\langle 1, \mathbf{2}, \mathbf{5}, \mathbf{6}, \mathbf{4}, \mathbf{3} \rangle \overset{\text{2-opt}}{\mapsto} \langle 1, \mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{5}, \mathbf{3} \rangle$$

the move is depicted in Figure 3.1. In the case of 2-opt, the delta costs are computed

$$\begin{aligned} \Delta_f(s, \text{2-opt}(i,j)) = &- [d_{i,i+1} + d_{j-1,j}] \\ &+ [d_{i,j-1} + d_{i+1,j}] \end{aligned}$$

in case of symmetric TSP (where $d_{i,j} = d_{j,i}$), but must reconsider all the intermediate customers between i and j in case of non-symmetric TSP.

3.2 Search

All neighborhood search methods share a common structure, which is described in Algorithm 3.

The first step consists in generating a starting solution (InitializeSolution). The initial solution can be built either randomly, or according to a greedy heuristic for the problem being solved. For example, for the TSP, one could build an initial solution by always choosing the closest node as the next customer (*nearest neighbor*).

Algorithm 3 Neighborhood search (NS)

```
procedure NeighborhoodSearch($\mathbf{S}, \mathcal{N}, f$)
    $i_{best} \leftarrow i \leftarrow 0$
    $s_{best} \leftarrow s \leftarrow$ InitializeSolution($\mathbf{S}$)
    while $\neg$StoppingCriterion($s, i, i_{best}$) do
        $m \leftarrow$ SelectMove($s, \mathcal{N}, f$)
        if $m =$ Null then
            return $s_{best}$
        end if
        if AcceptableMove($m, s, f$) then
            $s \leftarrow s \oplus m$
            if $f(s) < f(s_{best})$ then
                $s_{best} \leftarrow s$
                $i_{best} \leftarrow i$
            end if
        end if
        $i \leftarrow i + 1$
    end while
    return $s_{best}$
end procedure
```

Then, the procedure enters a loop in which the next move to apply is chosen by the function SelectMove, and, if the move is deemed acceptable by the function AcceptableMove, it is applied to s to obtain a new s. Note that the implementation of both SelectMove and AcceptableMove depends on the specific NS approach being used. Apart from the current solution s and the iteration number i, the procedure stores the best solution found from the beginning of the search s_{best}, along with the iteration i_{best} in which it was found.

The optimization run proceeds until a stopping criterion is met, or if a call to SelectMove returns a Null move. There are several possible choices for the StoppingCriterion, among the most common are

- stop when the allotted *timeout* t_{max} is depleted,

- stop when $i \geq iter_{max}$, where $iter_{max}$ is the maximum number of *iterations* allowed, and
- stop when $i - i_{best} \geq idle_{max}$, where $idle_{max}$ is the maximum number of *non-improving* iterations allowed.

In the following sections, we describe some of the most popular NS methods, and address some of the pitfalls of NS, and how to deal with them.

3.2.1 Hill climbing (HC)

The most basic NS method is *hill climbing* (HC), which gets its name from its application to maximization problems (see Figure 3.2). In HC, when the neighborhood $\mathcal{N}(s)$ of the current solution s is explored in random order, the first move that does not decrease the solution quality is selected. The process is described in Algorithm 4, where RandomMove samples a random move (without replacement) in the neighborhood $\mathcal{N}(s)$ of s, and returns Null if the whole neighborhood has been explored.

Algorithm 4 Components of hill climbing (HC)

```
function SelectMove($s, \mathcal{N}, f$)
    return RandomMove($s, \mathcal{N}$)
end function

function AcceptableMove($m, s, f$)
    return $\Delta_f(s, m) \leq 0$
end function
```

3.2.2 Steepest descent (SD)

Another basic NS method is *steepest descent* (SD), which, unlike HC, gets its name from its application to minimization problem. In SD, the search procedure always chooses the best move in the neighborhood $\mathcal{N}(s)$ of the current solution s.

Since the whole neighborhood must be explored in order to find the best move to apply, steepest descent is, in general, more computationally demanding than HC. However, SD is guaranteed to converge to a *local optimum* in a finite number of steps. The components of SD are shown in Algorithm 5. Note that, for consistency with the other NS algorithms presented in this chapter, the iteration counter i must be updated as $i \leftarrow i + |\mathcal{N}(s)$ after each move selection, as all the neighborhood of s is always explored. Moreover, the stopping criterion assumes that $idle_{max} = 0$, i.e., the algorithm stops after the first iteration without improvement.

Algorithm 5 Components of steepest descent (SD)

```
function SelectMove($s, \mathcal{N}, f$)
    $cost_m \leftarrow Inf$
    $m \leftarrow$ Null
    for all $m' \in \mathcal{N}(s)$ do
        $cost_{m'} = \Delta_f(s, m')$
        if $cost_{m'} \leq cost_m$ then
            $m \leftarrow m'$
            $cost_m \leftarrow cost_{m'}$
        end if
    end for
    return $m$
end function

function AcceptableMove($s, m, f$)
    return $\Delta_f(s, m) < 0$
end function

function StoppingCriterion($s, i, i_{best}$)
    return $i - i_{best} > idle_{max}$
end function
```

3.2.3 Local optima and plateaux

The mapping from solutions $s \in \mathbf{S}$ to their relative objective values $f(s)$, can be visualized as a *fitness landscape* (see Figure 3.2), where the components of each solution correspond to coordinates, and the cost corresponds to the elevation of the terrain.

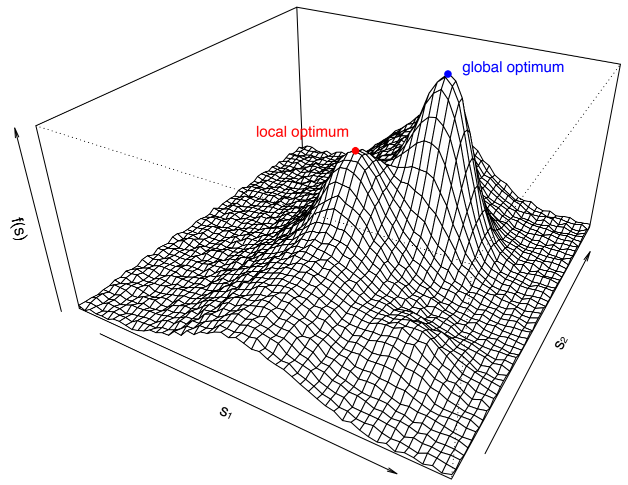


Figure 3.2: Example of a fitness landscape for a maximization problem.

The goal of optimization is to find a *global optimum*, i.e., the fitness landscape coordinates with the highest (in case of maximization), or lowest (in case of minimization), elevation. Note that, in general, there might exist multiple equivalent optima. One limitation of neighborhood search methods, is that, at each step, they only consider local knowledge about the neighborhood $\mathcal{N}(s)$ of a solution s. As such, they can be easily attracted by *local optima*, i.e., solutions which are the best in their neighborhood, but not the overall best. Such solutions constitute a hindrance to the overall optimization process, because without a policy to accept worse solutions, the search can get stuck. Many of the NS approaches described in the following sections deal with this aspect.

A similar complication is due to *plateaux*, i.e., areas of the fitness landscape where the objective value is constant. A simple mechanism to guarantee that the search is eventually able to exit a plateau is accepting *sideways moves*, i.e., moves which do not change the objective value of the current solution. This principle is applied in Algorithms 4 and 5, where weak inequality ($\leq$) is used to compare moves, instead of strict inequality (<).

3.2.4 Simulated Annealing (SA)

One of the most effective ways to escape local optima, is to accept, every now and then, moves that reduce the solution quality. *Simulated annealing* (SA) [67] achieve this by accepting a worsening move $m \in \mathcal{N}(s)$ with a probability depending inversely on $\Delta_f(s, m)$. That is, the greater is the decrease in quality induced by m, the less likely m will be selected. Of course, if a move improves the quality of the solution, it is always chosen.

To control the frequency of acceptance of worsening moves, a parameter t (for *temperature*) is introduced. At the beginning of the search t is initialized to a value t_0 (which can be computed heuristically, or tuned for the specific problem being solved), and is then updated according to a *cooling schedule* as the time passes. With t, the probability of selecting a move $m \in \mathcal{N}(s)$ is computed as

$$\mathbf{P}(m \mid s, t) = \begin{cases} e^{-\Delta_f(s,m)/t} & \text{if } \Delta_f(s, m) > 0 \\ 1 & \text{otherwise.} \end{cases}$$

In order to make SA behave stochastically based on $\mathbf{P}(m \mid s, t)$, a random number r is sampled *uniformly at random* in $(0, 1)$. If $r < \mathbf{P}(m \mid s, t)$ the move is accepted, otherwise it is rejected. Because of the effect of t, the algorithm changes its behavior over time. At the beginning of the search, when t is high, many worsening moves are accepted. As t decreases, the algorithm behaves less and less erratically until, in the final phases of the search, it performs standard hill climbing.

The customary way to decrease the temperature, is to use a *geometric cooling schedule*, i.e., to select a *cooling rate* parameter $0 < \lambda < 1$, and to update the temperature with the formula

$$t = \lambda \cdot t$$

after a number $neighbors\_sampled_{max}$ of moves have been tested for acceptance. The main algorithm components are described in Algorithm 6.

Algorithm 6 Components of simulated annealing (SA)

```
function SELECTMOVE($s, \mathcal{N}, f$)
    return RANDOMMOVE($s, \mathcal{N}$)
end function

function ACCEPTABLEMOVE($s, m, f$)
    $neighbors\_sampled \leftarrow neighbors\_sampled + 1$
    if $neighbors\_sampled \geq neighbors\_sampled_{max}$ then
        $t \leftarrow \lambda \cdot t$
        $neighbors\_sampled \leftarrow 0$
    end if
    return $\Delta_f(s, m) \leq 0$ or UNIFORMRANDOM$(0, 1) < e^{-\Delta_f(s,m)/t}$
end function
```

Traditionally, the search stops when $t < t_{min}$, where t_{min} is a parameter of the algorithm, however several others stopping criteria have been used in practice (maximum number of iterations, timeout, ...).

Variant: cutoffs

There are other possible ways to decrease t during the search. One, described by [60] under the name *cutoffs*, consists in decreasing t whenever a specific number of neighboring solutions have been *accepted* (rather than sampled, as in Algorithm 6). The rationale behind this technique, is that fewer iterations should be spent at the beginning of the search when the algorithm behaves erratically, in favor of the final phases of the search, where iterations are needed to obtain a good intensification.

Other variants

More information about variants of Simulated Annealing, and a thorough analysis of the appraoch, see [60, 61] and [108].

3.2.5 Tabu Search (TS)

A different approach to avoid getting stuck in local optima is the one taken by *tabu search* (TS) [49, 50]. In TS, the move selection procedure explores a subset of the neighborhood $\mathcal{N}(s)$ of a solution s, and always chooses the move with the lowest delta cost (see Algorithm 7).

While at a first glance it could seem that TS behaves like SD, there are two major differences between the two approaches. First, it is completely legitimate for TS to choose a move which increases the cost of the solution. As a consequence, local optima are not an issue for TS, as it can escape them by climbing the least steep slope. However, for the same reason, TS is prone to looping between neighboring solutions. The second difference with SD aims at fixing this problem, and consists in a prohibition mechanism to exclude, from $\mathcal{N}(s)$, the moves that would lead to looping, i.e., moves that reverse the effect of a previous move.

The prohibition mechanism is usually implemented as a FIFO list T (the *tabu list*), which is initially $T = \emptyset$, in which all the accepted moves m are enqueued. When the neighborhood $\mathcal{N}(s)$ is explored, all the moves $\textsc{InverseMoves}(m)$ of any move m in T are excluded from the selection.

The tabu list T has usually a limited size and, as such, it must be regularly freed of old moves, to make room for new ones. The basic approach to manage this process is to remove, at each iteration i the move m_{i-t} selected at iteration $i - t$, where t is a parameter to the algorithm (and also the length of the tabu list).

Another way of managing the tabu list, is to sample, at the moment of the insertion into T, a number $r \in [t - \delta, t + \delta]$, which represents the number of iterations that the move has to spend inside the tabu list.

Aspiration criteria

A further feature of TS involves exceptions to the prohibition mechanism. At each iteration, the procedure samples $\mathcal{N}(s) \setminus T$ as usual, however some of the moves in T can be selected, if they satisfy an *aspiration criterion* $A(s, m)$. A very common aspiration crite-

Algorithm 7 SELECTMOVE in TS

```
function SELECTMOVE($s, \mathcal{N}, f$)
    $cost_m \leftarrow Inf$
    $m \leftarrow$ NULL
    for all $m' \in \mathcal{N}(s) \setminus \{\text{INVERSEMOVES}(m) \mid m \in T\}$ do
        $cost_{m'} = \Delta_f(s, m')$
        if $cost_{m'} \leq cost_m$ then
            $m \leftarrow m'$
            $cost_m \leftarrow cost_{m'}$
        end if
    end for
    return $m$
end function

function ACCEPTABLEMOVE($s, m, f$)
    $\mathbf{T} \leftarrow (\mathbf{T} \setminus \{m_{i-t}\}) \cup \{m\}$
    return true
end function
```

rion involves enabling the prohibited moves that would generate a solution better than the best one s^* found so far.

Variants

The one presented in this section is a rather simple (and diffuse) variant of tabu search, for a more extensive analysis of the approach see the two original technical reports [49, 50] and the book by Glover and Laguna [51].

Conclusions

We have presented neighborhood search as an effective family of general methods to tackle large combinatorial optimization problems in a heuristic way. We discussed aspects related to problems modeling, highlighting issues related to local optima and plateaus, which can constitute a hindrance to the search process. Moreover, we presented the most popular algorithms to successfully handle such issues.

Chapter 4

Swarm intelligence

The algorithms presented in Chapter 3 operate on a a single solution, improving its quality as the time passes. *Population-based* algorithms, on the other hand, keep track of multiple solutions at once. One of the obvious advantages of such an approach is that the algorithm is less sensitives to issues, such as local optima, concerning a single solution. Another advantage, is that optimizing several solutions at once allows to get a broader coverage of the search space, which can possibly result in the discovery of areas of higher solution quality. Finally, population-based methods are naturally parallelizable, since the search can proceed more or less independently for the various solutions.

The class of population-based algorithms is very broad, and includes more specific classes of algorithms. Among these classes are *evolutionary algorithms*, e.g., genetic algorithms [109], evolution strategies [53], genetic programming [70], etc., which rely on operators inspired by genetics, such as crossover and mutation, to make the solutions evolve across successive generations. Another class is the one of *swarm intelligence* algorithms. In swarm intelligence algorithms, the processes in charge of optimizing each solution cooperate by sharing local information, in order to implement global reasoning.

In this chapter we introduce two representatives of this latter class, namely *particle swarm optimization* (PSO), and *ant colony optimization* (ACO), which are both inspired by foraging behaviors of animals in nature.

4.1 Particle Swarm Optimization (PSO)

Particle swarm optimization (PSO) is a perturbative population-based algorithm for optimization in continuous domain. The algorithm is inspired by the behavior of birds flocking as they look for food. The algorithm was first proposed in [62], and then extended in [47], as a method for training the weights of a multi-layer perceptron. A more general formulation, which is also the most popular, was later introduced in [98]. A revision of the several proposed variants of PSO can be found in [99]. Moreover, discrete domain versions of PSO [63] have been proposed over the years[1].

In the contex of this thesis, PSO has been used to optimize *viewpoint computation*, a continuous non-stationary problem which is found in many 3D applications (see Chapter 10).

Given a function $f : \mathbb{R}^d \mapsto \mathbb{R}$ to optimize, the idea behind PSO is to let a *swarm* of d-dimensional *particles*, representing solutions to the optimization problem, fly through the search space, exploiting their collective knowledge about the fitness landscape of f to locate the global optimum.

4.1.1 Overall procedure

The overall PSO procedure, as described in [99], is reported in Algorithm 8. First, both the position ($\vec{pos}_j$) and the velocity ($\vec{vel}_j$) of the n particles are initialized. In the general PSO formulation, the position is sampled uniformly at random inside the search space $\mathbf{S} \subset \mathbb{R}^d$, however, results [?] show that, having InitializePosition implement a problem- or instance-aware initialization strategy, can have a huge benefit on the performance of the algorithm. The velocity of each particle is typically initialized to a random d-dimensional vector, whose components are relatively small with respect to the range of the search space.

[1]The website http://www.swarmintelligence.org contains an rich bibliography about PSO and other swarm intelligence algorithms.

Algorithm 8 Particle Swarm Optimization (PSO)

```
procedure PSO($\mathbf{S}, f, n, c_1, c_2, w$)
    $g \leftarrow$ NULL
    $i \leftarrow 0$
    for $j \in \{1, \ldots, n\}$ do                    ▷ Initialize $n$ particles
        $vel_j \leftarrow$ INITIALIZEVELOCITY($S$)
        $best\_pos_j \leftarrow pos_j \leftarrow$ INITIALIZEPOSITION($S$)
        $best\_cost_j \leftarrow cost_j \leftarrow f(pos_j)$
        if $best\_cost_j < best\_cost_g$ then
            $g \leftarrow j$
        end if
    end for
    while ¬STOPPINGCRITERION($i$) do
        for $j \in \{1, \ldots, n\}$ do                ▷ Update particles, $g$
            $pos_j \leftarrow pos_j + vel_j$
            $cost_j \leftarrow f(pos_j)$
            if $cost_j \leq best\_cost_j$ then
                $best\_pos_j \leftarrow pos_j$
                $best\_cost_j \leftarrow cost_j$
                if $best\_cost_j \leq best\_cost_g$ then
                    $g \leftarrow j$
                end if
            end if
            $r_1 \leftarrow$ UNIFORMRANDOM($0, 1$)
            $r_2 \leftarrow$ UNIFORMRANDOM($0, 1$)
            $vel_j \leftarrow w \cdot vel_j$
                $+ \; c_1 r_1 \cdot (best\_pos_j - pos_j)$
                $+ \; c_2 r_2 \cdot (pos_g - pos_j)$
        end for
        $i \leftarrow i + 1$
    end while
    return $pos_g$
end procedure
```

At each time step t, each particle j is accelerated stochastically towards two points of the search space according to the update rule

$$\vec{vel}_j^{\,t} = w \cdot \vec{vel}_j^{\,t-1} + \tag{4.1}$$

$$c_1 r_1 \cdot (\vec{best\_pos}_j^{\,t-1} - \vec{pos}_j^{\,t-1}) + \tag{4.2}$$

$$c_2 r_2 \cdot (\vec{pos}_{g^{t-1}}^{\,t-1} - \vec{pos}_j^{\,t-1}). \tag{4.3}$$

The velocity update rule is composed of three terms. The first one (4.1) models the *inertia* of the particle, i.e., its resistance to steering. This term is controlled by the parameter w which represents the *weight* of the particle. According to [98], the use of w is fundamental to balance the trade-off between local and global search, and the parameter should be decreased over time to facilitate the convergence to the global optimum. The second term (4.2) accelerates the particle j towards $\vec{best\_pos}_j^{\,t-1}$, the best position visited by j itself since the start of the search, which consistutes a *local knowledge*. The final term (4.3) represents the acceleration towards $\vec{pos}_{g^{t-1}}^{\,t-1}$, the current position of the best particle of the swarm, i.e., the particle g that has visited the best position so far (according to f), which consistutes a *global knowledge*. The two acceleration terms are controlled, respectively, by the *cognitive parameter* c_1, which represents the trust of the particle in itself, and by the *social parameter* c_2, which represents the trust of the particle in the leader of the swarm. Moreover, the amount of acceleration towards $\vec{best\_pos}_j$ and $\vec{pos}_g$ is stochastic, because of the effect of r_1 and r_2, which are two numbers sampled uniformly at random in $[0, 1]$.

After the position of each particle has been updated, the particles are re-evaluated, to check whether there is a new leading particle g, and (possibly) to update the best past position of each particle. Then the main loop restarts, unless a stopping condition, e.g., maximum number of iterations exceeded, timeout, etc., is met.

4.1.2 Parameters

With respect to other approaches, PSO has many parameters which control its behavior. Setting (or *tuning*) such parameters appropriately, is clearly determinant for attaining good performance, and

Parameter name	Symbol	Suggested value
Number of particles	n	-
Cognitive parameter	c_1	2
Social parameter	c_2	2
Inertia weight	w	$[0.9, 1.2]$

Table 4.1: Parameters of PSO, with suggested values in [98]

the values should be determined in an application-dependent way. However, according to [98], there are some good default values for the PSO parameters, which are summarized in Table 4.1. While these values cannot possibly be valid for every application domain (see [112]), it is reasonable to pick them as starting points for a more thorough *parameter tuning*. The number of particles n reasonably depends on the size of the search space and on the number of dimensions, and there is no suggested setup for it. Also, consider that when w is decreased over time, at least one additional parameter must be used, i.e., w becomes w_{init} and w_{end}.

4.1.3 Variant: constriction factor

A known issue, with the previously described velocity update rule, is that the velocity is unbounded, and can grow arbitrarily large under certain conditions. One approach to mitigate this effect is to bound the magnitude of the velocity by a vector $\vec{vel}_{max}$ dependent on the size of the search space, e.g., 10% of the search space range in each component.

Another approach, reported in [99], is to use a slightly different velocity update rule

$$\begin{aligned}\vec{vel}_j^{\,t} = K \cdot [\vec{vel}_j^{\,t-1} + \\ & c_1 r_1 \cdot (\vec{best\_pos}_j^{\,t-1} - \vec{pos}_j^{\,t-1}) + \\ & c_2 r_2 \cdot (\vec{pos}_{g^{t-1}}^{\,t-1} - \vec{pos}_j^{\,t-1})]\end{aligned}$$

where K is a *constriction factor* that multiplies the update formula, reducing the risk of growing velocities. K can be computed as

$$K = \frac{2}{\left|2 - \phi - \sqrt{\phi^2 - 4\phi}\right|}$$

where $\phi = c_1 + c_2$, so that $\phi > 4$ holds. Note that the inertia of the particle is completely neglected in this update rule. Also, it should be recognized that the use of a constriction factor does not rule out completely the chance of growing velocities, only reduces it.

4.2 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) [44] is a constructive swarm intelligence meta-heuristic for combinatorial optimization, which simulates the foraging behavior of real ants. Although the first ACO contributions date back to the early '90s, in this section we focus on a more recent variant of the algorithm, dubbed *hyper-cube framework* (HCF) for ACO, which was originally proposed in [17], and further developed in [15]. This variant of ACO is insensitive to the scale of the objective function, and is thus more robust with respect to the original.

The fundamental principle behind ACO is that of *reinforcement learning* (see Appendix B). When a (real) ant finds a source of food, it lays down a trace of *pheromone* on its way back to the nest. The pheromone can be sensed by the other ants, which are thus able to follow the trace and, ultimately, to find the food. As more ants reach the food the trace gets reinforced, because every ant lays down additional pheromone. As soon as the food depletes, the ants stop laying down pheromone, which naturally evaporates erasing the trace. The pheromone trace is represents an information left by the ants in the environment in order to influence the behavior of the colony. Such kind of information is sometimes called *stigmergic* [101].

In the context of combinatorial optimization, food is represented by good solutions, i.e., sequences of assignments of values to variables of the form $c_{ij} = (x_i, d_j)$, where $d_j \in D_i, i \in \{1, \ldots, n\}$. Consequently, each *solution component* c_{ij} represents a segment of the path that lead to the food, whose trace must be reinforced. Following the same metaphor, ants are (possibly parallel) search processes that build up solutions according to a *state transition rule*. Intuitively, the underlying assumption of all ACO techniques, is that components of good solutions are good, and solutions built upon good components are good as well. This is true for some domains more than others, and is one of the reasons of the success of ACO algorithms in routing problems, where using short path segments also yield shorter, and thus better, paths.

4.2.1 State transition rule

Being ACO a constructive method, each of the n ants chooses, at every step, a variable among the unassigned ones, and assigns a value to it (possibly enforcing feasibility), until a complete solution is built. In particular, once a variable x_i has been chosen, its value is selected according to a probabilistic model called the *pheromone model*, parametrized by $|C = \{c_{ij} \mid x_i \in X, d_j \in D_i\}|$ *pheromone trail parameters* τ_{ij}. Given a partial solution s^p and a variable x_i to assign, the value τ_{ij} determines the probability of choosing the value $d_j \in D_i$ for x_i. The probability itself is calculated according to the following state transition rule

$$\mathbf{P}(c_{ij} \mid s_i^p) = \begin{cases} \dfrac{\tau_{ij}}{\sum_{c_{ik} \in J(s_i^p)} \tau_{ik}} & \text{if } c_{ij} \in J(s_i^p) \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

where $J(s_i^p)$ denotes the set of all the *feasible* components for s^p of the form $(x_i, d_k), d_k \in D_i$, i.e., the ones for which $s^p \cup \{(x_i, d_k)\}$ is a feasible solution.

4.2.2 Pheromone update rule

Many ACO variants use the state transition rule in Equation 4.4, but differ in the way the pheromone trail parameters are updated. In the HCF for ACO, after n solutions have been built, the pheromone model is updated according to the following *pheromone update rule*

$$\tau_{ij} = (1-\rho)\cdot\tau_{ij} + \rho\cdot \sum_{s\in U|c_{ij}\in s} \frac{F(s)}{\sum_{s'\in U} F(s')} \tag{4.5}$$

where U are the last solutions produced by the n ants, $\rho \in (0,1]$ is an *evaporation rate* that controls how fast the ant colony forgets about a non-reinforced trace, and F is a *quality function* that measures the goodness of a solution (usually chosen as $1/f$, where f is the cost function).

The terms highlighted in red, in Equation 4.5, represent the novelties introduced in the HCF for ACO, with respect to the former *Ant System* (AS) approach proposed in [44]. The introduced terms guarantee that the pheromone trail parameters τ_{ij} can only assume values in $[0,1]$. Apart from the obvious advantage regarding the independence on the scale of the objective function, this has another important implication. If we represent each solution s_k in the search space by a binary vector $\vec{v_k}$ of length $|C|$ specifying whether each component c_{ij} appears in s_k or not, then we can see the pheromone trail parameters as a vector $\vec{\tau}$ in the hypercube that has the $\vec{v_k}$ as vertices.

Each update

$$\begin{aligned}\vec{\tau} =& (1-\rho)\cdot\vec{\tau} + \rho\cdot\vec{m}\\ \vec{m} =& \sum_{\vec{s}\in U} \frac{F(\vec{s})}{\sum_{\vec{s'}\in U} F(\vec{s'})}\cdot\vec{s}\end{aligned}$$

to the pheromones slightly moves the head of $\vec{\tau}$ towards the solution composed by the best components in U (accumulated in $\vec{m}$), thus favoring the choice of some components over others in the construction of new solutions.

4.2.3 Overall procedure

The overall ACO procedure is described in Algorithm 9. The InitializePheromone procedure sets the τ_{ij} to small values. In HCF for ACO, since the range for the pheromone values is known ($\tau_{ij} \in [0,1]$), the pheromones trail parameters are initialized to 0.5, which guarantees a fair probabilistic choice at the beginning of the search. In approaches, such as AS, where the normalization term is missing in the update rule, setting an initial value for the pheromone trail parameters is much more difficult. Once all the n solutions have been built stochastically by the BuildSolutions procedure, they are used to update the pheromone trail parameters based on the Equation 4.5.

Algorithm 9 Ant Colony Optimization (ACO) in the HCF framework

```
procedure ACO($\mathbf{S}, f, n, \rho$)
    $\tau \leftarrow$ InitializePheromone$(0.5)$
    $i \leftarrow 0$
    $best \leftarrow$ Null
    while $\neg$StoppingCriterion$(i)$ do
        $\mathbf{U} \leftarrow$ BuildSolutions$(n, \tau)$            ▷ See Equation 4.4
        $\tau \leftarrow$ UpdatePheromone$(\tau, \mathbf{U}, \rho)$          ▷ See Eq. 4.5
        $best \leftarrow \arg\min_{j \in \mathbf{U} \cup \{best\}} f(j)$
        $i \leftarrow i + 1$
    end while
    return $best$
end procedure
```

4.2.4 Parameters

The only two parameters of the HCF for ACO are the number of ants n, and the evaporation rate ρ. Again, for the number of ants there is no clear setup, as it highly depends on the available computational resources. Intuitiveluy, using more ants is more computationally demanding, but improves exploration, and should converge faster to good solutions. As for ρ, values around zero (low evaporation rate) correspond to very small updates, and a high trust in the past solutions, while values close to 1 mean that the approach adapts very quickly to new solutions. This parameter should be tuned appropriately for the problem being solved.

4.2.5 Heuristic information

In many ACO implementations, *heuristic information* η_{ij} is used, along with pheromone values, to decide which component should enter a partial solution. Ideally, such information should be based on some knowledge about the problem, and should be fixed throughout the optimization run. The relative influence of τ_{ij} and η_{ij} is usually modeled through two parameters, α and β. The transition rule becomes then

$$\mathbf{P}(c_{ij} \mid s_i^p) = \begin{cases} \dfrac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{c_{ik}\in J(s_i^p)} \tau_{ik}^{\alpha}\eta_{ij}^{\beta}} & \text{if } c_{ij} \in J(s_i^p) \\ 0 & \text{otherwise.} \end{cases} \tag{4.6}$$

4.2.6 Variant: max-min ant system (MMAS)

Over the years, many attempts have been done to improve the performance of ACO algorithms. One of the main lines of research of this kind, consists in designing ACO algorithm that exploit more aggressively the gathered search experience.

A notable variant belonging to this class is the Min-Max Ant System (MMAS) [101]. In MMAS, a single solution s^{best} among the ones produced by the n ants is used, at each iteration, to update the pheromone model. This solution may be either be the global best solution found since the beginning of the search s^{gb}, or, more commonly, the best solution found during the current iteration s^{ib}. Also, the update function is slightly different from the one in Equation 4.5

$$\tau_{ij} = (1-\rho)\cdot\tau_{ij} + F(s^{best}). \tag{4.7}$$

Since the update rule in Equation 4.7 is unbounded, and because of the fact that the updates are on a single solution, MMAS is prone to early convergence. This can happen if, at each choice point, the level of pheromone in one component is much more high than the others. Because of this, the MMAS generates very often the same solution, which in turn contributes to reinforce its components. To cope with this issue, MMAS limits the pheromone level range to two values, τ_{min} and τ_{max} by *clamping* the new pheromone values at the moment of the update. The role of τ_{min} is to guarantee that even upon convergence, the ants still produce diverse solutions (the probability of choosing a component is always non-zero if $\tau_{min} > 0$). Moreover, the pheromones are initially set to τ_{max} to foster an initial exploration of the available components. This works because, even though the first solutions produced have a high quality, the pheromone levels of the non-used components are maximal, thus promoting the generation of solutions based on different components.

Note that an approach similar to MMAS can be implemented in the HCF for ACO.

Conclusions

We briefly discussed swarm intelligence methods, pointing out what are their advantages and disadvantages with respect to single-solution search mechanisms. We then presented two of the most popular swarm intelligence approaches, namely particle swarm optimization and ant colony optimization, which are notable examples of two very different ways of sharing information in order to carry out optimization.

Part II

Hybrid meta-heuristics

Chapter 5

Overview of hybrid methods

Over the years, the meta-heuristics community has developed many successful general purpose optimization algorithms. Because of this success, researchers have rarely sought to integrate meta-heuristics with approaches being developed in other communities, such as artificial intelligence (AI) or operations research (OR). However, in the past few years, the realization that a performance limit had been hit in the field of meta-heuristics, led the researchers towards the exploration of methods to integrate meta-heuristics with other techniques [16]. This trend originated the field of *hybrid meta-heuristics*, which brought together practitioners from many optimization paradigms, e.g., constraint programming, mathematical programming, machine learning. The hybrid meta-heuristics community has now come of age, and has its own set of conferences and journals. Moreover, many well-established hybrid search techniques, that can benefit from the complementary capabilities of a wide range of algorithms, have been developed in the recent years.

In this chapter, we will briefly overview some of the major research lines in hybrid meta-heuristics. However, for a proper discussion of many other hybrid approaches, we refer the reader to a recent survey [16]. The ideas behind some of the techniques presented in this chapter, and their implementation will be discussed in detail in Chapters 6 and 7.

5.1 Combining together meta-heuristics

A broad range of hybrid meta-heuristics are based on the principle of mixing together different meta-heuristic approaches. In the following sections, we briefly address the most popular ones.

5.1.1 Memetic algorithms

One of the main goals of hybrid meta-heuristics, is to integrate algorithms with different capabilities in order to get "the best of the many worlds". This idea underlies the integration of population-based algorithms, which are very good in the exploration of large search spaces and are little influenced by local optima, with neighborhood search meta-heuristics, which are good at locating and exploiting local optima.

Memetic algorithms are population based algorithms, mostly coming from the field of Evolutionary Computation, in which neighborhood search steps are taken in order to improve the quality of the solutions obtained, e.g., by cross-over and mutation. A straightforward example of such approaches is the extension of *iterated local search* (ILS) to a population of solutions. Standard ILS starts from a solution s, and iteratively improves its quality by repeatedly applying a neighborhood search technique until a local optimum is found, then the solution is perturbed and a new starting solution is obtained. The extension of ILS to multiple solutions consists in keeping a population of n solutions, each one evolved using the ILS scheme. At each generation additional n solutions are generated through neighborhood search, thus totaling $2n$ solutions. A straightforward selection scheme which discards the n worst solution is then used to generate a new population of n solutions.

5.1.2 Hyper-heuristics

Hyper-heuristics are high-level search methods for selecting or generating heuristics in order to solve optimization problems [21]. Unlike meta-heuristics, which directly explore the search space of a problem, hyper-heuristics explore the space of domain-specific heuristics and perform optimization by combining them together or activating them to operate on solutions. A hyper-heuristic thus works at a higher level of abstraction, and does not know anything about the underlying problem, which makes it applicable to a broader range of problems.

5.1.3 GRASP

Greedy randomized adaptive search procedures (GRASP) [48] is a constructive meta-heuristic that iteratively improves the quality of an incumbent solution by means of two alternated steps: a *construction step*, which builds a new solution in a biased but stochastic fashion, and a *local search* step which improves the newly constructed solution until a local optimum is found. The construction step is the peculiar feature of GRASP and works as follows. First, an empty solution is initialized, then each solution component is assigned probabilistically a value from a *restricted candidate list* (RCL). The RCL is a list of values containing the best ones according to a learned preference function. After each assignment, the preference function is updated based on the result of the assignment. The intuition behind GRASP is that obtaining a better initial solution at each iteration improves the effectiveness of the subsequent local search step.

5.1.4 Multi-level techniques

Multi-level techniques [110] are methods to tackle large optimization problems, based on the idea that finding a solution to a simplified version of a problem, and extending it to be a solution of the original problem, is much more tractable than solving the original problem in the first place. Multi-level techniques apply this principle repeatedly, simplifying the original problem through a problem-specific *coarsening* heuristic, until a number of *levels*, representing more and more simplified versions of the problem, are generated. Then, they proceed by finding a solution to the last level k, and they iteratively *refine* the solution so that it becomes a solution to level $k - 1$. The process continues until the solution is extended to a solution of the original problem.

Such techniques were originally developed in the field of graph theory, where several techniques exist to obtain coarser versions of a graph. However, their extension to other optimization problems has been widely explored. For instance, [110] shows examples of multi-level techniques applied to combinatorial optimization problems such as set covering and the TSP.

5.2 Combining meta-heuristics with CP

One of the most popular research lines in hybrid meta-heuristics consists in the integration of meta-heuristics with exact approaches. The motivation behind this, is that such classes of methods are good at different tasks, which suggests that integrating them could give birth to more robust approaches. In particular, exact methods are known to be very effective in solving constraint satisfaction problems, i.e., tackling hard constraints, but they are not very good at tackling objective functions. On the other hand, meta-heuristics are good at optimizing objective functions, but less good at finding feasible solutions [16].

Because of the popularity of constraint programming [89] among exact methods, it is natural that a number of hybrid meta-heuristics be devised to combine the power of strong constraint propagation techniques with the explorative power and performance of meta-heuristics.

In particular, when such an hybridization is considered, it is important to clarify which approach should be the *master* and which one the *slave*. On one side we have meta-heuristics approaches that use constraint programming as a sub-procedure. An example of such an approach is *large neighborhood search* (LNS), where CP is used within a general neighborhood search loop. On the other side we have constraint solver which use meta-heuristic techniques inside their flow. An example of this kind of approaches is ACO-driven onstraint programming. Both approaches are described in Chapter 7.

5.3 Other hybrid methods

Many other strategies to integrate meta-heuristics with other search approaches have been devised in the recent years. Some, called *math-heuristics* [77] aim at combining meta-heuristics with well-established mathematical programming techniques, for which high-performance solver exist, e.g., linear programming, mixed integer linear programming, and the like. Other approaches include the integration of meta-heuristics with dynamic programming (e.g, DynaSearch [33]), relaxations, decomposition techniques, and tree search (e.g., Beam-ACO [14]). Such approaches are, however, out of the scope of this thesis, and are not discussed here. For an overview of such methods, we again refer the reader to [16].

Conclusions

We presented a brief overview of hybrid meta-heuristic approaches, highlighting the main lines of research in this field, and pointing out some reasons for their adoption. Some of the presented approaches will be discussed in-depth later in this thesis, while for some others we provided references and reading directions.

Chapter 6

Reinforcement learning-based hyper-heuristics

Hyper-heuristics (see Section 5.1.2) are search methods for selecting or generating heuristics to solve computational search problems [21]. As such, they operate at a higher level of abstraction with respect to standard meta-heuristics. Specifically, while meta-heuristics operate directly on the space of solutions of a problem, hyper-heuristics operate on the space of heuristics or meta-heuristics for the problem. The primary goal of the research in hyper-heuristics, is to raise the level of generality of optimization algorithms, so that it is possible to devise solvers that can solve a problem without knowing the specific details about the problems being solved.

In this chapter we consider a particular family of hyper-heuristics, namely online-learning algorithms to select low-level heuristics. Specifically, we describe our effort in designing a hyper-heuristics to compete in the first Cross-Domain Heuristic Search Challenge (CHeSC'11) [20]. The results described in this chapter have been published in two papers, [41] and [42], which I co-authored, and which have been presented respectively at MIC'11, the 9th Metaheuristics International Conference, and at LION 6, the Learning and Intelligent OptimizatioN conference (2012).

6.1 The hyper-heuristics framework

In this section we describe the ideal hyper-heuristics framework [22], whose specification is concretized in a framework [83] for the design and analysis of cross-domain heuristic search.

The main idea behind the hyper-heuristic framework, is that there is a conceptual *domain barrier* between the hyper-heuristic algorithm and the modules implementing specific problem domains (see Figure 6.1). The domain modules must implement, at the very least, *i*) a method to *initialize* one or more solutions s_i with $i \in \{1, \ldots, k\}$, *ii*) an *objective function* f to measure the quality of a solution, and *iii*) a number of *low-level heuristics* h_j with $j \in \{1, \ldots, n\}$ that can be applied to a solution s_i to yield a new solution s_i'..

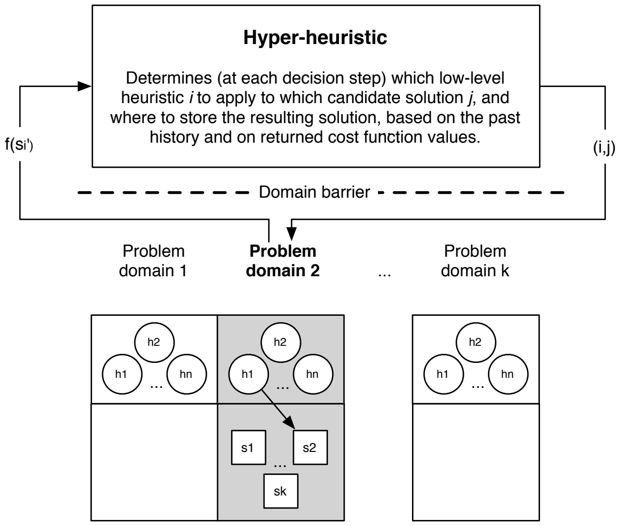

Hyper-heuristic
Determines (at each decision step) which low-level heuristic i to apply to which candidate solution j, and where to store the resulting solution, based on the past history and on returned cost function values.
f(si')
(i,j)
Domain barrier
Problem domain 1
Problem domain 2
...
Problem domain k
h2
h1
hn
s1
s2
sk

Figure 6.1: Hyper-heuristic framework.

The only way the hyper-heuristic can communicate with the domain modules, is by specifying, at each step, the index i of the next solution to modify, and the index j of the domain-specific heuristic to apply, whose nature is unknown to it. The domain module applies the chosen heuristic h_j to the specified solution s_i, and yields a new solution s'_i, which can be accepted or rejected by the hyper-heuristic based on its objective value $f(s'_i)$. Apart from these constraints, the hyper-heuristic algorithm can implement whatever logic in order to optimize the function.

6.2 CHeSC Competition

Concretely, the hyper-heristic framework has been implemented as a Java library called HyFlex, which also served as the battleground for the first Cross-Domain Heuristic Search Challenged (CHeSC'11). HyFlex is an API that provides basic functionalities for *i*) loading problem instances, *ii*) generating new solutions, and *iii*) applying low-level heuristics to solutions. Low-level heuristics are treated as black-boxes, and only information about their *family* is known (e.g., *mutations*, *local search moves*, *cross-overs*, …). Furthermore, it is allowed to tune the effect of low-level heuristics through the *intensity of mutation* and *depth of search* parameters, depending on the specific heuristic family.

The six problem domains considered in the competition are: *Boolean Satisfiability* (in the MAX-SAT formulation), *1-Dimensional Bin Packing*, *Permutation Flow Shop Scheduling*, *Personnel Scheduling*, *Traveling Salesperson Problem*, and *Vehicle Routing Problem* (however the last two were undisclosed until the day of the competition, to avoid overtuning on the other four). We refer the reader to the CHeSC'11 website[1] for further details, including the scoring system and the sources of the benchmark instances.

6.3 Our approach

The hyper-heuristic that we presented at the competition is based on a *reinforcement learning* (RL, see Appendix B for an overview of the basic idea and some advanced techniques), and has been selected automatically among a family of variants that were developed in the pre-competition stages. In order to describe them, we

[1] Address: http://www.asap.cs.nott.ac.uk/external/chesc2011

need to instantiate the following elements:

- the *environment*,
- the *reward function*,
- the set of *actions*,
- the *policy*, and
- the *learning function*.

In the following we describe our design choices about these aspects.

6.3.1 Environment

Given that problem domains are hidden by the HyFlex framework, at each decision step the only information available about the environment is the *objective value* of the current solution. Therefore, we need a way to build a state representation by only using this information. Unfortunately, these values have a completely different scale depending on the problem domain at hand, and on the search phase (start of the search, end of the search), moreover they don't convey enough information to drive the decisions of an agent (i.e., they are not *Markov states* [102]). For these reasons, the way a state s is represented inside the agent is non-trivial. After attempting some variants, we resorted to an adaptive state representation which tries to capture the concept of *relative recent reward* obtained by the agent, i.e. the reward *trend*. By "relative" we mean that each reward is normalized with respect to the problem's cost scale. In order to obtain the relative reward, we thus divide it by an (adaptive) measure of *low* reward (e.g., if 2.8 is the measure of low reward, an absolute reward of 8 will yield a relative reward of 2.86). The intuitive meaning of this operation is to obtain a reward measure which is both problem-independent and adaptive with respect to the search phase. With "recent" we mean that past experience is *discounted* [102], hence newer information is trusted more than old one. At each step, the new state is thus computed as

$$s_{i+1} = \lfloor (s_i + \beta * (r_i - s_i)) \rfloor$$

where the *reactivity* β is a parameter that will be explained later on and r_i is the last relative reward received.

6.3.2 Actions

We defined a possible *action* a as the choice of the heuristic family to be used, plus an intensity (or depth of search) value in the quantized set of values $0.2, 0.4, 0.6, 0.8, 1.0$. Once the family has been determined, a random heuristic belonging to that family is chosen and applied to the current solution with the specified intensity (or depth of search). The action application yields a new solution and a reward. Moreover, a special action performs a solution *restart* (without resetting the learned policy). In addition, heuristics belonging to the cross-over family require to operate on two solutions. For this reason we keep a number of independent agents, each one with its own solution, and use them for breeding when needed.

6.3.3 Reward function

The reward r is computed as the difference Δ_{cost} in the cost of the solution before and after the application of an action. More complex variants are possible, for instance

$$r = \Delta_{cost}/time_{action}$$

(reward over time), but this simple formulation is easier to understand, and makes the reward enough informative for our scopes.

6.3.4 Policy

As for the the policy, each state-action pair $\langle s, a\rangle$ is assigned an action $\omega_{s,a}$, which represent a degree of suitability of the action in the given state. We experimented with two classical reinforcement learning policies, namely a *softmax* action selection, using a Boltzmann distribution with an exponential cooling schedule, and an ϵ*-greedy* action selection. The ϵ-greedy selection policy scored overall better and was chosen to participate in the competition.

6.3.5 Learning function

As for the learning function, we have considered various techniques to update action values. In the simplest case, the action value after a specific $\langle s, a \rangle$ is executed, is set to the discounted average of the rewards (i.e. Δ_{cost}) received before, according to the formula

$$\omega_{s,a} = \omega_{s,a} + \alpha * (r - \omega_{s,a})$$

where the learning rate α is needed to tackle non-stationary problems, as explained in Appendix B.

A second investigated method is Sarsa [102], an on-line temporal -difference method in which the value of an action is also dependent on the stored value of the next action, thus implementing a sort of look-ahead strategy. However, the simplest method obtained better performance on the benchmark instances, and was selected for the competition.

6.4 Parameter tuning

We performed an extensive experimental analysis of the algorithm variants described in Appendix B, namely tabular reinforcement learning (RLHH), reinforcement learning with MLPs (RLHH-MLP), and reinforcement learning with eligibility traces (RLHH-ET), with the purpose of understanding the most relevant parameters (see Table 6.1) and their relationships. In order to properly tune all the algorithmic features, we ran an *F-Race* [13] on all the benchmark instances that were provided before the day of the competition (40). As for the experimental setup, we ran all the configurations to be tested on three different Intel machines equipped with quad core processors (resp. at 2.40, 2.40 and 3.00 GHz) and running Ubuntu 11.04. Differences in performance were leveled by using the CHeSC benchmarking tool provided by the organizers of the competition.

6.4.1 Common parameters

While each different approach has its own specific parameters, some of them (see Table 6.1) are common to every hyper-heuristic. In the following paragraph we are going to explain their meaning and list the possible values they can take.

Parameter name	**Domains**
Number of agents	$3, 4, 5, 6, 8, 10$
Cross-over with	`best_agent`, `optimum`
α (learning rate)	$0.1, 0.2, 0.3$
β (reactivity)	$0.05, 0.1, 0.25, 0.5, 0.9$
ϵ	$0.01, 0.05, 0.1$

Table 6.1: Common hyper-heuristic parameters.

The parameter *number of agents* determines how many learning agents are deployed in the search space. These agents share the same information about action values (either a table or a MLP) but have separate solutions to work on. An agent has access to the solutions of the other agents only during cross-over operations which, depending on the parameter *cross-over with*, can be carried out with the best solution overall (`optimum` mode) or with the agent which has the best current solution at the moment (`best_agent` mode). The *reactivity* β determines how promptly an agent updates its state when receiving a new reward, if the value is set to 1.0 the agent always substitutes its current state with the new reward, however as the value approaches 0.0 the state is updated more and more slowly. The *learning rate* α has a similar semantic, except that it models how fast action values are changed when receiving a new reward (see Appendix B). Finally, the probability for an agent to choose a random action instead than the one with highest value is denote by ϵ.

6.4.2 Parameters for RLHH-MLP

With respect to RLHH, RLHH-MLP requires a number of extra parameters (see Table 6.2) which are related to MLP learning. The parameters *hidden layers* and *hidden neurons* determine the complexity of the function that the MLP is able to approximate. Intuitively, having more hidden neurons allows to have a greater resolution (fit better the training data), and using more hidden layers leads to easier approximation of complex functions. The parameters *input scale* and *absolute maximum decay* are used to scale the MLP input in order to accelerate the convergence of gradient descent. Finally, *learning rate decay*, *learning rate change* and *error step size* are related to a technique known as *adaptive learning rate* [2] for the training of neural networks whose explanation is beyond the scope of this chapter.

Parameter name	**Domains**
Hidden layers	$1, 2$
Hidden neurons	$20, 30, 40$
Input scale	$1, 3$
Absolute maximum decay	0.9999
Learning rate decay	0.9999
Learning rate change	$0.001, 0.01$
Error step size	0.1

Table 6.2: RLHH-ET parameters.

6.4.3 Parameters for RLHH-ET

RLHH-ET only introduces two parameters: the *threshold* tr and *trace decay* λ (see Table 6.3), which are used to compute the length of the eligibility queue using Equation B.3.

Parameter name	**Domains**
tr (threshold)	0.01
λ (trace decay)	$0.5, 0.9$

Table 6.3: RLHH-MLP parameters.

Since the evaluation has to be performed across different domains and on instances with different scales of cost functions we decided to consider as the response variable of our statistical tests the *normalized* cost function value at the end of the run. That is, the cost value y is transformed by means of the following transformation, which is applied by aggregating on the same problem instance π

$$e(y, \pi) = \frac{y(\pi) - y^*(\pi)}{y_*(\pi) - y^*(\pi)}$$

where $y^*(\pi)$ and $y_*(\pi)$ denote the best known value and the worst known value of cost on instance π. This information has been computed by integrating the data gathered by our experiments with the information made public by CHeSC organizers[2].

6.4.4 Parameter influence

The first experimental analysis has the aim of understanding the influence of the different parameters on the outcome of the algorithms. For this purpose we perform an *analysis of variance* (ANOVA) on a comprehensive dataset including all configurations run throughout all the problem domains. Each algorithm variant has been run on each single instance for 5 repetitions.

We perform separate analysis for each variant of the algorithm and we set the significance level of the tests to 0.95. The outcome of this analysis will allow us to fix some of the parameters to "reasonable" values and to perform a further tuning of the relevant ones.

RLHH. The results of the ANOVA procedure on the RLHH variant are shown in Table 6.4. The model tested looked for the first-order effects of all RLHH parameters and the second-order interaction of *crossover with* and *number of agents*. The results show that the most relevant parameters are the selection of the crossover and the number of agents, but there seems to be no detectable interaction among them. As for this variant, the ϵ value is also significant.

RLHH-ET. Moving to the RLHH-ET parameters, the results of the ANOVA are reported in Table 6.5. Also in this case we test for the first-order effects of all RLHH parameters and the second-order interaction of the *crossover with* and *number of agents*.

[2]We have generated the equivalent normalized function plots with the data made public by the CHeSC organizers about the competition results, the plots are available (together with the R/GGPLOT2 code to generate them and the original data) at the addres http://www.diegm.uniud.it/urli/?page=chesc

	Sum Sq	**Mean Sq**	**F value**	**Pr(>F)**	
Number of agents (1)	0.74	0.18428	3.9860	0.003112	**
Crossover with (2)	2.41	2.41167	52.1648	5.509e-13	***
ϵ	0.33	0.16684	3.6087	0.027125	*
α	0.05	0.02666	0.5766	0.561834	
(1) $\times$ (2)	0.03	0.01464	0.3167	0.728581	
Residuals	425.15	0.04623			

Table 6.4: ANOVA for the RLHH variant.

The results show that in the case of this variant, the relevant parameter is the *trace decay*, apart of the selection of the agent for the crossover that was relevant also in the basic variant of the algorithm. For this variant, the number of agents seems not to be relevant and it has been set to 4 in the subsequent experiments.

	Sum Sq	**Mean Sq**	**F value**	**Pr(>F)**	
Number of agents (1)	0.034	0.0169	0.4074	0.6654	
Crossover with (2)	3.012	1.5060	36.2430	3.637e-16	***
ϵ	0.105	0.1046	2.5171	0.1128	
Trace decay λ	4.156	4.1562	100.0202	$<$ 2.2e-16	***
(1) $\times$ (2)	0.321	0.0803	1.9318	0.1026	
Residuals	76.792	0.0416			

Table 6.5: ANOVA for the RLHH-ET variant.

RLHH-MLP. Finally, for the RLHH-MLP variant, we tested all the parameters of the MLP and their second-order interactions. The results are summarized in Table 6.6 and they show that, apart *cross-over with*, the *input scale* and *hidden neurons* are relevant in explaining the differences in the performance of the algorithm. We discovered no significant second-order interaction among MLP parameters, and this lead to hypothesis that neural network parameters are quite orthogonal in this setting.

	Sum Sq	Mean Sq	F value	Pr(>F)	
Cross-over with	1.395	1.39471	28.6795	8.965e-08	***
ϵ	0.043	0.04305	0.8852	0.346840	
Hidden neurons (1)	1.814	0.90714	18.6536	8.555e-09	***
Learn. rate change (2)	0.425	0.42484	8.7361	0.003136	**
Reactivity (β)	0.220	0.11017	2.2655	0.103893	
Input scale	5.318	2.65898	54.6769	$< 2.2e-16$	***
(1) $\times$ (2)	0.249	0.12453	2.5608	0.077357	.
Residuals	219.130	0.04863			

Table 6.6: ANOVA for RLHH-MLP variant.

6.4.5 Tuning procedure

The winning parameter configurations found out through F-Race for the three variants of the algorithm are reported in Table 6.7 (parameter names and values have been compacted in a non-ambiguous way because of space issues).

Variant	ag	cw	ϵ	α	β	tr	λ	*is*	*hl*	*hn*	*lrc*
RLHH	5	opt.	0.05	0.2	0.5						
RLHH-ET	4	opt.	0.1	0.1	0.1	0.01	0.5				
RLHH-MLP	4	opt.	0.05	0.1	0.5			1	1	20	0.001

Table 6.7: Winning configurations (parameters with singleton domains have been omitted for brevity).

6.5 Comparison with others

To conclude, we perform a comparison with the other participants in the competition. In Figure 6.2 we report the distribution of results achieved by the three variants of our algorithm and by the other participants on the whole benchmark set (6 problem domains). The values reported are the normalized function values and the algorithms are sorted from bottom to top by the median value of the normalized cost function (the smaller the better).

From the figure it is possible to note that we improve our results

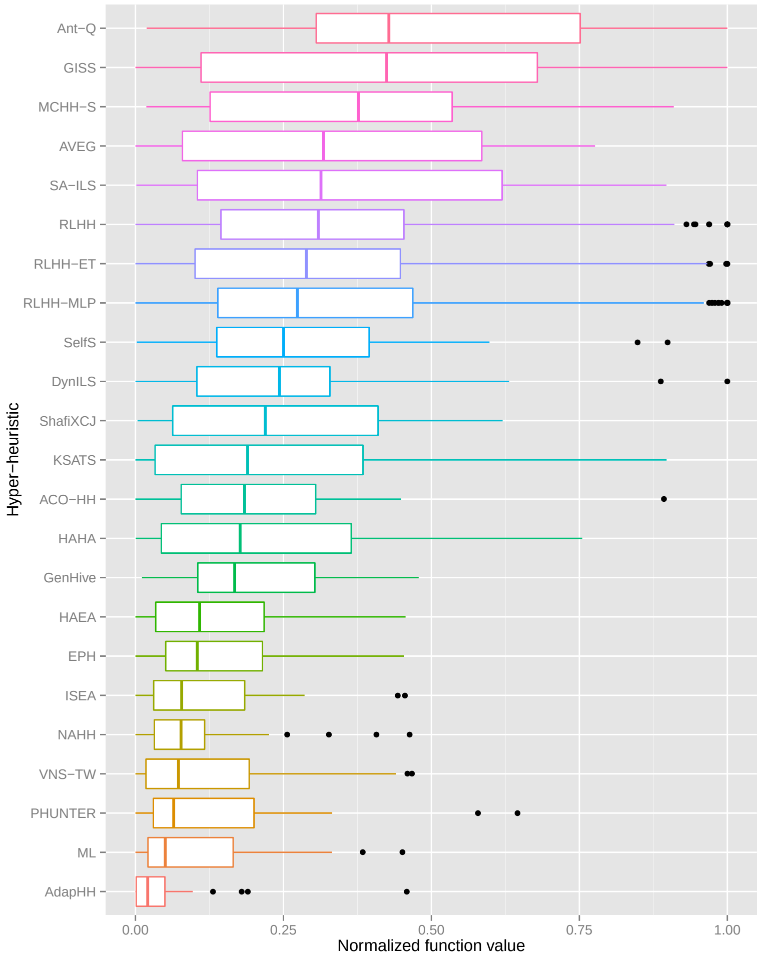


Figure 6.2: Comparison of our hyper-heuristics with the other participants.

with respect to the version of the algorithm we submitted to the competition (denoted by AVEG in the plot). The other interesting thing to point out is the fact that function approximation through the multi-layer perceptron has shown to be very useful and has lead to an improvement of the results. On the contrary, eligibility traces have not provided any improvement over the basic RLHH when all the variants have been properly tuned.

6.6 Other findings: ILS-like behavior

Iterated local search (ILS) consists in repeatingly improving the quality of a solution s by performing a neighborhood search until stagnation, and then restarting the search by strongly perturbing the obtained local optimum. Figure 6.3 is a visual log of a run of the hyper-heuristic on a SAT instance, and shows how the policy learned by RLHH implements an ILS-like behavior.

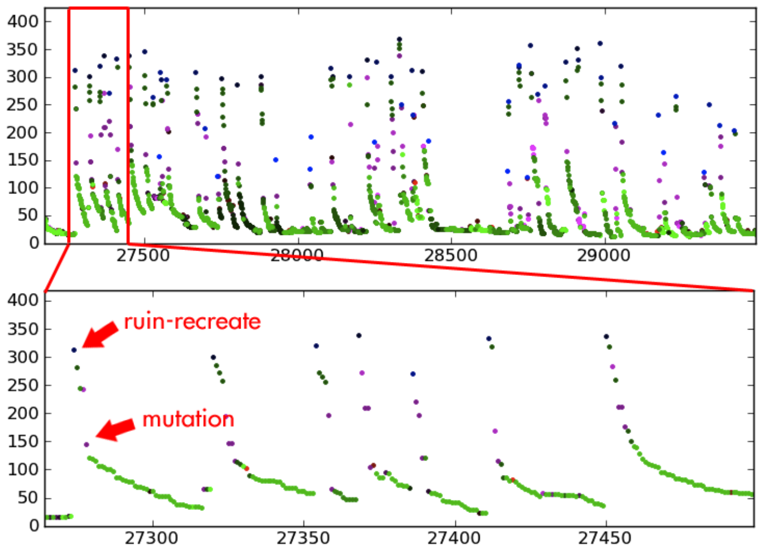


Figure 6.3: ILS-like behavior of RLHH on a SAT problem instance.

Each point corresponds to the application of a low-level heuristic, where the colors encode specific heuristic families (*local search* in green, *ruin recreate* in blue, *mutation* in purple, and *cross-over* in red) and the position on the y-axis the cost of the obtained solution. It is quite easy to see that the search is iterative, with an initial perturbation through a ruin-recreate heuristic, followed by a long sequence of local search steps. When the cost reaches a local optimum, the reward obtained by local search decreases, and the action values of local search heuristics decreases, and gets lower than the value of ruin-recreate heuristics, which are activated in order to restart the search.

Conclusions

We presented our results about reinforcement learning-based hyper-heuristics, in the context of the first cross-domain search challenge (CHeSC'11). Moreover, in Appendix B we discuss some of the main underlying concepts behind the presented hyper-heuristics. Finally, we provided some interesting findings that emerged from this research.

Chapter 7

Propagation-based meta-heuristics

In this chapter we discuss a popular class of hybrid heuristics, namely *propagation-based meta-heuristics*. Such methods take advantage of the extensive research about *constraint propagation* carried out in the constraint programming community, to improve the performance of meta-heuristics. In particular, we explore two different paradigms of integration. The first considered meta-heuristic is large neighborhood search, where constraint propagation is used as a sub-procedure to reduce the size of neighborhoods in a neighborhood search algorithm. The second one takes the opposite perspective, and uses the learning capabilities of ant colony optimization to bias the value selection heuristics of a master constraint solver.

In both cases, we provide a detailed explanation of the approach, and we present a practical implementation of the approaches as Gecode extensions.

7.1 Large Neighborhood Search (LNS)

Large Neighborhood Search (LNS) [97, 84] is a neighborhood search meta-heuristic based on the observation that exploring a *large neighborhood*, i.e., perturbating a significant portion of a solution, generally leads to local optima of much *higher quality* with respect to the ones obtained with regular neighborhood search. While this is an undoubted advantage in terms of search, it does not come without a price. In fact, exploring a large neighborhood structure can be *computationally impractical*, and, in general, requires a much higher effort than exploring of a regular neighborhood.

In order to cope with this aspect, LNS has been coupled more often than not with *filtering* techniques, aimed at reducing the size of the neighborhood by removing beforehand those moves that would lead to unfeasible solutions. In particular, a very natural way to implement LNS is to integrate the underlying neighborhood search mechanism with a constraint programming model, in order to leverage the power of constraint propagation to reduce the size of the neighborhood while traveling from one candidate solution to another in the search space. Such kind of approach has been successfully employed to tackle complex routing problems, e.g., VRP with time windows [10, 90].

In this section, we first present the large neighborhood search meta-heuristic as described in [84]. Then, we discuss some improvement to the basic algorithm, which can be used to improve the performance of the approach. Finally, we briefly overview a LNS meta-engine for the Gecode framework, which can be applied to any CP model, provided that it is extended with the right methods.

7.1.1 Algorithm idea

A the overall structure of LNS is shown in Algorithm 10, where the perturbative structure of a standard neighborhood search (NS) can be recognized.

Initialization

The algorithm is started with a COP $N = \langle X, D, C, f \rangle$ and, through an InitializeSolution function, it generates a starting feasible solution (10). In principle, whatever procedure can be used to generate the starting solution, e.g., a domain-specific heuristic or a tree search. The only requirement for InitializeSolution is that it

Algorithm 10 Large neighborhood search (LNS)

```
procedure LargeNeighborhoodSearch($N = \langle X, D, C, f \rangle, d$)
    $i \leftarrow 0$
    $s \leftarrow$ InitializeSolution($N$)
    while $\neg$StoppingCriterion($s, i$) do
        $N' \leftarrow$ Destroy($s, d, N$)
        $n \leftarrow$ Repair($N', f(s)$)
        if $n \neq$ Null then
            $s \leftarrow n$
        end if
        $i \leftarrow i + 1$
    end while
    return $s$
end procedure
```

generates a *feasible* solution. Of course, generating an initial solution which is also good provides a head start to the LNS procedure, and is therefore highly recommended.

Destroy-repair loop

Once the initial solution has been generated, the algorithm enters a refinement loop, which consists of two alternate steps. First, the *destroy* step, carried out by the Destroy function, unassigns (or *relaxes*) a subset of the decision variables, yielding a new COP N'. Second, the *repair* step, carried out by the Repair function, re-optimizes the relaxed variables, typically by means of an exhaustive tree search. In fact, N' is a domain-based tightening of N. The main idea behind LNS is that the problem N' faced by the repair step is much tractable than N for two reasons

1. the number of variables in N' is smaller, and often much smaller, than the number of variables in N, and
2. the domains of the free variables in N' are smaller than their counterparts in N, because of the constraint propagation due to the assigned variables in N'.

In the following we describe in more detail these two steps.

Variable relaxation. In the destroy step, a fraction of the variables is relaxed. There are many ways to specify the number of variables that are relaxed, a common one is to define a *destruction rate* $d \in]0,1]$, and then to relax $d \cdot |X|$ variables, where X is the set of decision variables. This allows to choose the fraction of variables to relax based on the size of the problem. Of course, different values of d originate different neighborhoods and imply different search efforts. For instance, at the most extreme cases, when $d = 1$ the original solution is completely replaced by a new one and local information is lost, while if $d \approx 0$ most of the solution is retained, and only a small neighborhood is explored. Note that if $d = 1$, them th method is equivalent to a full tree search (and is then complete).

As for the specific variables to relax, the Destroy function can either implement an unbiased strategy, in which the variables are chosen uniformly at random, or a heuristic strategy, in which the variables are chosen considering some knowledge of the problem. A typical choice, in the latter case, consists in relaxing those variables whose variables are responsible for the cost of the solution being high. A third family of relaxation strategies consists in mixing unbiased and heuristic strategies, e.g., follow an heuristic most of the time, but every now and then perform a random relaxation.

Re-optimization. Once a subset of the decision variables have been relaxed, a new solution is produced through a Repair function. Ideally, the repair function should return the *optimal* solution for the subproblem N', i.e., the assignment of the free variables that minimizes the cost function. Unfortunately, depending on the number and the choice of the relaxed variables, coming up with the optimal solution for N' might not be possible. A strategy, in this case, is to give a resource budget to the tree search, e.g., time limit or maximum number of dead-ends, and obtain a good solution instead than the best one.

The search proceeds until the StoppingCondition is met, which can be any of the ones described in Section 3.2 for standard NS.

7.1.2 Variants

Of course, the one presented in the previous section is just a basic variant of LNS. Some more sophisticated versions have been proposed over the years, here we brifly describe two possible ones.

Destruction rate adaptation

So far, except for custom destroy strategies, the performance of our LNS approach depends on a single parameter d. While this makes parameter tuning much easier, this also means that finding the right value for d is critical for the performance. On the one hand, if the destruction rate is too small, the search could become stuck, e.g., we relax 2 variables, but it is impossible to improve the solution by perturbing less than 3 variables. On the other hand, if the destruction rate is too large, the tree search might become slow, and disrupt the neighborhood search mechanism.

A common way to cope with this issue, is to *adapt* the destruction rate d as the search proceeds. In particular, the idea is to start the search with $d = d_{init}$. If this is sufficient to find an improving solution often enough, then the search continues until the stopping condition. However, if the search gets stuck in a situation as the one described above, then, after a number of iterations without improvements, the d is increased, up to a maximum of d_{max}. In the standard destruction rate adaptation scheme, when a new improving solution is found, d is reset to d_{init}.

Of course, this increases the number of parameter of the algorithm by one, as d is removed and d_{init}, d_{max} are introduced. Moreover, a strategy to increase d towards d_{max} has to be devised. A common one is to transform d as an integer however it provides to LNS an additional flexibility, which is a point for its use in practical applications.

Constraining the subproblems (solution acceptance)

A typical concern, when implementing LNS, is whether the Repair function should be constrained to provide solutions of better quality than the current one, or not. On the one hand, if the repair step is constrained, the tree search can benefit from cost-based constraint propagation (as in branch & bound), and it is impossible for the method to cycle and always return the same solution. However, an unfortunate choice of the variables to relax could block the search, especially when using a fixed heuristic for relaxation. On the other hand, leaving the tree search unconstrained is similar in spirit to Tabu Search, where the only requirement for a solution to be accepted is that it be the best in the neighborhood. As we have seen in Section 3.2.5, this is a good technique for avoiding local optima, but is prone to cycling, as there is no mechanism to prevent that the same solution is always returned. Moreover, the effect of cost-based constraint propagation is disrupted.

A number of trade-offs between these two alternatives can be devised. A family of techniques consist in allowing a *limited* increase of the cost value, without leaving the search completely unconstrained. The technique we describe here is inspired to simulated annealing (SA, Section 3.2.4), which in which a worsening solution is accepted with a probability $p = e^{-t/\Delta}$, where Δ is the difference in quality from the current solution and t is the current value of the temperature parameter. The mechanism we employ is the following. First, we draw a random number $p \sim \mathcal{U}(0, 1)$. This corresponds to the probability of acceptance in the classic SA. Then, we reverse the probability formula of SA, and we compute Δ as

$$\Delta = -(t \ln p)$$

where t is the current temperature. Δ can be used to constrain the cost of the next solutions to be generated by Repair, thus achieving cost-based constraint propagation, while allowing worse solutions to be generated. Note that when employing this strategy, all the parameters in SA (t_0, t_{min}, λ, and ρ in case of cutoffs) must be added to the algorithm and tuned.

7.1.3 Implementation: GECODE-LNS

We have implemented the LNS approach described in the previous sections as a generic Gecode *meta-engine* (Gecode-LNS) which uses the built-in branch & bound engine to implement the repair step. GECODE-LNS can be applied to every CP model implemented in gecode, provided that the following methods are implemented

INITIALSOLUTIONBRANCHING posts a random branching strategy on the model, that will be used to generate the initial feasible solution through the built-in branch & bound engine.

NEIGHBORHOODBRANCHING posts a heuristic (non random) branching strategy on the model, that will be used to generate the neighboring solutions of the current solution s through the built-in branch & bound engine.

RELAX(N', d) (activated on s) given an empty copy N' of the original COP N (*space* in GECODE) and the current solution s, assigns a fraction $1-d$ of N''s variables so that they are equal to the variables in s. This is equivalent to relaxing a fraction d of variables in s; the different perspective is a consequence of the fact that GECODE is a *copying* constraint system, as opposed to *trailing* constraint systems (fore more details on the difference, see [95]).

RELAXABLEVARS returns the total number of variables that can be relaxed, i.e., $|X|$. This is needed to compute the number of variables to relax based on d.

IMPROVING(s) (activated on n) returns whether the new solution n is improving with respect to s (this must be implemented because GECODE supports both minimization and maximization problems).

CONSTRAIN(s, Δ) (activated on N') constrains the cost of next solution n to be smaller than the cost of s plus a Δ. This method is necessary to implement the cost bound described in Section 7.1.2.

All the parameters described in the previous sections are accessible through GECODE's *command line interface* (CLI). The meta-engine has been open-sourced, and it is available under the permissive MIT License at the address https://bitbucket.org/tunnuz/gecode-lns.

7.2 ACO-driven CP (ACO-CP)

As we have seen in Section 4.2, ACO is a constructive meta-heuristic that, during the search process, learns how to assign values to variables in an optimal way. Because of its constructive nature, the way ACO builds the solutions resembles that of a CP solver. Moreover, similarly to ACO, many CP solvers employ dynamic value selection heuristics to choose the values for the decision variables. It is therefore natural to think of an integration of ACO and CP.

The first attempt in the literature is [79], where a method for solving a Job-Shop Scheduling problem is presented. The proposed procedure employs ACO to learn the branching strategy used by CP in the tree-search. The solutions found by CP are fed back to ACO, in order to update its probabilistic model. In this approach, ACO can be conceived as a master online-learning branching heuristic aimed at enhancing the performance of a slave CP solver. A slightly different approach has been taken in [64, 65]. Their algorithm works in two phases. At first CP is employed to sample the space of feasible solutions and the information collected is processed by the ACO procedure for updating the pheromone trails. In the second phase, the pheromone information is employed as the value ordering used for CP branching. Unlike the previous one, this approach uses the learning capabilities of ACO in an offline fashion.

In this section we present an integration between ACO and CP which shares some similarities with the one presented in [79], but is based on the hyper-cube framework (HCF) for ACO [17]. We then present an implementation of the approach, which can be applied to any GECODE model.

7.2.1 Algorithm idea

The overall structure of the algorithm is presented in Algorithm 11. First, the pheromone values are set to an initial value (in our case 0.5, as suggested in [17]). Then the algorithm enters the main refinement loop, in which, at each iteration, n_{ants} solutions are generated stochastically according to the transition rule described in Equation 4.4. The generated solutions are first checked to identify a new best, and then are added to a temporary set $\mathbf{U}$ which will be used to update the pheromones. After all the n_{ants} solutions have been added to $\mathbf{U}$, the pheromone trail parameters are updated by the UPDATEPHEROMONE function, according to the formula in Equation 4.5.

Since the one described in Algorithm 11 is essentially a special case of the HCF for ACO described in Section 4.2, the behavior of the approach is controlled by the same parameters (ρ, n_{ants}). Also, the discussed variants, i.e., MMAS and the heuristic information η, can be easily integrated in the method.

7.2.2 Implementation: GECODE-ACO

We have implemented this ACO-driven CP approach as a GECODE meta-engine (GECODE-ACO) which employs a custom branching strategy based on the stored pheromone matrix. The software de-

Algorithm 11 ACO-driven CP (ACO-CP)

```
procedure ACO($N = \langle X, D, C, \rangle n_{ants}, \rho$)
    $\tau \leftarrow$ INITIALIZEPHEROMONE()
    $i \leftarrow 0$
    $best \leftarrow$ NULL
    while $\neg$STOPPINGCRITERION($i$) do
        $\mathbf{U} \leftarrow \emptyset$
        for $a \in \{1, \ldots, n\}$ do
            $u_a \leftarrow$ TREESEARCH($N, \tau$)            ▷ See Equation 4.4
            if $f(u_a) < f(best)$ then
                $best \leftarrow u_a$
            end if
            $U \leftarrow \mathbf{U} \cup \{u_a\}$
        end for
        $\tau \leftarrow$ UPDATEPHEROMONE($\tau, \mathbf{U}, \rho$)            ▷ See Eq. 4.5
        $i \leftarrow i + 1$
    end while
    return $best$
end procedure
```

sign is similar to the one of GECODE-LNS: the solver can be used on any model, provided that some hook methods are implemented. Namely

VARS returns an array of all the decision variables that must be handled with ACO. Note that using GECODE-ACO on a subset of the variables is a sensible choice, since ACO can tackle some kind of problems better than some others.

QUALITY (activated on each u_k) measures the quality of a solution, by default $1/f(s)$ if the solution is complete, and 0 if the solution is incomplete (on minimization problems).

BetterThan tells whether the considered solution is better than another solution by comparing the cost (this must be implemented because GECODE supports both minimization and maximization problems).

All the parameters described in the previous sections are accessible through GECODE's *command line interface* (CLI). Moreover, some additional options are available to control the behavior of the solver, namely $-$`aco_constrain` $\in \{0, 1\}$ activates the bounding on the cost for the generation of the new solutions, and $-$`aco_update_on_best` $\in \{0, 1\}$ activates a variant of the algorithm in which the pheromones are updated every time a new best is found.

The meta-engine has been open-sourced, and it is available under the permissive MIT License at the address https://bitbucket.org/tunnuz/gecode-aco.

Conclusions

We discussed propagation-based meta-heuristics, a class of hybrid meta-heuristics standing at the cross-roads between constraint programming and meta-heuristics. In particular, we accurately described two approaches, namely large neighborhood search, which consists of a neighborhood search algorithm enhanced with constraint propagation, and ACO-driven constraint programming, a constraint programming solver driven by a learning branching heuristic. The advantage of using such methods is that the modeling phase can be easily separated from the solving phase, thus achieving *separation of concerns*, one of the main advantages in constraint programming. Moreover, this allows to use incomplete methods, while still taking advantage of the high-level modeling language provided by constraint programming.

Part III

Applications

Chapter 8

Balancing Bike Sharing Systems

Bike sharing systems are a very popular means to provide bikes to citizens in a simple and cheap way. The idea is to install bike stations at various points in the city, from which a registered user can easily loan a bike by removing it from a specialized rack. After the ride, the user may return the bike at any station (if there is a free rack). Services of this kind are mainly public or semi-public, often aimed at increasing the attractiveness of non-motorized means of transportation, and are usually free, or almost free, for the users. Such systems have recently become particularly popular, and an essential service, in many big cities all over the world, e.g., Vienna, New York, Paris, and Milan.

Depending on their location, bike stations have specific patterns regarding when they are empty or full. For instance, in cities where most jobs are located near the city centre, the commuters cause certain peaks in the morning: the central bike stations are filled, while the stations in the outskirts are emptied. Furthermore, stations located on top of a hill are more likely to be empty, since users are less keen on cycling uphill to return the bike, and often leave their bike at a more reachable station. These differences in flows are one of several reasons why many stations have extremely high or low bike loads over time, which often causes difficulties: on the one hand, if a station is empty, users cannot loan bikes from it, thus the demand cannot be met by the station. On the other hand, if a station is full, users cannot return bikes and have to find alternative stations that are not yet full. These issues result in substantial user dissatisfaction which may eventually cause the users to abandon the service. This is why nowadays most bike sharing system providers take measures to *rebalance* them.

Balancing a bike sharing system is typically done by employing a fleet of trucks that move bikes overnight between unbalanced stations. More specifically, each truck starts from a depot and travels from station to station in a tour, executing loading instructions (adding or removing bikes) at each stop. After servicing the last station, each truck returns to the depot.

Finding optimal tours and loading instructions for the trucks is a challenging task: the problem consists of a vehicle routing problem that is combined with the problem of distributing single-commodities (bikes) to meet the demand. Furthermore, since most bike sharing systems typically have a large number of stations ($\geq$ 100), but a small fleet of trucks, the trucks can only service a subset of unbalanced stations in a reasonable time, therefore it is also necessary to decide *which* stations should be balanced.

This chapter describes algorithms and methods to rebalance bike sharing systems, and it is based on the results described in two papers, [39] and [38], which I co-authored, and that have been presented respectively at HM'13, the 8th International Workshop on Hybrid Metaheuristics, and at CP'13, the 19th International Conference on Principles and Practices of Constraint Programming.

8.1 Related work

Balancing bike sharing systems (BBSS) has become an increasingly studied optimization problem in the last few years. A number of proposed approaches are based on ILP and MILP models of the BBSS problem. For instance, in [8], the authors consider the rebalancing as hard constraint and the total travel time as an objective to minimize. They study approximation algorithms on various instance types and derive different approximation factors for based on specific instance properties. Furthermore, they present a *branch*

& cut [80] approach based on an ILP model including subtour elimination constraints. In [34] the dynamic variant of the problem is considered, and a MILP model with alternative Dantzig-Wolfe (column generation) and Benders (row generation) decompositions is proposed to tackle large instances of the problem. [88] describes two different MILP formulations for the static BBSS and also consider the stochastic and dynamic factors of the demand. In [30], a branch & cut approach based on a relaxed MILP model is used in combination with a Tabu Search solver that provides upper bounds.

Another research line focuses on very efficient local search methods for BBSS. In [85] a heuristic approach for the BBSS is described, which couples variable neighbourhood search (VNS) for route optimization, with an helper algorithm that takes care of finding the optimal loading instructions. Finally, [94] propose a new cluster-first route-second heuristic, in which the clustering problem simultaneously considers the service level feasibility constraints, and the approximate routing costs. Furthermore, they present a constraint programming model for the BBSS that is based on a scheduling formulation.

8.2 Problem formulation

In the following, we consider the static case of the BBSS, in which it is assumed that no bikes are moved independently between stations during the rebalancing operations (in other words, no customers are using the service during rebalancing, which is a valid approximation for balancing systems overnight).

Bike sharing systems consist of a set bike stations $\mathbf{S} = \{1, \ldots, S\}$ that are distributed all over the city. Each station $s \in \mathbf{S}$ has a maximum capacity of C_s bike racks and holds, at any instant, b_s bikes where $0 \leq b_s \leq C_s$. The target value t_s for station s states how many bikes the station should ideally hold to satisfy the customer demand. The values for t_s must be derived in advance from a user demand model, usually based on historical data, so that $0 \leq t_s \leq C_s$ (see [103] for an example). Please note that, depending on the demand and on the formulation, stations can be considered either as “sinks” or “sources”. This means that bikes cannot be removed from "sink" stations, and bikes cannot be added to "source" stations.

A fleet of vehicles $\mathbf{V} = \{1, \ldots, V\}$ with capacity $c_v > 0$ and initial load $\hat{b}_v \geq 0$ for each vehicle $v \in \mathbf{V}$, can be used to move bikes between stations $s \in \mathbf{S}$ to reach the target values t_s. Each vehicle starts its tour from the depot, denoted by D, and must go back to it at the end of the service. Thus, the set of possible stops in a tour is denoted $\mathbf{S}_d = \mathbf{S} \cup \{D\}$. The vehicles have a budget of $\hat{t} > 0$ time units to complete the balancing operations (after which every vehicle must have reached the depot). The traveling times between all possible stops are given by the matrix $travel\_time_{u,v}$ where $u, v \in \mathbf{S}_d$. Note that, in some formulations, the traveling times matrix also includes an estimate of the processing time needed to serve the station.

The goal is to find a tour for each vehicle, including loading instructions for each visited station. The loading instructions state how many bikes have to be removed from, or added to every station. Clearly, the loading instructions must respect the maximum capacity and current load of both the vehicle and the station. Furthermore, each vehicle can only operate within the overall time budget, and, in the considered formulation [85], must distribute all the loaded bikes before going back to the depot (i.e., the trucks must be empty at the end of their tours).

After every vehicle has returned to the depot, each station $s \in \mathbf{S}$ has a new load of bikes, denoted b'_s. The closer b'_s is to the the desired target value t_s, the better the solution. Thus, the objective is to find tours that manipulate the station states such that they are as close as possible to their target values. Moreover, among all the possible routes, we are interested in finding the lower-cost one r_v for each vehicle $v \in \mathbf{V}$. For this reason, the cost function also includes a time component.

The objective function f contains two components: the sum of the deviation of b'_s from t_s over all stations $s \in \mathbf{S}$, and the travel time for each vehicle

$$f(\sigma) := w_1 \sum_{s \in \mathbf{S}} |b'_s - t_s| + w_2 \sum_{v \in \mathbf{V}} \sum_{(u,w) \in r_v} travel\_time_{u,w}.$$

Note that this defines a scalarization over a naturally multi-objective problem. As such, some points in the Pareto optimal set are hence neglected by construction. The main reason for this choice is the need to compare with the current best approaches [85], which employ an equivalent scalarization. Furthermore, multi-objective propagation techniques are still a relatively unexplored research area.

8.3 Modeling

In this section, we describe two different CP models for the BBSS problem, namely the *routing* model and the *step* model. The first one is an adaptation of the constraint model for the classical *Vehicle Routing Problem* (VRP) proposed in [66]. The second considers BBSS as a planning problem with a fixed horizon (number of steps).

For each model, we present the decision and auxiliary variables, the involved constraints, and a custom branching stragtegy that is used to explore the search tree.

8.3.1 Routing model

The routing model employs successor and precessor variables to represent the path of each vehicle on a special graph G_{VRP} that consists of three different kinds of nodes: *i*) the starting node for each vehicle (the respective depot, which is typically the same for all vehicles), *ii*) the nodes that should be visited in the tour, and *iii*) the end node for each vehicle, again the respective depot. In summary, G_{VRP} contains $2V + S$ nodes, where V is the number of vehicles and S is the number of nodes to visit. This graph structure allows to easily define successor and predecessor variables to represent paths. Note that, in this encoding, the starting and ending depots of the vehicles (which are mapped on the unique depot D) are treated as separate nodes.

This VRP-based model is extended to allow unvisited stations and to capture loading instructions on a per-station basis. To achieve this, we introduce a dummy vehicle v_{dummy} that (virtually) visits all the unserviced stations. This artifice allows us to treat unvisited stations as a cost component, and makes it easier to ensure that no operations are scheduled for unvisited stations, by constraining the load of the dummy vehicle to be always zero. This results in an extension of the G_{VRP} graph to a graph G_{BBSS} that contains $2(V+1)+S$ nodes, where $V+1$ is the number of vehicles including the dummy vehicle.

Such encoding is illustrated in Figure 8.1, where the basic structure is shown on the lower layer, and the encoded G_{BBSS} and a possible solution is shown on the upper layer.

We store all the nodes of G_{BBSS} in an ordered set $\mathbf{U}$ which is defined as follows

$$\begin{array}{lll|l}
\mathbf{U} = \{ & 0, \ldots, V, & & \mathbf{V}_s\text{: start nodes} \\
 & V+1, & & \text{station } 1 \\
 & V+2, & & \text{station } 2 \\
 & \ldots, & & \ldots \\
 & V+S, & & \text{station } S \\
 & V+S+1, \ldots, 2V+S+2 & & \mathbf{V}_e\text{: end nodes} \\
\} & & &
\end{array}$$

Thus, $\mathbf{U}$ contains first the starting nodes (depots) for the V vehicles and the dummy vehicle, followed by the S regular stations, and finally the end nodes (also depots) for the V real vehicles plus the dummy one. Note, that $\mathbf{V}_s = \{0, \ldots, V\}$ is the set of start nodes of vehicles and $\mathbf{V}_e = \{V+S+1, \ldots, 2V+S+2\}$ is the set of end nodes of each vehicle. In summary, the tour of vehicle $v \in \mathbf{V}$ starts at a depot in $\mathbf{V}_s$, continues to some station nodes in $\mathbf{S}$ and ends at a depot in $\mathbf{V}_e$.

Variables

The vehicle routes are represented by $|\mathbf{U}|$ successor variables `succ` with domains $\{1, \ldots, |\mathbf{U}|\}$, where each $\texttt{succ}_i$ represents the node following node $i \in \mathbf{U}$. Moreover, we morel the inverse relation using $|\mathbf{U}|$ predecessor variables `pred`, with the obvious semantics. Though redundant, `pred` variables result in stronger propagation [66] when channeled with `succ` variables.

Vehicles are associated to the respective nodes by means of $|\mathbf{U}|$ vehicle variables `vehicle` ranging over $\{0, \ldots, V\}$. Note that, as

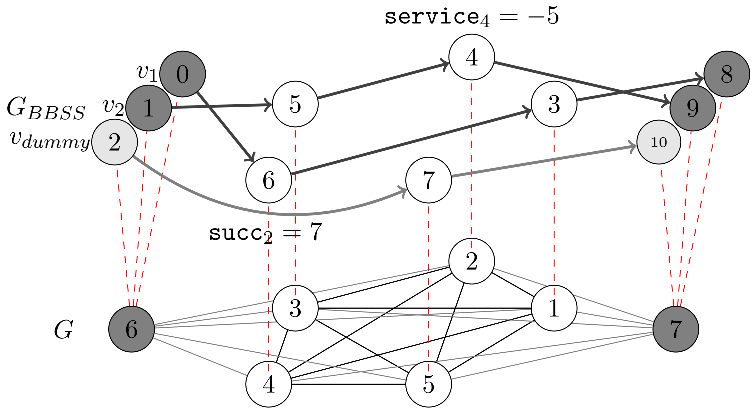


Figure 8.1: Graph encoding of the BBSS problem employed in the routing CP model. The lower layer shows the original graph, whereas the upper layer shows the encoded graph in the case of two vehicles, and a solution. The path starting at node 2 and ending at node 10 (i.e., the dummy vehicle) corresponds to the set of unserved nodes.

a consequence, every node $i \in \mathbf{U}$ must be visited by exactly one vehicle (that is, $\texttt{vehicle}_i$).

The loading instructions for each node are captured by $|\mathbf{U}|$ operation variables `service`, representing the number of bikes that are added or removed at node $i \in \mathbf{U}$, and ranging over $[-c, +c]$, where $c = max(C_{max}, c_{max})$, and C_{max} and c_{max} are respectively the maximum capacities of stations and vehicles. The amount of bikes on the trucks after visiting each node $i \in \mathbf{U}$, is modeled by $|\mathbf{U}|$ load variables `load`.

In order to model time constraints, we introduce $|\mathbf{U}|$ time variables `time`, where $\texttt{time}_i$ constitutes the arrival time at which $\texttt{vehicle}_i$ vehicle arrives at node i. Recall that in the considered problem formulation, the arrival time also includes the processing time, i.e., the time for handling the bikes at the node i. Moreover, for the problem variant in which loading and unloading times must be considered in the time constraints, we have $|\mathbf{U}|$ `processing` variables which measure how much time is needed to perform service

Name	Dim.	Dom.	Description
`succ`	U	$\mathbf{U}$	successor of $i \in \mathbf{U}$
`pred`	U	$\mathbf{U}$	predecessor of $i \in \mathbf{U}$
`vehicle`	U	$\mathbf{U}$	vehicle serving $i \in \mathbf{U}$
`service`	U	$[-c, +c]$	transferred bikes at $i \in \mathbf{U}$
`load`	U	$[0, c_v]$	load of vehicle v after $i \in \mathbf{U}$
`time`	U	$[0, \hat{t}_v]$	arrival time of v at $i \in \mathbf{U}$
`processing`	U	$[0, \hat{L}]$	processing time at $i \in \mathbf{U}$
`deviation`	U	$\mathbf{S}$	deviation at $s \in \mathbf{S}$
`cost`	1	$[l, u]$	overall cost

Table 8.1: Variables in the CP Model, i is the index of a node.

at station $iin\mathbf{U}$.

Finally, we use S deviation variables `deviation` to represent the deviation from the target values (unbalance) at station $s \in \mathbf{S}$ after the balancing tours. Such variables are used to model the deviation component of the cost function (`cost` variable).

Of the above variables, only `service`, `vehicle`, and `succ` are decision variables, the others are auxiliary variables for modeling. All the variables of the routing model are summarized in Table 8.1.

Constraints

In the following, we present the constraints of the routing model, separating the essential constraints, that are required to comprehensively model the problem, from the redundant constraints, that are used to help the solution process.

Essential constraints. In order to build valid routes for the vehicles, we use two circuit constraints, that cause the `succ` and `pred` variables to form a Hamiltonian path covering all nodes. Note that this requires a post-processing step to split the single Hamiltonian path into $V + 1$ paths, one for each vehicle, plus one for the dummy vehicle. In a previous version of the model, we employed two alldifferent constraints to model a similar restriction, however circuit has better sub-tour elimination properties.

$$\text{circuit}(\texttt{succ})$$
$$\text{circuit}(\texttt{pred})$$

The next step is to establish the successor-predecessor chain for each regular station

$$\texttt{pred}_{\texttt{succ}_s} = s \;\; \forall s \in \mathbf{S}$$
$$\texttt{succ}_{\texttt{pred}_s} = s \;\; \forall s \in \mathbf{S}$$

and the successor-predecessor chain for the start and end nodes where $\hat{s} = V + S$ represents the index of first end node in $\mathbf{U}$

$$\texttt{pred}_v = \hat{s} + v \;\; \forall v \in \mathbf{V}_s$$
$$\texttt{succ}_{\hat{s}+v} = v \;\; \forall v \in \mathbf{V}_s.$$

As for the vehicle constraints, we first set the respective vehicle $v \in \mathbf{V}_s$ for each start- and end-node (depots) in the path

$$\texttt{vehicle}_v = v \;\; \forall v \in \mathbf{V}_s$$
$$\texttt{vehicle}_{\hat{s}+v} = v \;\; \forall v \in \mathbf{V}_s$$

and then we propagate the vehicle chain over the path variables, to ensure that each separate route (sequence of nodes) is served by the same vehicle

$$\texttt{vehicle}_{\texttt{succ}_i} = \texttt{vehicle}_i \;; \forall i \in \mathbf{U}$$
$$\texttt{vehicle}_{\texttt{pred}_i} = \texttt{vehicle}_i \;; \forall i \in \mathbf{U}.$$

Regarding the loading constraints, we first fix the initial load to $\hat{b}_v$ (except for the dummy vehicle, which is constrained to be always empty)

$$\texttt{load}_v = \hat{b}_v \quad \forall v \in \mathbf{V}_s \setminus \{V\}$$
$$\texttt{load}_V = 0$$

then, we state the relation between load and service along a path

$$\texttt{load}_{\texttt{succ}_i} = \texttt{load}_i - \texttt{service}_i \; \forall i \in \mathbf{U}.$$

and we constrain every vehicle to be completely empty at the end of the route

$$\texttt{load}_v = 0 \quad \forall v \in \mathbf{V}_e. \tag{8.1}$$

Furthermore, we constrain the load of the vehicle after visiting station $s \in \mathbf{S}$ so that it doesn't exceed its capacity $c_{\texttt{vehicle}_s}$:

$$\texttt{load}_s \leq c_{\texttt{vehicle}_s} \quad \forall s \in \mathbf{S}.$$

Next come the operation constraints. At first, we model the *operation monotonicity*, i.e., services at station s should either force loading or unloading bikes depending on the current number of bikes b_s and the target value of bikes t_s (“sinks” and “sources”)

$$\texttt{service}_s \leq 0 \quad \forall_{s \in \mathbf{S}} : b_s > t_s \tag{8.2}$$

$$\texttt{service}_s \geq 0 \quad \forall_{s \in \mathbf{S}} : b_s < t_s. \tag{8.3}$$

Note that a service value of 0 is allowed in both cases since a station could remain unserved (e.g., because of the time budget constraints). Additionally, if station s is not served by the dummy vehicle (V), then the service must not be zero, and vice versa

$$(\texttt{vehicle}_s \neq V) \iff (\texttt{service}_s \neq 0) \;\; \forall s \in \mathbf{S}.$$

Then, the service and processing time at the start and end nodes (depots) i is set to zero for all vehicles

$$\begin{aligned} \texttt{service}_i &= 0 \, \forall i \in \mathbf{V}_s \\ \texttt{service}_i &= 0 \, \forall i \in \mathbf{V}_e \\ \texttt{processing}_i &= 0 \, \forall i \in \mathbf{V}_s \\ \texttt{processing}_i &= 0 \, \forall i \in \mathbf{V}_e. \end{aligned}$$

Moreover, the service is limited by the maximum number of bikes in the station, and forbidden to yield a negative number of bikes

$$\begin{aligned} b_s + \texttt{service}_s &\leq C_s \, \forall s \in \mathbf{S} \\ b_s + \texttt{service}_s &\geq 0 \, \forall s \in \mathbf{S} \end{aligned}$$

Finally, we state the time constraints. First, the arrival time (and processing time) at the start depots is fixed to zero

$$\texttt{time}_v = 0 \ \forall v \in \mathbf{V}_s$$

then, the time chain for the successor and predecessor variables is established

$$\texttt{time}_v = \texttt{time}_{\texttt{pred}_v} + \texttt{processing}_{\texttt{pred}_v} + \mathit{travel\_time}_{\texttt{pred}_v, v} \ \forall v \in \mathbf{S} \cup \mathbf{V}_e$$
$$\texttt{time}_{\texttt{succ}_v} = \texttt{time}_v + \texttt{processing}_v + \mathit{travel\_time}_{v, \texttt{succ}_v} \ \forall v \in \mathbf{V}_s \cup \mathbf{S}.$$

At last, the overall working time for each vehicle must be within its time budget

$$\texttt{time}_{\hat{s}+v} \leq \hat{t}_v \ \forall v \in \mathbf{V}. \tag{8.4}$$

The model can be enhanced by some redundant constraints, that will take care of some particular substructures of the problem.

Revisits. In our model, we allow a vehicle to visit a station more than once and with a maximum limit of visits M. In order to model this aspect, we replicated each *non depot* node of the G_{BBSS} graph M times. This addition makes it necessary to add two more types of constraints to keep the model consistent with the formulation. In particular, also for symmetry breaking, we want to make sure that the replicas of a station are used in order, i.e., it does not make sense to visit a replica of a station if the station itself is not visited. This is stated through a set of `activity` variables, that tell whether the service at a station is different from zero, and through a DFA-based regular expression constraint, that forces the sequence of `activity` variables on a station to only have zeroes at the end.

$$\texttt{reg}(1 * 0 *, [\texttt{activity}_s, \ldots, \texttt{activity}_{s+M-1}]) \ \forall s \in \mathbf{S}$$

The second type of constraints regarding revisits is a unary scheduling constraint, which ensure that multiple visits to the same station do not overlap in time.

$$\begin{aligned} \texttt{unary}([&\texttt{time}_s, \ldots, \texttt{time}_{s+M-1}], \\ &[\texttt{processing}_s, \ldots, \texttt{processing}_{s+M-1}], \\ &[\texttt{activity}_s, \ldots, \texttt{activity}_{s+M-1}]) \qquad \forall s \in \mathbf{S} \end{aligned}$$

Redundant constraints. First, because of the monotonicity constraints (8.2,8.3), the stations requiring the unloading of bikes can be removed from the successors of the starting depots

$$\texttt{succ}_i \neq j \; \forall i \in \mathbf{V}_s, j \in \{s \in \mathbf{S} \,|\, b_s < t_s\}.$$

Similarly, because of constraint (8.1), which requires empty vehicles at the end of the path, the stations requiring the loading of bikes must be removed from the predecessors of the ending depots

$$\texttt{pred}_i \neq j \; \forall i \in \mathbf{V}_e, j \in \{s \in \mathbf{S} \,|\, b_s > t_s\}.$$

Finally, we integrate an *early failure detection* for the working time constraint (8.4): if the working time of the current partial solution plus the time to reach the final depot exceeds the total time budget, then the solution cannot be feasible

$$\texttt{time}_i + \texttt{processing}_i + \mathit{travel\_time}_{i,\hat{s}+\texttt{vehicle}_i} \leq \hat{t}_{\texttt{vehicle}_i} \; \forall i \in \mathbf{S}.$$

This is also enforced by a custom propagator that will perform a one-step look-ahead of this constraint. The idea of this propagator (see Figure 8.2) is to prune a node s from the $\texttt{succ}_i$ variable if all two-step-paths from i to an ending depot d_v passing through s will exceed the time budget $\hat{t}_v$.

Cost function. The cost function of the problem is a hierarchical one, and comprises two different major components: the level of unbalancing and the working effort.

The unbalancing component is defined in terms of the `deviation` variables, which are set to be the absolute value of the deviation from the target number of bikes at each station after service has been performed, i.e.:

$$\texttt{deviation}_s = |b_s + \texttt{service}_s - t_s| \; \forall s \in \mathbf{S}$$

The working effort is the sum of the total traveling time (i.e., the sum of the times at which each vehicle reaches its ending depot) plus the overall activity performed throughout the path (i.e., the

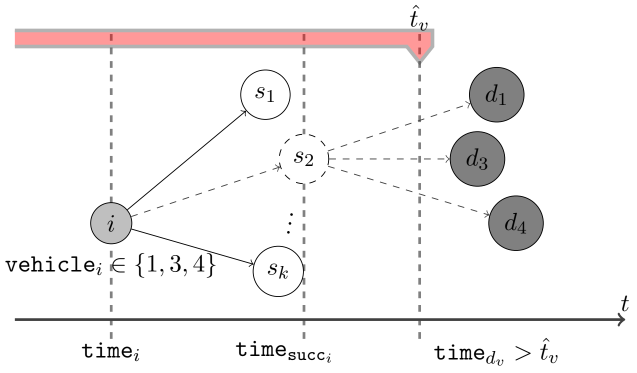


Figure 8.2: Illustration of the look-ahead propagator. Value s_2 can be removed from the domain of $\texttt{succ}_i$ if *all* two steps paths from i through s_2 to a compatible ending depot d_v will exceed the corresponding time budget $\hat{t}_v$.

absolute value of the `service`). The cost function is the weighted aggregation of the two components, i.e.:

$$\begin{aligned}\texttt{cost} = w_1 \sum_{s \in \mathbf{S}} \texttt{deviation}_s \\ + w_2(\sum_{v \in \mathbf{S}} \texttt{time}_{\hat{s}+v} + \sum_{s \in \mathbf{S}} |\texttt{service}_s|)\end{aligned}$$

where $w_1 = 1$ and $w_2 = 10^{-5}$ [85], so that the satisfaction of the first component prevails over the second one. Note that the two objectives are slightly conflicting, as in order to reduce deviation, the service must increase.

8.3.2 Step model

The step model considers BBSS as a planning problem with an horizon of K steps. Thus, the aim is to find a route (with the respective loading instructions) with a maximum length of K for each vehicle, where the first and the last stop are the depot D. As such we introduce a set of steps $\mathbf{K} = \{0, \ldots, K\}$, where 0 is the initial state

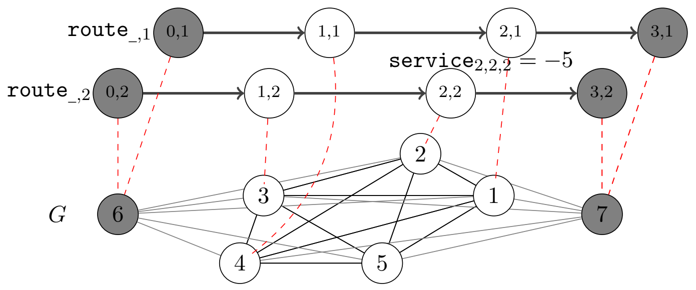


Figure 8.3: Solution representation for the step model. The lower layer shows the original graph, whereas the upper layer shows the decision variables of the step model, i.e., the routes variables for two vehicles, and an example of the service variables.

and step K is the final state, thus each vehicle visits at most $K-1$ stations. K is set to an estimated upper bound

$$K = \left\lceil \frac{\hat{t}}{\tilde{t}} \right\rceil + 1$$

where $\tilde{t}$ is the median of all travel times.

In contrast to the routing model, this formulation allows to directly represent the route of each vehicle by a sequence of stations, as shown in Figure 8.1. By using this encoding, we can formulate some of the constraints more naturally.

Variables

In the step model, we model the routes using $K \cdot V$ route variables `route`, ranging over the possible stops $\mathbf{S}$, where $\texttt{route}_{k,v}$ denotes the k-th stop in the tour of vehicle $v \in \mathbf{V}$.

The service at the various stations is modeled by the `service` variables where $\texttt{service}_{k,s,v}$ represents the number of bikes that are removed or added to station $s \in \mathbf{S}$ at step $k \in \mathbf{K}$ by vehicle $v \in \mathbf{V}$ and therefore ranges over $[-c, +c]$, where $c = C_{max} = max_{s \in \mathbf{S}} C_s$ denotes the maximum capacity over all stations. The load of a vehicle is represented by the `load` variables where $\texttt{load}_{k,v}$ is the load of vehicle $v \in \mathbf{V}$ at step $k \in \mathbf{K}$. The variables $\texttt{n\_bikes}_{k,s}$

Name	Dim.	Domain	Description
`route`	$K \cdot V$	$\mathbf{S}$	stop of vehicle $v \in \mathbf{V}$ at step $k \in \mathbf{K}$
`service`	$K \cdot S \cdot V$	$[-c, +c]$	transferred bikes at station $s \in \mathbf{S}$ by vehicle $v \in \mathbf{V}$ at step $k \in \mathbf{K}$
`activity`	$K \cdot S \cdot V$	$[0, c]$	transfers at stop $s \in \mathbf{S}$ by vehicle $v \in \mathbf{V}$ at step $k \in \mathbf{K}$
`load`	$K \cdot V$	$[0, c]$	load of vehicle $v \in \mathbf{V}$ after step $k \in \mathbf{K}$
`time`	$K \cdot V$	$\mathbf{T}$	time when vehicle $v \in \mathbf{V}$ arrives at station at step $k \in \mathbf{K}$
`processing`	$K \cdot S$	$[0, S]$	time spent at stop $s \in \mathbf{S}$ at step $k \in \mathbf{K}$
`n_bikes`	$K \cdot S$	$[0, c]$	bikes at stop $s \in \mathbf{S}$ after step $k \in \mathbf{K}$

Table 8.2: Variables of the step model

model how many bikes are stored at station $s \in \mathbf{S}$ at step $k \in \mathbf{K}$. Additionally to $\mathbf{K}$, the set $\mathbf{K}_{-1} = \{0, \ldots, K-1\}$ represents the set of steps excluding the last step and $\mathbf{K}_S = \{1, \ldots, K-1\}$ is the set of steps that concern stations, but not the depots (first and last step).

All in all, only the `route` and `service` variables are in fact decision variables. All the model variables are summarized in Table 8.2

Constraints

In the following, we present the constraints of the step model, separating the essential constraints, that are required to comprehensively model the problem, from the redundant constraints, that are used to help the solution process. Note that revisits are implicit in the step model, and thus only need to be limited to a maximum number of visits.

Essential constraints. First, we constrain the initial state of the solution: the first stop of the route of each vehicle $v \in \mathbf{V}$ is the depot, and the initial load of v is $\hat{b}_v$

$$\texttt{route}_{0,v} = D \qquad \forall v \in \mathbf{V}$$
$$\texttt{load}_{0,v} = \hat{b}_v \qquad \forall v \in \mathbf{V}$$

moreover, the initial service is set to zero, as well as the initial time, and the initial number of bikes at station s equals b_s

$$\begin{aligned}\texttt{service}_{0,s,v} &= 0 \quad \forall s \in \mathbf{S}, v \in \mathbf{V}\\ \texttt{time}_{0,v} &= 0 \quad \forall v \in \mathbf{V}\\ \texttt{n\_bikes}_{0,s} &= b_s \quad \forall s \in \mathbf{S}.\end{aligned}$$

Second, we make the formulation consistent by constraining the `activity` at station $s \in \mathbf{S}$ for vehicle $v \in \mathbf{V}$ at step $k \in \mathbf{K}$ to be the absolute value of the respective `service`

$$\texttt{activity}_{k,s,v} = |\texttt{service}_{k,s,v}| \qquad \forall k \in \mathbf{K}, s \in \mathbf{S}, v \in \mathbf{V}$$

moreover, every vehicle $v \in \mathbf{V}$ may only perform actions on at most *one* station at each step $k \in \mathbf{K}$, thus the activity is zero in at least $S - 1$ stations

$$\mathsf{atleast}(\texttt{activity}_{k,v}, 0, S-1) \qquad \forall k \in \mathbf{K}, v \in \mathbf{V}.$$

As for time constraints, we constrain `time` to be always increasing, and we enforce the monotonicity (sink and source stations) by stating that the stations that need to receive bikes to reach their target value must have *positive* `service` (8.5), while stations from which bikes need to be removed to reach their target value, must have *negative* `service` (8.6)

$$\texttt{time}_{k,v} \leq \texttt{time}_{k+1,v} \;\; \forall k \in \mathbf{K}_S, v \in \mathbf{V}$$

$$\texttt{service}_{k,s,v} \geq 0 \;\; \forall k \in \mathbf{K}, v \in \mathbf{V}, s \in \mathbf{S} \; s.t. \; b_s - t_s \leq 0 \tag{8.5}$$

$$\texttt{service}_{k,s,v} \leq 0 \;\; \forall k \in \mathbf{K}, v \in \mathbf{V}, s \in \mathbf{S} \; s.t. \; b_s - t_s \geq 0. \tag{8.6}$$

Then, we define the relation linking `service` variables and the `load` of a vehicle $v \in \mathbf{V}$ at consecutive time steps (8.7), and similarly are the number of bikes at a station after the visit of a vehicle (8.8), and the time of arrival of a vehicle at a station (8.9)

$$\texttt{load}_{k+1,v} = \texttt{load}_{k,v} + \sum_{s \in \mathbf{S}} \texttt{service}_{k+1,v,s} \forall k \in \mathbf{K}_{-1}, v \in \mathbf{V} \tag{8.7}$$

$$\texttt{n\_bikes}_{k+1,s} = \texttt{n\_bikes}_{k,s} - \sum_{v \in \mathbf{V}} \texttt{service}_{k+1,v,s} \forall k \in \mathbf{K}_{-1}, s \in \mathbf{S} \tag{8.8}$$

$$\texttt{time}_{k+1,v} \geq \texttt{time}_{k,v} + \mathit{travel\_time}_{\texttt{route}_{k,v}, \texttt{route}_{k+1,v}} \forall k \in \mathbf{K}_{-1}, v \in \mathbf{V} \tag{8.9}$$

Note that the syntax here is simplified, e.g.

$$travel\_time_{\texttt{route}_{k,v}\texttt{route}_{k+1,v}}$$

is actually expressed using a element constraint.

The `route` and `activity` variables are then linked, and it stated that if vehicle $v \in \mathbf{V}$ has returned to the depot before reaching the maximum number of steps K, then it may not leave it anymore. This allows more flexibility, as each vehicle can visit up to K nodes, but does not necessarily have to

$$(\texttt{activity}_{k,s,v} \geq 0) \Leftrightarrow (\texttt{route}_{k,v} = s) \qquad \forall k \in \mathbf{K}, v \in \mathbf{V}, s \in \mathbf{S}$$
$$(\texttt{route}_{k,v} = D) \Rightarrow (route_{k+1,v} = D) \qquad \forall v \in \mathbf{V}, k \in \mathbf{K}_{-1}.$$

Then, we make the model time consistent: two different vehicles $v_1 \neq v_2 \in \mathbf{V}$ cannot visit the same station at the same time (equivalently, can only visit the same station at different times $k_1, k_2 \in \mathbf{K}_{-1}$). We model this through a unary global constraint, similarly to the routing model

$$\begin{aligned}(\texttt{route}_{k1,v1} = \texttt{route}_{k2,v2} \wedge \texttt{route}_{k1,v1} \neq D) \Rightarrow\\ \mathsf{unary}([\texttt{time}_{k1,v1}, \texttt{time}_{k2,v2}], [\texttt{processing}_{k1,v1}, \texttt{processing}_{k2,v2}])\\ \forall k_1, k_2 \in \{1 \dots K-1\}, v_1, v_2 \in \mathbf{V}, v1 \neq v2\end{aligned}$$

moreover, we use a count constraint (plus a temporary variable c) to ensure that a station is visited at most v_{max} times in a solution. For the current formulation $v_{max} = 1$, however using a different value is legitimate in the step model, and effectively enables station revisiting

$$\mathsf{count}(\texttt{route}_v, c),\ \mathsf{dom}(c, 0, v_{max})\ \ \forall v \in \mathbf{V}.$$

As required by the formulation, the last station visited by each vehicle $v \in \mathbf{V}$ must be the depot, where the load is constrained to be zero, as the service

$$\begin{aligned}\texttt{load}_{K,s,v} =&\quad 0 &\quad \forall s \in \mathbf{S}, v \in \mathbf{V}\\ \texttt{route}_{K,v} =&\quad D &\quad \forall v \in \mathbf{V}\\ \texttt{service}_{K,s,v} =&\quad 0 &\quad \forall s \in \mathbf{S}, v \in \mathbf{V}\end{aligned}$$

Branching Strategies

We have implemented two different branching strategies on both models. Here we describe their logic and the conditions for employing them.

The “cost-wise” strategy (Figure 8.4) cycles through the available vehicles, assigning, at each turn, first the *service* and then the *successor* of the last station in the route of the selected vehicle (except for the initial depot, where the service is fixed, Figure 8.4a). Both the *service* (Figures 8.4b and 8.4d) and the *successor* (Figures 8.4c and 8.4e) of each station are chosen so as to greedily optimize the objective function. To do so, the strategy always chooses the service which maximizes the balancing, and then chooses the successor that can yield the highest balancing at the next step, consistently with the current load of the vehicle. As a consequence of using this branching strategy, in the run depicted in Figure 8.4, the solver achieves a final unbalance of 14 in 10 steps (some of them have been omitted for brevity). This brancher typically obtains very good solutions, but can generate a great number of failures if the vehicles are required to be empty at the end of their service. This happens because, when behaving greedily, it is often necessary to do a lot of backtracking in order to obtain a solution with empty vehicles at the end.

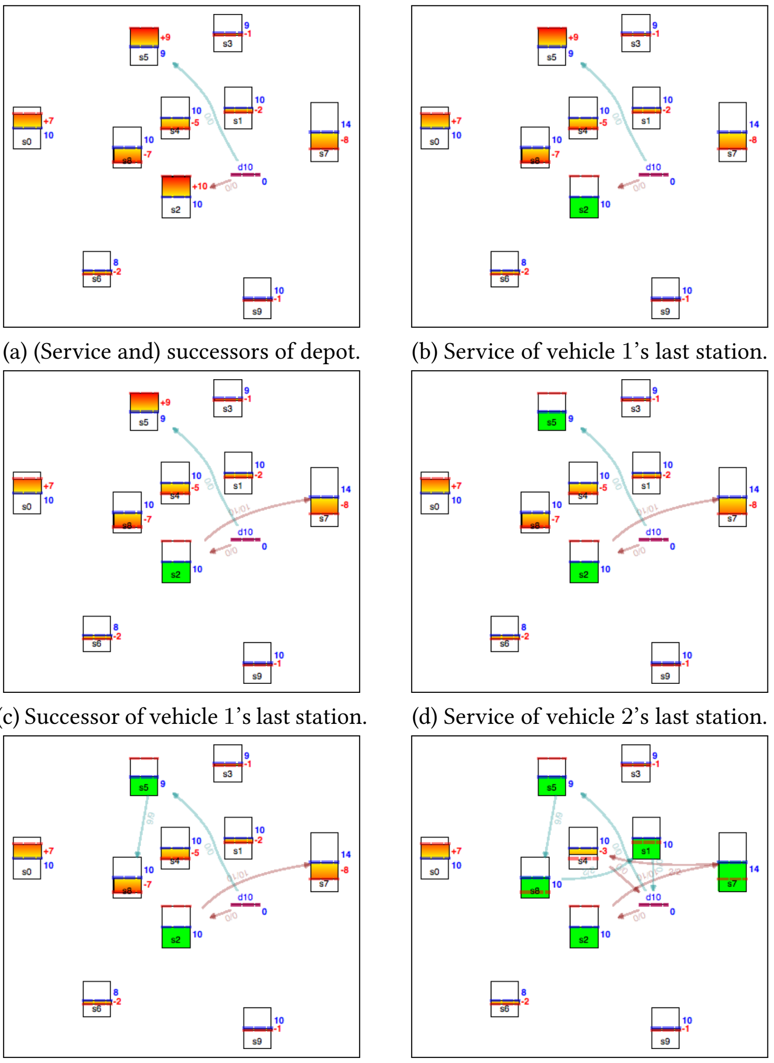
The “feasibility-wise” strategy (Figure 8.5) does not aim at optimizing the objective function. Instead, it attempts to obtain a feasible solution fast, by taking advantage of a simple invariant. After *every* branching step (which corresponds to extending a route by two legs), the load of each vehicle must be *zero*, so that the vehicles can always choose to go back to the depot, yielding a feasible solution. This allows to always satisfy the constraint requiring the vehicles to be empty before returning to the depot. To do so, the strategy considers all the pairs of nodes whose unbalance is complementary, and chooses the one whose mutual rebalancing would be maximal. Thus, at each single branching step, two *successors* and two *services* are fixed (Figures 8.5a, 8.5b, and 8.5c). This, of course, is a restriction over the initial problem Ãš formulation, i.e., this branching strategy does not explore the whole search space. Nevertheless, this is an appropriate brancher to find a reasonably good initial solution, or for priming a heuristic search strategy (such as LNS, described in then next Section) procedure, when the formulation requires that the vehicles be empty at the end of the service. As a consequence of using this branching strategy, in the run depicted in Figure 8.5, the solver achieves a final unbalance of 36 in only 4 steps.

8.4 Search by Large Neighborhood Search

As mentioned in Section 7.1, LNS is a template method whose actual implementation depends on problem-specific details. In particular LNS requires to specify the following aspects

- how the *solution initialization* is carried out;
- the way in which the Destroy procedure is implemented, i.e., *which variables* are chosen for relaxation;
- the way in which the Repair step is defined, i.e., which tree search method is used (branch & bound, depth-first search, ...), which branching heuristics are employed;
- whether the search for the *next solution* stops at the first feasible solution, at the first improving solution or continues until a local optimum is found;
- whether *d is evolved* during the search or not and the range of values it can assume;
- whether the *acceptance criterion* is strict improvement, equal quality or it is stochastic, e.g., as described in 7.1.2;
- the employed *stopping criterion.*

Note that the LNS version implemented for this problem differ slightly from the one described in Section 7.1, as d directly represents the number of variables to be freed at the next Destroy step. As such, d_{min} and d_{max} are integer parameters which must be tuned. While this does not scale along with the dimension of the instance being solved, it makes the duration of the Repair step much controllable.



(a) (Service and) successors of depot.

(b) Service of vehicle 1's last station.

(c) Successor of vehicle 1's last station.

(d) Service of vehicle 2's last station.

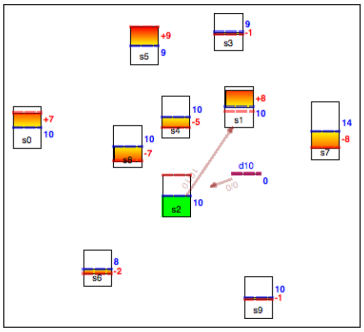
(e) Successor of vehicle 2's last station.

(f) Vehicles return to depot.

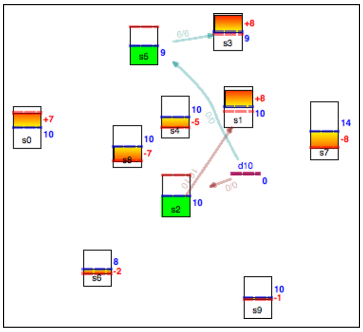
Figure 8.4: Illustration of the "cost-wise" brancher operations.

8.4.1 Common components

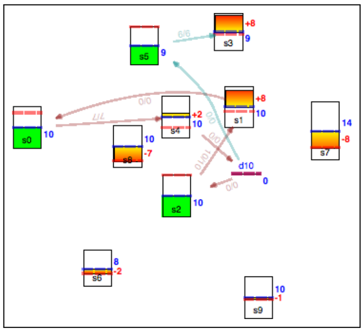
In our approach, most of these aspects are common to both CP models. However, some components, in particular the destroy steps,



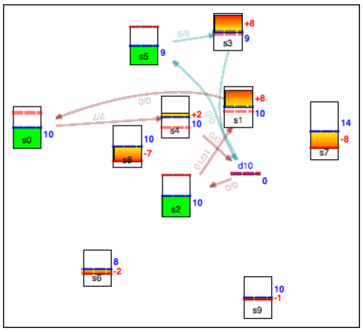
(a) Complementary nodes matched.



(b) Complementary nodes matched.



(c) Complementary nodes matched.



(d) Vehicles return to depot.

Figure 8.5: Illustration of the "feasibility-wise" brancher operations.

are model-specific because they depend on the variables employed for modeling or on the branching strategy. We defer the description of the model-specific components to the last part of this section.

Solution initialization

We obtain the initial solution by performing a tree search with a custom branching strategy tailored for each model. The idea behind our branching strategy is to choose the next station and amount of service so that the total reduction of unbalancing is maximal. Search is stopped after finding the first feasible solution.

Repair step

Similarly to the initialization, the repair step consists of a branch & bound tree search with a time limit, subject to the constraint that the next solution must be of better quality than the current one. The search starts from the relaxed solution and the time budget is proportional to the number of free variables ($t_{var} \cdot n_{free}$) in it. The tree search employs the same branching strategy used for solution initialization.

Acceptance criterion

Our implementation of the algorithm supports various different acceptance criteria, described in the following.

Accept improvement (strict): A repaired solution x_t is accepted if it strictly improves the previous best x_{best}. If the repair step cannot find a solution in the allotted time limit, then an *idle iterations* counter ii is increased. When ii exceeds the maximum number of idle iterations ii_{max}, d is updated.

Accept improvement (loose): A repaired solution x_t is accepted if it is equal or improves the previous best x_{best}, i.e., sideways moves are allowed. If the repair step cannot find an *improving* solution in the allotted time limit, then the idle iterations counter ii is increased (an equivalent solution does not constitute an improvement). When ii exceeds the maximum number of idle iterations ii_{max}, d is updated.

Simulated annealing (SA): First, we draw a number $p \sim \mathcal{U}(0, 1)$ uniformly at random, then we compute the allowed cost increase of the new solution as $\Delta = -(t \ln p)$ (reversed SA rule), where t is the typical temperature parameter in SA. The temperature t is updated as $t = t \cdot \lambda$, with $0 < \lambda < 1$ after ρ solutions have been accepted at such temperature. Once Δ has been computed, we use it to bound the cost of the relaxed solution, and we accept whatever solution results from the branch & bound step. If the repair step cannot find an improving solution in the allotted time limit, then the idle iterations counter ii is increased. When ii exceeds the maximum number of idle iterations ii_{max}, d is updated.

A repaired solution n is accepted as the new best only if it is strictly improving over the previous best $best$. If the repair step cannot find an improving solution in the allotted time limit, then the non-improving iterations counter $iter_{idle}$ is increased. When the iteration counter exceeds the maximum number of idle iterations $idle_{max}$ a new initial solution is designated by using a random branching, and the search is restarted.

Adaptive d

The destruction rate d evolves during the search in order to implement an intensification / diversification strategy and to avoid stagnation of the search. In our implementation, at each step its value is updated as follows

$$d = \begin{cases} min(d+1, d_{max}) & \text{if } x_t > x_{best} \text{ and } ii > ii_{max} \\ d = d_{init} & \text{otherwise} \end{cases} \tag{8.10}$$

where ii is current the number of idle iterations performed with destruction rate d, and ii_{max} is the maximum number of such iterations. This update scheme will increase the *radius* of the neighborhood to allow solution diversification when the repair step cannot find an improving solution in a given neighborhood. When a new best solution is found, the original initial neighborhood radius is reset, so that the exploration of the newly discovered solution region is intensified. When d is updated, the counter ii is reset. If $d = d_{max}$ and the maximum number of idle iterations have been used, the search restarts by perturbing the solution with a $d = 2d_{max}$ destruction rate.

As soon as a new best solution is found, the original initial neighborhood radius is reset, so that the exploration of the newly discovered solution region is intensified.

Stopping criterion

We allow the algorithm to run for a given timeout, when the time is up, the algorithm is interrupted and the best solution found is returned.

8.4.2 Destroy step

As mentioned before, the only model-specific component of our implementation is the destroy step. In fact, this is the most relevant aspect of LNS since it requires a careful selection of the variables that have to be relaxed to define the neighborhood. This selection strongly depends on some specific knowledge about the problem structure in order to avoid unmeaningful combinations.

Destroy step for the routing model.

In the case of the routing model, the relaxed solution is generated by selecting d stations from each route R_i and resetting the `succ`, `service`, and `vehicle` variables of these stations to their original domains. Moreover, also the `succ` variable of the stations preceding the relaxed ones are reset to their original domain to allow for different routes. Note that since we are considering also these variables the final fraction of variables relaxed is in fact greater than d.

Destroy step for the step model.

The relaxed solution of the step model is produced by selecting d internal nodes (i.e., excluding the depots) among all the routes and resetting the `route` and `service` variables.

8.4.3 Experimental evaluation

In this section we report and discuss the experimental analysis of the algorithms. All the experiments were executed using JSON2RUN [105] on an Ubuntu Linux 12.04 machine with 16 Intel Xeon CPU E5-2660 (2.20GHz) cores. For fair comparison, both the CP and the LNS algorithms were implemented in GECODE (v3.7.3), the LNS variant consisting of a specialized search engine and two specialized branchers.

The LNS parameters (ii_{max}, d_{init}, d_{max} and t_{var}) have been tuned by running an F-Race [13] with a confidence level of 0.95 over a pool of 150 benchmark instances from Citybike Vienna. Each instance, featuring a given number of stations $S \in \{10, 20, 30, 60, 90\}$, was considered with different number of vehicles $V \in \{1, 2, 3, 5\}$ and time budgets $\hat{t} \in \{120, 240, 480\}$, totaling 900 problems. The tuning was performed by letting the algorithms run for 10 minutes. The best configurations were $ii_{max} = 30$ for the routing model and $ii_{max} = 18$ for the step model, $t_{var} = 350$ and $d_{min} = 2$ for both models, $d_{max} = 20$ for the routing model and $d_{max} = 10$ for the step model. As for the acceptance criterion, the loose improvement strategy was the best for both models.

For benchmarking, we let the winning configurations for LNS and the pure CP models run for one hour, the results are summarized in Table 8.3.

Model and solution method comparison

The main goal of this comparison is to understand and analyze the behavior of the branch & bound and LNS solution methods for the two problem models. Figure 8.6 shows exemplarily the evolution of the best cost within one search run on an instance from the Citybike Vienna benchmark set featuring 30 stations. The pink and turquoise dashed lines represent the resolution using branch and bound respectively on the routing and the step model. The solid lines represent the median of 10 runs of LNS on the two models. The dark areas represent the interquantile range at each time step, while the light areas represent the maximum range covered by LNS over the 10 runs.

From the plot it is possible to see that, regarding the pure CP approaches, the routing model is clearly outperforming the step model. As for the LNS-based solvers, the situation is quite the opposite, with the step model outperforming the routing model on the median run. However, it should be considered that performance data collected on a single instance is of limited statistical

Instance features			**CP**		**MILP**		**LNS**		**VNS**
			Routing	**Step**	**Bounds**		**Routing**	**Step**	
S	V	$\hat{t}$	$\overline{f}$	$\overline{f}$	$\overline{ub}$	$\overline{lb}$	$\overline{f}$	$\overline{f}$	$\overline{f}$
10	1	120	**28.3348**	**28.3348**	**28.3348**	**28.3348**	**28.3348**	**28.3348**	**28.3348**
10	1	240	4.6027	6.2027	**4.2694**	0.0042	**4.2028**	4.3361	4.2694
10	1	480	**0.0032**	4.3363	0.0033	0.0028	0.2699	**0.0032**	**0.0032**
10	2	120	10.6026	10.6026	**9.8027**	9.4377	10.2027	10.6026	**9.9360**
10	2	240	**0.0034**	4.0033	**0.0034**	0.0032	0.2701	**0.0034**	**0.0034**
10	2	480	**0.0032**	4.6032	0.0033	0.0028	0.2699	0.2033	**0.0032**
20	2	120	58.0029	57.8696	**55.8029**	26.4201	56.2029	55.9363	**55.3363**
20	2	240	**11.6057**	12.0057	19.7388	0.0038	4.9391	6.2724	**4.2058**
20	2	480	6.2062	6.8063	**1.8091**	0.0036	0.8729	0.6731	**0.0061**
20	3	120	42.0041	41.4041	**37.3376**	1.3478	34.0043	34.5376	**31.7376**
20	3	240	7.9398	8.1399	**6.1408**	0.0040	0.8066	0.2067	**0.0065**
20	3	480	**8.2732**	8.4733	13.3419	0.0032	0.8063	0.4067	**0.0061**
30	2	120	112.3362	111.6029	**106.9363**	55.9491	106.2030	105.8697	**104.7363**
30	2	240	48.0726	**47.6726**	74.9389	0.0049	39.8060	40.6726	**34.6061**
30	2	480	**7.8095**	8.0764	69.7407	0.0046	1.3428	0.1430	**0.0093**
30	3	120	91.9375	**89.6042**	90.4042	16.3045	82.7377	80.8710	**78.1377**
30	3	240	20.0751	**19.4085**	61.6072	0.0046	11.9419	12.4085	**7.0752**
30	3	480	**7.4099**	8.8766	175.4000	0.0002	1.0763	0.4099	**0.0093**
60	3	120	**270.6710**	273.0042	274.2710	157.3735	263.2711	264.2710	**253.8046**
60	3	240	**163.3423**	171.5424	370.2000	0.0000	151.2091	147.5426	**126.7428**
60	3	480	40.0175	**37.9508**	–	–	28.3508	21.8841	**6.6176**
60	5	120	**250.2735**	250.7401	289.2711	34.6978	217.6070	211.9405	**196.6075**
60	5	240	**104.2144**	125.0811	370.2000	0.0000	92.5478	63.8813	**41.4816**
60	5	480	**22.9547**	36.8216	–	–	16.0219	3.8210	**0.0190**
90	3	120	**466.0043**	470.7376	492.2032	290.5999	453.9378	452.3378	**441.6047**
90	3	240	**343.1426**	359.6760	566.2667	0.0000	327.1427	319.0095	**294.4765**
90	3	480	**172.5516**	177.7517	–	–	164.6182	135.8851	**100.9522**
90	5	120	**428.6736**	436.2736	566.2667	0.0000	402.4071	393.7406	**376.0743**
90	5	240	**265.9483**	304.3482	–	–	253.9482	206.4820	**174.2157**
90	5	480	102.2955	**95.0287**	–	–	127.8287	20.0954	**1.4285**

Table 8.3: Comparison of our approaches with the MILP and the best VNS approach of [85]. These results are published in [40].

significance. As for the comparison between pure CP approaches and LNS-based ones, the latter exhibit better anytime properties, reaching low areas of the objective function much faster then their branch & bound counterparts. Of course this comes at the price of completeness, and we expect CP approaches to rival with or outperform the LNS-based ones given enough time. It is worth notic-

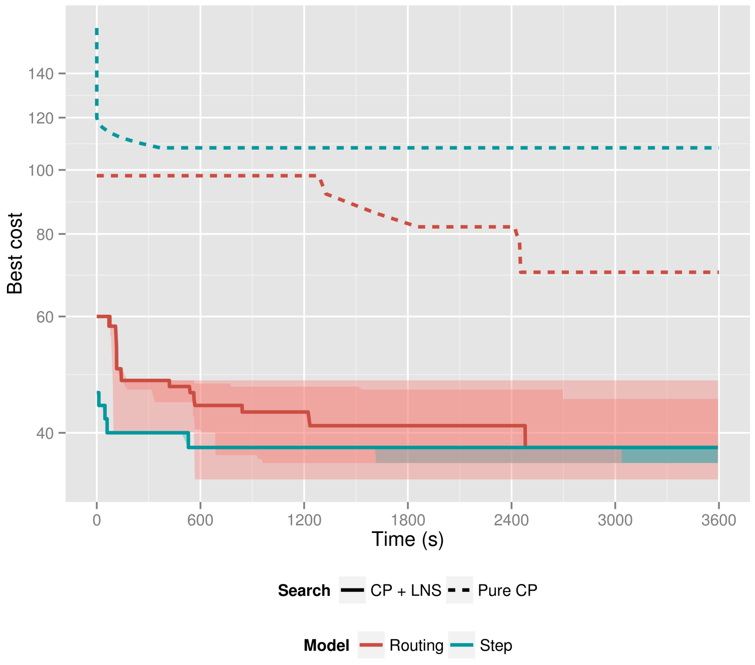


Figure 8.6: Evolution of the best cost for the pure CP (branch & bound) and LNS solution methods for the routing and the step model (30 stations, 2 vehicles, time budget 480 minutes)

ing that this result is quite consistent across the whole benchmark set.

Comparison with other methods

In this second experiment, we compare our CP and LNS solution methods with the state-of-the-art results of [85], who solved the same set of instances using a Mixed Integer Linear Programming (MILP) solver and a Variable Neighborhood Search (VNS) strategy. The result of the comparison against the best of the three different VNS approaches in [85] are reported in Table 8.3. The reported results in each row are averages over 150 instances, grouped by size, number of vehicles and available time for the trucks to complete the tour. Cells marked with a dash refer to instance classes for which the algorithm cannot reach a feasible solution within a hour. In these cases it makes no sense to compute a mean.

We first proceed in comparing approaches belonging to the same family of methods: i.e., exact / heuristics. As for the exact methods (namely the two CP variants and MILP), it is possible to observe that the CP models consistently reach better results than MILP for mid- and big-size instances ($S \geq 30$). Moreover they are able to find at least one solution on instances for which MILP was not able to find any result. In these settings, the routing model performs better than the step model.

Moving to the comparison of heuristic methods, the clear (and overall) winners are the VNS procedures [85]. Nevertheless, it is possible to notice that LNS is further improving over the solutions found by CP, justifying its use on this problem.

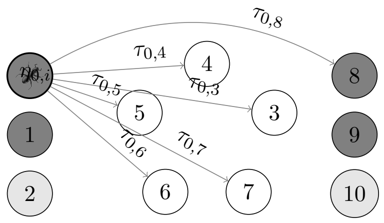
Overall, our LNS approach appears more robust with respect to the largest instances, where pure CP often fails to find even a feasible solution. However, similarly to the pure CP method, also in this case there is no clear winner.

8.5 Search by ACO-driven CP

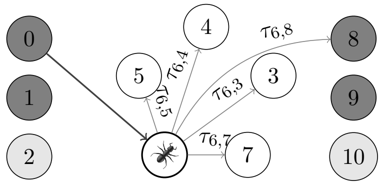
In our CP model for the BBSS problem, there is a natural partition of the decision variables into two families, i.e., routing and operation variables.

8.5.1 Handling of routing variables

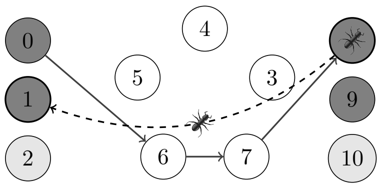
The first set of variables, $\texttt{succ}_i$, is handled very naturally by ACO, which has been shown to be particularly effective in solving routing problems. In our approach, ACO is embodied by a two-phase branching strategy which takes care both of variable and value selection. This process is illustrated in Figure 8.7.



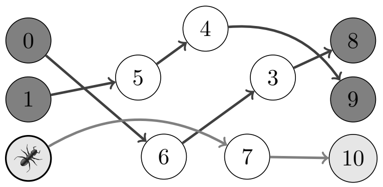
(a) The ant is first placed at the starting depot of the first vehicle.



(b) The value of the $\texttt{succ}_i$ variable is selected according to the pheromones $\tau_{i,j}$.



(c) Once the ending depot is reached, the ant starts with the route of the next vehicle.



(d) All remaining nodes are assigned to the dummy vehicle (i.e., they are left unserved).

Figure 8.7: Illustration of the graph traversal performed by one ant.

Variable selection

The first variable to be selected, according to the heuristic, is the `succ` of the first vehicle starting depot (Fig. 8.7a). As for the next variable to assign, we always choose the one indicated by the value of the last assigned variable, i.e., the `succ` of the last assigned node (Fig. 8.7b). By following this heuristic, we enforce the completion of existing paths first. If the successor of the last assigned node is a final depot (Fig. 8.7c), then we cannot proceed further on the current path, and we start a new one by assigning the successor of the next starting depot. Once the paths of all vehicles are set, the remaining unserved nodes will be assigned to the dummy vehicle (Fig. 8.7d).

Value selection

Once the next variable to assign is chosen, all the values in its current domain are considered as candidates. Note that, in this, we are in fact exploiting problem-specific knowledge, as the domain of a variable is, at any time, determined by the constraint propagations activated earlier in the search.

The next step is where ACO comes into play. For our approach we have used the HCF for ACO algorithm described in Section 7.2. As most other ACO approaches, HCF for ACO maintains a pheromone table in which each $\langle X_i, v_j \rangle$ (variable, value) pair has a corresponding $\tau_{i,j}$ pheromone value indicating the desirability of value v_j for the variable X_i.

In line with the majority of ACO variants, our value selection heuristic is stochastic, with the probability of choosing a specific value being proportional to the corresponding τ-value. In particular, our transition rule is the one described in Equation 4.4.

8.5.2 Handling of operation variables.

The operation variables are assigned through a depth-first tree-search based on `deviation` variables, which are the main component of our cost function. We employ a *minimum value* heuristic which gives priorities to lower values when assigning `deviation` variables. As a consequence, lower cost solutions are produced before bad ones. Note that this is possible as the deviation does not appear in any hard constraint, and so there is no danger of generating unfeasible solutions.

While other choices are possible, e.g. a full exploration of the tree by branch & bound, in this context we aim at finding quickly feasible solutions, so that they can be used for learning. The rationale behind this choice is that decisions taken towards the root of the search tree have a greater impact than the ones taken towards the leaves, and τ-updates are the only way to improve our ACO-based value selection heuristic.

8.5.3 Pheromone update

The pheromone update implements the update rule described in Section 7.2. All the solutions $\mathbf{U}$ generated by the n_{ants} ants are thus used to update the τ-values.

8.5.4 Experimental evaluation

For fair comparison and convenience, both the pure CP and the ACO-CP methods were implemented in Gecode (v3.7.3), the ACO variant consisting in specialized branching and search strategies.

All the experiments were run with json2run on an Ubuntu Linux 12.04 machine with 16 Intel Xeon CPU E5-2660 (2.20GHz) cores.

All pheromones were initially set to $\tau_{\max} = 1$, as suggested by [101] in order to foster initial exploration. The ρ parameter and the number of ants n_{ants} have been tuned by running an F-Race with a confidence level of 0.95 over a pool of 210 benchmark instances from Citybike Vienna. Each instance, featuring a given number of stations, was considered with different number of vehicles ($V \in \{1, 2, 3, 5\}$) and time budgets ($\hat{t} \in \{120, 240, 480\}$). Moreover, the algorithms were allowed to run for three different timeouts (30, 60, 120 seconds), totaling 7560 problems.

We tuned the number of ants $n \in \{5, 10, 15, 20\}$ and the learning rate ρ together, as we expected an interaction between the two parameters. The 8 candidate values for ρ were instead sampled from the low-discrepancy Hammersley point set in $[0.4, 0.8]$. This interval was chosen according to previous exploratory experiments, with $\rho \in [0, 1]$ and 32 samples.

The result of the tuning process is that, for the considered set of problems, the best setup involves 5 ants and $\rho = 0.65$.

Comparison between CP and ACO-CP.

The main goal of this comparison is to understand if a dynamic branching strategy based on ACO can indeed outperform a static branching strategy. Figure 8.8 shows the results on an instance from the Citybike Vienna benchmark set featuring 30 stations. The choice of this instance has been driven by the fact that a time budget of 2 minutes was too low for CP to obtain even a single solution on larger instances.

The results of the tuning show that ACO-CP clearly outperforms the pure CP approach. In fact, the CP solver is declared significantly inferior by the F-Race procedure after just 15 iterations.

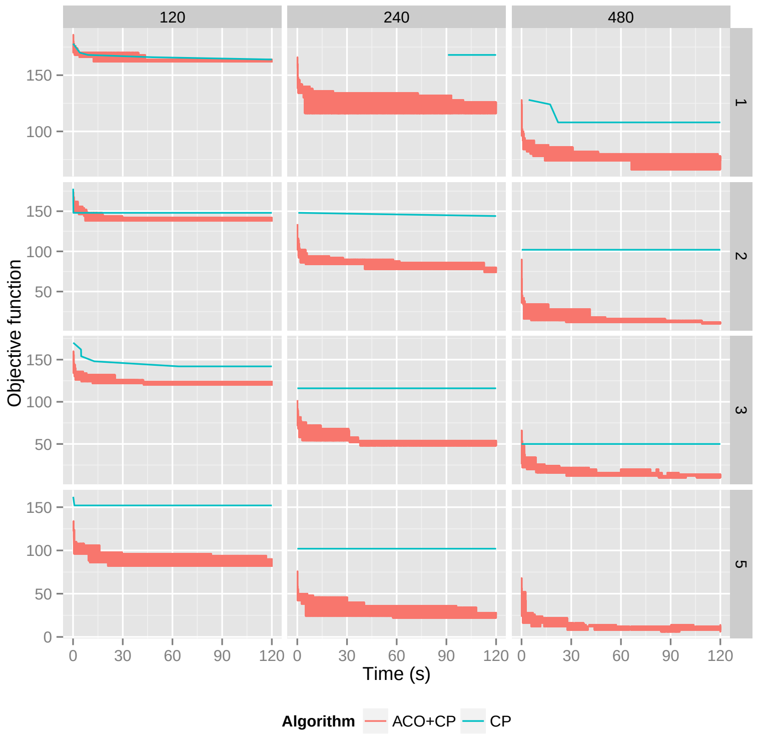


Figure 8.8: Comparison between ACO-CP (dark, solid lines) and CP (light, thin lines) on a problem instance with 30 stations. The columns of the graph matrix represent the vehicle time budget and the rows represent the number of available vehicles.

The superior behavior of ACO-CP is confirmed also from the analysis reported in Figure 8.8, for the variants of a single problem instance with 30 stations. Note that the ACO-CP data is based on 5 repetitions of the same experiment, as the process is intrinsically stochastic.

It is possible to see that the cost values achieved by ACO-CP are always lower than those of CP and in one case (namely time budget 480 and 5 vehicles) CP is even not able to find a solution within the granted timeout despite the fact that it is somehow a loosely constrained instance.

Comparison with other methods.

In this second experiment, we compare both ACO-CP and CP with state-of-the-art results of [85]. Note that these experiments were carried out with a previous version of the model, described in [39], and upon which the ACO-CP approach was developed (this is the reason why the results in the CP columns of Table 8.3 differ from the ones in the next table). The main difference in the two versions of the model regard the "feasibility-wise" strategy for branching and the better use of global constraints. The results of the comparison are reported in Table 8.4, where we compare against the best of the three different VNS strategies described in [85]. The results reported are averages across instances with the same number of stations.

In this respect, the results are still unsatisfactory, since the best VNS approach is outperforming our ACO-CP on almost all instances. Nevertheless, our ACO-CP is able to do better than the MIP approach for mid- and large-sized instances.

Conclusions

We discussed a specific combinatorial optimization problem, namely the problem of balancing bike sharing systems (BBSS). We presented two orthogonal constraint programming models for the same problem formulation, and showed how to solve those model through propagation-based hybrid meta-heuristics. Our results were compared with the state-of-the-art results on this problem.

Instance			CP			ACO+CP			MIP [85]			VNS [85]	
S	V	$\hat{t}$	$\overline{obj}_{30}$	$\overline{obj}_{60}$	$\overline{obj}_{120}$	$\overline{obj}_{30}$	$\overline{obj}_{60}$	$\overline{obj}_{120}$	$\overline{ub}$	$\overline{lb}$	$\overline{time}$	$\overline{obj}$	$\overline{time}$
10	1	120	28.3477	28.3477	28.3477	28.5344	28.7477	28.5478	**28.3477**	28.3477	4	**28.3477**	2
10	1	240	14.0908	11.4915	9.5589	5.2276	4.7609	4.4810	4.2942	0.0424	3600	**4.2941**	10
10	1	480	14.8247	13.2922	9.8942	0.4322	0.9120	0.6052	0.0320	0.0276	3600	**0.0317**	17
10	2	120	10.2266	10.2266	10.2266	10.4001	10.6667	10.4268	**9.8269**	9.4768	911	9.9601	3
10	2	240	5.3652	4.6987	2.7662	0.4342	0.9274	0.1009	0.0340	0.0322	856	**0.0339**	19
10	2	480	5.3637	4.8971	3.2976	0.4854	0.4586	0.8584	**0.0317**	0.0313	1245	**0.0317**	15
20	2	120	72.4942	70.4279	68.7614	64.2558	62.9492	61.4561	**5.8294**	26.9012	3600	55.3628	8
20	2	240	74.2422	72.3754	71.5087	18.3904	17.3372	15.3907	19.7884	0.0383	3600	**4.2575**	58
20	2	480	74.1093	72.7093	72.5756	3.9748	2.7743	2.8943	1.8906	0.0403	3600	**0.0615**	142
20	3	120	67.5712	64.1051	61.5053	48.4287	47.1622	45.0557	37.3759	1.4770	3600	**31.7763**	13
20	3	240	74.3813	74.1811	74.1143	7.5511	5.8310	4.7641	6.2083	0.0401	3600	**0.0650**	65
20	3	480	74.3814	74.1811	74.1144	4.5878	2.5211	2.5874	13.4191	0.0316	3600	**0.0614**	114
30	2	120	127.5604	126.4939	125.5608	122.4552	119.2022	118.0823	106.9631	56.3908	3600	**104.7633**	12
30	2	240	117.2520	116.5857	116.3188	75.7637	73.2173	70.4309	74.9886	0.0487	3600	**34.6608**	109
30	2	480	101.8650	101.8650	101.4652	13.5173	11.1311	9.3847	69.8069	0.0432	3600	**0.0925**	491
30	3	120	–	117.7058	115.6393	107.7879	105.1748	102.0554	90.4419	16.6454	3600	**78.1773**	21
30	3	240	104.6052	104.4719	104.0054	46.0564	42.7502	40.5769	61.6715	0.0461	3600	**7.1523**	191
30	3	480	100.7422	100.6089	100.6089	–	10.7450	8.5595	175.4000	0.0015	3600	**0.0925**	399
60	3	120	–	–	–	307.8148	304.4283	300.2154	274.3101	157.7350	3600	**253.8462**	45
60	3	240	–	–	–	245.3374	238.1644	233.6585	370.2000	0.0000	3600	**126.8282**	521
60	3	480	205.8871	205.8871	205.8870	127.2744	122.7286	117.6223	–	–	3600	**6.7758**	3600
60	5	120	–	–	–	283.0537	278.0540	272.8145	289.3111	34.9784	3600	**196.6749**	99
60	5	240	–	–	–	184.7371	179.0572	173.6710	370.2000	0.0000	3600	**41.6161**	1556
60	5	480	–	–	–	–	–	–	–	–	3600	**0.1902**	3600
90	3	120	–	–	–	511.8807	507.2943	504.2013	492.2319	290.8990	3600	**441.6473**	82
90	3	240	–	–	–	451.7232	445.4705	438.2044	566.2667	0.0000	3600	**294.5646**	985
90	3	480	–	–	–	334.6610	326.4350	319.5826	–	–	3600	**101.1221**	3600
90	5	120	–	–	–	490.3193	480.7739	473.9345	566.2667	0.0000	3600	**376.1432**	169
90	5	240	–	–	–	393.4433	383.3375	375.5915	–	–	3600	**174.3566**	3304
90	5	480	–	–	–	213.3140	202.3017	192.3832	–	–	3600	**1.6855**	3600

Table 8.4: Comparison of CP and ACO-CP with MIP and the best VNS approach in [85].

Chapter 9

Curriculum-Based Course Timetabling

Course Timetabling (CTT) [93] is a popular combinatorial optimization problem, which deals with generating university timetables by scheduling weekly lectures, subject to conflicts and availability constraints, while minimizing costs related to resources and user discomfort. Thanks mainly to the international timetabling competitions ITC-2002 and ITC-2007 [78], two formulations have, to some extent, arisen as “standard”. These are the so-called *Post-Enrollment Course Timetabling* (PE-CTT) [72] and *Curriculum-Based Course Timetabling* (CB-CTT) [37]. These two formulations have received attention in the research community, so that many recent articles deal with either one of them. The distinguishing difference between these two formulations is the origin of conflicts between courses, which is based on student enrollment, in PE-CTT, and on predefined curricula, in CB-CTT. This is however only one of the differences, which actually include also many other distinctive features and cost components. For example, in the PE-CTT, each course is a self-standing event, whereas in CB-CTT a course consists of multiple lectures. Consequently, the soft constraints are different: in PE-CTT they are all related to events, penalizing late, consecutive, and isolated ones; in CB-CTT they mainly involve curricula and courses, ensuring compactness in a curriculum, trying to evenly spread the lectures of a course in the weekdays, and possibly preserving the same room for a course.

In this chapter, we address the CB-CTT variant of the problem, which has been used up to recently to schedule the courses at the University of Udine. More specifically, we show two different solution methods, namely a neighborhood search by Simulated Annealing (SA), and a Large Neighborhood Search (LNS) based on a novel CP model for CB-CTT. The chapter is based on the results described in two papers, [5] and [104], which I respectively co-authored and authored, and have been presented to the 6th Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA'13), and to the Doctoral Program of CP'13. Moreover, a follow-up of [5] has been recently submitted to a relevant journal of the field.

9.1 Related work

In this section, we briefly discuss the literature on CB-CTT. The section is organized as follows: first we report meta-heuristic and constraint-based resolution techniques. Then, we revise exact methods and lower bounds. Finally, papers that investigate additional aspects related to the CB-CTT problem, such as instance generation, are discussed.

9.1.1 Meta-heuristic approaches

In [81], the author presents a constraint-based solver which incorporates several local search algorithms operating in three stages: a construction phase which uses an *Iterative Forward Search* algorithm to find a feasible solution, a first search phase delegated to a *Hill Climbing* algorithm, followed by a *Great Deluge* or *Simulated Annealing* to escape from local minima. The algorithm won two out of three tracks of ITC-2007 and was among the finalists in the remaining track. Also the *Adaptive Tabu Search* proposed by [74] follows a three-stage scheme: in the initialization phase a feasible timetable is built using a fast heuristic; then the intensification and diversification phases are alternatively applied through an adaptive tabu search, in order to reduce the soft constraints. A novel hybrid meta-heuristic technique, obtained combining *Electromagnetic-like Mechanisms* and the *Great Deluge* algorithm, has been applied by [1]. They obtained good results on both CB- and PE-CTT testbeds. Finally, [75] investigated the search performance of different neighborhood relations used by local search algorithms. The behavior of neighborhood is compared using different evaluation criteria, and new combinations of neighborhoods are explored and analyzed.

9.1.2 Constraint programming

In [26] a hybrid approach for PE-CTT is described, which shares many similarities with the hybrid approach described later in this chapter. Among other things, the authors propose a CP model, coupled with a LNS search strategy, to tackle complex instances of the problem. Some interesting insight is given on the approach. First, the authors stress the importance of releasing the right variables during the LNS step. Second, they propose a Simulated Annealing (SA) acceptance criterion to escape local minima. Third, they handle feasibility and optimization in separate search phases. In *et al.* [32] a framework for the integration of a CP solver with LNS is presented, anda simple CTT variant, where conflicts are solely determined by the availability of teachers, is solved.

9.1.3 Exact methods and lower bounds

Several authors employed exact methods with the twofold objective of finding solutions and lower bounds.

In [24] a hybrid method based on the decomposition of the whole problem in different sub-problems, each one solved using a mix of different IP formulations is implemented. Subsequently [19], the same authors presented a new MIP formulation based on the concept of “supernode” which is used to model graph coloring problems. This new encoding has been applied also to CB-CTT benchmarks, and compared with the standard two-index MILP model developed in CPLEX, showing that the supernodes formulation is able to considerably reduce computational time. Lastly, in [23]

Instance	[24]	[71]	[23]	[54]	[54]♮	[3]♭	[25]	Best known
comp01	**5**	4	4	4	4	0	**5**	5 *
comp02	6	11	11	0	12	**16**	**16**	24
comp03	43	25	25	2	38	28	**52**	66
comp04	2	28	28	0	**35**	**35**	**35**	35 *
comp05	183	108	108	99	183	48	166	29
comp06	6	12	10	0	22	**27**	11	27
comp07	0	**6**	**6**	0	**6**	**6**	**6**	6 *
comp08	2	**37**	**37**	0	**37**	**37**	**37**	37 *
comp09	0	47	46	0	72	35	92	96
comp10	0	**4**	**4**	0	**4**	**4**	2	4 *
comp11	0	0	0	0	0	0	0	0 *
comp12	5	57	53	0	**109**	99	100	30
comp13	0	41	41	3	**59**	**59**	57	59 *
comp14	0	46		0	**51**	**51**	48	51 *
comp15					**38**	28		66
comp16					16	**18**		18 *
comp17					48	**56**		56 *
comp18					24	**27**		62
comp19					**56**	46		57
comp20					2	**4**		4 *
comp21					**61**	42		75

Table 9.1: Lower bounds for the comp instances. Provably optimal results in the "Best known" column are denoted by an asterisk.

a procedure is developed and lower bounds are obtained for various formulations. In [71], the authors propose an IP approach that decomposes the problem in two stages: the first one, whose goal is to assign courses to periods, is focused mainly on satisfying the hard constraints; the second one takes care of soft constraints and assigns lectures to rooms by solving a matching problem. [54] describes a partition-based approach to compute new lower bounds: The original instance is divided into sub-instances through an *Iterative Tabu Search* procedure, and each subproblem is solved via an ILP solver using the model proposed by [71]. The lower bound for the original problem is obtained summing up the lower bounds of the sub-instances. Recently, [25] computed new lower bounds using an approach somewhat similar to the one by [54], however in this case the partition is based on soft constraints. Once the initial problem is partitioned, two separated problem are formulated as ILPs and then solved to optimality by a *Column Generation* technique. In [3], the authors present an application of several satisfiability (SAT) solvers to the CB-CTT problem. The difference among these SAT-solvers is in the encoding used, that defines in each case which constraints are considered soft or hard. Using different encondings, they were able to compute new lower bounds and obtain new best solutions for the benchmarks. Finally, [4] translate the CB-CTT formulation into an Answer Set Programming (ASP) problem and solve it using the *clasp* ASP solver.

A summary of the results of the cited contributions is given in Table 9.1. There are reported lower bounds for the instances of the ITC-2007 testbed, called comp, where the tightest ones are highlighted in bold. In the table, we also report the best results known at the time of writing. Best values marked with an asterisk are guaranteed optima (i.e., they match the lower bound).

9.1.4 Instance generation

The first instance generator for CB-CTT has been devised in [19], based on the structure of the comp instances. This contribution has been later improved in [73] by Lopes and Smith-Miles, who base their work on a deeper insight on the features of the instances. Some of the results in the rest of this chapter, specifically the tuning process, takes advantage of the generator developed in the latter work, which is publicly available.

9.2 Problem formulation

The formulation of the problem that we use in this paper is the one proposed for the ITC-2007, which is by far the most popular one. Alternative formulations are described in [18]. The CB-CTT formulation of ITC-2007 can be also found in [37]. However, for the sake of self-completeness, we briefly report it here.

The problem consists of the following entities

- **Days $\times$ Timeslots = Periods.** We are given a set $\mathbf{D}$ of teaching *days*, each one partitioned in a set of *timeslots* $\mathbf{T} \subseteq \mathbb{N}$. Each $p \in \mathbf{P} = \mathbf{D} \times \mathbf{T}$ defines a *period* which is unique within a week.
- **Rooms.** Lectures can be scheduled in a set $\mathbf{R}$ of *rooms*, each one with a specific *capacity* k_r for $r \in \mathbf{R}$. Additionally, a *roomslot* $rs \in \mathbf{RS} = \mathbf{R} \times \mathbf{P}$ represents a room in a specific period.
- **Courses.** A course $c \in \mathbf{C}$ is composed of a set $\mathbf{L_c}$ of *lectures*, that must be scheduled at different times. Each course is taught by a *teacher* t_c and is attended by a set of *students* $\mathbf{S_c}$. In addition, the lectures of a course should be scattered over a *minimum number of working days* w_c, and must not be scheduled in any period $u \in \mathbf{U_c} \subset \mathbf{P}$ declared as unavailable. The complete set of lecture is $\mathbf{L} = \bigcup \mathbf{L_c}$ for all $c \in \mathbf{C}$.
- **Curricula.** Courses are organized in *curricula* $q \in \mathbf{Q}$ that students can enrol to. Each curriculum has a set of courses $\mathbf{C_q} \subseteq \mathbf{C}$. Lectures pertaining courses in the same curriculum cannot be scheduled together, in order to allow students of each curricula to attend all courses.

A **feasible** solution of the problem is an assignment of a *period* and a *room*, to each lecture, that satisfies the following **hard** constraints

- **Lectures.** All lectures $\mathbf{L}$ must be scheduled.
- **Room occupancy.** Two lectures $l_1, l_2 \in \mathbf{L}, l_1 \neq l_2$ cannot take place in the same roomslot, no matter what course or curriculum they pertain to.
- **Conflicts.** Lectures in the same course or in conflicting courses i.e., same teacher or same curriculum, may not be scheduled at the same time.
- **Availabilities.** A course may not be taught in any of its unavailable periods.

Feasible solutions can be ranked on the basis of their violations of the following **soft constraints**

Violation	Weight	Symbol
Room capacity	1	rc
Room stability	1	rs
Minimum working days	5	mwd
Isolated lectures	2	il

Table 9.2: Weights of the various types of violations [18]

- **Room capacity.** Every lecture $l \in \mathbf{L_c}$ for $c \in \mathbf{C}$ should be scheduled in a room $r \in \mathbf{R}$ so that $k_r \leq |\mathbf{S_c}|$, which can accomodate all of its students. If this is not the case, $|\mathbf{S_c}| - k_r$ *room capacity violations* are considered.
- **Room stability.** Every lecture $l \in \mathbf{L_c}$ for $c \in \mathbf{C}$ should be given in the same room. Every additional room used for a lecture in course c generates a *room stability violation.*
- **Minimum working days.** Lectures $\mathbf{L_c}$ of a course $c \in \mathbf{C}$ should be scattered over a minimum number of working days w_c. Each time a course is scheduled in $d < w_c$ days, counts as $w_c - d$ *working days violations.*
- **Isolated lectures.** When possible, lectures of the same curriculum should be adjacent within a day. Each time a lecture is not preceded or followed by lectures of courses in the same curriculum counts as a *isolated lectures violation.*

Therefore, in addition of being feasible, a solution s should have the minimal linear combination of soft violations (see weights in Table 9.2)

$$cost(s) = rc(s) \cdot w_{rc} + rs(s) \cdot w_{rs} + mwd(s) \cdot w_{mwd} + il(s) \cdot w_{il} \quad (9.1)$$

For all the details, including input and output data formats and validation tools, see [37].

9.3 Modeling

In this section we present two possible models for the CB-CTT problem, one based on neighborhood search, the other on constraint programming.

9.3.1 Neighborhood search

The proposed neighborhood search method is based on *Simulated Annealing* (see Section 3.2.4), in particular, the approach is based, and improves, the one in [7]. As for all local search methods, we need to define a search space, a proper neighborhood relation, and a cost function.

Search space

The search space is composed of all the assignments of lectures to rooms and periods for which the hard constraint *availabilities* is satisfied. On the contrary, hard constraints *conflicts* and *room occupancy* are considered in the cost function, however their violation is highly penalized (see Equation 9.2).

Neighborhood relations

We employ a composite neighborhood relation, defined by the set of solutions that can be reached by applying either of the following moves to a solution

MoveLecture (ML) Move one lecture from its currently assigned period and room to another period and/or another room.

SwapLectures (SL) Take two lectures of distinct courses in different periods and swap their periods and their rooms.

The overall neighborhood relation is the union of ML and SL. However, since previous studies [7] revealed that the ML neighborhood is more effective by restricting to moves towards an *empty* room in the new timeslot, the same restriction is applied where possible, i.e., where the room occupancy is less than 100%, as there are no empty rooms in such case.

In order to control how often each neighborhood is used, we use a *swap rate* parameter $sr \in [0, 1]$. In detail, the move selection strategy proceeds in two stages: first the neighborhood is randomly selected with a non-uniform probability with bias sr, then a random move in the selected neighborhood is uniformly drawn.

Cost function

In our model, the cost function is augumented with some of the hard constraints (the ones constituting a hindrance to the generation of the first solution), namely *conflicts* ($co(s)$) and *room occupancy* ($ro(s)$)

$$cost'(s) = cost(s) + co(s) \cdot w_{hard} + ro(s) \cdot w_{hard} \tag{9.2}$$

where w_{hard} is a parameter to the algorithm (see Table 9.3). In fact, this value should be high enough to give precedence to feasibility over the objectives, but it should not be too high so to allow the SA meta-heuristic (whose move acceptance criterion is based on the difference in the cost function) to select also moves that will increase the number of violations in early stages of the search.

9.3.2 Constraint programming

In the following, we present the variables and constraints that define the CP model for CB-CTT. At the best of our knowledge, this is the first CP model for this formulation of the CTT problem.

Variables

We represent a solution as a set of l roomslot variables `roomslot`, that encompass both the room and the period a lecture $l \in \mathbf{L}$ is scheduled in. The variable domains are initialized as $dom(\texttt{roomslot}_l) = \{1 \ldots |\mathbf{R}| \cdot |\mathbf{P}|\}$. Additionally, we use some redundant variables (namely $\texttt{day}_l$, $\texttt{period}_l$, $\texttt{timeslot}_l$, $\texttt{room}_l$ with the obvious channellings) as modeling sugar.

Hard constraints

Since we are using exactly $|\mathbf{L}|$ decision variables, expressing the *lectures* and *room occupancy* constraints is trivial

$$\mathsf{alldifferent}(\texttt{roomslot})$$

To model the *Conflicts* constraint, we must take into account pairs of conflicting courses and constrain their lectures to be scheduled at different periods

$$\mathsf{alldifferent}(\{\texttt{roomslot}_l \mid l \in \mathbf{L_{c_1}} \cup \mathbf{L_{c_2}}\})$$

$\forall c_1, c_2 \in \mathbf{C}, \texttt{conflicting}(c_1, c_2)$ where `conflicting` checks whether two courses belong to the same curriculum or have the same teacher (note that this holds when checking a course against itself). Finally, the *availabilities* constraint can be modeled by imposing that

$$\texttt{period}_l \notin \mathbf{U_c}, \quad \forall c \in \mathbf{C}, l \in \mathbf{L_c}$$

Note that, for our purposes, some of the hard constraints are available both as hard and soft constraints. In particular, the *lectures* and *conflicts* constraint can be transformed into soft constraints by imposing nvalues and count constraints between the `roomslot` and `period` variables and two auxiliary variables, and then embedding the auxiliary variables in the cost function.

Soft constraints

For each of the soft constraints, we define an auxiliary variable to accumulate the violations of the constraints. For some of them, such as *room stability*, this involves counting, for each $c \in \mathbf{C}$ how many different values (nvalues) were taken by $\{\texttt{roomslot}_l \mid \mid l \in \mathbf{L_c}\}$, and then subtracting it from $|\mathbf{L_c}|$ to calculate how many extra rooms were used by the course. A similar approach is taken to compute the violations for *minimum working days*, while set variables and cardinality constraints are used for *isolated lectures*. The complete model is publicly available at https://bitbucket.org/tunnuz/cp ctt.

9.4 Search by Simulated Annealing

In this section, we describe the solution method for the neighborhood search model presented in Section 9.3.1. In [7] the metaheuristic that guides the search is a combination (token-ring) of Tabu Search and a "standard" version of SA. The method presented in this section shows that an enhanced single-stage version of the

SA, once properly tuned, can outperform such a combination. The main differences of the SA approach implemented in this algorithm with respect to the SA in [7] are the following

1. a *cutoff-based* temperature cooling scheme [60], and
2. a different *stopping condition* for the solver, based on the maximum number of allowed iterations.

Another major difference with respect to previous work consists in the statistical analysis and the tuning process (see Section 9.4.1). In particular, the tuning is carried out on a very broad set of instances, described in Section 9.4.1. The statistical analysis aims at distinguishing a *fixed setting* of the parameters that generalizes reasonably well over the whole set of instances, and an automatic procedure to *predict* the ideal parameter setup, on the basis of computable features of the instance at hand.

In the rest of this section we describe the main algorithmic aspects of the SA method, and defer the explanation of the statistical analysis and the tuning process to Section 9.4.1.

Cutoff-based cooling scheme. In order to better exploit the time at its disposal, the implemented approach employs a cutoff-based cooling scheme. In practice, instead of sampling a fixed number ns of solutions at each temperature level (as it is customary in SA implementations), the algorithm is allowed to decrease the temperature prematurely, i.e., by multiplying it for the cooling rate cr, if a portion $na \leq ns$ of the *sampled* solutions has been *accepted* already. This allows to speed-up the search in the initial stages of the search, thus saving iterations that can be used in the final stages, where intensification sets in.

Stopping condition. To allow a fair comparison with the existing work in literature, the considered SA variant stops when an iteration budget (which is roughly equivalent to a time budget, given that the cost of one iteration is approximately constant) expires, rather than when a specific temperature t_{min} is reached. This has the drawback that when the budget runs out, the temperature might still be too high. In order to overcome this problem, the *expected* minimum temperature t_{min} is fixed to a reasonable value and the number ns (see Equation 9.3) of solutions sampled at each temperature is computed so that the minimum temperature is reached when the maximum number of iterations is met.

$$ns = iter_{max} \bigg/ \left(\frac{-\log\left(t_0/t_{min}\right)}{\log cr} \right) \qquad (9.3)$$

Because of the cutoff-based cooling scheme, at the beginning of the search the temperature decreases before all ns solutions have been sampled, thus t_{min} is reached k iterations in advance, where k depends on the cost landscape and on the ratio na/ns. These k iterations are thus saved to be exploited at the end of the search, i.e., when t_{min} has been reached, to carry out further (intensification) moves.

Moreover, given the dependence of the cutoff-based scheme on the ratio na/ns, in order to simplify the parameters of the algorithm, we decided to specify indirectly the value of the parameter na by employing a real-valued parameter $\rho \in]0, 1]$ that represents the ratio na/ns of the number of sampled solutions that will be accepted.

Summary of parameters. Our algorithm accepts many parameters. In order to refrain from making any "premature commitment" [58], all the parameters are scrutinized in our statistical analysis. All the parameters are summarized in Table 9.3, along with the ranges involved in the experimental analysis which have been fixed based on preliminary experiments.

Note that the iterations budget has been fixed to the single value ($iter_{max} = 2.31 \cdot 10^8$) that provides the algorithm with a running time (on our test machines) which is equivalent to the one allowed by ITC-2007 computation rules (the original ITC-2007 benchmarking tool was used to perform this measurement, allowing a running time of 408 seconds).

Parameter	**Symbol**	**Interval**
Starting temperature	t_0	$[1, 100]$
Neighbors accepted ratio (na/ns)	ρ	$[0.01, 1]$
Cooling rate	cr	$[0.99, 0.999]$
Hard constraints weight	w_{hard}	$[10, 1000]$
Neighborhood swap rate	sr	$[0.1, 0.9]$
Expected minimum temperature	t_{min}	$[0.01, 1]$

Table 9.3: Parameters of the search method

9.4.1 Feature-based tuning

In order to come up with a successful algorithm, we carry out an extensive and statistically principled parameter tuning. The aim is to investigate the possible relationships between the instance features, reported in Table 9.4, and the ideal setup of the solver parameters. The ultimate goal of this study is to find, for each parameter, either a fixed value that works well on a broad set of instances, or a formula to predict the best value based on measurable features of each instance. Ideally, the results of this study should carry over to unseen instances, thus making the approach more general than typical parameter tuning. This is, in fact, an attempt to alleviate the effect of the *No Free Lunch Theorems for Optimization* [112], which state that, for any algorithm (resp. any parameter setup), any elevated performance over one class of problems is exactly paid for in performance over another class.

In this section we first present the instances involved in the analysis, then proceed to describe the statistical metodology and the most important result. Finally, we compare the results with the ones of the best approaches in literature.

Instances

According to the customary *cross-validation* guidelines [55], we split the considered instances into two sets: a set of *training instances* used to tune the algorithm, and a set of *validation instances* used to evaluate and possibly revise the tuning .

Training instances. The first group is a large set of artificial instances, created using the generator by [73], which has been specifically designed to reproduce the features of real-world instances. In order to avoid *overtuning* phenomena, only this set has been used for the tuning phase and the individual results on these instances will not be reported. The generator is parametrized upon two features of the instances: the total number of lectures and the percentage occupation of the rooms. For each pair of values of the two parameters we generate 5 instances. As for the number of number of lectures, our instances range from 50 to 1200, using step 50, while the percentage of occupation takes the four values $\{50\%, 70\%, 80\%, 90\%\}$. On overall, the full testbed consists of 480 instances ($5 \cdot 24 \cdot 4$). After screening them in detail, we realized that not all instances generated were useful for our parameter tuning purposes. In fact, four classes of instances were excluded from the analysis in an early phase, namely *provably infeasible* instances, instances with *unrealistic room endowment* (featuring courses with more users than the capacity of the rooms), *too hard* instances (not feasible with limited runtime, possibly infeasible), and *too easy* isntances (where the 0-cost solutions can be found rather quickly).

Validation instances. The second group is the set of instances employed in the literature. It comprises the usual set of instances, the so-called comp ones, which is composed by 21 elements, mainly from the University of Udine, that have been used for ITC-2007. These instances were used to validate the goodness of the approach emerging from the parameter tuning, against the other approaches in literature.

Summary of features. Table 9.4 summarizes the available families of instances, highlighting some aggregate indicators (i.e., minimum and maximum values) of the most relevant features of the instances belonging to each family. All instances employed in this work are available from the CB-CTT Problem Management System `http://satt.diegm.uniud.it/ctt`.

Family	#I	Co	Le	R	Pe	Cu
comp	21	30 – 131	138 – 434	5 – 20	25 – 36	13 – 150
test	4	46 – 56	207 – 250	10 – 13	20 – 25	26 – 55
DDS	7	50 – 201	146 – 972	8 – 31	25 – 75	9 – 105
Udine	9	62 – 152	201 – 400	16 – 25	25 – 25	54 – 101
EasyAcademy	12	50 – 159	139 – 688	12 – 65	25 – 72	12 – 65
Erlangen	4	738 – 850	825 – 930	110 – 176	30 – 30	738 – 850

Family	RO	Co	Av	RS	DL
comp	42.6 – 88.9	4.7 – 22.1	57.0 – 94.2	50.2 – 72.4	1.5 – 3.9
test	86.2 – 100.0	5.7 – 6.2	76.8 – 97.6	69.8 – 87.2	2.0 – 2.1
DDS	20.1 – 76.2	2.6 – 23.9	21.3 – 91.4	53.6 – 100.0	1.9 – 5.2
Udine	50.2 – 76.2	4.0 – 6.6	70.1 – 95.5	57.5 – 71.3	1.7 – 2.7
EasyAcademy	17.6 – 52.0	4.8 – 22.2	55.1 – 100.0	41.8 – 70.0	2.7 – 7.7
Erlangen	15.7 – 25.1	2.3 – 2.9	66.7 – 71.4	49.5 – 56.0	1.0 – 1.2

Table 9.4: Minimum and maximum values of the features for the families of instances (#I: number of instances): courses (Co), total lectures (Le), rooms (R), periods (Pe), curricula (Cu), room occupation (RO), average number of conflicts (Co), average teachers availability (Av), room suitability (RS), average daily lectures per curriculum (DL).

Experimental setup

Our analysis is based on the 480 training instances described in Section 9.4.1. The compared parameter setups are sampled from the *Hammersley point set* [52], for which the ranges whose bounds are reported in Table 9.3. This choice has been driven by two properties that make this point generation strategy particularly suitable for parameter tuning. First, the Hammersley point set is *scalable*, both with respect to the number of sampled parameter setups, and to the dimensions of the sampled space. Second, the sampled points exhibit *low discrepancy*, i.e., they are space-filling, despite being random-like. For these reasons, by sampling the sequence, one can generate any number of representative combinations of any number of parameters. Note that the sequence is deterministic, and must be seeded with a list of prime numbers. Also, the sequence generates points $p \in [0, 1]^n$, which must then be re-scaled in their desired intervals.

All the experiments were generated and executed using JSON2 RUN [105] on an Ubuntu Linux 13.04 machine with 16 Intel® Xeon® CPU E5-2660 (2.20 GHz) physical cores, hyper-threaded to 32 virtual cores. A single virtual core has been dedicated to each experiment.

Exploratory experiments

Before carrying out the experimental analysis, we executed two preparatory steps. The first involved running a *F-Race(RSD)* tuning [13] over the training instances with a 95% confidence, in order to establish a baseline for further comparisons. The race ended with more than one surviving setups, mainly differing for the values of w_{hard} and cr, but giving a clear indication about the good values for the other parameters. This suggested that setting a specific value for w_{hard} and cr, at least within the investigated intervals, was essentially irrelevant to the performance, which allowed to simplify the analysis, by fixing $w_{hard} = 100$ and $cr = 0.99$ (see Table 9.5). Observe that removing a parameter from the analysis has the double benefit of removing some experimental noise, and to allow a finer-grained tuning of the other parameters, at the same computational cost. We thus repeated the race, which resulted in a single winning setup, fixing $\rho = 0.0364$, $t_0 = 30$, $t_{min} = 0.16$ and $sr = 0.43$.

The second step consisted in testing all the sampled parameter setups against the whole set of training instances. This allowed us to further refine the study in two ways. First, from the result it was clear that the initial estimates for the parameters intervals were too conservative, encompassing low-performance areas of the parameters space. In particular, a notable finding was that, on the whole set of training instances, a golden spot for sr was around 0.43, the same value found with F-Race; this parameter was thus fixed, along with w_{hard} and cr. Table 9.5 summarizes the whole parameter space after this preliminary phase (parameters in boldface have not been fixed in this phase, and are the subject of the following analysis). Second, we ran a Kruskal-Wallis test [56] with significance 90% on the dependence of the cost distribution on parameter values, which revealed that some of the instances were irrelevant to our analysis. As a consequence, the analysis was furthered only on a significant subset of 314 original instances.

Parameter	**Symbol**	**Interval**
Starting temperature	t_0	$[1, 40]$
Neighbors accepted ratio (na/ns)	ρ	$[0.034, 0.05]$
Cooling rate	cr	$\{0.99\}$
Hard constraints weight	w_{hard}	$\{100\}$
Neighborhood swap rate	sr	$\{0.43\}$
Expected minimum temperature	t_{min}	$[0.015, 0.21]$

Table 9.5: Revised intervals for investigated parameters

Once the parameter domains were pruned, and the instances reduced to the significant ones, we sampled 20 parameter setups from the remaining 3-dimensional (t_0, ρ, t_{min}) Hammersley point set, and we executed 10 independent runs of each parameter setup on every instance. The following analysis is based on this data.

Statistical analysis

In order to train a model to *predict* the ideal parameters values for each instance, two elements are needed. The first is a set of instances with known or measurable features (the training set described in Section 9.4.1). The second is the known ideal parameter setup for each of these instances.

Per-instance parameter tuning. For each instance, the basic idea is to approximate the cost as a function of the algorithmic parameters by a regression model, with the parameters coded as experimental factors assuming values in the [-1,1] range. Since it is not possible to exclude from the analysis any interactions between the three algorithm parameters under study, all the parameters need to be considered together.

In particular, in our analysis, we took into account three different models.

Linear model (M_1) A simple model approximating the cost as a linear function of the algorithmic parameters. Namely, for $\{i = 1, \dots, n\}$, with n equal to the sample size for each instance ($n = 20 \cdot 10$), the deterministic component of the model is given by

$$g_1(x_i, \beta) = \beta_0 + \sum_{j=1}^{3} x_{ij}\, \beta_j$$

where x_{ij} is the i-th value for the j-th coded algorithmic parameter ($j = 1, 2, 3$) and $x_i = (x_{i1}, x_{i2}, x_{i3})$, whereas $\beta = (\beta_0, \beta_1, \beta_2, \beta_3)$ are coefficients that are estimated from the experimental data.

Quadratic model (M_2) This model extends the previous one by including quadratic terms for each coded algorithmic parameter and interaction term

$$g_2(x_i, \beta) = \beta_0 + \sum_{j=1}^{3} \beta_j\, x_{ij} + \sum_{j=1}^{3} \beta_{j+3}\, x_{ij}^2 + \beta_7\, x_{i1}\, x_{i2} + \beta_8\, x_{i1}\, x_{i3} + \beta_9\, x_{i2}\, x_{i3}.$$

Group-effect model (M_3) These models simply assume a constant level of cost at each different experimental point, namely

$$g_3(x_i, \beta) = \beta_j \text{ , with } j = m(i)$$

where $m(i)$ is the function that returns the experimental point associated to each observation, namely $m(i) \in \{1, \dots, 20\}$.

The last model can be seen as the model corresponding to one-way ANOVA analyses. Based on the idea of experimental response surfaces [7], we first fitted both M_1 and M_2, and we compared them with M_3. From the result of the preliminary Kruskal-Wallis test it was already known that the selected instances were those where M_3 was a meaningful model, therefore the task performed at this step aimed at checking whether a lower-dimensional linear or quadratic function of the algorithmic parameters could approximate sufficiently well the fit provided by the group-effect model.

All the models were fitted by median regression, rather than ordinary least squares. Median regression assumes that the deterministic function $g_j(x_i, \beta)$, $j = 1, 2, 3$ approximates the median cost, rather than the mean cost, corresponding to a certain combination of algorithmic parameters. Denoting by y_i the cost of the i-th observation, the estimated β coefficients are the coefficients that minimize the objective function

$$\sum_{i=1}^{n} |y_i - g_j(x_i, \beta)| \tag{9.4}$$

and therefore the technique is also known as Least Absolute Deviation (LAD) regression. Being based on the median rather than the mean, median regression is much less influenced by outliers in the response then ordinary least squares. As a consequence, median regression is suited for robust estimation of regression models from experimental data where outliers may arise in the response, such as in the algorithm under study here. Inferential usage of the method does not require the normality assumption of the response variable, hence it is not crucial to choose the right scale for the cost. This is a clear advantage when data from multiple instances are analyzed, as the normalizing transformations of cost varies widely across the instances. Last but not least, median regression models can be easily trained by casting the optimization of (9.4) as a linear programming problem, as done by the R package QUANTREG [69]. All in all, median regression, or, more generally, quantile regression, seems reasonably suited for the analysis of the performances of stochastic algorithms.

The model among M_1, M_2, and M_3 that provided the best fit was selected by means of the Akaike Information Criterion (AIC) for model selection [68], which is known to have good performances for prediction. When the linear or quadratic regression models were selected, the parameter setup corresponding to the

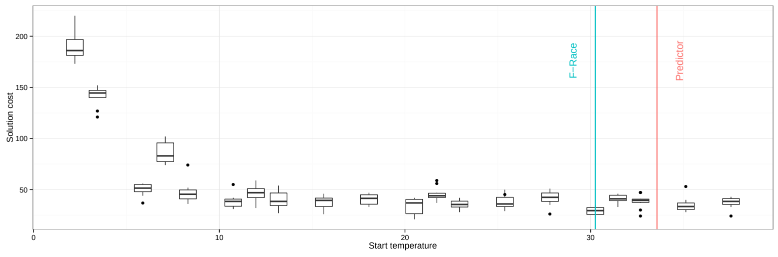


Figure 9.1: Relation between t_0 and cost distribution on a single training instance with $Le = 1200$. Shown are also the ideal t_0 found by F-Race and by the feature-based predictor.

minimum predicted cost was then chosen, and the corresponding parameters deemed as being *optimal*. When the group-effect model was selected by the AIC criterion, we chose the design point corresponding to the smallest sample median, as M_3 model fitted by median regression corresponds to computing a different sample median at each design point. Figure 9.1 displays the box-plots of cost values on a single training instance corresponding to each design point, projected on t_0.

Feature-based regression of optimal parameter values. Once we indentified the ideal parameter setup for each instance, instance features were included in the regression. We thus built one median regression model for each parameter to predict, and the instance features, together with the identified ideal setups, were used to train the model. Clearly, not all features are equally relevant for the prediction of the parameters, therefore the process started from a model involving the complete set of features and, again by using the AIC criterion, we sorted out one feature at once until the best fitted model was reached, i.e., the one with smallest AIC.

As it turns out, only three of the considered features are significant for predicting the algorithm parameters, namely the total number of lectures (Le), the number of curricula (Cu), and the average number of daily lectures per curriculum (DL). Moreover, different parameters are predicted by different features. In particular,

Param.	Intercept	Le	Cu	DL
t_0	16.5	0.019	-0.027	0
ρ	0.043	$-9.95 \cdot 10^{-6}$	$1.99 \cdot 10^{-5}$	0
t_{min}	0.073	$-1.17 \cdot 10^{-5}$	0	$4.58 \cdot 10^{-2}$

Table 9.6: Coefficients for parameter predictors. *Le* is the number of lectures, *Cu* the number of curricula, *DL* the average number of daily lectures per curriculum.

in order to predict the ideal t_0 and ρ one needs to know the total number of lectures and the number of curricula, while to compute the ideal t_{min} the number of lectures and the average number of daily lectures per curriculum are needed. Table 9.6 reports the coefficients of our fitted linear models, together with the intercept term.

By looking at the predicted parameter values for the *validation* instances, with respect to their features (see Figure 9.2), we can draw come conclusions. First, t_0 and ρ are highly correlated with the number of lectures, a feature which, in literature, is commonly considered as a measure of instance *hardness*. In particular, as the number of lectures increases, the predicted initial temperature increases as well, suggesting that, when instances become more difficult, it is beneficial to accept a lot of worsening solutions at the beginning of the search. This was, in fact, an expected result of the analysis, as raising the initial temperature fosters the exploration of the search space in the hope of finding basins of attraction of lower cost. Moreover, the ideal number of neighbors that have to be accepted before decreasing the temperature gets lower when the problem gets harder. This indicates that the *cutoffs* mechanism described in Section 9.4 plays a determinant role in achieving good performances on the larger instances. Finally, the ideal minimum temperature increases linearly with the increase of the average number of lectures per curriculum, which is another measure related to instance hardness. The consequence of this choice is that the temperature will decrease more quickly on larger instances, supporting the effect of cutoffs, and performing a lot of

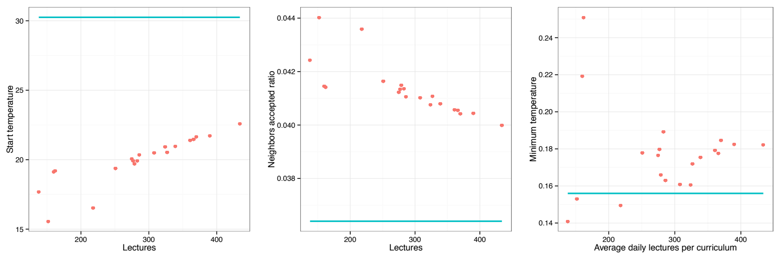


Figure 9.2: Comparison between the parameters found by F-Race (cyan line), and the ones predicted by the model based on the features of each instance (pink dots). The points represent instances in the *validation set*, and consider the most influential feature for each parameter.

intensification towards the end of the search. Note that, in order for this strategy to perform a sufficient amount of diversification, the initial temperature must be higher (as suggested by the predictor).

9.4.2 Results

In the following, we report the comparison of our approach both against the F-Race baseline, as well as some results against the best approaches in literature.

Comparison with the baseline

In order to compare the quality of feature-based tuning against the standard F-Race approach, we ran our algorithm, tuned with both, on the validation instances.

The experiments revealed that there is almost no difference in the cost distributions, with each approach outperforming the other about half of the times. This result was somewhat surprising, since feature-based tuning, in virtue of its use of feature information, is expected to exhibit a better generalization behavior over unseen instances. We looked for an explanation of this effect in the training phase of our regression models, and found out that, in fact, the per-instance tuning (ideally the best possible, see Subsection 9.4.1), has itself a quality comparable to standard F-Race. By looking at Figure 9.1, which is representative of a large portion of the training instances, it is possible to see why. The parameter space (in this case for t_0) is split in two well-distinct parts. One part (the leftmost in Figure 9.1) yields poor results, while any choice of values inside the other part is reasonably safe. In our scenario, the portion of the parameter space leading to poor results is typically very narrow, while the portion leading to better results is broader. This suggests that the chosen algorithm is rather robust with respect to parameter choice, at least for this kind of problem. As a consequence, it is possible, for F-Race, to find a parameter setup that works consistently well across a large set of instances.

From a practical point of view, achieving feature-based tuning is, in general, much more expensive than running a parameter race. However, the outcomes of this process are both an insight on the meaning and relevance of parameters, and a mechanism (i.e., our parameter predictors) that will expectedly scale to a higher degree on new, unseen, problem instances. For these reasons, while recognizing the effectiveness and time-efficiency of F-Race, we chose to resort to feature-based tuning for the comparison with the state of the art.

Comparison with other approaches

In order to validate the quality of our approach, we compared its results against the best ones in literature using the ITC-2007 timeout and instances. For the sake of fairness, the results that are obtained by allotting a higher runtime, for example those of [3], who use 10'000 to 100'000 seconds instead of about 300-500 seconds as established by the competition rules, have been excluded from the comparison.

Table 9.7 shows the average of 31 runs of the algorithm, in which we have highlighted in bold the lowest average costs. For the sake of completeness, we have reported the results obtained with standard F-Race tuning as well, but we have not considered them in the comparison, as our method of choice is the one using feature-based tuning. The figures show that our approach matches or outperforms the state of the art algorithms in more than half of the instances, also improving on our previous results.

Inst.	[81]	[74]	[1]	[7]	**Us**	**Us (F-Race)**
comp01	**5.00**	**5.00**	**5.00**	**5.00**	5.16	5.26
comp02	61.30	60.60	53.90	**51.60**	55.93	55.49
comp03	94.80	86.60	84.20	82.70	**80.87**	82.55
comp04	42.80	47.90	51.90	47.90	**39.48**	39.61
comp05	343.50	**328.50**	339.50	333.40	340.87	338.97
comp06	56.80	69.90	64.40	55.90	**55.64**	55.98
comp07	33.90	28.20	**20.20**	31.50	28.68	27.87
comp08	46.50	51.40	47.90	**44.90**	45.03	44.77
comp09	113.10	113.20	113.90	108.30	**106.96**	107.16
comp10	**21.30**	38.00	24.10	23.80	23.26	24.58
comp11	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.00
comp12	351.60	365.00	355.90	346.90	**337.80**	336.29
comp13	73.90	76.20	**72.40**	73.60	74.70	74.22
comp14	61.80	62.90	63.30	60.70	**58.51**	58.94
comp15	94.80	87.80	88.00	89.40	**79.93**	80.68
comp16	41.20	53.70	51.70	43.00	**39.54**	40.35
comp17	86.60	100.50	86.20	83.10	**79.29**	78.74
comp18	91.70	82.60	85.80	84.30	**80.90**	82.16
comp19	68.80	75.00	78.10	71.20	**67.80**	69.06
comp20	**34.30**	58.20	42.90	50.60	47.74	49.00
comp21	108.00	125.30	121.50	106.90	**104.19**	103.55
avg	87.22	91.26	88.13	85.06	**83.44**	83.58

Table 9.7: Best results and comparison with other approaches over the *validation* instances. Values are averages over multiple runs of the algorithms.

We consider this outcome as very representative of the quality of the presented approach, especially considering that validation instances were not involved in the tuning process, to avoid overtuning, and that the algorithm itself is very simple.

9.5 Search by Large Neighborhood Search

Our CP model for the CB-CTT problem (see Section 9.3.2) was implemented in GECODE (v4.2.0), enabling its resolution with branch

& bound. However, the tackled problem revealed to be quite hard to solve for branch & bound alone, as all but the easiest instances failed to provide even a feasible solution within a time budget equal or comparable to the one allowed by the ITC-2007 rules. In particular, the essential *Lectures* and *Conflicts* hard constraints proved to be very hard to tackle. This prompted us to seek for a different search strategy.

9.5.1 Algorithm overview

The approach we describe in this section is a mix of branch & bound and Large Neighborhood Search (LNS, see Section 7.1) based on the CP model described in Section 9.3.2. In particular, in our approach, we deal with the *Lectures* and *Conflicts* hard constraints by relaxing them, and including them in the cost function as soft constraint. Then we proceed to solve the *lexicographic* minimization problem where first all the violations of the relaxed constraints are eliminated, and then the objective function as defined in the formulation (Equation 9.1) is minimized.

The algorithm iteratively improves an incumbent solution s. Each resolution starts with a 10-seconds branch & bound attempt to solve the complete model, in order to get an advantage on easier instances. This is reasonable as, for such instances, it is convenient to use a search mechanism with strong propagation properties. Then an iterative phase starts, where, at each step, we select a number d of `roomslot` variables from s, we reset their domains to the original ones (see Section 9.3.2), and we re-optimize them through *branch & bound*. The algorithm stops when the allotted time budget expires.

In the rest of this section, we present the specific components of our LNS technique, some of which are specialized from CB-CTT, as suggested in Section 7.1.

Biased relaxation

When the variables must be chosen for relaxation, a fraction of them is chosen heuristically based on the constraints being violated by the solution. For instance, if a lecture $l \in \mathbf{L}$ causes a conflict, we release $\texttt{roomslot}_l$ as well as the `roomslot` variables related to lectures conflicting with it. In general, whenever we detect a variable causing a violation, we release it. The rest of the variables is chosen uniformly at random until the number of variables to relax d is reached.

Branch & bound time budget

Once d variables have been relaxed, a new solution is generated by re-optimizing them. This corresponds to solving a domain-based tightening of the original problem, i.e., a subproblem induced both by the variables that are already fixed, and by the propagation activated by these fixations. In general, one could try to get the optimal solution for the subproblem, however, according to our experiments, it is much more beneficial to save time and execute many iterations. As such, our branch & bound is time bounded by

$$t = t_{var} \cdot d$$

where t_{var} is a parameter of the algorithm. Thus, the more variables are relaxed by the *destroy* step, the more time is given to the *repair* step to come up with a good solution.

Simulated annealing-like cost bound

Being LNS a special case of neighborhood search, we need a strategy to deal with local optima. For this problem, we decided to use the simulated annealing-like cost bound, described in Section 7.1.2. Note that, by using this technique, we can still get the cost-based propagation (since the cost of the next solution is bounded), while being able to escape local optima. However, this introduces four parameters in the algorithm, since we use the SA with cutoffs.

Variable neighborhoods

The search process starts by relaxing $d = d_{min}$ variables. After a number $iter_{max} \cdot d$ of iterations have been spent at a certain d without any improvement, d is increased by one, and up to $d = d_{max} \cdot |L|$. Here $iter_{max}$, d_{min}, and d_{max} are parameters of the algorithm, that must be tuned. Whenever $d = d_{max}$, the search restarts with a new solution differing from the original one in $2 \cdot d_{max}$, and the d is reset to d_{min}. The restart solution is found using a random branching strategy.

Adaptive d_{min}

In some situations, it may be necessary to release more than d_{min} variables to get any solution improvement. This means that, until $iter_{max}$ iterations have passed, it is not possible to improve. To cope with this aspect, at each restart, the new d_{min} is set to the d that yielded the highest number of (temporary) best solutions in the past iterations.

9.5.2 Parameter tuning

The presented LNS approach involves a number of parameters, which are summarized in Table 9.8, together with their initial intervals. Here we split the parameters with continuous domains (which have to be sampled) from the parameters with discrete domains, for which every value in the domain is tried. Note that ρ and $iter_{max}$ are considered as a continuous parameters, even though their values are truncated to their integer part after sampling. Moreover, some of the parameters, such as d_{min}, d_{max}, and t_{var} were fixed or discretized after some preliminary experiments.

Parameter	**Symbol**	**Domain**	
Min. free variables	d_{min}	$\{2\}$	discrete
Max. free variables (ratio)	d_{max}	$\{0.05\}$	discrete
Max. iterations	$iter_{max}$	$[150.0, 350.0]$	continous
Msec to fix a variable	t_{var}	$\{7, 10, 20, 50\}$	discrete
Init. temperature	t_{init}	$[1.0, 100.0]$	continuous
Cooling rate	λ	$\{0.95, 0.97, 0.99\}$	discrete
Neighbors accepted before cooling	ρ	$[1, 50]$	continuous

Table 9.8: LNS parameters

For the tuning process, we have used the same approach described in Section 9.4.1 to compute the F-Race baseline. We thus sampled 32 parameters setups from the Hammersley point set based on the continuous parameters, and assigned each value in turn to the discrete parameters. Then we ran an F-Race with 95% confidence, over a set of instances including the ones in the ITC-2007 competition. The result of the race is the following winning setup: $t_{var} = 10$, $t_{init} = 35$, $\rho = 5$, $iter_{max} = 250$ and $\lambda = 0.97$.

9.5.3 Results

Table 9.9 shows our results (grey column) against the current best ones in literature, on the ITC-2007 testbed. Overall, the proposed approach is outperformed by most approaches in literature. This is likely explainable by the fact that most approaches are based on very fast neighborhood search strategies, while LNS involves cloning constraint networks and performing propagation, and probably requires more time (i.e., dropping the competition rules) in order to attain good performance.

One advantage of using an underlying CP model is that adding side constraints to the model is rather trivial, which makes the approach flexible and much more applicable to real-world situations. However, the rather noticeable difference in performance with respect to the other approaches is still an important limit to overcome.

Conclusions

In this chapter, we presented a popular combinatorial optimization problem, namely curriculum-based course timetabling (CB-CTT), which arises in many universities every semester. We solved the problem using two very different approaches. The first one is a simulated annealing neighborhood search, whose parameters are tuned on the fly based on the features of the instance being solved. The second approach is a large neighborhood search based on a novel CP model for the problem.

Inst.	**[37]∪[81]**		**[74]**		**[1]**		**[7]**		**CP+LNS**		**Best**
	avg	best	avg	best	avg	best	avg	best	med	best	
comp01	**5.0**	**5**	**5.0**	**5**	**5.0**	**5**	**5.00**	**5**	6.0	5	5
comp02	61.3	43	60.6	**34**	53.90	39	**53.0**	40	219.5	158	24
comp03	94.8	72	86.6	**70**	84.20	76	**79.0**	**70**	226.0	158	66
comp04	42.8	**35**	47.9	38	51.90	**35**	**38.3**	**35**	92.0	62	35
comp05	343.5	**298**	**328.5**	**298**	339.5	315	365.20	326	931.5	637	290
comp06	56.8	**41**	69.9	47	64.40	50	**50.4**	**41**	174.0	130	27
comp07	33.9	14	28.2	19	**20.20**	**12**	23.8	17	156.5	97	6
comp08	46.5	39	51.4	43	47.90	**37**	**43.6**	40	162.5	70	37
comp09	113.1	103	113.2	99	113.90	104	**105.0**	**98**	216.0	173	96
comp10	21.3	**9**	38.0	16	24.10	10	**20.5**	11	137.5	91	4
comp11	**0.0**	**0**	**0.0**	**0**	**0.0**	**0**	**0.00**	**0**	0.0	0	0
comp12	351.6	331	365.0	**320**	355.90	337	**340.5**	325	716.0	616	300
comp13	73.9	66	76.2	65	72.40	**61**	**71.3**	64	152.0	120	59
comp14	61.8	**53**	62.9	52	63.30	**53**	**57.9**	54	131.0	103	51
comp15	94.8	–	87.8	**69**	88.00	73	**78.8**	70	226.5	150	66
comp16	41.2	–	53.7	38	51.70	32	**34.8**	**27**	124.5	93	18
comp17	86.6	–	100.5	80	86.20	72	**75.7**	**67**	198.5	152	56
comp18	91.7	–	82.6	**67**	85.80	77	**80.8**	69	144.5	116	62
comp19	68.8	–	75.0	**59**	78.10	60	**67.0**	61	199.0	141	57
comp20	**34.3**	–	58.2	35	42.90	**22**	38.8	33	185.0	137	4
comp21	108.0	–	125.3	105	121.50	95	**100.1**	**89**	257.5	209	75

Table 9.9: Comparison with the best approaches in literature on ITC-2007 instances. Timeout (5 minutes) has been calculated using the competition benchmarking tool.

Chapter 10

Other contributions

During the course of my doctorate, I have worked with local and international researchers on research projects involving optimization at different levels. This chapter is an attempt to give a very brief overview of the attained results, however an in-depth discussion of these works is out of the scope of this thesis.

Section 10.1 is devoted to Virtual Camera Control, a research topic at the cross-roads between optimization and computer graphics. The results addressed in this section are based on work, carried out before and during my doctoral course with, Prof. Roberto Ranon[1], and Prof. Marc Christie[2].

10.1 Virtual Camera Control

In a computer graphics application, such as a videogame or a scientific visualization, the user experiences the virtual world through the lenses of a *virtual camera*. *Virtual camera control* (VCC) is a branch of computer graphics which deals with positioning and moving a virtual camera within a virtual environment, and also encompasses other aspects such as shot editing and virtual cinematography. Since the quantity and quality of information perceived by the user is dependent on the way the camera is handled by the system,

[1]Department of Mathematics and Computer Science, University of Udine, Italy
[2]IRISA/INRIA Rennes Bretagne Atlantique, France.

camera control is one of the fundamental aspects of the interaction of the user with the virtual world.

In its most common form, a virtual camera (see Figure 10.1) is a geometric object which can be defined through seven parameters: *position* (x, y and z), *orientation* (through the Euler angles ϕ, θ, and ψ), and *field of view* (γ, which represents the zoom).

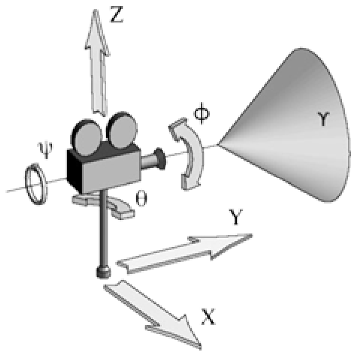


Figure 10.1: A virtual camera.

A common way to achieve automatical control of the virtual camera is the following. First, a set of objective functions that measure the degree of satisfaction of some desired visual properties of the final image (or shot) are defined. Then, a general-purpose solver is used to maximize the linear combination of such objective functions, thus transforming the problem of finding a good virtual camera setup in a continuous-domain optimization problem. When the problem consists in finding a virtual camera at a specific instant in time, it is often referred to as *viewpoint computation*.

Two main issues arise when such an approach is used to solve viewpoint computation problems. First, in order to obtain good results, the algorithms computing the satisfaction of the visual properties should return *accurate* measures. Second, in order for an automatic approach to be useful, its performance should be compatible with real-time environments, i.e., scenes where the objects might quickly change their position. We briefly address these issues in the following sections, and defer the interested reader to the relative papers.

10.1.1 Pixel-accurate evaluation of visual properties

In [86] we describe a language for reasoning about visual properties in terms of operations, e.g., counting and overlap assessment, on pixel sets. The described language allows to define new properties as sub-routines, and upon execution over a virtual environment, employs the capabilities of the *graphical processing unit* (GPU) to assess the satisfaction of such properties.

This approach can be used to assess the accuracy attained by automatic virtual camera control algorithms, but it cannot be used in a solving phase, because of the high computational cost of the executed operations.

10.1.2 Efficient viewpoint computation with PSO

In [**?**], we present a viewpoint composition library based on particle swarm optimization (PSO, see Section 4.1). The library is based on a previous work by some of the same authors, but the focus of the new research is on the performance. In particular, we propose a new technique for initializing PSO particles, which makes use of problem-specific information to guess good candidate positions. Moreover, we carry out an extensive parameter tuning and performance analysis in order to obtain performances that are compliant with real-time applications.

The described library is publicly available under the permissive MIT license at https://bitbucket.org/rranon/smart-viewpoint-computation-lib.

10.2 Runtime analysis of Genetic Programming

In the last decade, *genetic programming* (GP) [70] algorithms have found various applications in a number of domains. However, their runtime behaviour is hard to understand in a rigorous manner. In particular, the implicit stochasticity in many components of GP algorithms, make it difficult to establish clear upper bounds for the application of GP to a lot of problems.

In the last few years, researchers in the field of runtime analysis and evolutionary computation, have devised a number of probabilistic techniques, e.g., drift analysis [43] and fitness-based partitions [111], to come up with *expected* upper bounds to the running time.

In [106], we provide experimental evidence to the theoretical results obtained in a recent work [46], which analyzed the runtime of GP on two simple study problems. The purpose of our experimental analysis is twofold. On the one hand we want to confirm the expected runtimes obtained through the theoretical study. On the other hand, we want to conjecture expected runtimes for situations in which a proven upper bound has not been found.

In [82], we analyze, both from a theoretical standpoint, and from an experimental one, the runtime of single- and multi-objective GP algorithms on generalized versions of the said study problems. We give proofs for new bounds for some algorithmic setups, and provide experimental evidence for some aspects for which a bound is currently missing.

A complete discussion of the results in [106, 82] out of the scope of this thesis.

Conclusions

We presented two additional bodies of work, which are not directly related with the topic of this thesis, but constitute original research work carried out during my doctoral course. These include an application of particle swarm optimization to the problem of generating snapshots of a 3D environment which satisfy some desired visual properties, and some interesting results in the runtime analysis of both single- and multi-objective genetic programming.

Appendix A

Parameter tuning

All of the approaches described in this thesis, and in general most optimization algorithms, expose a number of parameters which control their behavior. To make an example, the initial temperature t_0 and the cooling rate λ in simulated annealing (see Section 3.2.4) determine the balance between diversification and intensification obtained by the algorithm. Of course, depending on the explored fitness landscape, it might be preferable to diversify or to intensify, but this is not known in advance. In general, the success of a specific approach over a given problem instance, depends on setting correctly the values for all its parameters.

Unfortunately, in most cases, there is no general rule of thumb concerning the correct setting of the parameters. It is thus necessary to use automatic techniques that are able to use experiments in order to come up with a good parameter setup. Such techniques belong to the class of *parameter tuning* algorithms.

In this appendix, we describe two techniques for parameter tuning, namely *single-point parameter tuning* and *feature-based parameter tuning*. We present a popular algorithm to achieve the former, and the general idea behind the latter. Concerning feature-based parameter tuning, since it is still a domain-dependent technique, we refer the reader to a possible implementation in the context of curriculum-based course timetabling (CB-CTT) in Section 9.4.1.

A.1 Defining the competing parameter setups

Regardless of the employed tuning technique, a common preparatory step consists in defining the alternative parameter setups that are compared. This preparatory step involves two phases. The first one consists in defining the *parameter ranges*. For some parameters, this is trivial, because the range is implicit in the parameter semantics. For instance, the implicit rate for the cooling rate λ in simulated annealing is $]0, 1]$. However, note that even for such parameters, more restrictive ranges can be devised in order to concentrate the tuning in better areas of the parameters space. To further the previous example, reasonable values of the cooling rate are usually in $[0.9, 1.0]$. For other parameters, the reasonable ranges must be defined with *exploratory experiments*. Exploratory experiments consist in running the algorithm under study, with various parameter setups, on a reasonable number of problem instances. These exploratory experiments should roughly sample the space of parameters, hence the initial ranges for the parameters should be very large.

Note that so far we have not considered *categorical parameters*, i.e., discrete parameters that define alternative implementations of algorithms or their components. For instance, in a neighborhood search algorithm, whether to only solutions that strictly improve upon the incumbent one, or accept sideways moves. Ideally, such variants should be exposed as parameters and optimized through parameter tuning, as suggested in [58] in the philosophy of *programming by optimization*.

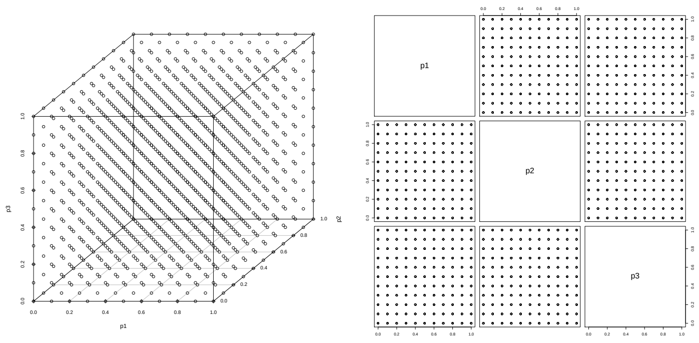
Also, note that it is not possible, to optimize one parameter at a time, as in most algorithms the parameter interact and concur in determining the behavior of the algorithm. This is the single most important underlying assumption of parameter tuning.

A.1.1 Full factorial

One way to come up with a number of candidate parameter setups, is to define discretizations over the identified parameter ranges, and then considering the Cartesian product of such discrete sets.

Such approach is called the *full factorial.* This method allows to obtain space-filling setups, i.e., parameter setups that cover all the space of possible parameters. Specifically, let n be the number of parameters, for every set of $n-1$ parameters, the remaining parameter takes all the values in its discretized range.

The disadvantage of such approach is obvious, as the number of generated parameter setups is very large (see Figure A.1, which depicts $1'000$ parameter setups generated from 3 parameters in $[0,1]$ with 10 values per parameter).



(a) 3D view of the parameter space with the generated parameter setups.

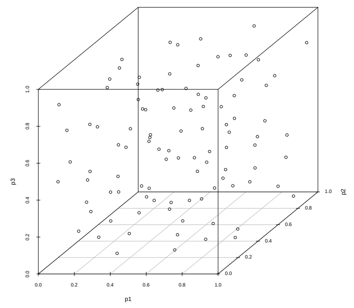
(b) Projections of the parameter setups on each pair of dimensions of the parameter space.

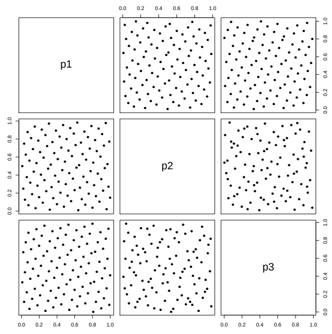
Figure A.1: Setups generated as Cartesian product of ranges.

Of course, generating that many parameter configurations just to adequately represent the parameter space has its drawbacks, since in order to choose the best parameter setup, many experiments must be carried out.

A.1.2 Sampling

In order to avoid generating the full factorial, one approach involves *sampling* a number of configurations from the parameter space. This can be done by using techniques that allow to generate representative sets of points, while keeping the number of generated setups low.





(a) 3D view of the parameter space with the generated parameter setups.

(b) Projections of the parameter setups on each pair of dimensions of the parameter space.

Figure A.2: Setups sampled using the Hammersley point set.

One such parameter is the Hammersley point set [52], which we employed for the first time in [5]. Two properties, in particular, make this point generation strategy particularly suitable for parameter tuning. First, the Hammersley point set is *scalable*, both with respect to the number of sampled parameter setups, and to the dimensions of the sampled space. Second, the sampled points exhibit *low discrepancy*, i.e., they are space-filling, despite being random-like. For these reasons, by sampling the sequence, one can generate any number of representative combinations of any number of parameters. Note that the sequence is deterministic, and must be seeded with a list of prime numbers. Also, the sequence generates points $p \in [0, 1]^n$, which must then be re-scaled in their desired intervals. Figure A.2 shows 100 space-filling parameter setups generated with this method.

Other techniques for sampling include the *nearly-orthogonal latin hypercubes* [31], which have been successfully employed in [7].

A.2 Finding the best configuration with race

Once a set of alternative parameter setups has been identified, we need to find the best one (or ones). One popular approach for this task is to employ an approach called *F-Race* [12].

An F-Race consists in a controlled execution of experiments, in which, at each step, some of the parameter setups are possibly sorted out. The general idea of the algorithm is the following. First, all the competing configurations are run against an initial block of instances of fixed size. A typical value for the block size is around *10*, but it strongly depends the available resources, i.e., instances, time budget. After the experiments have been executed, each instance is considered and a natural ranking of the parameter setups is calculated. Then, each parameter setup is considered and the sum of the ranks obtained in the various instances is calculated. At this point, a Friedman rank sum test with a given confidence is executed on the parameter setups, and a post-hoc analysis is used to sort out the ones that are "provably" inferior.

After the first set of parameter setups have been pruned, the experimentation restarts with the remaining ones, and at each new instance a new pruning is attempted. Note that the number of parameter setups is only reduced if the Friedman test produces a sufficiently low p-value. Also, note that with this mechanism, the overall cost of parameter tuning is reduced, as not all parameter setups are tried against all instances.

The race stops when there is only one configuration remaining, or all the instances have been tried. Note that the result of this tuning is a configuration which is overall better than the others at tackling the whole set of instances. By definition, this means that, on some specific instance, a different approach might be better than the winning one, but not overall. This makes sense as, if we identify an algorithm with a given parameter setup, then the NFL theorems hold.

A.3 Feature-based parameter tuning

A more involved statistical analysis can be carried out to produce a more instance-specific tuning, i.e., which is less doomed by the NFL theorems. In order to to this, one needs

- the *best parameter configuration for each instance*, and
- a set of *features* that represent each instance.

Features are computable metrics about a problem instance, that can be computed offline or quickly computed online when the instance is loaded in the solver. For instance, Table 9.4 describes some of the most relevant features in CB-CTT instances. The overall procedure works as follows. First, a per-instance *overtuning* is carried out, to find the parameter setup that is the best for each instance. Then, a statistical model is fitted using the features of each instance as inputs, and the ideal parameter values as outputs.

If the model is constructed correctly, it can be used perform regression, i.e., to predict the ideal parameter values based on measurable information about the instance. In practice, many features are not actually needed in estimating the correct parameter values. Moreover, it is possible that the noise in the experimental data is too high to allow any reliable modeling.

The process of building a feature-based predictor for simulated annealing parameters based on CB-CTT instance features, is described in Section 9.4.1.

A.4 JSON2RUN

JSON2RUN [105] is a Python-based command line tool to automate the running, storage, and analysis of experiments. The main advantage of JSON2RUN (over a home-brewed experiment suite) is that it allows to describe a parameter space concisely as a JSON[1]-formatted pseudo-logical parameter tree, such as the following (note the presence of parameter definitions as well as logical operators to combine them).

[1]Website: http://www.json.org

```
{
    "type": "and",
    "descendants": [
        {
            "type": "discrete",
            "name": "p1",
            "values": [ "foo", "bar", "baz" ]
        },
        {
            "type": "or",
            "descendants": [
                {
                    "type": "discrete",
                    "name": "p2",
                    "values": { "min": 0.0, "max": 1.0, "step": 0.25 }
                },
                {
                    "type": "continuous",
                    "name": "p3",
                    "values": { "min": 2.0, "max": 3.0 }
                }
            ]
        }
    ],
    "postprocessors": [
        {
            "type": "hammersley",
            "points": 20
        }
    ]
}
```

Parameter generations work as a *coroutine*, i.e., every call to the parameter generation facility produces the *next* parameter setup. The nodes of the parameter tree are handled recursively as follows

Discrete nodes are leaf nodes which generate one discrete parameter at a time, as defined in their `values`. The `values` can be specified extensionally (see `p1` in the example), or intensionally through a `step` parameter (see `p2`).

Continuous nodes are special nodes, which describe (but do not generate) discrete parameters. Instead, they generate interval parameters that must be later be processed by `postprocessors`.

And nodes generate the Cartesian product of the `discrete` parameters generated by their descendants. That is, they first activate once all their descendants to generate the first parameter setup. At the subsequent calls, they only activate the last descendant and keep the first ones fixed, until the last descendant has produced all the possible values (of combination of values if it contains more `and` nodes). The process goes on until all the configurations are generated.

Or nodes activate their descendants one at a time, thus can be used to generate alternatives.

Post-processors can perform additional operator on the parameter setups generated by the node they are attached to. For instance, they can process the `continuous` parameter in order to implement sampling, they can perform rounding operations on specific parameters, and generate parameters based on custom Python expressions.

Thus, the above parameter tree would generate the following sequence of experiments

```
./solver --p1 foo --p2 0.0
./solver --p1 foo --p2 0.25
./solver ...
./solver --p1 foo --p3 2.05
./solver --p1 foo --p3 2.1
./solver ...
./solver --p1 bar --p2 0.0
./solver --p1 bar --p2 0.25
./solver ...
./solver --p1 bar --p3 2.05
./solver --p1 bar --p3 2.1
./solver ...
```

Note that parameter generation is recursive, thus the parameter tree can be arbitrarily complex. Post-processors can be attached to every non-leaf node (thus `and` and `or` nodes only).

No matter how a parameter tree is specified, it can be used either to run an F-Race (between all the defined parameter setups), or run an exhaustive experimental analysis. The results of the analysis are automatically stored in a MongoDB[2], where they can be accessed by parameter values. The JSON2RUN distribution also contains an R script with functions to retrieve the experiments from the database and convert them to R data frames.

JSON2RUN is publicly available under the permissive MIT license, and its main features are described in [105].

[2]Website: http://www.mongodb.org

Appendix B

Reinforcement learning and neural networks

In this appendix, we briefly introduce the basics of reinforcement learning (RL, [102]) and multi-layer perceptrons (MLP, [2] a kind of neural networks), a special kind of artificial neural networks (ANN). We then describe how a RL agent can use a multi-layer perceptron to store the information about its acting policy, and why this is necessary.

B.1 Reinforcement learning

Reinforcement learning (RL) is a machine learning method whose goal is to train a *learning agent* to behave optimally inside an initially unknown environment. The learning is only based on positive and negative rewards that the environment gives to the agent in response of its actions. The main components of a reinforcement learning system are therefore

Environment whose observable features can be sensed by the learning agent in form of *states*.

Actions the operations that the agent is allowed to perform in the environment.

Reward function which is a numeric feedback that the environment gives to the agent in response of its actions.

Policy i.e., the rule according to which the agent chooses a specific action based on the state of the environment. The policy is usually parametrized by a number of weights $\omega_{s,a}$ that encode the desirability of applying a specific action a, given a specific state s of the environment.

Learning function the rule which is used to update the parameters of the policy.

The implicit goal of a learning agent, is to *maximize* its *long-term utility*, i.e., the sum of all the rewards it receives during its lifetime. The various choices for the above components define a range of different algorithms. In fact, RL should be seen more as a family of techniques, rather than a single one.

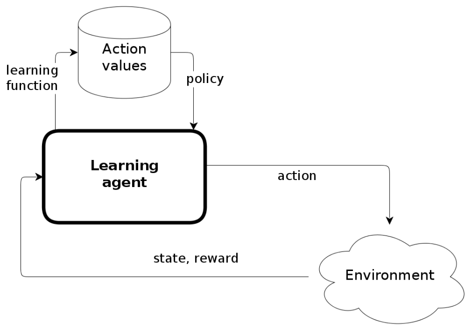


Figure B.1: The agent-environment architecture

B.1.1 The acting-learning loop

Reinforcement learning is intrinsically iterative. By performing actions and updating its knowledge, the agent learns how to behave more and more optimally in order to achieve its goal. This acting-learning loop is shown in Figure B.1.

Algorithm 12 describes the structure of the basic reinforcement learning loop, in which knowledge is updated right after each action application. This way of updating knowledge is known as *online learning* as opposed to *batch learning* where knowledge is only updated after a number of iterations (and thus actions) have been executed.

Algorithm 12 Reinforcement learning loop

```
procedure RL($env$)
    $\omega \leftarrow$ InitializeActionValues()
    while $\neg$StoppingCriterion() do
        $s \leftarrow$ Observe($env$)
        $a \leftarrow$ Policy($s, \omega$)
        $r \leftarrow$ Execute($a, env$)
        Learn($s, a, r, \omega$)
    end while
end procedure
```

State update. The state of the environment is observed (Observe) at the beginning of each iteration. The state represents the agent's knowledge about the environment at a specific time during its trip towards the goal, and representing the state in the proper manner is key to a correct learning.

The representation of the state is subject to a typical trade-off. On the one hand, if the state representation is coarse-grained (few dimensions), it will be difficult for the agent to distinguish to states which are similar, however, learning how to behave optimally in every state will be easier. On the other hand, if the state representation is finer (many dimensions), the agent will be more informed, but it will have a hard time learning how to behave properly in every state. Unfortunately there is no rule of thumb about this aspect, as it strongly depende on the environment's constraints and observability. Ideally, one state should report important features of the environment, without describing every single detail of it, i.e., it should be able to *generalize*.

Policy. At each iteration, the agent selects which action to take in the current state according to its Policy. The policy is a function for selecting an action s, given a state s, based on its recorded *action value* $\omega_{s,a}$. For instance, a simple policy could consists in always selecting the action with highest value (the so-called *greedy* policy), or selecting a random action with probability $\epsilon \in]0, 1]$ and behaving greedily with probability $1 - \epsilon$ (ϵ-*greedy* policy). An agent behaving greedily is said to *exploit* its knowledge, while an agent behaving randomly is said to *explore*. Balancing exploitation and exploration is one of the key points of achieving a good reinforcement learning. Note that, once a particular policy (greedy, ϵ-greedy, …) has been selected, the behavior of the agent is completely determined (except for stochastic effects) by the set of action values ω.

The action values are initialized at the beginning of the loop, by the INITIALIZEACTIONVALUES() function. Here, a good heuristic is to set high initial action values, so to promote an initial exploration of the possible actions. Moreover, in order to keep the learning effective over time, one should always ensure that each action has a non-zero probability of being selected. This allows to choose actions which were maybe bad when they were first tried, but could be good in the present, and avoid early convergence of the policy. The randomness in *ϵ-greedy* policies has precisely this function.

Reward function. Once the action has been selected the agent executes it in the environment, possibly changing its state. The action execution (EXECUTE) yields a *reward* r, a real-valued indicator of the action goodness in that particular situation. The reward can be seen as a measure of how much the action application has moved the agent towards its goal. In principle, nothing guarantees that the same action in the same state will produce the same reward, because the environment could be *non-stationary*, or the information in the state not sufficient to completely describe the it. For this reason, the knowledge of the agent, which is represented by action values in specific states, must be updated continuously or until convergence.

Learning function. Knowledge update is carried out through the learning function. This function modifies the values of actions in order to change the outcome of the policy and thus the agent's behavior. A simple but effective way of updating knowledge is the one in Equation B.1, which is similar in spirit to the update rule of ACO (Equation 4.5).

$$\omega_{s,a} += \alpha(r - \omega_{s,a}) \tag{B.1}$$

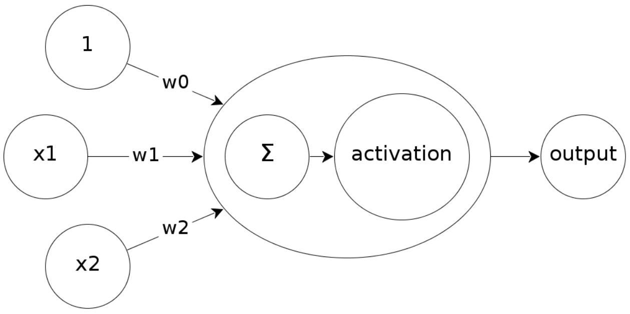


Figure B.2: (Single-layer) perceptron.

Here r is the reward obtained in the last iteration, by applying action a in state s, and α is a the *learning rate*, most commonly a number in the interval $]0, 1]$ which determines how fast the agent learns information from the newly obtained rewards.

If α is low, e.g. 0.01, the agent will converge very slowly to an optimal policy, while if the value is high, e.g. 1.0 the reward will completely replace the previous action value. Since one usually wants to avoid sudden changes in the action values, a popular value for learning rate is around 0.1, however this strongly depends on the environment and the goal. A constant learning rate allows to tackle non-stationary problems, since the policy never really converges.

Two things must be noted here. First, if the action value and the reward are equal (the prediction about the value of the action was correct), the update doesn't change the ω. Second, information about action values is kept on a per-state basis, therefore when we refer to *action values*, we actually mean *state-action values*.

B.2 Multi-layer perceptrons

Multi-layer perceptrons (MLP) are a function approximation mechanism which belongs to the class of *supervised* machine learning algorithms. We are going to briefly revise MLPs in this section, by starting from the simpler concept of *perceptron*.

A (single-layer) perceptron is a processing unit with a number of weighted inputs (one of which has a constant value of 1.0) and an output. Its only operational capabilities consist in computing the weighted sum of its inputs, process this sum with an activation function and output it.

The algorithm implemented by the perceptron in Figure B.2 can be summarized with the formula

$$h(\vec{x}) = \textsc{Activation}(\vec{w}^T \vec{x})$$

where $\vec{w}$ is the vector of weights and $\vec{x}$ is the vector of inputs which is of the form $[1, x_1, x_2, \dots]$ (remember that the first input is always 1.0).

The nature of the Activation function determines the kind of function which is possible to approximate. For instance, by using the *identity* as the activation function, a perceptron is able to approximate, after it is trained, any linear function in any number of variables since, by expanding the equation, we get

$$h(\vec{x}) = \vec{w}_0 + \vec{w}_1\vec{x}_1 + \vec{w}_2\vec{x}_2 \dots$$

which is the equation of a line.

Training a perceptron means setting the weights of the arcs so to *minimize the error* function between the output of the perceptron and the desired function of the input. A popular choice for the error function is the squared error, i.e.,

$$E_{\vec{w}}(\vec{x}, y) = \frac{1}{2}(h(\vec{x}) - y)^2$$

where the *training instance* $(\vec{x}, y)$ represents an input-output pair in which y is the correct result that we would like the perceptron to compute for $\vec{x}$. The fractional term doesn't change the behavior of the learning algorithm and simplifies the whole equation later.

Note that the error function is parametrized on $\vec{w}$, therefore in order to minimize it we need to update $\vec{w}$. To do so, we must consider the error over the entire set of training instances (sum of squared errors), i.e.,

$$E_{\vec{w}} = \frac{1}{2n}\sum_{i=1}^{n}(h(\vec{x}_i) - y_i)^2$$

There are a number of ways to minimize the error, depending on the properties of the error function, some of the most popular being variations of the *gradient descent* method. The idea behind gradient descent is that if we update $\vec{w}$ in the direction of the negative gradient of the error function, we will eventually reach the minimum of the function. Fixing $h(x) = \vec{w}^T\vec{x}$, the gradient of the error function, with respect to $\vec{w}$ and a single training instance is computed as:

$$\nabla E_{\vec{w}}(\vec{x}, y) = \frac{\partial (h(\vec{x}) - y)^2}{\partial \vec{w}} = (h(\vec{x}) - y)^2 \vec{x}$$

therefore the update on $\vec{w}$ is:

$$\vec{w} = \vec{w} - \alpha (h(\vec{x}) - y)^2 \vec{x}$$

where α is a learning rate. This kind of update makes sense if we don't know in advance the whole set of training instances however, if we do, a batch update (in which the error is averaged over the whole set of training instances) can be used:

$$\vec{w} = \vec{w} - \alpha \frac{1}{n} \sum_{i=1}^{n} (h(\vec{x}) - y)^2 \vec{x}$$

Note that this kind of training will only do if the function we want to approximate is linear. For approximating non-linear functions, however, the *logistic* function $h(\vec{x}) = \frac{1}{1+e^{-\vec{w}^T\vec{x}}}$ is mostly used.

MLPs (Figure B.3) are layered networks of perceptrons in which the output of each perceptron in a layer k is connected to all the inputs of perceptrons in the layer $k+1$. Because perceptrons are actually inspired to neural cells, these networks are commonly known as *feed-forward neural networks* (FFNN), where *feed-forward* refers to the fact that signals are always propagated from the input to the output layer, and perceptrons are usually referred to as *neurons.*

There are no constraints on the number of hidden layers, or the number of neurons in a layer, however it has been demonstrated [59] that MLPs with a large-enough single hidden layer are able to approximate any non-linear function of the input, thus learning a

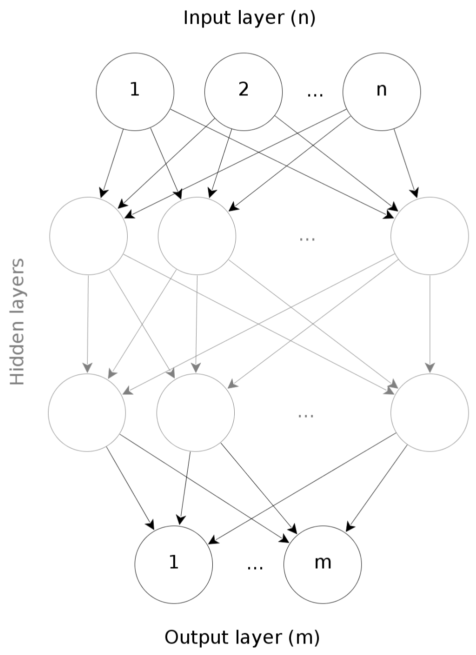


Figure B.3: Multi-layer perceptron.

mapping $f : \mathbb{R}^n \mapsto \mathbb{R}^m$. Unfortunately, there is no rule of thumb on the right number of hidden neurons, which must be worked out with parameter tuning.

Tranining a MLP with gradient descent is a bit more involved than training a single perceptron, however the theoretical foundation is the same. The difference is that the error function must be *back-propagated* through the neural network since we don't have correct values for the hidden neurons. We are not going to cover back-propagation in this appendix (see [2] for more information).

B.2.1 Storing action-values in a multi-layer perceptron

When the states and actions of a RL system are discrete and finite, a simple way to store action values is to keep them in a bi-dimensional table. This solution is sometimes called *tabular reinforcement learning*. However, when the states or the actions are continuous, or the number of states is just too large, this technique, although simple, is no more feasible. In these cases, a popular choice is to treat action values as a continuous function and use techniques to approximate it. MLPs have been used to approximate action values in many applications [102].

One simple approach for using a MLP as action values store, is to apply the selected action, collect the reward and then train the MLP by using the $\langle s, s \rangle$ pair as input, and the obtained reward as the output. This can be done either after each action application (online learning) or after a number of steps (batch learning). Note that the state representation can be composed by several *features* (distinct pieces of information about the environment), which can be fed in different inputs of the MLP.

The main advantage of this approach is that the amount of weights one must update depends only on the structure (i.e., number and layout of the perceptrons) of the MLP, rather than on the number of states and actions. A second advantage is that MLPs are good at generalizing. In tabular RL, the update of an action value $w_{s,a}$ affects only the selection of action a in state s. As a consequence, since the size of the table can be very large, the algorithm can take a lot of iterations before updating each single value. When using MLPs, similar states share the effect of updates, thus leading to faster convergence of the policy.

B.2.2 Eligibility traces

Eligibility traces (ET) are a mechanism used in RL for *temporal credit assignment*. Let's look at the simple example in Figure B.4. Suppose that the cost of a solution s at time t is $f(s^t)$, then the reward of the action applied at time t is calculated as $f(s^t) - f(t+1)$, i.e., the more the cost is reduced, the higher the reward. In this example, actions $1, 2$ would receive a positive reward since they reduce the value of the cost function, but in fact they bring the agent in a local optimum. Conversely, actions $3, 4, 5$, and 6 would obtain a very bad reward, because their activation causes the cost function to increase, but they bring the agent out of the local optima, and eventually towards the global optimum.

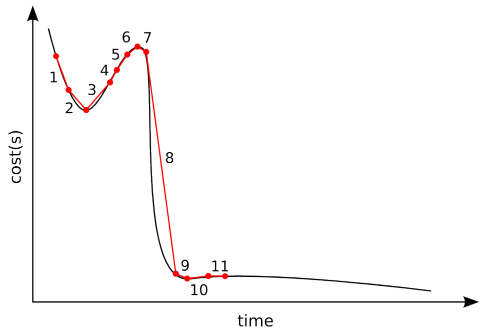


Figure B.4: Sequence of actions during exploration of a cost function.

The idea of temporal credit assignment is that the reward (either positive or negative) should be used to update the whole sequence of $\langle s, a\rangle$ pairs which led to the current situation, rather than just the last one. Following this idea, actions $1, 2$ should be punished (and thus made harder to choose) for bringing the agent to the local optimum, and actions $3, 4, 5$, and 6 should be rewarded (and thus made easier to choose) for leading the agent out of the local optimum, and eventually to a better optimum. In this case, making actions $3, 4, 5$, and 6 easier to choose would make easier, in the future, to escape from local optima by applying the same (or a similar) sequence of actions.

In practice, eligibility traces can be easily implemented by keeping a table of values which are updated as follows

$$e_{s,a} = \begin{cases} 1 & \text{if state is } s \text{ and action is } a \\ \lambda e_{s,a} & \text{otherwise} \end{cases}$$

where λ is a decay factor in $[0, 1[$ which determines how much previous $\langle s, a\rangle$ pairs should be updated towards the latest reward. If the decay is set to 0, the agent has no memory about past actions and the update is described by Equation B.1. As the decay factor approaches 1, temporal credit assignment is enforced. The value of $e_{s,a}$ is then used in a simultaneous update of all $\langle s, a\rangle$ pairs according to

$$\omega_{s,a} += \alpha(r - \omega_{s,a})e_{s,a} \tag{B.2}$$

This kind of eligibility traces is commonly known as *replacing traces*, because 1 always replaces the previous value of $e_{s,a}$ when we perform action a in state s. This mechanism can be implemented efficiently by keeping a queue with the the last

$$\lceil\frac{\log(tr)}{\log(\lambda)}\rceil \tag{B.3}$$

pairs where tr is the value of e under which the update is considered negligible, and updating pairs proportionally to λ^p where p is their position in the queue.

Conclusions

This thesis investigated two classes of hybrid meta-heuristics (see Chapter 5), namely hyper-heuristics (Chapter 6 and propagation-based meta-heuristics (Chapter 7). We described their application to several combinatorial optimization problem that arise in real-world situations. Specifically, we showed how to solve the problem of balancing bike sharing systems (BBSS) by means of CP-based large neighborhood search (LNS) and ACO-driven constraint programming (ACO-CP). Moreover, we showed how to tackle the problem of generating highly constrained university timetables (CB-CTT) by means of a simulated annealing approach which is tuned at optimization time based on measurable information about the instance being solved (Chapter 9). Such results were obtained by means of statistical model and machine learning techniques. Moreover, the CB-CTT problem was also tackled by means of CP-based large neighborhood search (LNS), albeit without outstanding results. As for hyper-heuristics, we discussed the development of the algorithms based on reinforcement learning (Appendix B) that we submitted to the first cross-domain heuristic search challenge (Chapter 6).

In this thesis, we tried to keep the discussion of search methods independent from the discussion of specific domains of applications, so to highlight the necessity of separating the modeling and the solving phases in order to increase the applicability of the described apparoaches. We have shown that such separation of concerns can be achieved very easily with the investigated hybrid meta-heuristics. Specifically, hyper-heuristic attain it by reasoning at a higher level, without exploiting any domain-specific information about the problem being solved. The considered propagation-based meta-heuristics, on the other hand, are based on constraint programming models, and little or no information is needed in general to apply them to those models.

Apart from the obtained results, in this thesis we have presented (see Chapter 7 and Appendix A) several optimization and research support tools and libraries which have been developed in the context of this research. Among other things, this effort allows us and other researchers to repeat our results, in the hope of improving the quality of research and the comparison of different approaches.

The findings discussed in this thesis, sparked new research directions for the future. We can identify at least four research lines which look promising for extending our research.

A first research direction is the one concerning the *initialization of algorithms*, an aspects which has been often received limited attention in our previous research. Results regarding particle swarm optimization for viewpoint computation [?], revealed that having a good initialization policy can be determinant for obtaining good time-effectiveness and solutions of good quality.

A second line of research, concerns the use of *multi-level techniques* (Section 5.1.4) as an effective method for reducing the complexity of large instances of combinatorial optimization problems. Such approaches share the same underlying idea of large neighborhood search, and could be integrated easily in our framework, for instance, by coarsening the problem through constraint relaxation and then refining the solutions through LNS.

Another promising research direction is the one of *feature-based tuning* (Section 9.4.1), which we already applied in the context of CB-CTT. Feature-based tuning would allow to alleviate the effect of the *no free lunch* (NFL) theorems for optimization. However, the effort needed to implement it is still a hindrance to its diffusion, and a clear methodology has yet to be developed. Our previous effort in this sense can be seen as the first step in this direction.

Finally, in this thesis we have treated all problems as single-objective optimization problems. However, as pointed out in Sections 1.4.3 and 1.4.4, many real-world combinatorial optimization problems are multi-objective in nature. As a consequence, transforming them, e.g., through scalarization, into singe-objective ones, eliminates some of the optimal solutions. Therefore, a possible line of research consists in tackling them directly through multi-objective algorithms. Particularly in the context of propagation-based hybrid meta-heuristics, this constitutes a real challenge, since, to the best of our knowledge, multi-objective propagation techniques are still a relatively unexplored research area.

Apart from the above directions for further research, the application of the presented hybrid meta-heuristics should be extended to different problems. In particular, we are interested in problems concerning sustainability, both in the production and distribution of goods, e.g., delivery problems with particular constraints, and in sustainable mobility, e.g., car pooling and intermodal public transportation.

Bibliography

[1] Salwani Abdullah, Hamza Turabieh, Barry McCollum, and Paul McMullan. A hybrid metaheuristic approach to the university course timetabling problem. *Journal of Heuristics*, 18(1):1–23, 2012.

[2] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2010.

[3] Roberto Asín Achá and Robert Nieuwenhuis. Curriculum-based course timetabling with SAT and MaxSAT. *Annals of Operations Research*, 2012.

[4] Mutsunori Banbara, Takehide Soh, Naoyuki Tamura, Katsumi Inoue, and Torsten Schaub. Answer set programming as a modeling language for course timetabling. *Theory and Practice of Logic Programming*, 13(4-5):783–798, 2013.

[5] Ruggero Bellio, Sara Ceschia, Luca Di Gaspero, Andrea Schaerf, and Tommaso Urli. A simulated annealing approach to the curriculum-based course timetabling problem. In *MISTA'13: the 6th Multidisciplinary International Conference on Scheduling: Theory and Applications*, 2013.

[6] Ruggero Bellio, Sara Ceschia, Luca Di Gaspero, Andrea Schaerf, and Tommaso Urli. Feature-based tuning of simulated annealing applied to the curriculum-based course timetabling problem. ***Submitted***, 2014.

[7] Ruggero Bellio, Luca Di Gaspero, and Andrea Schaerf. Design and statistical analysis of a hybrid local search algo-

rithm for course timetabling. *Journal of Scheduling*, 15(1):49–61, 2012.

[8] Mike Benchimol, Pascal Benchimol, Benoît Chappert, Arnaud De la Taille, Fabien Laroche, Frédéric Meunier, and Ludovic Robinet. Balancing the stations of a self service bike hire system. *RAIRO – Operations Research*, 45(1):37–61, 2011.

[9] Frédéric Benhamou, David A. McAllester, and Pascal Van Hentenryck. Clp(intervals) revisited. Technical report, 1994.

[10] Russell Bent and Pascal Van Hentenryck. A two-stage hybrid local search for the vehicle routing problem with time windows. *Transportation Science*, 38(4):515–530, 2004.

[11] Christian Bessiere. *Handbook of Constraint Programming: Constraint Propagation.* Elsevier, New York, NY, USA, 2006.

[12] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of GECCO'02: the Genetic and Evolutionary Computation Conference*, pages 11–18. Morgan Kaufmann Publishers Inc., 2002.

[13] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-race and iterated f-race: An overview. In *Experimental methods for the analysis of optimization algorithms*, pages 311–336. Springer, 2010.

[14] Christian Blum. Beam-acoâĂŤhybridizing ant colony optimization with beam search: An application to open shop scheduling. *Computers & Operations Research*, 32(6):1565–1591, 2005.

[15] Christian Blum and Marco Dorigo. The hyper-cube framework for ant colony optimization. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(2):1161–1172, 2004.

[16] Christian Blum, Jakob Puchinger, Günther R. Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151, 2011.

[17] Christian Blum, Andrea Roli, and Marco Dorigo. Hc–aco: The hyper-cube framework for ant colony optimization. In *Proceedings of MIC'01: the 4th Metaheuristics International Conference*, volume 2, pages 399–403, 2001.

[18] Alex Bonutti, Fabio De Cesco, Luca Di Gaspero, and Andrea Schaerf. Benchmarking curriculum-based course timetabling: formulations, data formats, instances, validation, visualization, and results. *Annals of Operations Research*, 194(1):59–70, 2012.

[19] Edmund Burke, Jakub Mareček, Andrew Parkes, and Hana Rudová. A supernodal formulation of vertex colouring with applications in course timetabling. *Annals of Operations Research*, 179(1):105–130, 2010.

[20] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Barry McCollum, Gabriela Ochoa, Andrew J. Parkes, and Sanja Petrovic. The cross-domain heuristic search challenge–an international research competition. In *Learning and Intelligent Optimization*, pages 631–634. Springer, 2011.

[21] Edmund K. Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R. Woodward. A classification of hyper-heuristic approaches. In *Handbook of Metaheuristics*, pages 449–468. Springer, 2010.

[22] Edmund K. Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. *Handbook of Metaheuristics*, pages 457–474, 2003.

[23] Edmund K. Burke, Jakub Mareček, Andrew J. Parkes, and Hana Rudová. A branch-and-cut procedure forâĂătheâĂău-

dine course timetabling problem. *Annals of Operations Research*, 194:71–87, 2012.

[24] Edmund K. Burke, Jakub Mareček, Andrew J. Parkes, and Hana Rudová. Decomposition, reformulation, and diving in university course timetabling. *Computer and Operations Research*, 37(3):582–597, 2010.

[25] Valentina Cacchiani, Alberto Caprara, Roberto Roberti, and Paolo Toth. A new lower bound for curriculum-based course timetabling. *Computers & Operations Research*, 2013.

[26] Hadrien Cambazard, Emmanuel Hebrard, Barry OâĂŹSullivan, and Alexandre Papadopoulos. Local search and constraint programming for the post enrolment-based course timetabling problem. *Annals of Operations Research*, 194(1):111–135, 2012.

[27] Sara Ceschia, Luca Di Gaspero, and Andrea Schaerf. Tabu search techniques for the heterogeneous vehicle routing problem with time windows and carrier-dependent costs. *Journal of Scheduling*, 14(6):601–615, 2011.

[28] Sara Ceschia and Andrea Schaerf. Local search and lower bounds for the patient admission scheduling problem. *Computers & Operations Research*, 38(10):1452–1463, 2011.

[29] Sara Ceschia, Andrea Schaerf, and Thomas Stützle. Local search techniques for a routing-packing problem. *Computers & Industrial Engineering*, 66(4):1138–1149, 2013.

[30] Daniel Chemla, Frédéric Meunier, and Roberto Wolfler Calvo. Bike sharing systems: Solving the static rebalancing problem. *Discrete Optimization*, 2012.

[31] Thomas M. Cioppa and Thomas W. Lucas. Efficient nearly orthogonal and space-filling latin hypercubes. *Technometrics*, 49(1), 2007.

[32] Raffaele Cipriano, Luca Di Gaspero, and Agostino Dovier. A multi-paradigm tool for large neighborhood search. In *Hybrid Metaheuristics*, volume 434 of *Studies in Computational Intelligence*, pages 389–414. Springer, 2012.

[33] Richard K. Congram, Chris N. Potts, and Steef L. Van De Velde. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *Computing, INFORMS Journal on*, 14(1):52–67, 2002.

[34] Claudio Contardo, Catherine Morency, and Louis-Martin Rousseau. Balancing a Dynamic Public Bike-Sharing System. Technical Report CIRRELT-2012-09, CIRRELT, Montreal, Canada, 2012.

[35] George B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity Analysis of Production and Allocation*, 1951.

[36] Luca Di Gaspero. *Local Search Techniques for Scheduling Problems: Algorithms and Software Tools.* PhD thesis, Dipartimento di Matematica e Informatica, Universt'a degli Studi di Udine, 2013.

[37] Luca Di Gaspero, Barry McCollum, and Andrea Schaerf. The second international timetabling competition (ITC-2007): Curriculum-based course timetabling (track 3). Technical report, Queen's University, Belfast, UK, August 2007.

[38] Luca Di Gaspero, Andrea Rendl, and Tommaso Urli. Constraint-based approaches for balancing bike sharing systems. In *CP'13: The 19th International Conference on Principles and Practice of Constraint Programming*, pages 758–773. Springer, Berlin-Heidelberg, 2013.

[39] Luca Di Gaspero, Andrea Rendl, and Tommaso Urli. A hybrid aco+cp for balancing bicycle sharing systems. In *HM'13: The 8th International Workshop on Hybrid Metaheuristics*, pages 198–212. Springer, Berlin-Heidelberg, 2013.

[40] Luca Di Gaspero, Andrea Rendl, and Tommaso Urli. Balancing bike sharing systems with constraint programming. *Constraints* ***(submitted)***, 2014.

[41] Luca Di Gaspero and Tommaso Urli. A reinforcement learning approach for the cross-domain heuristic search challenge. *MIC'11: the 9th Metaheuristics International Conference, Proceedings of*, 2011.

[42] Luca Di Gaspero and Tommaso Urli. Evaluation of a family of reinforcement learning cross-domain optimization heuristics. In *LION 6: the Learning and Intelligent Optimization Conference, Proceedings of*, pages 384–389, Berlin-Heidelberg, 2012. Springer.

[43] Benjamin Doerr, Daniel Johannsen, and Carola Winzen. Multiplicative drift analysis. *Algorithmica*, 64(4):673–697, 2012.

[44] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41, 1996.

[45] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. MIT press, 2004.

[46] Greg Durrett, Frank Neumann, and Una-May O'Reilly. Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics. In *FOGA'11: the 11th Workshop on Foundations of Genetic Algorithms, Proceedings of*, pages 69–80. ACM, 2011.

[47] Russell C. Eberhart and James Kennedy. A new optimizer using particle swarm theory. In *Micro Machine and Human Science, 1995., Proceedings of MHS'95: the Sixth International Symposium on*, pages 39–43. IEEE, 1995.

[48] Thomas A. Feo and Mauricio G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, 1995.

[49] Fred Glover. Tabu search - part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[50] Fred Glover. Tabu search - part ii. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[51] Fred Glover and Manuel Laguna. *Tabu search*, volume 22. Springer, 1997.

[52] John Michael Hammersley, David Christopher Handscomb, and George Weiss. Monte carlo methods. *Physics today*, 18:55, 1965.

[53] Nikolaus Hansen. The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer, 2006.

[54] Jin-Kao Hao and Una Benlic. Lower bounds for the ITC-2007 curriculum-based course timetabling problem. *European Journal of Operations Research*, 212(3):464–472, 2011.

[55] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning, 2nd Edition.* Springer, New York, 2009.

[56] Myles Hollander and Douglas A. Wolfe. *Nonparametric Statistical Methods, 2nd Edition.* Wiley, New York, 1999.

[57] Holger Hoos and Thomas Stüzle. *Stochastic Local Search: Foundations & Applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[58] Holger H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.

[59] Kurt Hornik, K. Stinchombe, Maxwell, and Halber White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[60] David S. Johnson, Aragon Cecilia R., Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing:

an experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865–892, 1989.

[61] David S. Johnson, Aragon Cecilia R., Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

[62] James Kennedy and Russel C. Eberhart. Particle swarm optimization. In *Neural Networks, 1995., Proceedings of the IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

[63] James Kennedy and Russell C. Eberhart. A discrete binary version of the particle swarm algorithm. In *Systems, Man, and Cybernetics, 1997., Proceedings of the IEEE International Conference on*, volume 5, pages 4104–4108. IEEE, 1997.

[64] Madjid Khichane, Patrick Albert, and Christine Solnon. Integration of aco in a constraint programming language. In *Ant Colony Optimization and Swarm Intelligence*, pages 84–95. Springer, 2008.

[65] Madjid Khichane, Patrick Albert, and Christine Solnon. Strong combination of ant colony optimization with constraint programming optimization. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *CPAIOR'10: the 7th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*, volume 6140 of *Lecture Notes in Computer Science*, pages 232–245, Berlin-Heidelberg, Germany, 2010. Springer.

[66] Philip Kilby and Paul Shaw. *Handbook of Constraint Programming: Vehicle Routing*. Elsevier, New York, NY, USA, 2006.

[67] Scott Kirkpatrick, Daniel C. Gelatt, and Mario P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[68] Roger Koenker. *Quantile regression.* Cambridge University Press, 2005.

[69] Roger Koenker. `quantreg`*: Quantile Regression*, 2013. R package version 5.05.

[70] John R. Koza. *Genetic Programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[71] Gerald Lach and Marco Lübbecke. Curriculum based course timetabling: new solutions to Udine benchmark instances. *Annals of Operations Research*, 194(1):255–272, 2012.

[72] Rhyd Lewis, Ben Paechter, and Barry McCollum. Post enrolment based course timetabling: A description of the problem model used for track two of the second international timetabling competition. Technical report, Cardiff University, Wales, UK, 2007.

[73] Leo Lopes and Kate Smith-Miles. Pitfalls in instance generation for udine timetabling. In *LION 4: the Learning and Intelligent Optimization Conference, Proceedings of*, 2010.

[74] Zhipeng Lü and Jin-Kao Hao. Adaptive tabu search for course timetabling. *European Journal of Operations Research*, 200(1):235–244, 2009.

[75] Zhipeng Lü, Jin-Kao Hao, and Fred Glover. Neighborhood analysis: a case study on curriculum-based course timetabling. *Journal of Heuristics*, 17(2):97–118, 2011.

[76] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[77] Vittorio Maniezzo, Thomas Stèutzle, and Stefan Voss. *Matheuristics: hybridizing metaheuristics and mathematical programming*, volume 10. Springer, 2009.

[78] Barry McCollum, Andrea Schaerf, Ben Paechter, Paul McMullan, Rhyd Lewis, Andrew J. Parkes, Luca Di Gaspero,

Rong Qu, and Edmund K. Burke. Setting the research agenda in automated timetabling: The second international timetabling competition. *INFORMS Journal on Computing*, 22(1):120–130, 2010.

[79] Bernd Meyer and Andreas Ernst. Integrating aco and constraint propagation. In Marco Dorigo, Mauro Birattari, Christian Blum, Luca Gambardella, Francesco Mondada, and Thomas Stützle, editors, *Ant Colony Optimization and Swarm Intelligence*, volume 3172 of *Lecture Notes in Computer Science*, pages 166–177. Springer, Berlin-Heidelberg, Germany, 2004.

[80] John E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of Applied Optimization*, pages 65–77, 2002.

[81] Tomáš Müller. Itc2007 solver description: a hybrid approach. *Annals of Operations Research*, 172(1):429–446, 2009.

[82] Anh Nguyen, Tommaso Urli, and Markus Wagner. Single- and multi-objective genetic programming: new bounds for weighted order and majority. In *FOGA'13: the 12th Workshop on Foundations of Genetic Algorithms, Proceedings of*, pages 161–172. ACM, 2013.

[83] Gabriela Ochoa, Matthew Hyde, Tim Curtois, Jose A. Vazquez-Rodriguez, James Walker, Michel Gendreau, Graham Kendall, Barry McCollum, Andrew J. Parkes, Sanja Petrovic, et al. Hyflex: a benchmark framework for cross-domain heuristic search. In *Evolutionary Computation in Combinatorial Optimization*, pages 136–147. Springer, 2012.

[84] David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of Metaheuristics*, pages 399–419. Springer, 2010.

[85] Marian Rainer-Harbach, Petrina Papazek, Bin Hu, and Günther R. Raidl. Balancing bicycle sharing systems: A variable neighborhood search approach. In Martin Middendorf

and Christian Blum, editors, *Evolutionary Computation in Combinatorial Optimization*, volume 7832 of *Lecture Notes in Computer Science*, pages 121–132, Berlin-Heidelberg, 2013. Springer.

[86] Roberto Ranon, Marc Christie, and Tommaso Urli. Accurately measuring the satisfaction of visual properties in virtual camera control. In *the 10th International Symposium on Smart Graphics, Proceedings of*, pages 91–102. Springer, Berlin-Heidelberg, 2010.

[87] Roberto Ranon and Tommaso Urli. Improving the efficiency of viewpoint composition. *Visualization and Computer Graphics, IEEE Transactions on*, 2014.

[88] Tal Raviv, Michal Tzur, and Iris A. Forma. Static repositioning in a bike-sharing system: Models and solution approaches. *Transportation and Logistics, EURO Journal on*, 2012.

[89] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, New York, NY, USA, 2006.

[90] Louis-Martin Rousseau, Michel Gendreau, and Gilles Pesant. Using constraint-based operators to solve the vehicle routing problem with time windows. *Journal of Heuristics*, 8(1):43–58, January 2002.

[91] John E. Savage. *Models of computation*, volume 136. Addison-Wesley, Reading, MA, USA, 1998.

[92] John E. Savage. *Models of computation*, volume 136. Addison-Wesley, Reading, MA, USA, 1998.

[93] Andrea Schaerf. A survey of automated timetabling. *Artificial Intelligence Review*, 13(2):87–127, 1999.

[94] Jasper Schuijbroek, Robert Hampshire, and Willem-Jan van Hoeve. Inventory rebalancing and vehicle routing in bike sharing systems. Technical Report 2013-E1, Tepper School of Business, Carnegie Mellon University, 2013.

[95] Christian Schulte. Comparing trailing and copying for constraint programming. In *ICLP*, volume 99, pages 275–289, 1999.

[96] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and programming with gecode, 2010.

[97] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael J. Maher and Jean-Francois Puget, editors, *CP'98: the 4th International Conference on Principles and Practice of Constraint Programming, 1998, Proceedings of*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998.

[98] Yuhui Shi and Russell C. Eberhart. A modified particle swarm optimizer. In *Evolutionary Computation, 1998., Proceedings of the IEEE International Conference on*, pages 69–73. IEEE, 1998.

[99] Yuhui Shi and Russell C. Eberhart. Particle swarm optimization: developments, applications and resources. In *Evolutionary Computation, 2001., Proceedings of the Congress on*, volume 1, pages 81–86. IEEE, 2001.

[100] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[101] Thomas Stützle and Holger H. Hoos. Max–min ant system. *Future generation computer systems*, 16(8):889–914, 2000.

[102] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge University Press, 1998.

[103] Tommaso Urli. Balancing bike sharing systems (bbss): instance generation from the citibike nyc data. *arXiv preprint arXiv:1305.1112*, 2013.

[104] Tommaso Urli. Hybrid cp+lns for the curriculum-based course timetabling problem. *CP'13: Doctoral Program*, page 73, 2013.

[105] Tommaso Urli. json2run: a tool for experiment design & analysis. *arXiv preprint arXiv:1305.1112*, 2013.

[106] Tommaso Urli, Markus Wagner, and Frank Neumann. Experimental supplements to the computational complexity analysis of genetic programming for problems modelling isolated program semantics. In *PPSN'12: Parallel Problem Solving from Nature, Proceedings of*, pages 102–112. Springer, Berlin-Heidelberg, 2012.

[107] Peter van Beek. *Handbook of Constraint Programming: Backtracking Search Algorithms.* Elsevier, New York, NY, USA, 2006.

[108] Peter J. M. Van Laarhoven and Emile H. L. Aarts. *Simulated Annealing: Theory and Applications.* Springer, 1987.

[109] Markus Wagner and Frank Neumann. A fast approximationguided evolutionary multi-objective algorithm. In *Proceedings of GECCO'13: the 15th Annual Conference on Genetic and Evolutionary Computation.* ACM Press, 2013.

[110] Chris Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1-4):325–372, 2004.

[111] Ingo Wegener. Methods for the analysis of evolutionary algorithms on pseudo-boolean functions. *Evolutionary Optimization*, pages 349–369, 2003.

[112] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.