Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN
COMPUTER SCIENCE AND ENGINEERING
Ciclo: XXXI

Settore Concorsuale : 01/B1
Settore Scientifico Disciplinare: INF/01

Innovative Applications of Constraint Programming

Presentata da: Tong Liu

Coordinatore Dottorato:
Paolo Ciaccia

Relatore:
Maurizio Gabbrielli

Correlatore:
Jacopo Mauro

Esame finale anno 2019

Abstract

Constraint programming (CP) is a declarative paradigm that enables us to model a problem in the form of *constraints* to be satisfied. It offers powerful constraint solvers which, by implementing general-purpose search techniques, are fast and robust to address complex constraint models automatically. Constraint programming has attracted the attention of people from various domains. By separating the definition of a problem from its solution, it is more natural for people to implement the program directly from the problem specification, reducing the cost of development and future maintenance significantly. Furthermore, CP provides the flexibility of choosing a suitable solver for a problem of a given nature, which overcomes the limitations of a unique solver. Thanks to this, CP has allowed many non-domain experts to solve emerging problems efficiently.

This thesis studies the innovative applications of CP by examining two topics: constraint modeling for several novel problems, and automatic solver selection. For the modeling, we explored two case studies, namely the (sub)group activity optimization problem, and the service function chaining deployment problem that comes from the Software Defined Network (SDN) domain. Concerning the solver selection, we improved an algorithm selection technique called "SUNNY", which generates a schedule of solvers for a given problem instance. In this work, we demonstrate with empirical experiments that the procedure we have designed to configure SUNNY parameters is effective, and it makes SUNNY scalable to an even broader range of algorithm selection problems not restricted to CP.

Acknowledgements

I would like to express my deep gratitude to Professor Maurizio Gabbrielli, my research supervisor, for his endless advice and patient guidance throughout my time as a graduate student. I thank also my co-supervisor Professor Jacopo Mauro; although we have been working remotely, I learned enormously from many enlightening discussions with him, which has significantly influenced my thinking and research.

I am greatly indebted to Professor Gabbrielli and Professor Mauro for all that I learned from them both in terms of technical knowledge and in terms of research style. Their willingness to give their availability so generously has been very much appreciated. This work would have been impossible without their help and support.

I am also very grateful to Dr. Roberto Amadini, Professor Roberto Di Cosmo and Professor Franco Callegati for their support and advices; working with them was also a hugely positive and enriching experience.

My experience as a Ph.D. student at University of Bologna would have been much poorer but for the colleagues and friends that I had in Bologna. I would like to thank particularly Mariagrazia Amato, Luca Bedogni, Giacomo Bergami, Flavio Bertini, Marcos Caetano, Stefano Cammelli, Angelo Di Iorio, Adrien Durier, Andrea Melis, Federico Montori, Andrea Nuzzolese, Francesco Gavazzo, Saverio Giallorenzo, Micheal Lodi, Vicenzo Lomonaco, Stefano Pio Zingaro, Stefano Rizzo, Rahimeh Rouhi, Daniel Russo, Wilson Sakpere, Luca Sciullo, Angelo Trotta, YF Chen, YC Fan, XL Xu, Chun Tian and YW Zhang. I extend my thanks also to the friends I met at the Doctoral Training Center in Paris (EIT Digital), Wenqin Shao, José

Horta, Gianni Barlacchi, Mourad Leslous and Quang Pham.

The author acknowledges the Ph.D. fundings sponsored by the Italian Ministry of Education (MIUR) and trainings offered by the European Doctoral School, EIT Digital.

List of Acronyms

AI Artificial Intelligence

AS Algorithm Selection

ASlib Algorithm Selection Library

COSEAL COnfiguration and SElection of ALgorithms

CP Constraint Programming

CSP Constraint Satisfaction Problem

ETSI European Telecommunications Standards Institute

ILP Integer Linear Programming

***k*-NN** *k*-Nearest Neighbors algorithm - a non-parametric machine learning method

LCG Lazy Clause Generation

MANO Management and Orchestration

MILP Mixed Integer Linear Programming

MP Mathematical Programming

NBI North Bound Interface

NFV Network Function Virtualization

OASC Open Algorithm Selection Challenge

QoS Quality of Service

SA Simulated Annealing - an approximative search technique in AI

SBI	South Bound Interface
SDN	Software Defined Network
SFC	Service Function Chain
SUNNY	SUb-portfolio Nearest Neighbor lazY - a technique for algorithm selection
VIM	Virtualized Infrastructure Managers
VNF	Virtual Network Function
WAN	Wide Area Network

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This thesis studies the application of constraint programming (CP) in solving novel problems and the improvement of solving efficiency with the aid of solver selection.

As mentioned in a Chinese idiom that "Nothing can be accomplished without norms or standards" (Mencius, 300 BC), it can be noted that humans are surrounded by *constraints*, which arise naturally from their endeavors. The constraints, being a medium of expression for formalizing regularities, limit, oblige, or even prevent people when they make decisions. Today, with the use of of artificial intelligence (AI), many problems that involve constraints can be modeled as constraint satisfaction problems (CSPs). Within this framework, a problem can be represented concisely by a set of constraints over a set of variables with a finite domain, and a solution includes values that are assigned to all variables that contemporaneously satisfy all the constraints.

When applying CP to the CSPs, the principal task of a user is to *describe* the elements that combine to form the problem and the properties of a solution that can be found. The language used for this description typically belongs to the declarative programming paradigm which differs from the imperative programming mainly in the fact that the specific instructions which express the control flow needed to obtain the solution are no longer necessary. Indeed, the user needs only to formulate "declaratively" the problem model, while the control flow and the results generation are taken care of by a black-box engine. For CP, in particular, results are computed by a *constraint solver*, which interprets the variables and constraints and automatically adopts appropriate search strategies that lead to the solution.

Due to the characteristics of the declarative languages, developers who use CP can focus on building complex relations between problem entities, which allows them to develop the problem model more naturally than by using the imperative language paradigm. As a result of the freedom of utilizing an independent CP solver, users can identify among the state-of-the-art solvers the most appropriate one for their specific problem. Given these advantages, people have been attracted to CP from various domains, and they embed CP as a core component in their projects to solve practical, applicative, and industrial problems. These problems range from scheduling to packaging problems, from production to design problems, and from entertainment to financial problems [131].

In the first part of this thesis we present two main contributions for CP application. First, we use CP to solve the group activity optimization problem. Specifically, we generate (sub)group activity schedules by forming groups of users with similar preferences. At the same time, we ensure group activity synchronization by considering the time and topological constraints. Afterward, we compare the CP solution with that of an approximative approach (local search method), and we show the computation limit of CP for large scale instances and how a heuristic-based approximative solution can scale better. The second contribution is the application of CP to the service function chain deployment problem. To be brief, in a typical software defined network (SDN) scenario, there exist several domains (such as data centers and Internet service providers) that offer network services at different costs and under various conditions. A service function chain is a sequence of network services, possibly across domains, that satisfies the order requested by the user, meets the conditions of the domain policy, and at the same time, minimizes the deployment cost. To tackle this problem, we propose a general framework using CP, showing the feasibility of handling non-trivial service chain problems in real time.

We compare the search technique of CP with mixed integer linear programming (MILP), concluding that CP is faster than the MILP technique for this problem, although MILP is currently the most accredited and renowned method in the SDN field [108, 58].

The second part of the thesis addresses some efficiency problems in using solvers. The efficiency of solving problems modeled with CP depends not only on how a problem is modeled but also on the implementation method of constraint solvers. Indeed, different kinds of solvers may have their strengths in solving different categories of problems or even different instance cases. As a result, users are usually encouraged to compare the available solvers manually to identify the most suitable for their specific problem. The process of selecting suitable solvers (or algorithms) can be improved significantly with the help of machine learning. Considering each solver's historical performances regarding different types of instances, it is possible to predict the best solver, or schedule of solvers (each solver being used in an assigned time slot), to be applied to an unseen instance. The study of algorithm selection (AS) originated from Rice [123], and recently, it is becoming more attractive with the release of the AS library ASlib [23]. This library collects AS problems from a number of domains and aims to understand the scalability and robustness of various AS approaches. In this thesis, we present several improvements to the SUNNY [10] AS technique. SUNNY generates a schedule of solvers for solving problem instances based on the k-NN [3]. Initially, this technique has been studied only within the CP domain and, in particular, with the MiniZinc challenge dataset [8]. SUNNY has proven its effectiveness by winning the first-place prize in the open track of the MiniZinc Challenge (2015-2017). However, it performed poorly with the ASlib benchmark in the ICON challenge. In this work, we introduce our improvements to SUNNY with an automatic parameter configuration, with which SUNNY yields promising results in the second challenge of the ASlib benchmark, which is the OASC challenge.

1.1 Thesis outline and contributions

In this section, we provide a brief overview of the content of this thesis. Essentially, we divide this thesis into two main parts. The first part (Chapters 3 and 4) introduces two application problems with CP, which are a (sub)group activity optimization problem and a flexible service function chain deployment problem. The second part (Chapter 5) describes the improvements to the SUNNY AS technique to solve ASlib problems.

In more detail, in

Chapter 2 we offer an overview of the CP and portfolio-based algorithm selection for CP problems.

Chapter 3 we present an application tool for (sub)group activities, describe the problem model, prove the problem hardness, and implement the solution using both CP and a local search approach (simulated annealing). Furthermore, we use empirical experiments to compare the performances of the two techniques.

Chapter 4 we describe the problem of flexible service function chaining (SFC), which is becoming attractive in the SDN domain [28, 108]. We define a general model for this problem and solve it by comparing the CP and MILP techniques. In particular, we show how CP can outperform the conventional MILP approach for solving SFC problems.

Chapter 5 we introduce the portfolio-based AS technique, SUNNY, which exploits the synergy of available solvers to improve the efficiency of CP solving. We also describe and justify the improvements made to SUNNY to make it scalable to a broader range of AS problems (ASlib).

Chapter 6 contains concluding remarks and the direction of future research.

All of the original contributions in this dissertation have either already been published or are in preparation for review. In particular, the work presented in Chapter 3 has been published in [99], while the work in Chapter 4 will appear in

[98]. Part of the work in Chapter 5 has been published in [4, 5, 97], and a journal version is in preparation.

Chapter 2

Background

Intuitively, a *constraint* can be regarded as the restriction over a space of possibilities, it is something that restricts, limits or regulates. This notion gives birth to an important field of Artificial Intelligence: Constraint Programming (CP).

Constraint Programming became attractive not only for its strong theoretical foundation but also for its potentials to solve hard real-life problems. Its success comes from the fact that on one hand as declarative presentation it allows to model a problem in a way easy-to-read and on the other hand it is supported by general-purpose algorithms which are efficient for wide range of problems.

The literature on Constraint Programming is vast [125, 105, 27, 20, 143, 142, 126], among them we would mention the Books [125, 105] which provide a complete and comprehensive presentation of CP. In this chapter we select the most relevant topics concerning our works, namely, Constraint Satisfaction Problems, Modeling CSP and theory of solving techniques. Afterwards, we review the applications of CP and point out its limitations. In the end, we briefly discuss other related techniques: mathematical programming (well-known in Operations Research) and techniques that improve the boundary of solving effectiveness such as local search and portfolio-approaches. First of all let us start with a brief history about CP.

2.1 Brief History of CP

Constraint Programming (CP) has a long tradition, the initial ideas leading to CP can be found in the Artificial Intelligence (AI) field dating back to 1960s and 1970s [20, 142].

For instance, the application for *interactive graphics* - Sketchpad - was developed in early 1960s by Ivan Sutherland [136] (who was then awarded the Turing Prize in 1988). The application allows users to draw and manipulate constrained geometric figures on computer's display, at that time, a constraint language for graphical interaction was introduced. This work has also contributed to the notion of local propagation and constraint compiling. In the following 20 years, based on the languages such as Fikes' REF-ARF [46], Laurière's Alice [90], Sussman's CONSTRAINTS [135] and Borning's ThingLab [25] the language for Constraint Programming has slowly taken shape and reflects the common properties of these languages:

- declarative problem modeling
- propagation of the effects of decision
- efficient search for feasible solution

The milestone towards CP was achieved in the 1980s where Gallaire [53] and Jaffar & Lassez [81] recognized *Logic Programming* as a special kind of Constraint Programming since the basic idea behind Logic Programming (declarative modeling) is similar to CP. Therefore constraints and logic programming have been naturally combined and yielded languages such as: Prolog III, CLP(R), and CHIP. However, this does not implies that constraint programming is restricted to CLP. Constraints can be integrated via software libraries to typical imperative languages like c++ or Java as well.

Over a long period and extensive research, CP presents an inner interdisciplinary nature. It combines and exploit the ideas from a number of different fields including for example, Artificial Intelligence, Combinatorial Algorithms, Computational Logic, Discrete Mathematics, Operations Research, Programming Languages and Symbolic computation etc.

2.2 Constraint Satisfaction Problems (CSPs)

Constraint Satisfaction Problems have been a subject of research in AI for plenty of years. A Constraint Satisfaction Problem (CSP) is defined as:

- a set of variables $X = x_1, \ldots, x_n$
- for each variable x_i, a finite set D_i of possible values (its domains) like integers or strings.
- a set of constraints restricting the values that the variables can simultaneously take.

Example 2.1 *Let us see how the famous crypto-arithmetic game can be modeled. Let us consider for instance "F A T H E R + M O T H E R = P A R E N T" authored by David J. Porter. The game consists of a mathematical equation among unknown numbers, whose digits are represented by letters. To solve the game we need to associate to every letter in "father mother parent" a different number from 1 to 10 in a way that the sum of "father" and "mother" is equal to "parent".*

Since we need to find what numbers are associated to every letter we can model these numbers with variables which domain is the set $\{0, \ldots, 9\}$. *Let be* $(F, A, T, H, E, R, M, O, P, N)$ *a set of variables each with domain* $\{0, \ldots, 9\}$.

The set of constraints to consider are:

- $(100000 * F + 10000 * A + 1000 * T + 100 * H + 10 * E + R) + (100000 * M + 10000 * A + 1000 * T + 100 * H + 10 * E + R) = (100000 * P + 10000 * A + 1000 * R + 100 * E + 10 * N + T)$
- *alldifferent*$(F, A, T, H, E, R, M, O, P, N)$
- $F \neq 0$

- $M \neq 0$
- $P \neq 0$

This problem allows the assignment $(F, A, T, H, E, R, M, O, P, N) = (2, 9, 6, 7, 5, 3, 1, 8, 4, 0)$ *as solution.*

A solution to a CSP is a labeling, i.e. an assignment of a value from its domain to every variable, in such a way that all constraints are satisfied at once. We may want to find:

- just one solution, with no preference
- all solutions
- an optimal, or at least a good solution, given some objective function defined in terms of some or all of the variables

Solutions to a CSP can be found by searching (systematically) through the possible assignments of values to variables. Search methods divide into two broad classes, those that traverse the space of partial solutions (or partial value assignments), and those that explore the space of complete value assignments (to all variables) stochastically.

2.2.1 Constraint Optimization Problem

In many real-life applications we are not just interested in finding "one" solution but "the" optimal solution, or at least a good one. The quality of the solutions is usually measured by an application-dependent function called objective function which can score a solution numerically. In this case, the goal is to find a solution in which the objective function gets minimized or maximized. These kinds of problems are referred to as Constraint Optimization Problems (COPs). [1]

[1]Note that sometimes the COP may also refer to Combinatorial Optimization Problem [64].

In the rest of this thesis, without loss of generality, we will always consider a COP as a minimization problem. Indeed, it is always possible to switch from a maximization problem to an equivalent minimization problem by simply negating the objective function. Formally, a COP can be defined as follows:

Definition 2.1 (COP) *A Constraint Optimization Problem (COP) is a quadruple* $P := (X, D, C, f)$ *where:*

- $P' := (X, D, C)$ *is a CSP;*
- $f : D \to R$ *is the objective function of* P*.*

The goal is normally to find a solution of P' that minimizes f. Indeed, a COP is a special case of the CSP; a COP can be regarded as a CSP in which f is a constant over D. By guessing the values of objective function, a CSP can eventually find a solution in which f is minimized.

2.3 Solving CSP

The searching algorithm designed for CSPs [126] is based on the structure of *states* that the values are assigned to each variable, therefore, it is a kind of *general-purpose* algorithm rather than *problem-specific* heuristics. This distinguishes CP from other popular techniques tailored for specific disciplinary problems.

From the theoretical point of view, solving CSP is trivial using the systematic exploration of the solution space. Even if systematic search methods without additional improvements seem straightforward and non-efficient, they still worth mentioning since they are the foundation of more advanced and efficient algorithms.

The basic constraint satisfaction algorithm that searches the space of complete labelings, is called generate-and-test. The idea is simple: complete labeling of variables is generated and, consequently, if this labeling satisfies all the constraints then the solution is found; otherwise, another labeling is generated. The generate-and-test algorithm is a weak generic algorithm that is used if everything else failed. Its efficiency is poor due to non-informed generator and late discovery of inconsistencies. Consequently, there are two ways to improve its efficiency:

- the generator of valuations is smart, i.e., it generates valuations in such a way that the conflict found by the test phase is minimized.

- the generator is merged with the tester, i.e. the validity of the constraint is tested as soon as its respective variables are instantiated. This method is used by the backtracking approach. Backtracking [125] is a method of solving CSP by incrementally extending a partial solution that specifies consistent values for some of the variables, towards a complete solution, by repeatedly choosing a value for another variable consistent with the values in the current partial solution. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. As a result, backtracking is strictly better than generate-and-test. However, its running complexity for most nontrivial problems is still NP-hard.

There are three major drawbacks of the basic backtracking:

1. thrashing, i.e., repeated failures due to having not identified the real reason of the conflict (e.g., conflict variables).

2. redundant work, i.e., the variable values that cause conflict are not remembered.

3. late detection of the conflict, i.e., the potential conflict is not detected until it occurs.

Next, we present some of the improvements to backtracking discussed in the literature.

2.3.1 Consistency Techniques

One alternative approach for solving CSP is based on removing inconsistent values from variables' domains until a solution appears. [2] These methods are called consistency techniques. There are several consistency techniques [89, 101] but most of them are not complete, i.e., they can not be used alone to solve a CSP completely. The names of basic consistency techniques are derived from the graph notions. The CSP is usually represented as a constraint graph or hyper-graph (sometimes called constraint network) where nodes correspond to variables and edges / hyper-edges are labeled by constraints.

The simplest consistency technique is referred to as a node consistency. It removes values from variable domains that are inconsistent with unary constraints on respective variables. The most widely used consistency technique is called *arc consistency* (AC). This technique removes values from variables domains that are inconsistent with binary constraints. There exist several arc consistency algorithms starting from AC-1 based on repeated revisions of arcs till a consistent state is reached or some domain become empty. The most popular among them are AC-3 and AC-4. AC-3 works with deleting inconsistent values from variable domains while AC-4 keeps in memory a list that tracks unsupported values. It is claimed that, in many cases, AC-3 works better than AC-4 in establishing arc consistency [146].

Even more inconsistent values can be removed by path consistency techniques. Path consistency is a property similar to arc consistency but considers pairs of variables instead of only one. A pair of variables is path-consistent with a third variable if each consistent evaluation of the pair can be extended to the other variable in such a way that all binary constraints are satisfied. There exist several path consistency algorithms like PC-1 and PC-2 but, compared to algorithms for arc consistency, they need an extensive representation of constraints that is memory consuming.

All above-mentioned consistency techniques are covered by a general notion of

[2]Although consistency techniques are outside the scope of this thesis, we still mention them here since they are the fundamentals of constraint solvers.

k-consistency [50] and strong k-consistency. A constraint graph is k-consistent if, for every system of values for $k-1$ variables satisfying all the constraints among these variables, there exist a value for an arbitrary k-th variable such that the constraints among all k variables are satisfied. A constraint graph is strongly K-consistent if, it is j-consistent for all $j \leq k$. We have that:

- node consistency is equivalent to strong 1-consistency.
- arc consistency is equivalent to strong 2-consistency.
- path consistency is equivalent to strong 3-consistency.

Algorithms exist for making a constraint graph strongly k-consistent for $k > 2$, but in practice, they are rarely used because of efficiency issues.

Although these algorithms remove more inconsistent values than any arc-consistency algorithm they do not eliminate the need for the search in general. Restricted forms of these algorithms removing a similar amount of inconsistencies with a greater efficiency have been proposed. For example, directional arc consistency revises each arc only once, requires less computation than AC-3 and less space than AC-4 but is still able to achieve full arc consistency in some problems. It is also possible to weaken the path consistency in a similar way.

2.3.2 Constraint Propagation

Either systematic search or consistency techniques can be used alone to completely solve the CSP but this is not suggested in practice. A combination of both approaches is more commonly used. To avoid some problems of backtracking like thrashing or redundant work, look-back schemes were improved. Backjumping [55] for instance is a method to avoid thrashing. The control of backjumping is exactly the same as backtracking except when assignment conflict takes place. Both algorithms pick one variable at a time and look for a value for this variable making sure that the new assignment is compatible with values committed so far. However, when backjumping finds an inconsistency, it analyses the situation in order to identify the *source* of inconsistency. It uses the violated constraints as guidance to find out the conflicting variable. If all the values in the domain are explored then the backjumping algorithm backtracks to the most recent conflicting variable. This is the main difference from the backtracking algorithm that backtracks to the immediate past variable.

Another look-back schema called backchecking [70] avoids redundant work. Backchecking and its evolution backmarking are useful algorithms for reducing the number of compatibility checks. For example, if the algorithm finds that some label Y/b is incompatible with any recent label X/a then it remembers this incompatibility. As long as X/a is still committed, the Y/b will not be considered again. Backmarking is an improvement over backchecking since it reduces the number of compatibility checks by remembering for every label the incompatible recent labels and avoids repeating compatibility checks which have already been performed.

All look-back schemes share the disadvantage of late detection of the conflict. Indeed, they solve the inconsistency when it occurs but they do not prevent the inconsistency to occur. For this reason look-ahead schemes were proposed. For instance forward checking, the simplest example of look ahead strategy, performs arc-consistency between pairs of a non-instantiated variable and an instantiated one removing temporarily the values that the non instantiated variable can not assume. It maintains the invariance that for every unlabeled variable there exists at least one value in its domain that is compatible with the values of instantiated/labeled variables. Even though forward checking does more work than backtracking when each assignment is added to the current partial solution, it is almost always a better choice than chronological backtracking.

Further future inconsistencies are removed by the partial look-ahead method. While forward checking performs only the checks of constraints between the current variable and the not defined variables, the partial look-ahead extends this consistency checking even to variables that have not direct connection with labeled variables, using directional arc consistency. The approach that uses full arc-consistency after each labeling step is called *full* look ahead.

2.3.3 Lazy Clause Generation (LCG)

Lazy clause generation combines the strengths of CP propagation and SAT solving. The key idea is to mimic the underlying rules of FD propagators by properly generating corresponding SAT clauses. The clause generation is "lazy" since it is not performed a priori, but it occurs during the search. This approach enables a strong nogood learning, able to detect and analyze the conflicts that occur during the search. Typical advantages of LCG have been discussed on the RCPSP/max problem [128]. Moreover, the lazy clause generation solver Chuffed [57] has dominated the MiniZinc Challenges 2012–2014, and the Google Or-tools which adopted the LCG has boosts its performance significantly in the MiniZinc Challenges 2017-2018.

2.4 Applications of CP

With the hope of reducing development time while preserving the efficiency of procedural language, CP has been found attractive in many application domains, for instance, CP for DNA structure analysis, time-tabling for hospitals or industry scheduling. It proved to be well adapted for solving real-life problems because many application domains evoke constraint descriptions naturally.

The first type of industrial application of CP was perhaps the assignment problems. A typical example is the stand allocation for airports, where aircraft must be parked on the available stand during the stay at airport or counter allocation for departure halls. Another example is berth allocation to ships in the harbor or refinery berth allocation.

Another typical constraint application area is personnel assignment where work rules and regulations impose difficult constraints. The important aspect in these problems is the requirement to balance work among different persons. Systems like Gymnaste [32] were developed for production of rosters for nurses in hospitals, for crew assignment to flights or stuff assignment in railways companies.

Successful applications for finite domain constraint are the once that solve scheduling problems, where, again, constraints express naturally the real life limitations.

Constraint based software is used for well-activity scheduling, forest treatment scheduling, production scheduling in plastic industry or for planning production of military and business jets. The usage of constraints in Advanced Planning and Scheduling systems is increasing due to current trends of on-demand manufacturing.

Another large area of constraint application is network management and configuration. These problems include planning of cabling of the telecommunication networks in the building or electric power network reconfiguration for maintenance scheduling without disrupting customer services. Another example is optimal placement of base stations in wireless indoor telecommunication networks [51]. There are many other areas that have been tackled using constraints. Recent applications of constraint programming were used in computer graphics, natural language processing, database systems, molecular biology, business applications, electrical engineering and transport problems.

2.4.1 Limitations

Since many problems solved by CP are NP-hard problems, the identification of restrictions that make the problem tractable is very important for both the theoretical and the practical points of view. Unfortunately, the efficiency of constraint programs is still unpredictable and the intuition is usually the most important part of deciding when and how to use constraints. A common problem for CP users is the stability of the constraint model. Even small changes in a program or in the data can lead to a dramatic change in performance. The process of performance debugging for a stable execution over a variety of input data is currently not well understood.

Another problem is choosing the right constraint satisfaction technique for a particular problem. Sometimes fast blind search like chronological backtracking is more efficient than more expensive constraint propagation and vice versa.

A particular problem in many constraint models is the cost optimization. Sometimes, it is very difficult to improve an initial solution, and a small improvement takes much more time than finding the initial solution.

Finally constraint programs can add constraints dynamically but they do not support the on-line constraint solving required for instance in a changing environment. For instance the possibility of deleting a constraint at runtime has been considered by some extensions like the ones described in [145] but this kind of operation are yet too costly to be performed.

2.5 Other techniques to Solve CSP

Although CP is efficient in solving plenty of practical problems, it worth knowing that there exist similar techniques with longer history, for instance the Linear Programming from Operations Research (OR). In this section we briefly introduce OR as well as other methods which are able to enhance CP efficiency: Local Search and Portfolio Approaches.

2.5.1 Operations Research (Mathematical Programming)

Briefly speaking, Operations Research (OR, a.k.a Operational Research) is the discipline that helps to make better decisions by the application of advanced analytical methods. The study originated in military efforts during World War I, and subsequently widely applied to civilian purposes in a huge variety of fields including business, finance, logistics, and society. OR encompasses a wide range of problem-solving techniques and methods applied in the pursuit of improved decision-making and efficiency, such as simulation, mathematical optimization, queueing theory and other stochastic-process models, Markov decision processes, econometric methods, data envelopment analysis, neural networks, expert systems, decision analysis, and the analytic hierarchy process. [3] In particular, the COPs are well studied and used in practice in many areas such as services, logistics, transports, economics, as well as in other industrial applications. Operations research has proved to be useful for modeling problems of planning, scheduling, assignment, routing and design. In this

[3]From http://en.wikipedia.org/wiki/Operations_research.

section an overview of the classical OR optimization approaches and a comparison between CP and OR techniques are described.

Linear Programming

Linear programming (LP) is a general OR optimization method in which both the constraints and optimization function are linear. The canonical form of a LP problem is defined as:

$$\texttt{maximize } c^T x \texttt{ subject to } Ax \leq b, x \geq 0 \tag{2.1}$$

where $x \in \mathbb{R}^n$ is the vector of the variables to be assigned, $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ are vectors of known coefficients (c^T is the transpose of c) while $A \in \mathbb{R}^{mn}$ is the matrix of the constraints coefficients. The inequalities $Ax \leq b$ are constraints that specify a convex polyhedron (the feasible region) over which the objective function $f(x) = c^T x$ has to be maximized.

Every LP problem (or linear program), referred to as a primal problem, can be converted into a corresponding dual problem, which provides an upper bound to the optimal value of the primal problem [26]. Note that the dual of a dual linear program is the original primal linear program. Given the above definition of primal problem, the corresponding dual is:

$$\texttt{minimize } b^T y \texttt{ subject to } A^T y \geq c, y \geq 0 \tag{2.2}$$

The theory of the duality shows some interesting properties (e.g., the duality theorems) and it is also exploited by the simplex algorithm [112]. This method, devised by George Dantzig in 1947, makes use of the concept of simplex (i.e., a polytope of $n + 1$ vertices in n dimensions) for solving LP programs. Other effective techniques for solving LP problems are instead based on interior point methods [119].

According to the variables domain, the LP problem can be specialized in different problems. For instance, when all of the variables values are required to be integers, it becomes the *Integer Linear Programming* (ILP) problem. Comparing to LP, which can be solved efficiently in the worst case, ILP problems are NP-hard in many practical situations; When only some of the variables are required to be integers, then it becomes a *Mixed Integer Programming* (MIP) problem. These are also NP-hard since they are even more general than ILP programs. However, despite the NP-hardness, some important subclasses of ILP and MIP problems are efficiently solvable [17, 140]. The algorithms for solving LP problems include for instance the Simplex algorithm, the cutting-plane method and the column generation [72, 41].

Constraint Programming vs. Mathematical Programming

Constraint Programming and Mathematical Programming are regarded as different approaches for solving combinatorial problems. Both of these techniques have strengths as well as weaknesses, for which reason it is not possible to determine which is the best technique to be adopted in general.

The two approaches come from different nature, and their basic differences are considered as follows.

- CP models the problem with variables of discrete values (integer or Boolean) while MP supports both discrete and continuous variables.
- CP natively supports logical constraints as well as a full range of arithmetic expressions including modulo, integer division, or the element expression which indexes an array of values by a decision variable. In contrast, MP supports only linear constraints, linearized logical constraints, or quadratic convex constraints.
- CP models have no limitation on the arithmetic constraints that can be set on decision variables, while an MP engine is specific to a class of problems whose solution space satisfies certain mathematical properties.
- CP provides an easy way to deal with inference methods, logic processing, high-level problem modeling; MP works well with relaxation methods, duality theory and atomistic problem modeling.

Table 2.1: Mathematical Programming vs. Constraint Programming

Features	Mathematical Programming	Constraint Programming
Domain relaxation	YES	NO
Optimality proof	YES	YES
Modeling limitation	Restricted to linear and Quadratic problems	Discrete problems
Optimality proof	YES	YES
Specialized constraints	NO	YES
Logical constraints	YES	YES
Theoretical basis	Algebra	Graph theory and algorithmic
Model and solver are independent	YES	YES

A compact comparison [74] of the two approaches is described in Tab. 2.1.

In practice, the general advantages of CP consist in being better at sequencing and scheduling, in the more natural modeling, in the use of global constraints, and in a natural way to locally control the constraints, however, it is weak in treating continuous variables as well as over-constrained optimization problems.

An answer to the question "when should we prefer CP than MP and vice versa" is given by a guideline from Google [75]. They suggest that MP works faster than CP for problem model with less alternatives, i.e. *all* the constraints must hold for a solution to be feasible (e.g. constraints are connected only by the "and" statements); on the contrary, CP is generally faster at solving the problem model where constraints require only one property to be satisfied (constraints connected by "or" statements).

Some researchers claim that CP and MP have complementary strengths. And in order to achieve better performances and solve large combinatorial problems, it has been natural to try to integrate these two approaches [109]. The emerging research field of the integration between OR techniques and CP is promising and stimulating.

Some of the main challenges involves the interaction between the user and the solving process, the resolution of partially unknown or ill-defined problems, the processing of large scale over-constrained problems, and the improvement of the CP solving process, both in the constraints propagation and in the solution search.

2.5.2 Local Search

Due to the large size and the heterogeneous nature of real-world combinatorial problems, it is sometimes impracticable to use exact approaches. A possible approach to challenge this is the use of Local Search (LS) methods. LS methods are generally greedy approaches relying on a simple idea: trying to improve a solution at hand by moving step to step towards a possibly better solution. When no better solutions can be found by partial solution modifications, it means that a *local optimum* was reached. To avoid getting stuck in a local optimum, several heuristics can be employed. In the work [47], different hybrid methods are reported which combine the LS and CP taking the advantage of LS efficiency and the flexibility of CP paradigm. Some local search methods (e.g., [30, 40, 114]) used CP as a way to efficiently explore large neighborhoods with side constraints. Others, such as [31], used LS as a way to improve the exploration of the search tree. In the particular context of the CSPs, a LS approach iteratively tries to improve an assignment of the variables until all the constraints are satisfied. The local search is therefore performed in the space D of the possible assignments, by means of a proper evaluation function for measuring the quality of the assignments (e.g., in terms of the number of violated constraints). Two main classes of local search algorithms exist: non-randomized and randomized. The non-randomized uses the greedy technique, well-known examples are the Hill Climbing [130], Variable Neighborhood Search [111] and the Tabu Search [59]. Their drawback concerns the possibility of getting stuck in a sub-optimal state. The randomized LS aims to overcome this issue, example algorithms in this fashion are Evolutionary Algorithms [19] and Simulated Annealing [144].

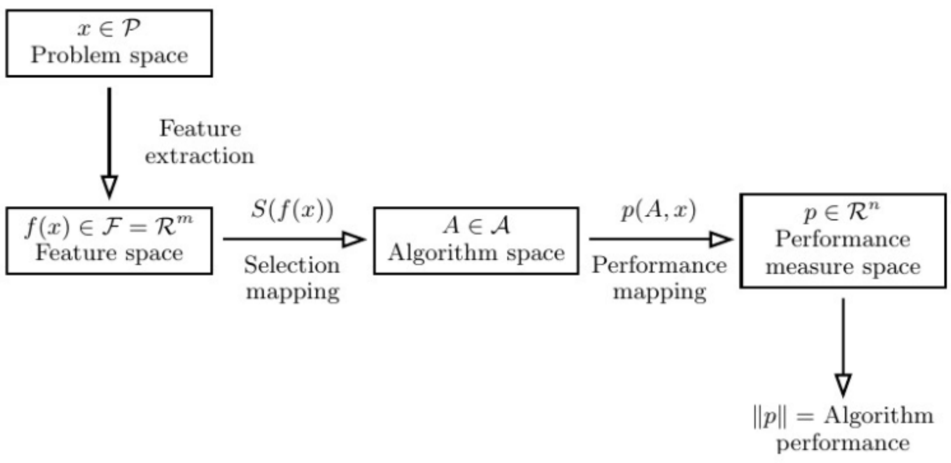


Figure 2.1: Refined model for the Algorithm Selection Problem.

2.5.3 Portfolio Approaches

Portfolio Approaches are an alternative way to improve solving efficiency by exploiting the usage of more solvers. It is well recognized in the field of AI that different algorithms have different performance on different categories of problems (or even problems belonging to the same category). As pointed out also by the "No Free Lunch" theorems [147], it is evident that a single algorithm can not be a panacea for all possible problems. Given a problem x and a collection of different algorithms $A_1, A_2, ..., A_m$, the algorithm selection (AS) problem basically consists in selecting which algorithm A_i performs better on x. This problem was originally introduced by John R. Rice in 1976 [123]. An overall diagram to represent his model is depicted in Figure 2.1. Here, given an input problem x, a vector $f(x)$ of features which is extracted from x, the problem is finding the algorithm(s) from a set of available ones which are supposed to have good performance on x. The notion of "good performance" is not self-contained but defined according to suitable metrics to represent the algorithm performance. Formally, the performance of algorithm A on x is mapped by a performance function P to a measure space $p = P(A, x) \in \mathbb{R}^n$. It is then a measure $|p| \in \mathbb{R}$ obtained from $P(A, x)$ to be maximized or minimized.

The Algorithm Portfolio [60] can be regarded as a particular approach to CP solving. The boundary between Algorithm Selection and Algorithm Portfolios is not evident and these two concepts could be considered as synonyms. According to [87], by definition, algorithm portfolios can be seen as particular instances of the more general AS framework in which the algorithm selection is performed case-by-case. Within the context of CP, the algorithm space consists of a portfolio $s_1, s_2, ..., s_m$ of different CP solvers, we can thus consider a portfolio solver as a particular constraint solver that exploits the strengths of constituent solvers inside its portfolio. When dealing with an unseen problem p, the portfolio solver, based on the instance features, tries to predict which are the best constituent solvers $s_1, s_2, ..., s_k$ $(k \leq m)$ for solving p and then apply them to p in a sequential or parallel way.

Coming back to practice, there are several surveys to show the effectiveness of applying Algorithm Portfolio to CP [6, 14]. In this thesis we will focus on the portfolio-based Algorithm Selector SUNNY [10] which has been based on k-NN techniques and proved effective in recent MiniZinc Competitions [14], i.e., the yearly international competition for CP solvers. SUNNY is a per instance algorithm scheduling strategy based on k-NN algorithm. Roughly speaking, for each test instance SUNNY selects k training instances which are similar to the test instance in terms of Euclidean Distance (on instance features). Based on the selected instances, SUNNY generates a schedule of solvers that maximize the number of instances solved by the selected solvers. Then, a time slot proportional to the fraction of solved instances is assigned to each solver. Finally, the proposed solvers are ordered according to the average solving time on the selected instances. In 2015, SUNNY was compared with other solver selectors in the first ICON Challenge on algorithm selection with less satisfactory performance, in the Chapter 5, we will present the advancements made on SUNNY which finally achieved promising results in the 2017 OASC Challenge on algorithm selection.

Part I

Applications of Constraint Programming

Chapter 3

CP for (sub)group activity optimization

Humans are social animals and usually organize activities in groups. However, they are often willing to split temporarily a bigger group into subgroups to enhance their preferences. In this Chapter we present NightSplitter, an on-line tool that is able to plan movie and dinner activities for a group of users, possibly splitting them in subgroups to optimally satisfy their preferences. We first model and prove that this problem is NP-complete. We then use Constraint Programming (CP) or alternatively Simulated Annealing (SA) to solve it. Empirical results show the feasibility of the approach even for big cities where hundreds of users can select among hundreds of movies and thousand of restaurants.

Structure of this chapter. In Section 3.1 we introduce the problem. In Section 3.2 we describe `NightSplitter` from the user perspective. In Section 3.3 we first formalize the problem solved by `NightSplitter` proving its NP-hardness while in Section 3.4 we present how CP and SA techniques are used to solve it. Section 3.5 presents the experiment results that validate the use of `NightSplitter`. Related work and conclusions are in Section 3.6 and 3.7 respectively.

3.1 Problem Introduction

Nowadays, most of the city activities such as restaurants, cinemas, museums, theaters have complete and detailed information on web pages and offer a variety of online services and options for consulting programs, making reservations, buying tickets, etc. One of the main problems that the customer has to face in order to take advantage of this huge offer is to master the information overload which comes with it. For example, in Paris, our reference town for this work, there are more than 13500 restaurants and around 100 cinemas with 150 movies each night. Hence, the apparently simple task of organizing a night out with a movie followed by a dinner can already turn into a serious planning exercise.

When there are several persons involved, e.g., a family or a group of friends, with different ideas, preferences, and needs, coordinating the activities of the group becomes significantly more complex.

It is quite natural, in order to satisfy all the preferences of the members of a group, to take a pragmatic approach and split the group of persons into several sub-groups performing different activities, in order to enhance the individual satisfactions: some groups will watch the latest Hollywood blockbusters, while some others will prefer an Indie movie, provided, of course, this can take place approximately at the same time, and in the same movie theater, or in movie theaters not too far apart.

And that's not all: one needs to take into account both time constraints (e.g., we need to be home before midnight) and spatial constraints (e.g., we do not have the car and we do not want to walk for one hour). The planning of a night out can therefore easily become a daunting task.

Recommender systems and planners provide tools that can help users to manage these difficulties by filtering information, suggesting solutions, predicting some needs and planning the activities. However, most of the existing tools focus on a single user, so they cannot be used when several users interact and participate in a group activity [22, 43]. Tools considering group experiences exist [16, 24, 106] but they mainly focus on methods for aggregating preferences for a fixed group of users in order to optimize (some notions of) group satisfaction.

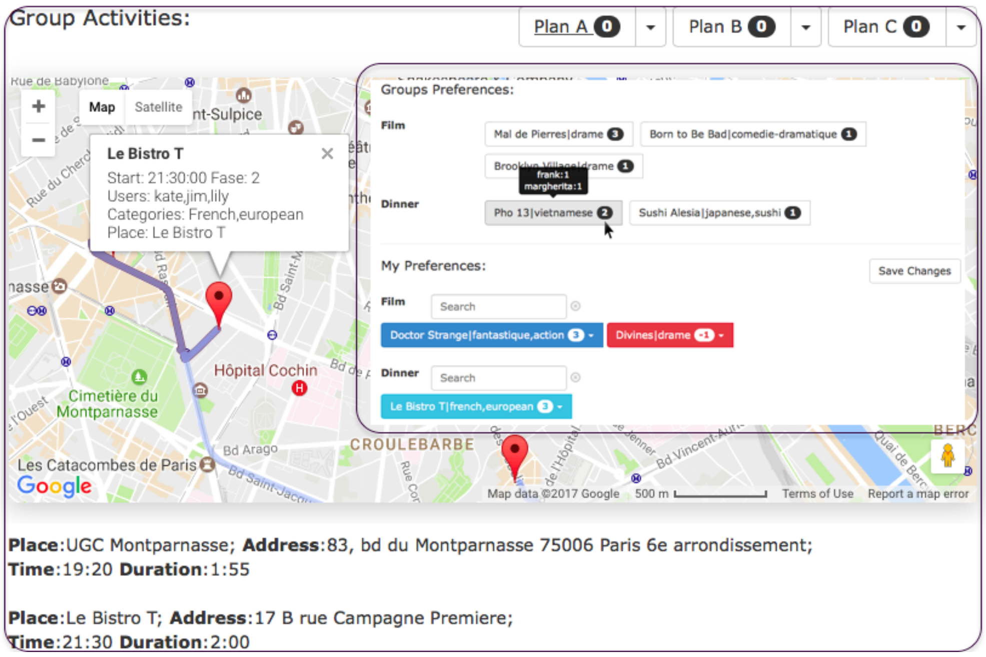
Only a few research papers [21, 92] consider the problem of sub-group formation and group splitting, but they do not take into account time and space constraints or they impose the same subgroups for all the activities, thus forbidding the most interesting cases, like a group that splits into subgroups to see different movies, but then joins at the same restaurant.

In this work we present `NightSplitter`, an on-line tool that is able to plan movie and dinner activities for a group of users, possibly splitting them in subgroups to optimally satisfy their preferences. We first model this problem and prove that it is NP-complete. We then use Constraint Programming (CP) or alternatively Simulated Annealing (SA) to solve it. Empirical results, obtained on real data for the city of Paris, show the feasibility and scalability of the approach even when hundred of users can select among hundreds of movies and thousand of restaurants.

It is worth noticing that even though, for the sake of clarity and concreteness, in this work we focus on the above mentioned activities, our approach is completely general and our tool can be easily adapted to any problem which has the following features: 1) there is a group of users who have to perform a sequence of n activities; 2) each user can express some preferences on these activities; 3) the group can be divided in several sub-groups, each one performing a different activity at a given time frame; 4) temporal and spacial constraints can be added on the different activities; 5) the aim of the tool is to optimize the overall satisfaction of all the users involved in the activities.

3.2 NightSplitter

`NightSplitter`, the tool we have developed and that we present in this Section, is a web application for planning movie and restaurant activities in the city of Paris. It may be used by a group of users and it can split them in subgroups to optimally satisfy their preferences. The application uses real data for (currently) 13598 restaurants and 93 cinemas with 153 movies, which are stored in a database and are constantly updated by a crawler embedded in the application. Using `NightSplitter`, an initial user dubbed *group initiator* can create a "group event" for a certain date. The group initiator is able to tune several parameters and constraints such as the number of possible subgroups, the size of subgroups, the total time window for performing the



Place:UGC Montparnasse; **Address**:83, bd du Montparnasse 75006 Paris 6e arrondissement;
Time:19:20 **Duration**:1:55

Place:Le Bistro T; **Address**:17 B rue Campagne Premiere;
Time:21:30 **Duration**:2:00

Figure 3.1: NightSplitter Screenshot.

activities, the maximal time one is forced to wait between the activities. The group initiator can then invite other members to participate to the group by sharing a reference link. The invited member, by clicking on the link, is included automatically into the group and will be able to express his/her preferences, possibly inviting other persons to join the group.

As can be seen from Fig. 3.1 showing a screenshot of `NightSplitter`, by using some simple menus each user can express preferences on movies and restaurants in Paris. Social interaction among group members is possible, since each user can see the preferences of others and can instantly see the results of updating or modifying his/her own preferences. The main interface is divided in two parts: a dashboard for preferences and a digital map for showing the solutions. In the preference dashboard (right side of Fig. 3.1), users can input their preferred movie and restaurant names (or alternatively movie and cuisine categories). The introduction of this information is facilitated by an autocomplete function that suggest possible values. The expressed preference is represented by a tag with color, where the tag shows the name of the preference and the color indicates its scale: deep blue to signal a strong like, light blue for like, yellow for dislike, red for strong dislike, and gray for neutral. On the top of the dashboard, there is a summary of the group preferences, where in each tag, next to the activity name, there is an aggregated score. Each time a user enters or modifies a preference, the preference dashboard will be updated in real time and the system will start to compute a new solution. [1] The computation, as later detailed in Section 3.4, uses either a Constraint Programming or Simulated Annealing technique. The averages of the individual preferences and the public ratings of the selected activities are weighted and combined to form a unique evaluation metric to establish the quality of every solution (cf. Definition 3.6). The 3 solutions with highest aggregated preference are provided and displayed on-the-fly to the users, both in textual form and on the digital map. The text informs the user about their tentative scheduled activities while the map provides a global view of the subgroups activities with their cinema-restaurant paths. Given the different solution plans, group members have the option to like or dislike them by clicking "Plan A/B/C" as shown in the upper part of Fig. 3.1. Based on these votes the group initiator can finalize the decision and pick up the plan for the entire group.

The online version of `NightSplitter` is available at [139]. [2]

3.3 NightSplit

In this section we formalize the definition of the optimization problem solved by `NightSplitter` and dubbed `NightSplit`. The key elements of `NightSplit` are the users and the activities that users can perform. We therefore assume the following finite disjoint sets: $\mathcal{U}$ for users range over by $u_1, u_2, \ldots$, $\mathcal{A}_M$ and $\mathcal{A}_R$ for the movie and restaurant activities respectively. We will denote with $\mathcal{A} = \mathcal{A}_M \cup \mathcal{A}_R$ a generic activity ranged over by $a_1, a_2, \ldots$.

[1]Currently preferences are visible to all the users. However, mechanisms to hide the individual preferences such as differential privacy [45] are under consideration.

[2]We are developing the tool for commercial use.

Activities have properties such as a possible starting time or the location where they are performed. The planning problem therefore needs to consider two dimensions: time and space. As far as the time is concerned, for `NightSplit` we consider only a fixed time window assuming that we want to plan all the activities within a given time range. In particular, for simplicity we use a discrete notion of time dividing the time window in time slots of fixed duration. Similarly, we discretize also the space by dividing it into a finite number of different locations. The granularity of the time and the space can be arbitrarily improved by reducing the duration of the time slot or considering smaller locations. In the following we denote with $TIME = \{1, \ldots, T_{max}\}$ and $Loc = \{1, \ldots, Loc_{max}\}$ the time slots and the locations where T_{max} and Loc_{max} are the number of time slots and the number of locations. In our examples, we consider 5 min as the time slot unit. We can therefore define the general properties of an activity as follows.

Definition 3.1 (Activity Proprieties) *Given a set of activities $\mathcal{A}$ we denote with:*

- `startTime` *the total function $\mathcal{A} \to TIME$ that associates to an activity its starting time slot (i.e., when the movie starts or when the restaurant opens),*
- `endTime` *the total function $\mathcal{A} \to TIME$ that associates to an activity its finishing time slot (i.e., when the movie ends or when the restaurant closes),*
- `duration` *the total function $\mathcal{A} \to TIME$ that associates to an activity the user's duration in time slots.*
- `area` *the total function $\mathcal{A} \to Loc$ that associates to an activity the location where it takes place.*
- `publicRating` *a complete function $\mathcal{A} \to \mathbb{N}$ that associates to an activity a possible rating.* [3] *Ratings are represented with natural numbers: the bigger the rating, the better the activity is considered.*

[3]Specifically, the rating value of activity ranges from 0 to 5, where 0 means "no rating information is given".

With a slight abuse of notation, given an activity a and a property p we denote with $a.p$ (rather than with p(a)) the value of the propriety p for activity a.

Example 3.1 *A* restaurant activity $a \in \mathcal{A}_r$ might be characterized by $a.$`startTime` $=$ 228, meaning that the restaurant opens at 19:00 (assuming a time slot of 5 minutes 228 corresponds to 19), $a.$`endTime` $= 276$, meaning that the restaurant closes at 23:00, $a.$`duration` $= 18$ meaning that the dinner will last 90 minutes, $a.$`area` $= 5$ meaning that the location is identified with id 5, and $a.$`publicRating = 3` meaning that the public rating is 3. □

As far as preferences are concerned, based on findings such as those reported in [120], we avoid using a very refined scale and we allow only 5 values: from -2 indicating a strong dislike to a +2 indicating a strong preference, and 0 indicating a neutral opinion. Formally user preferences are defined as follows.

Definition 3.2 (Activity Preferences) *Given a set of users $\mathcal{U}$ and a set of activities $\mathcal{A}$, an activity preference is a total function* `pref` $: \mathcal{U} \times \mathcal{A} \to \{-2, -1, 0, 1, 2\}$.

Since the user has to move between different locations, to properly define a valid plan we need a metric that evaluates the distance between different activities. We are only interested in the time to go from one activity to another. Hence, we abstract from physical details such as GPS coordinates and means of transportation and we simply consider a distance metric between locations which is given in terms of times slots (needed to go from one location to the other).

Definition 3.3 (Distance metric) *Given a set of locations Loc and a set of time slots $TIME = \{1, \ldots, T_{max}\}$ a distance metric is a total function* `dist` $: Loc \times Loc \to TIME$.

We are now ready to define what is a plan: a simple association of activities to the users.

Definition 3.4 (Plan) *Let us consider a set of users $\mathcal{U}$, two sets of activities $\mathcal{A}_M$ and $\mathcal{A}_R$ and a set of time slots $TIME$. A plan is a total function* $\texttt{plan} : \mathcal{U} \to (\mathcal{A}_M \times TIME) \times (\mathcal{A}_R \times TIME)$ *that associates to a user a movie and restaurant activity with their beginning time slots.*

Example 3.2 *A* plan $\texttt{plan}(u) = ((a_1, 108), (a_2, 138))$ means that to the user u is assigned the activity a_1 that starts at 9:00 and the activity a_2 at 11:30. □

Not all the plans are valid: For instance a plan may schedule two overlapping activities for a user. For this reason, we introduce the notion of plan validity that captures the constraints that a feasible plan must possess.

Definition 3.5 (Plan Validity) *Given a positive integer* maxGroupNum *representing the maximal number of sub-groups allowed, a positive integer* minCardinality *representing the minimal size of a group, and a positive integer* $\texttt{maxWait} \in TIME$ *representing the maximal waiting time between two activities, a plan* plan *is said valid iff:*

- *starting and ending time are satisfied. Formally, for each user $u \in \mathcal{U}$, if* $\texttt{plan}(u) = ((a_m, t_m), (a_r, t_r))$ *then* $\texttt{startTime}(a_m) \leq t_m \leq \texttt{endTime}(a_m) - \texttt{duration}(a_m)$ *and* $\texttt{startTime}(a_r) \leq t_r \leq \texttt{endTime}(a_r) - \texttt{duration}(a_r)$*;*

- *activities do not overlap. Formally, $\forall u \in \mathcal{U}$, if* $\texttt{plan}(u) = ((a_m, t_m), (a_r, t_r))$ *then* $t_r \geq t_m + \texttt{duration}(a_m) + \texttt{dist}(\texttt{area}(a_m), \texttt{area}(a_r))$*;*

- *activities are not too far apart. Formally, $\forall u \in \mathcal{U}$, if* $\texttt{plan}(u) = ((a_m, t_m), (a_r, t_r))$ *then* $t_r \leq t_m + \texttt{duration}(a_m) + \texttt{maxWait}$*;*

- *the number of groups is limited by* maxGroupNum*. Formally,* $|\{(a_m, t_m) \mid \forall u \in \mathcal{U} \,.\, \texttt{plan}(u) = ((a_m, t_m), (a_r, t_r))\}| \leq \texttt{maxGroupNum}$ *and* $|\{(a_r, t_r) \mid \forall u \in \mathcal{U} \,.\, \texttt{plan}(u) = ((a_m, t_m), (a_r, t_r))\}| \leq \texttt{maxGroupNum}$ *;*

- *the cardinality of the group is bounded by* minCardinality*. Formally, for all activities $a_m \in \mathcal{A}_m$, and time slots $t_m \in Time$* $|\{u \mid \forall u \in \mathcal{U} \,.\, \texttt{plan}(u) = ((a_m, t_m), (a_r, t_r))\}|$ *is 0 or greater or equal than* minCardinality*. Similarly,* *for all activities* $a_r \in \mathcal{A}_R$*, and time slots* $t_r, \in Time$ $|\{u \mid \forall u \in \mathcal{U} \; . \; \mathtt{plan}(u) = ((a_m, t_m), (a_r, t_r))| $ *is 0 or greater or equal than* `minCardinality`*.*

In order to simplify the presentation, given a plan $\mathtt{plan}(u) = ((a_1, t_1), (a_2, t_2))$ in the following we will use $\mathtt{plan}(u).a_m$ for denoting a_1, $\mathtt{plan}(u).a_r$ for a_2, $\mathtt{plan}(u).t_m$ for t_1, and $\mathtt{plan}(u).t_r$ for t_2 (m stands for movie, r for restaurant).

We are now ready to define the `NightSplit` optimization problem. Intuitively, the `NightSplit` goal is to find a valid plan that optimizes the individual activity preferences and the public activity preferences. Different criteria may be used to combine these preferences. `NightSplit` allows a great flexibility combining all these objectives into one by summing them according to some weights.

Definition 3.6 (`NightSplit`) *Let* η *be a real number* $\in [0,1]$ *representing the weight associated to the individual activity preferences and the public preferences.* [4] *The* `NightSplit` *problem is to find the valid plan* $\mathtt{plan}^*$ *that maximizes the following objective function.*

$$\mathtt{obj}(\mathtt{plan}) = \eta \cdot \mathsf{sum}_{act}(\mathtt{plan}) + (1-\eta) \cdot \mathsf{sum}_{pub}(\mathtt{plan}) \tag{3.1}$$

where sum_{act} *and* sum_{pub} *are the sum of the individual activities preferences and public preferences as define below:*

$$\mathsf{sum}_{act}(\mathtt{plan}) = \sum_{u \in \mathcal{U}} (\mathtt{pref}(u, \mathtt{plan}(u).a_m) + \mathtt{pref}(u, \mathtt{plan}(u).a_r)) \tag{3.2}$$

$$\mathsf{sum}_{pub}(\mathtt{plan}) = \sum_{u \in \mathcal{U}} (\mathtt{plan}(u).a_m.\mathtt{publicRating} + \; \mathtt{plan}(u).a_r.\mathtt{publicRating}) \tag{3.3}$$

As can be expected, even tough this formulation is rather simple, `NightSplit` is an NP-hard problem.

Theorem 3.1 (NP-hardness) *The* `NightSplit` *is NP-hard.*

[4] Public preferences are useful to break the ties when users have very general individual preferences (e.g., I like all the movies)

Proof: To prove hardness, we reduce the NP-complete problem Perfect Expected Component Sum (PECS) [21] to the decision version of `NightSplit`, i.e., the problem to find whether there exists a valid plan such that the objective function `obj` is greater or equal than a given value. [5] An instance of PECS consists of a collection V of m-dimensional boolean vectors, i.e., $V \subset \{0,1\}^m$ and a number k. The problem is to determine whether there exists a disjoint partition of V into k subsets $V_1, \ldots, V_k$ such that $\sum_{i=1}^{k} \max_{1 \leq j \leq m}(\sum_{\bar{v} \in V_i} \bar{v}|j|) = |V|$.

Given an instance of PECS we map every vector $\bar{v}_i \in V$ as a user u_i having some preferences over m different movies. The intuition behind the hardness proof is to exploit the planning of the movie activities to find a solution for PECS. We assume that there is only one location, that the m movie activities start at the time slot 0 and end at time slot 1 with duration 1. Similarly, we assume that there are m different restaurant activities that start at time slot 1 and end at time slot 2 with duration 1. We set `maxGroupNum` to k, `minCardinality` to 1, `maxWait` to 1, and we assume that the function `dist` is the constant function 0. In this way all the movie activities are compatible with the restaurant activities and all the possible plans that have a maximal number of k groups are valid. We set the preferences of the movie activities to reflect the values of the vector $\bar{v}$. Formally, for all $1 \leq i \leq |V|$ and $1 \leq j \leq m$ we define $\texttt{pref}(u_i, a_j) = \bar{v}|j|$. We set to 0 instead the preferences for all the restaurant. We set the weight of the user preferences η to 1 while we discard the public preferences with $1 - \eta = 0$.

Based on the definition of `NightSplit`, it is easy to see that $\sum_{i=1}^{k} \max_{1 \leq j \leq m}$ $(\sum_{\bar{v} \in V_i} \bar{v}|j|) = |V|$ iff the `obj` of the `NightSplit` problem is equal to $|V|$. The partition induced on the users performed by `NightSplit` corresponds to the partition of V into the k set of vectors $V_1, \ldots, V_k$. □

[5] The decision version of the problem requires the "greater or equal" operator. Similar to the theorem presented in [21], our theorem holds because the sum of the preferences is never greater than V.

3.3.1 Useful Extensions

While `NightSplit` is already NP-hard, there are some useful extensions of it that do not alter its complexity class and its nature. In the following we just comment on some of them that are considered in the online `NightSplitter`.

First observe that the notion of a valid plan can be further restricted considering additional constraints. For example, it may be useful to allow users to indicate that they are not available before or after a given time. Moreover, the minimal number of people required to form a group or the number of groups can vary depending on the activity (e.g., it may be the case that for going to the movie we accept to split the group in two while to eat in a restaurant we do not allow any split). Other useful extensions concern the definition of different kinds of user preferences. For instance, usually users like to hang out in certain locations and they want to minimize the traveling time between the activities, minimize the waiting time, start the activities as soon as possible, etc. All these preferences may be considered by adding further terms to the objective function that we optimize in `NightSplit`, possibly reducing its weight by an appropriate parameter. `NightSplitter` has been designed to be easily extensible and take into account new sources of user preferences or constraints. For instance, the preferences over some areas can can be easily defined in the profile menu of the user and then taken into account when generating the plans.

Finally, we could also relax the limit of two activities, considered in this work, and we could extend our system to applications where more activities can be performed in sequence, especially in the tourism industry, following, e.g., [129, 92].

3.4 Solution Approaches

To solve the `NightSplit` problem we propose two different approaches. The first one relies on Constraint Programming (CP) and allows us, in principle, to obtain the optimal solution. The second approach uses Simulated Annealing (SA), a probabilistic local search procedure which, under certain conditions for its parameters, is known to find the optimal solution with a probability approaching one. In this section we briefly describe the CP and SA approaches, while we defer to Section 3.5 for their comparison.

3.4.1 NightSplit and Constraint Programming

Constraint Programming (CP) [125, 132] is a widely adopted approach for solving NP-hard problems. The CP paradigm enables to express complex relations in form of constraints to be satisfied. In particular a Constraint Satisfaction Problem (CSP) $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ consists of a finite set of variables $\mathcal{X}$, each of which associated with a domain $D_x \in \mathcal{D}$ of possible values that it could take, and a set of constraints $\mathcal{C}$ that defines all the admissible assignments of values to the variables [101]. Given a CSP the goal is normally to find a solution, i.e, an assignment to the variables that satisfies all the constraints of the problem. When an objective function needs to be minimized or maximized we deal instead with a Constraint Optimization Problems (COPs), i.e., a generalized CSP where the goal is not only to find a solution but among all possible solutions the one that maximizes or minimizes the objective function.

Clearly the `NightSplit` problem can be seen as a COP. For every user u we have introduced:

- a variable M_u representing the selection of the movie activity. The domain of this value is the finite domain of all the possible movie activities;
- a variable R_u representing the selection of the restaurant. The domain of this value is the finite domain of all the possible restaurant activities;
- two variables $S_{u,1}, S_{u,2}$ representing the beginning of the activities. The domain of these variables is the finite set of the possible time slots;
- two variables $G_{u,1}, G_{u,2}$ representing the subgroup to which user u belongs (for the first and second activity respectively). The domain of these variables depend on the maximal number of groups allowed for activity.

With these variables it is possible to state all the constraints as listed in Definition 3.5. For instance, the first constraint bounding the starting time of the activities might be expressed by stating that movie_start$[M_u] \leq S_{u,1}$ where movie_start is the array storing the movies starting time. This constraint is simply a disequality between two expressions: the first retrieves the concrete value from an array while the second is the variable $S_{u,1}$. Note that CP solvers can employ efficient techniques to handle this kind of equalities or disequalities (global constraints). Moreover, for this particular case, the constraint setting x as the value taken by the y-th value of the array is known as element constraint [125], which is often supported by constraint solvers that adopt ad-hoc propagation algorithms to speed up the search of solutions.

To model all the constraints we used MiniZinc [113], which is the de-facto language to define CSPs and COPs and is supported by a huge variety of constraint solvers. Since the majority of the solvers does not support real variables, we restrict the use of the preference weights η to rational numbers only. A complete explanation of the the MiniZinc model and all the constraints defined is outside the scope of this work. For more information we invite the reader to consult [80].

The CP model is composed by a set of variables, a set of pseudo-Boolean constraints and an objective function. The variable assignment that maximizes the objective function corresponds to the best solution. Let $\mathcal{G} = g_1, g_2, ..., g_{\mathsf{maxGroupNum}}$ be a set of groups that partitions the users. In our model, we adopt array of variables that associates a property to each user (properties like, group, activity, travel distance etc), and we use a set of them for movie activity and a set for restaurant activity. And we denote with variable_name1 and variable_name2 respectively. If not mentioned explicitly, we use variable_name to represent both variable_name1 and variable_name2.

- user_group_map$_u$ for $u \in \mathcal{U}$, user_group_map$_u \in \mathcal{G}$, such variable array maps each user to a group.

- user_act_map$_u$ for $u \in \mathcal{U}$, user_act_map$_u \in \mathcal{A}_c$, maps each user to an activity of movie and restaurant.

- $\texttt{user\_start\_time\_map}_u$ for $u \in \mathcal{U}$, $\texttt{user\_start\_time\_map}_u \in TIME$, represents the schedule activity start time for each user.
- $\texttt{user\_duration\_map}_u$ for $u \in \mathcal{U}$, $\texttt{user\_duration\_map}_u \in TIME$, activity duration of each user
- $\texttt{user\_begin\_map}_u$ for $u \in \mathcal{U}$, $\texttt{user\_begin\_map}_u \in TIME$, it represents the official start time of assigned activity for each user.
- $\texttt{user\_end\_map}_u$ for $u \in \mathcal{U}$, $\texttt{user\_end\_map}_u \in TIME$, for each user, it associates the official end time for the activity user assigned.
- $\texttt{user\_location\_map}_u$ for $u \in \mathcal{U}$, $\texttt{user\_location\_map}_u \in Loc$, stands for user's location for activity
- $\texttt{user\_pub\_rating\_map}_u$ for $u \in \mathcal{U}$, $\texttt{user\_pub\_rating\_map}_u \in \{0, ..., 5\}$, the activity's public rating which assigned to the user.
- $\texttt{user\_preference\_map}_u$ for $u \in \mathcal{U}$, $\texttt{user\_preference\_map}_u \in \{-2, ..., 2\}$, indicates each user's preference for her given activity.
- $\texttt{user\_distance\_map}_u$ for $u \in \mathcal{U}$, $\texttt{user\_distance\_map}_u \in TIME$, reflects each user's travel from movie activity to restaurant activity. Not like other variables which are valid for both 2 activities, this one is unique.

The model's constraints are listed below:

$$\begin{split}\max \sum_{u \in \mathcal{U}} \eta \texttt{user\_preference\_map}_u \\ + (1-\eta)\texttt{user\_pub\_rating\_map}_u\end{split} \tag{3.4}$$

$$\begin{split}\forall u \in \mathcal{U}, \{&\texttt{user\_act\_map}_u, \\ &\texttt{user\_begin\_map}_u, \texttt{user\_duration\_map}_u, \\ &\texttt{user\_end\_map}_u, \texttt{user\_location\_map}_u, \\ &\texttt{user\_pub\_rating\_map}_u\} \in Activities\end{split} \tag{3.5}$$

$$\begin{aligned}\forall u \in \mathcal{U}, \{&\texttt{user\_act\_map}_u, \\ &\texttt{user\_preference\_map}_u\} \in Preferences\end{aligned} \tag{3.6}$$

$$\begin{aligned}\forall u \in \mathcal{U}, \{&\texttt{user\_location\_map1}_u, \\ &\texttt{user\_location\_map2}_u, \\ &\texttt{user\_distance\_map}_u\} \in Locations\end{aligned} \tag{3.7}$$

$$\begin{aligned}\forall u \in \mathcal{U}, &\texttt{user\_start\_map}_u \geq \texttt{user\_begin\_map}_u \\ &\wedge \texttt{user\_start\_map}_u \leq \texttt{user\_end\_map}_u \\ &\qquad - \texttt{user\_duration\_map}_u\end{aligned} \tag{3.8}$$

$$\begin{aligned}\forall u \in \mathcal{U}, &\texttt{user\_start\_map2}_u \\ &\geq (\texttt{user\_start\_map1}_u + \texttt{user\_duration\_map1}_u \\ &+ \texttt{user\_distance\_map}_u) \wedge \texttt{user\_start\_map2}_u \\ &\leq \texttt{user\_start\_map1}_u + \texttt{user\_duration\_map1}_u \\ &+ maxWait)\end{aligned} \tag{3.9}$$

$$\begin{aligned}\forall g \in \mathcal{G} |\{&\texttt{user\_group\_map}_u | u \in \mathcal{U} \\ &\wedge g \in \texttt{user\_group\_map}_u\}| > \texttt{minCardinality}\end{aligned} \tag{3.10}$$

$$\texttt{user\_group\_map}_{u_1} = g_1 \tag{3.11}$$

$$\begin{aligned}&\texttt{user\_group\_map}_{u_2} = g_1 \\ \vee &\texttt{user\_group\_map}_{u_2} = g_2\end{aligned} \tag{3.12}$$

Constraint 3.4 is the objective function to optimize which measures the users satisfaction. Constraints 3.5, 3.6, 3.7 ensure that the variable domains correspond to the input data. Constraint 3.9 regulates the user's start time in the schedule, **??** ensures the schedule's temporal validity. Constraint 3.10 guarantees the number of users in each subgroup is not less than the `minCardinality`. 3.11 and 3.12 are used for symmetry breaking where, they ensure that the first user stays in the first group and the second user stay either in the first or the second group. [6] This reduces the search domain and the resulting solution still belongs to the optimal solutions.

According to our encoding the number of constraint added is linear w.r.t. the number of users.

Remark 1 *Beside CP, we have also tried to encode the* `NightSplit` *to exploit Satisfiability-Modulo-Theories (SMT) solvers. SMT solving extends and improves upon SAT solving by introducing the possibility of stating constraints in some expressive theories, e.g., arithmetic or bit-vector expressions. While all the constraints of* `NightSplit` *can be encoded in SMT, we were not able to provide an encoding linear w.r.t. the number of activity locations. Indeed, differently to what happens in CP where the* element *constraint can be used [125], in the SMT case the encoding of the traveling time between two activities requires the introduction of a quadratic number of constraints w.r.t. the number of locations. Based on our test, since we had more than 300 locations, the addition of these quadratic number of constraints hindered the use of SMT solvers. For this reason, in Section 3.5, we will compare only the performances of the CP and SA approaches.*

3.4.2 NightSplit and Simulated Annealing

Simulated Annealing (SA) [1] is a local search technique inspired by the annealing process in metallurgy. SA has been widely used for approximating the global optimum of a given function. Given an initial solution, random moves are made to produce new potential solutions. A new solution that improves the previous one is (usually) always accepted. Solutions that worsen the current solution are instead accepted with a probability that, like the temperature in the annealing process, is gradually decreasing. Accepting worse solutions is a fundamental property because it allows for a more extensive search for the optimal solution, possibly avoiding getting stuck in local optima.

[6] We note that a stronger but more sophisticated constraint for symmetry breaking could be the value_precede_chain where all the symmetries in group names would be eliminated.

Contrary to the CP technique described before, SA can not guarantee that the final solution obtained is optimal. However, for discrete and large search spaces, SA scales better and could produce (sub)optimal solution very quickly.

Among all the different implementations of SA available we rely on the re-implementation in PHP of the python SA module [116]. After some manual tuning, we have fixed the parameters to control the decreasing of the temperature and the number of iterations (50000). The temperature exponentially decreases as the algorithm progresses. As customary, a move causing a decrease in state energy (i.e., an improvement of the `NightSplit` objective function) was always accepted. Moves instead increasing the state energy (i.e., a worse solution) but within the bounds of the temperature are also accepted.

The initial solution is obtained by randomly generating the assignments from users to activities. To obtain instead a valid plan from a current solution we proceed the move method as follows: (i) we randomly select movie activity assignments or restaurants activity assignments and modify them; (ii) we randomly select a subset of users U; (iii) we assign a new activity a to the selected users in U. This activity is randomly chosen among all the activities for which the aggregated preference of the U users is positive. Intuitively, this avoids selecting an activity that no user in U wants to perform; (iv) if the assigned activity is not compatible with other existing ones (e.g., if for user u we select a movie activity a that overlaps with his/her restaurant activity) we delete the old activities; (v) for every user u that has no activity assigned we look at the activities assigned to other users, check if any of them is compatible with the updated activity and if so we assign this activity to the user u assuming that this does not violate the group constraints. To have a unified picture, the SA pseudo-code has been attached as Appendix A.1.

3.5 Empirical Experiment

In this section we describe the experiments performed in order to validate the scalability of `NightSplitter` and we discuss the results.

We have considered for the experiments real data from the city of Paris: The movies information - for 93 cinemas and currently 153 different movies (with 1950 projections a day) - is retrieved from Allociné [2, 56], restaurant data - for 13598 restaurants - from TripAdvisor [141]. OpenStreetMap and GoogleMaps were also used to identified 317 positions of metro stations: for each activity we considered its nearest metro station as its location. [7]

Our Activity data are structured to contain the following three fields:

- Name (of a movie or a restaurant);
- Category (indicates the kind of movie and the type of cuisine);
- Intervals (indicate the time frame in which the activity is available).

Note that if an activity with the same name has two separate intervals (e.g. a restaurant is open from 11:00 to 15:00 and from 19:00 to 23:00) we consider two separate activities in our data. This means that there may exists in the data several activities with the same name.

The data related to the preferences was collected from Movielens [71] and Yelp [151]. These datasets, originally defined for activities in the U.S., were converted for Paris activities. This was done by mapping the names of the Paris activities to the activity existing in the preference dataset while preserving the activity category and the public rating. After that, we randomly sampled 8,000 users for the restaurant activity and 5.300 users for the movies activity to use their individual preferences for the experiments. The statistics related to the activities and preference data are summarized in Table 3.1 where the last column indicates the average preferences of the users. Note that if a restaurant was open for two separate intervals (e.g., from 11 to 15 and from 19 to 23) this was captured by considering two separate activities.

Since the goal is to provide a responsive tool, for the experiments we fixed a timeout of 60 seconds taking the best solution found by the tested approach within

[7] Alternatively, we can use the actual location and store the effective travel time between any pair of activities. However, the amount of activities that we considered will generate millions of records; this exceeds our experiment resources.

Activity Type	Activities	Users	Avg. pref
Movies	1950	5300	6
Restaurants	17069	8000	2

Table 3.1: Summary Statistics of the Dataset.

this time frame. For each testing scenario we repeated the experiment 30 times. For every experiment we match the chosen number of user with random user from the dataset using their preferences. We allow the subgroups to be formed by at least 2 people, the time slot unit to be 5 minutes assuming that the duration for a dinner/lunch is 90 minutes. The CP model is encoded in MiniZinc which is then translated - with different instance data - to independent fzn files. Then each fzn file is delivered to the CP solver. The SA algorithm is implemented by using PHP5. The experiments were run on an Ubuntu Intel Core 3.30GHz machine with 8 GB of RAM.

We compared the performance of three different state-of-the-art CP solvers, namely Chuffed [34], Or-Tools [62], and HCSP [79], [8] and the SA method described in the previous section.

We first compare the three different CP solvers for different number of users, assuming to have only 2 subgroups and not taking into account the public ratings (i.e., $\eta = 1$). Fig. 3.2 shows the average times needed by the solvers to find the optimal value by varying the number of users, where the filled icons mean that the solver has proven the optimality of the solution for all the 30 repeated tests. Chuffed has always computed the optimal solution for values up to 9 users and it is the fastest among the three solvers. The Or-Tools cannot find the optimal solution within the timeout for more than 5 users, while the HCSP solver performs slightly better than

[8]We selected these solvers based on the recent results of the MiniZinc Challenge 2016 [137]. In particular Or-Tools won a golden medal in the Fixed category and HCSP won a golden medal in Free and Parallel category. Chuffed was the second best solver of the entire Challenge after LCG-Glucose-free which is not publicly available. We would remark also that our problem instances have been submitted to the incoming MiniZinc Challenge 2017 [138].

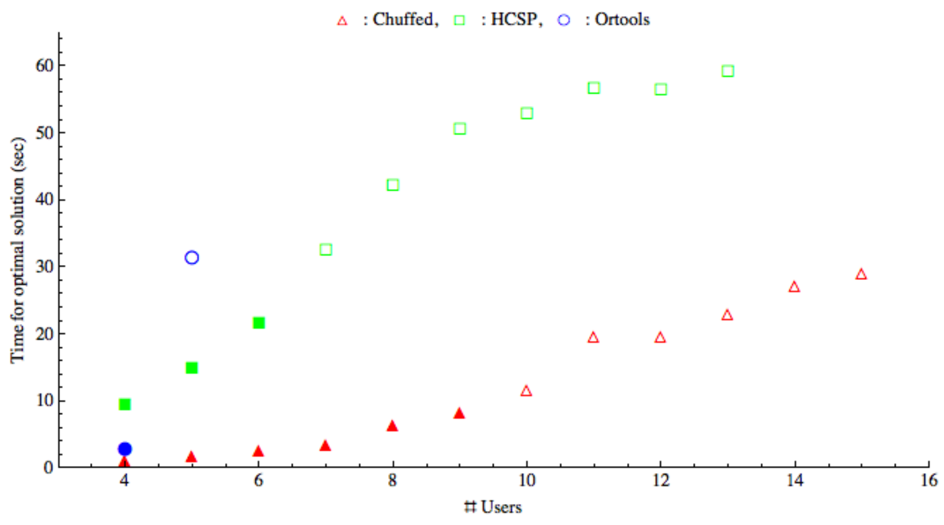


Figure 3.2: CP Solvers comparison.

Or-Tools and occasionally it is still capable to prove optimal solution for up to 13 users. Similar results are obtained when increasing the number of subgroups or when public ratings are taken into account by lowering the value of the η parameter. Since Chuffed outperforms the other solvers in our application, in the following we show only the performance of this solver for the comparison with SA approach.

We compare the performance of Chuffed and SA in terms of quality of the solution for a number of users ranging between 4 and 40, assuming 2 subgroups could be formed, and the weight associated to the individual preference η to be 1. (i.e., public rating were not taken into account). In this test we limit the number of subgroups to 2 since we believe that especially for small groups users would not like to be split in many subgroups. Fig. 3.3 and 3.4 depicts respectively the average solution score and the average time needed to find the best solution for the 30 repeated tests (the green dot in Fig. 3.3 representing the number of tests such that CP proves solution optimality). The plots show that for a limited number of users SA is competitive with Chuffed, while for more than 15 users SA is definitely better. The advantage of the CP solution is that for less than 10 users the solutions are proven optimal while some SA solutions were suboptimal. From the plot it is however possible to see that the number of solutions that could be proven optimal in less than 60 second

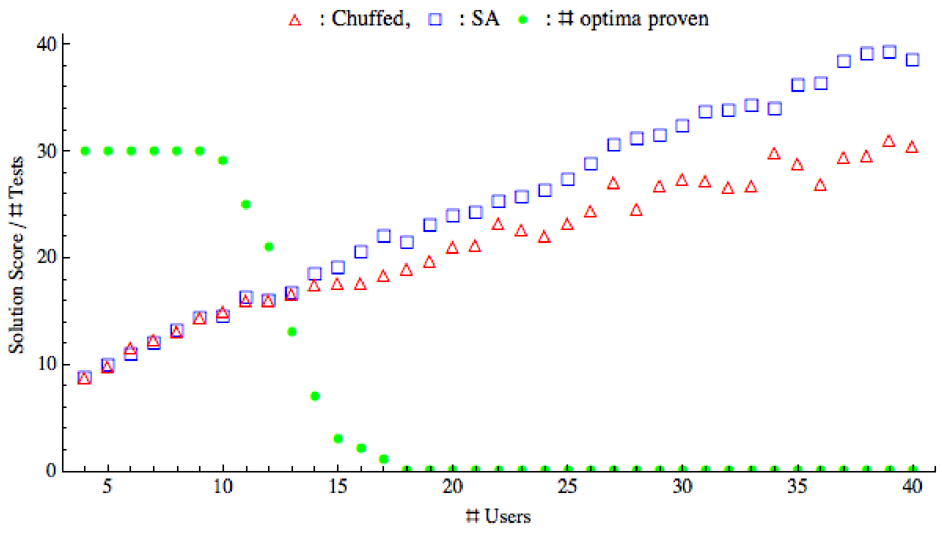


Figure 3.3: CP vs SA comparison.

decreases at the increase of the number of users. With more than 20 users no solution was proven optimal. It is clearly visible that Chuffed is better only for a limited number of users while the SA is often able to find the best solution within the first 15 seconds.

We then compare the two approaches by varying the number of possible subgroups from 1 to 8. In Fig. 3.5 we present the plots obtained considering 32, 64, and 128 users. From the plots it can be seen that the CP technique is only suitable with few users and when no more than 2 subgroups can be formed. When the number of users increases or more than 2 groups can be formed the solutions provided by the CP solver within 60 seconds are worse than the ones produced by the SA. In our biggest scenario, considering 128 users, the SA is the only viable choice because unfortunately the CP solver is not even able to provide a single solution (hence the lack of points for Chuffed in Fig. 3.5(c)). We conduct experiments also varying the weights used to aggregate the individual and public preferences. In these cases there are no significant changes, except that the final score increases.

Fig. 3.6 shows for instance the performances of Chuffed and SA while varying the parameter η considering 32 users and 2 subgroups. In particular, Fig. 3.6(a) presents the average time when the best value is found while Fig. 3.6(b) presents the average

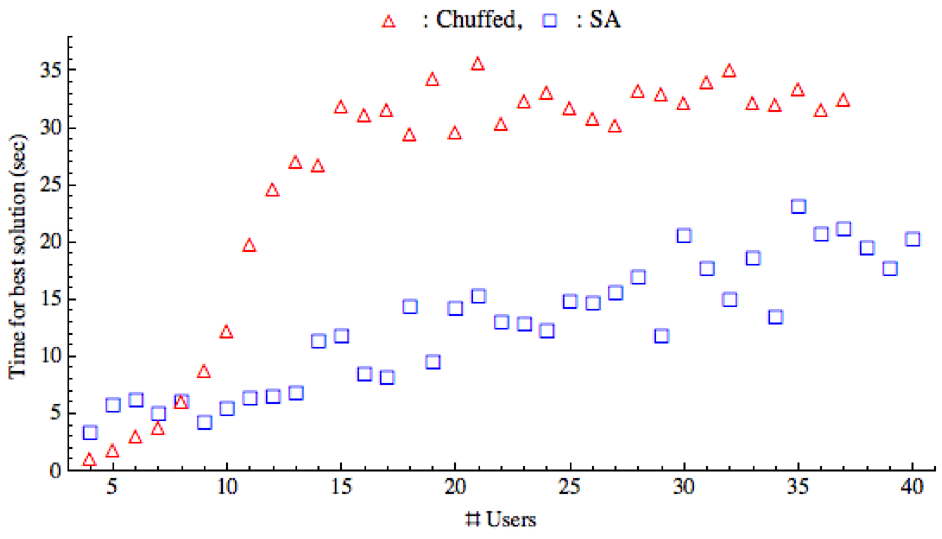
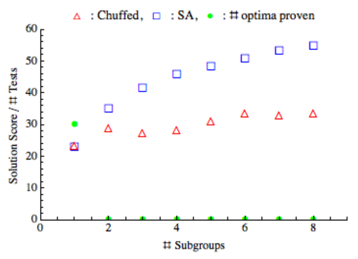
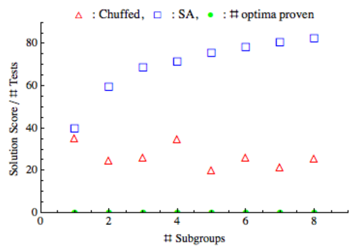


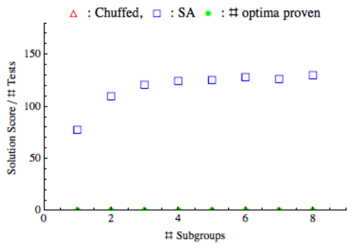
Figure 3.4: Time to find the best solution.



(a) 32 users



(b) 64 users

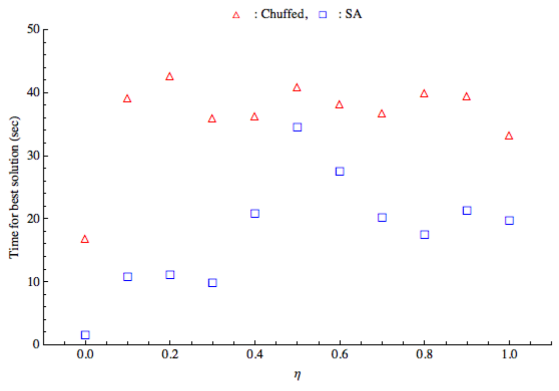


(c) 128 users

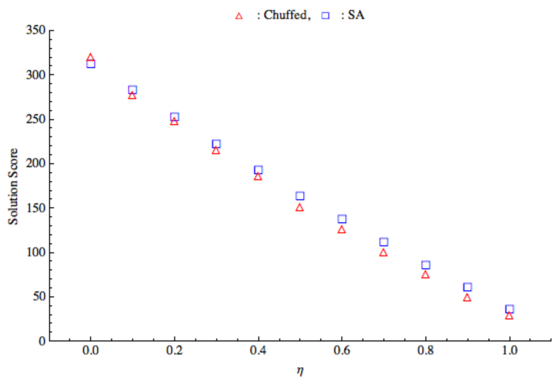
Figure 3.5: Comparison of CP and SA varying the number of subgroups.

score found after 60 seconds. As long as the user's preferences are accounted for (i.e., $\eta \neq 0$), it is immediately visible that with this amount of users the SA approach is better than Chuffed since SA is able to find better values in a short amount of time and Chuffed is not able to prove the optimality of the solutions within 60 seconds.

Summarizing, we may conclude that when considering two subgroups and few users the CP approach may be useful and even prove the optimality of the solution. For more subgroups and more users the SA approach is better. For those experiments where the optimality of the solutions was proven, the SA approach was able to propose competitive solutions. We conjecture that this holds also for big instances where we were not being able to prove the optima.



(a) Time to find the best solution



(b) Average score of best solutions

Figure 3.6: Comparison of CP and SA varying η.

3.5.1 `NightSplit` in MiniZinc Challenge 2017

To explore the existence of better solvers than those we have examined so far, we submitted five problem instances to MiniZinc Challenge 2017. In this challenge, there are 23 constraint solvers in total that are implemented by more than 15 teams worldwide. The submitted instances have fixed parameter values except for the users' preferences, which are randomly generated (as we did in the conducted experiments), and the different combinations of the parameters: the number of users and the number of subgroups. Tab. 3.2 provides a detailed overview of these instances.

	Instance 1	Instance 2	Instance 3	Instance 4	Instance 5
Number of users	5	6	12	12	15
Number of subgroups	1	3	1	1	3

Table 3.2: Summary of `NightSplit` instances in the MiniZinc Challenge 2017.

The overall results of the MiniZinc Challenge have confirmed that, in general cases, Chuffed is still the best solver among all. However, there are also some exceptions. In instances whose number of subgroups is bigger than one, the Or-Tools is a little bit faster, while in those whose number of subgroups is one, the solver Choco4 sometimes is more competitive than Chuffed. On the whole, in all of the instances that are solved within the timeout (20 minutes), the three solvers Chuffed, Or-Tools, and Choco4 have outperformed all others. For more challenge details, we refer the interested readers to the Challenge website [39].

3.6 Related work

The literature on recommender or planning systems is very large and we omit all the references to works which consider the case of a single user only, with the exception of [129], which uses CSP techniques for building a tourist recommendation and planning application. Concerning group recommender systems, [24] provide a survey on several existing approaches while [54] presents a recommender system for tourism based on the tastes of the users, their demographic classification and the places they have visited in former trips. More recently, the idea of group splitting has appeared in some papers. Notably [21] proposes an approach for forming groups of users in order to maximize satisfaction. The work [92] introduces the problem of group tour recommendation which includes the problem of forming tour groups whose members have similar interests. Differently from our case, all the above mentioned papers consider groups or sub-groups as fixed entities, which once are created cannot be modified. With our approach, instead, for each activity we have a different group formation, that is, we can have two users who are in the same group for the first activity (the movie) and are in different groups for the second one (the dinner). Moreover, the above papers focus on the theoretical aspects rather than presenting a tool.

There exist also several works which address the problem of group preference modeling and the definition of an appropriate notion of "group satisfaction" [106, 85]. In general these are difficult tasks, since it is hard to find a definition which takes into account all the various aspects involved in the group dynamics.

An interesting approach is presented in [16], where the notion of disagreement between group members is formally defined and, on its basis, a consensus function is introduced in order to formally define a satisfactory semantics for group recommendation. In some cases, users preferences depend on the contextual information in a dynamic domain, thus making even more difficult to make recommendation for groups. Recently Context-Aware Recommender Systems [83] have been proposed in order to address this issue. All the above mentioned approaches to the modeling of preferences, while interesting and relevant, are somehow orthogonal to the problem that we are considering in our work. Indeed, we could easily change the preference model without major changes in our tool.

To conclude we would like to mention also the works conducted in [43, 22, 129] which present recommendation and planning systems targeting a single user only but are interesting for us since they consider models of generating itineraries (for touristic applications) which could be integrated with our tool.

3.7 Summary and future prospectives

We have presented `NightSplitter`, an on-line tool that is able to plan movie and dinner activities for a group of users, possibly splitting them in subgroups to optimally satisfy their preferences. The tool is based on a formal model and two different technologies - Constraint Programming and Simulate Annealing - which can be easily adapted to other applications. The tests we have conducted show that our tool can be effectively used on real data for the city of Paris, with thousands of activities and hundred of users. The comparison between CP and the simulated annealing approach show that the latter can scale up to consider larger number of users, making our approach feasible also for quite different social applications.

We are now extending our work along several directions: First, we are considering a greater number of different activities and we are adding some more features such as, e.g., the selection of a preferred limited area for the activities (this is done by selecting an area on the map). Second, the recommendation semantics adopted in our model is aggregated preference: we are now exploring different notions of group recommendation semantics such as least misery, most pleasure, Borda count, etc. [106]. In particular we would like to see whether the semantics proposed in [21] with the related algorithms could improve our approach. Third, we would like to investigate techniques for group definition using social factors and group dynamics as those suggested in [85]. Fourth, we would like to explore possible improvements for the CP approach by using, e.g., linearizion of the constraints, column generation methods, or the use of pre-solve.

Chapter 4

Flexible Service Function Chaining Deployment with CP

Network Function Virtualization (NFV) and Software Defined Networking (SDN) are technologies that recently acquired a great momentum thanks to their promise of being a flexible and cost-effective solution for replacing hardware-based, vendor-dependent network middleboxes with software appliances running on general purpose hardware in the cloud. Delivering end-to-end networking services across multiple NFV/SDN network domains by implementing the so-called Service Function Chain (SFC) i.e. the sequence of Virtual Network Functions (VNFs) that will compose the service, is a challenging task.

In this chapter we address two crucial sub-problems of this task, namely i) the language to formalize the request of a given SFC to the network and ii) the solution of the SFC design problem, once the request is received. As for i) in our solution the request is built upon the intent-based approach, with a syntax that focuses on asking the user "what" she needs and not "how" it should be implemented, in a simple and high level language. Concerning ii) we define a formal model describing network architectures and VNF properties that is then used to solve the SFC design problem by means of Constraint Programming (CP), a programming paradigm which is often used in Artificial Intelligence applications. We argue that CP can be effectively used to address this kind of problems because it provides very expressive and flexible modeling languages which come with powerful solvers, thus providing efficient and scalable performance. We substantiate this claim by validating our tool on some typical and non trivial SFC design problems.

Structure of this chapter. In Section 4.1 we introduce the problem, in Section 4.2 we provide background knowledge and a detailed description of NFV/SDN-based frameworks, introducing the elements of the problem. In Section 4.3 we set the general problem framework and present our model to specify user desiderata and domain-level properties. In Section 4.4 we describe how to translate a given model into a MiniZinc finite domain specification, reporting in Section 4.5 validation experiments and performance results. Finally, in Section 4.6 we consider related work, we draw conclusions, and delineate future work.

4.1 Problem Introduction

Following the recent innovations brought about by Cloud Computing and resource virtualization, current advances in communication infrastructures show an unprecedented central role of software-based solutions [104]. On the one hand, Network Function Virtualization (NFV) [108] supports the deployment of network functions—e.g., load balancers, firewalls, intrusion detection devices, and traffic accelerators—as pieces of software running on off-the-shelf hardware. On the other hand, Software Defined Networking (SDN) [73] decouples the software-based control and management plane from the hardware-based forwarding plane, turning traditional infrastructures into fully programmable communication platforms. A SDN is hence a network whose topology can be orchestrated dynamically. By taking advantage of the complementary features of NFV and SDN it fosters the provision of flexible and cost-effective network services—from now on, referred simply as *services*.

As detailed in Section4.2, in an NFV/SDN framework, services are deployed as Service Function Chains (SFC) [48], i.e., the concatenation of some basic functions, typically running in some form of virtual environment (virtual machine, container etc.). These are called Virtual Network Functions in short VNFs. Essentially, an SFC corresponds to the sequence of VNFs that a traffic flow traverses from its source to its destination. In this context, multiple network configurations can coexist over the same physical infrastructure, bypassing the need for specialized hardware and physical network reconfigurations. Moreover the software-based SFCs can be instantiated, controlled, modified, and removed over a small time scale which is impossible to achieve in traditional networks typically requiring physical or manual reconfiguration to modify topology and/or forwarding. However, one of the main problems linked to SFC planning is that it is complex to define and apply SFC configurations that both respect multiple domain-level properties (QoS, etc.) and also avoid misbehaviors over contrasting or incompatible service desiderata. This calls for both suitable, high-level languages to easily describe SFC requests and for tools to efficiently design SFC—once the request is received—given the available VNFs and network resources.

Contribution. Answering this call, in this work we propose two contributions. The first is a model to describe both SFC user requests and the holding domain-level constraints over a multi-domain network scenario—since the model is intended for (possibly automated) user interaction (both customers and network administrators) it is expressed using the familiar JavaScript Object Notation (JSON). The second is a tool based on Constraint Programming (CP) which solves the SFC design problem. The tool uses a MiniZinc specification which is a direct translation of the JSON specification. While there exists another paper [91] using CP techniques for routing problems, ours is the first proposal of applying CP to the SFC design problem in its full generality. We argue that CP can be effectively used to address this kind of problems, as it provides very expressive and flexible modeling languages to harness the complexity of SFC design. This, together with the outstanding performance of modern CP solvers, has promising aspects in terms of scalability, opening the market to operators offering ad-hoc just-in-time SFC configurations to users. To substantiate our claims we validated our tool by solving some typical and non-trivial SFC design problems and considering its performance.

4.2 Application Context: NFV/SDN Networking

NFV/SDN paradigms promise to revolutionize network management through the concept of *network programmability*, i.e., the possibility to run network services in a similar way as running software in a computer. Indeed, traditional network functions are bound to hardware devices, in which actions like instantiating a new service or modifying a service instance are rather complex and require specialized operations. Contrarily, the combination of recent NFV/SDN technologies paves the way to fully programmable communication networks. The expected benefits of programmable networks are reduced operation costs, as well as increased flexibility and responsiveness.

Network Function Virtualization. In NFV network functionalities, currently mostly implemented by means of dedicated appliances (the so called *middleboxes*, like firewalls, NATs, packet inspectors, traffic conditioners, etc.) are turned into software applications, called Virtual Network Functions (VNFs). These are shipped inside virtual machines or containers and hosted into cloud computing infrastructures equipped with off-the-shelf hardware (i.e., not specialized for a specific networking function) [108]. The basic concept it is briefly sketched in Fig. 4.1.

Software Defined Networking. SDN decouples the network control plane from the data forwarding plane. The former is placed into a so called *SDN controller*, defining all the forwarding logics in a centralized way and injecting them into the networking devices. The main protocol proposed for SDN is Openflow [107], which is designed to support the dialog between network controllers and appliances.

The ETSI NFV-MANO Framework. NFV became subject of standardization by ETSI in the NFV Management and Orchestration (MANO) framework. ETSI launched the initiative by bringing together seven leading telecom operators in 2012. Currently over 300 individual companies [76], including many global service providers, joined the initiative, which is the reference standardization framework in this field. We provide in Fig. 4.2 a conceptual representation of the approach proposed by the ETSI NFV-MANO framework—from now on called MANO [77]. In MANO,

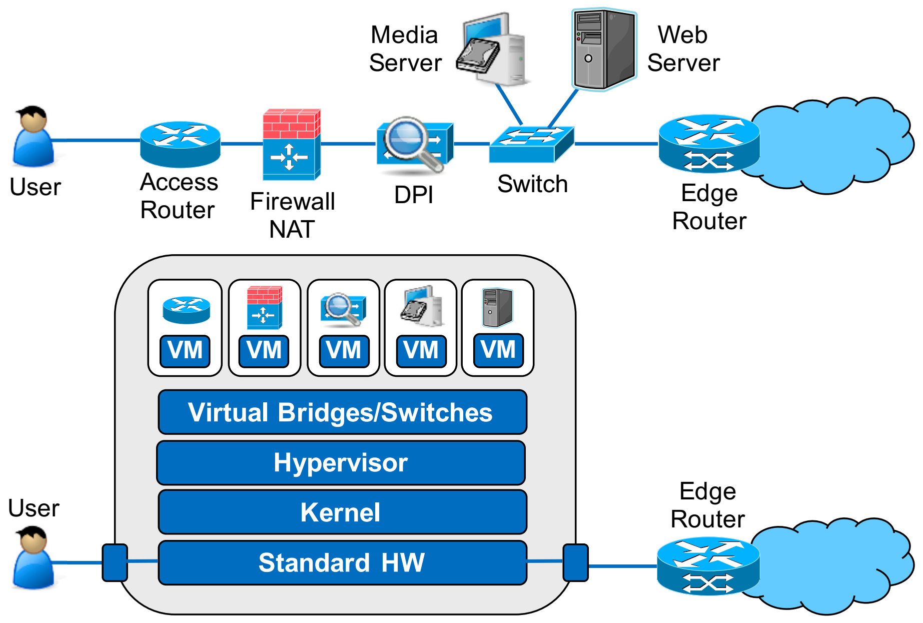


Figure 4.1: General concept of NFV.

VNFs are deployed over a set of cloud data centers that may be either closely or remotely located, depending on the specific service implementation scenario. The data centers are managed by a specific cloud infrastructure management system chosen by the owner/provider, e.g., the renowned OpenStack [36] platform, while general networking services are managed by SDN controllers. MANO addresses both cloud and network controllers as Virtualized Infrastructure Managers (VIMs).

The NorthBound Interface. The components in Fig. 4.2 must interact by means of suitable Application Programming Interfaces (APIs) and, roughly speaking, the the API offered by a given functional block to the one that is logically above it (providing increased abstraction) is usually called a *NorthBound Interface (NBI)* while the interface with one logically below (closer to the specific implementation) is usually called a *SouthBound Interface (SBI)*. [1]

The Service Function Chain. In this context, a *service* is a specific combination of VNFs and communication capabilities that are requested by a user and that

[1]For completeness, interfaces between functional blocks at the same architectural level are usually addressed as East/West-bound interfaces.

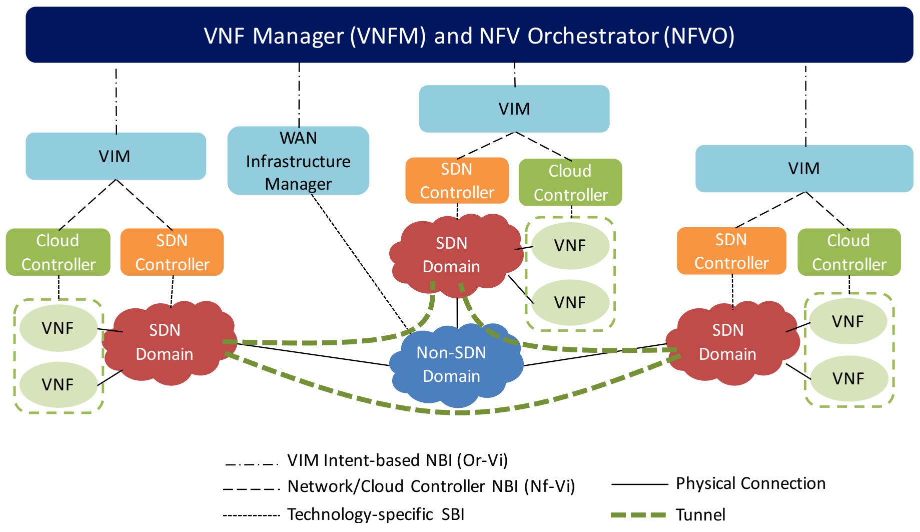


Figure 4.2: General concept of MANO.

must be implemented in the available infrastructure. [2] This is the *Service Function Chain (SFC)*, i.e. the implementation of a composite service as the concatenation of basic services, typically implemented via VNFs. For instance an SFC could be the sequence of a NAT and a Firewall at the edge of the provider network, serving a set of customers. In essence, an SFC is the series of VNFs that a traffic flow must traverse from its source to its destination. *Thanks to the capabilities offered by SDN and NFV, SFCs can be dynamically controlled and modified over a relatively small time scale, both increasing the flexibility of service provisioning and reducing the management burden.*

SFC deployment planning. The aspect we focus in this work is SFC deployment planning, also called Service Function Chaining (SF-Chaining). Within a single technological and administrative domain, e.g., a single data center, SF-Chaining can be successfully achieved with the help of the native domain management system, i.e. the VIM [28]. However, when the SFC spans across multiple network domains, (c.f., Fig. 4.3) each owned by a different player and characterized by different technology stacks, the dimensional and logical complexity of the problem increases. With many

[2] Here, *users* may either be *customers* (residential or business) requiring a specific networking service or *network operators* configuring specific services for their customers.

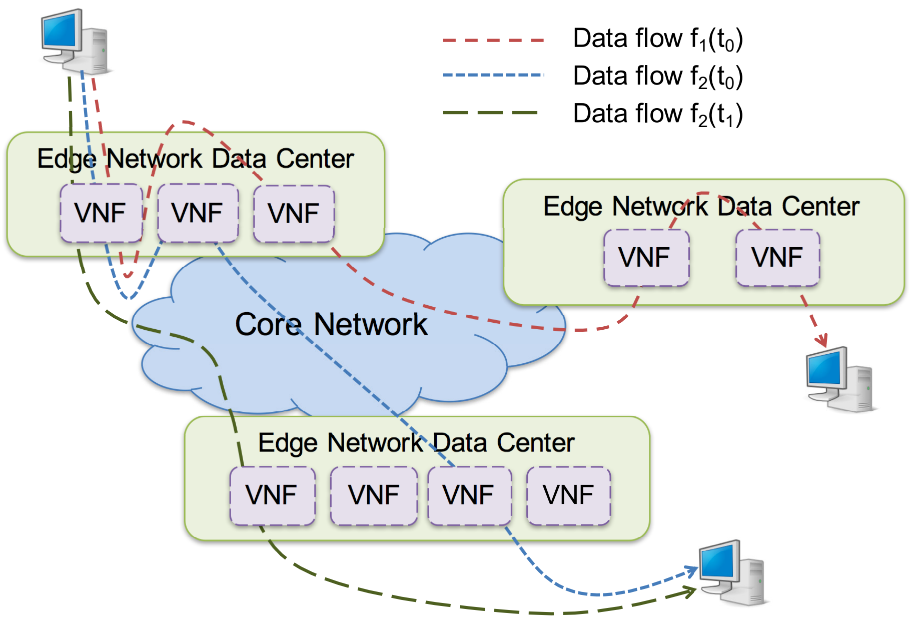


Figure 4.3: General example of dynamic Service Function Chaining.

domains and many VNFs per domain the space of possible solutions to a specific SF-Chaining problem becomes very large as formally shown in the following section. Moreover the specification of the SF-Chaining request in a general way, that can be mapped over the various domains is also non trivial [124, 117, 133].

MANO provides a general architectural framework for the implementation of NFV but does not provide implementation details for the various interfaces of logical levels, that are still matter of study and testing.

Regarding the specification of the SF-Chaining request, solutions have been recently proposed to implement a vendor-agnostic, and interoperable NBI interface for the MANO according to the *intent-based* approach [35]. Very briefly the intent-based approach goal is to provide a semantic at the interface that allows the user to focus on *what* he/she wants to achieve and not on *how* it will be implemented, thus hiding all the technology-specific details and making the service request as general as possible. In this work we extend and better formalize this approach by providing a general schema for the semantics of the interface that can be easily translated into technology dependent specifications.

While the intent-based specification solves the problem of applying a global plan over multiple domains, it does not answer the problem of engineering the SF-Chaining. Generally speaking this problem consists of: *i*) SFC design: addressing the issue of selecting the set of VNFs to be chained to implement the SFC, with the goal of optimizing some notion of cost; *ii*) VNF activation and placement: addressing the issue of where to execute VNFs when more options are available, for instance with the goal to maximize performance or distribute the workload.

SF-Chaining is a crucial part of the Resource Allocation problem in an NFV environment and has been mostly studied by means of Mixed Integer Linear Programming [58]. Unfortunately the complexity of the problem makes such solutions viable just for small networks. Usually heuristics are proposed and tailored to some specific optimization goal, thus limiting their applicability or generality. The problem is that, when designing an SFC, beside standard shortest-path problems, one has to solve additional constraints arising from the specific nature of the service functions involved. For example, if a Virtual Private Network (VPN) function is present, which encrypts a message before it leaves the source domain, then a complementary VPN function should appear before the final destination, to decrypt the message.

In this work we propose an efficient, general and scalable tool, based on Constraint Programming (CP), for the engineering of SFC plans over multiple domains. We will show that complex SFC plans can be computed in a small time-frame, turning the engineering and application of SFC plans from a manual, time-consuming activity to an automatic and just-in-time task.

4.3 Problem Definition

With reference to what explained above, in this section we set the general problem framework following the schematic presented in Fig. 4.3. In particular we assume the following.

- *Network architecture.* The network is divided into a number of *Domains*, defined according to administrative and/or technological boundaries. For the purpose of this work a Domain is an infrastructure that is managed homogeneously by a single actor. The Domain has one or a set of Virtual Infrastructure Managers that are properly coordinated and thus acts as a single entity. The resources of the Domain are managed as a whole.

- *Inter-Domains interconnection.* We assume that the various Domains are interconnected by Domain border gateways and interconnection links. Domain interconnections may be at the geographical as well as at the local level, depending on topological and administrative constraints. Domain interconnection can be related to some form of QoS objective, either cost, latency, bandwidth availability, etc. depending on the specific scenario.
- *Intra-Domains interconnection.* The networking among VNFs of the same domain is not a subject of this work. We assume that, within a domain, connectivity is granted at a level of Quality of Service sufficient for the purpose. If the various domains are data centers, their management platforms provision the resources needed in terms of computation, networking etc.
- *VNFs.* The Virtual Network Functions are devoted to specific networking tasks. In this work we assume that one VNF performs just one task, therefore we will talk of VNF types to specify which tasks are performed. The VNF types considered in the following are briefly described below.
- *VNF location.* VNFs are executed in the data centers hosted in the various Domains. In principle the Domains are not homogeneous in terms of connectivity, computing capabilities and functionalities, therefore a Domain may or may not be suitable to execute some VNFs. Moreover it may be that a given VNF has to be executed into a specific domain. Without loss of generality, we restrict the choice of the location of each VNF in an SFC to three options: the source Domain, the destination Domain or unspecified; the latter meaning that the VNF can be located in any available Domain, including source and destination.

The set VNF types is a set of network functions that we consider to be part of common networking practice, obviously the work can be extended to include other types of VNFs.

- *Deep Packet Inspector (DPI).* Looks into the content of the packets and takes specific forwarding decisions according to specific predefined patters.
- *Network Address Translator (NAT).* Translates IP addresses mostly used to interconnect areas with private IP addressing from the public Internet.
- *Traffic Shaper (TS).* May enforce specific packet and/or bit rate limitations to a traffic flow.
- *Wide Area Network Accelerator (WANA).* Compresses packet content to provide higher transfer speed.
- *Virtual Private Network Endpoint (VPN).* Encrypts data flows and authenticate users over a specific public network section.

Note that *gateway* VNFs do not appear in the user desiderata, however they are needed, as discussed before, to provide inter-domain connections. The VNF set that we will consider to construct a solution will then include also gateways.

4.3.1 Service Function Chain specification

In the remainder, to distinguish between customer and network operator SFC desiderata, we call the former *user requests* and the latter *domain constraints*. In order to provide a concrete and simple model for specifying SFC user requests, immediately usable in practice, we rely on the JSON [38] notation, defining the model using the generic formalism of JSON Schema [52] as follows.

Definition 4.1 (SFC user request) *A Service Function Chain user request is any JSON specification compliant with the JSON Schema below, where we assume that the cardinalities of* `vnfList`*,* `prox_to_src`*, and* `prox_to_dst` *are equal.*

```
{
  "VNFs":{
    "type":"array",
    "items":{
      "type":"string",
      "enum":["DPI","NAT","TS","WANA","VPN"]
        }
  },
  "Mask":{
    "type":"array",
    "item":{"type":"boolean"}
  },
  "type":"object",
  "properties":{
    "src":{"type":"string"},
    "dst":{"type":"string"},
    "qos":{"type":"string"},
    "qos_type":{"type":"string"},
    "qos_thr":{"type":"string"},
    "qos_value":{"type":"integer"},
    "vnfList":{
      "$ref":"#/VNFs"},
      "dupList":{"$ref":"#/VNFs"
    },
    "prox_to_src":{
      "$ref":"#/Mask"
    },
    "prox_to_dst":{
      "$ref":"#/Mask"
    }
  }
}
```

Briefly, the highlighted elements in Definition 4.1 represent:

- `src` and `dst` the start and target domain of the service chain;

- `qos` the QoS feature to be provided with the service chain;
- `qos_type` a high-level unique identifier of a QoS metric;
- `qos_thr` the QoS threshold to be applied to the specified metric;
- `qos_value` the value assigned to the threshold;
- `vnfList` is the ordered list of VNFs to be traversed for the requested service. We `enum`erate them in type `VNFs` as strings representing the VNFs we support in our model (and mentioned at the beginning of Section4.3);
- `dupList` is the set of VNF types where the traffic needs to be duplicated.

Finally, `prox_to_src` and `prox_to_dst` are `Mask`s on the `vnfList`, i.e., they are arrays of booleans with the same cardinality of `vnfList` that indicate if a VNF should be respectively located in the domain of the `src` or of the `dst`.

Example 4.1 *To* complete Definition 4.1, we report an example of SFC user request. In the code below, the user requests a chain between domains `s` and `d`, indicating a `qos` on the `speed` of the connection, measured in terms of `bandwidth` with a threshold of `90`% on the `throughput` of transmitted data. The service request consists of (in this order): a `DPI` (whose traffic is duplicated, as per `dupList`), a `VPN` in the domain of `s` and a complementary `VPN` function in the domain of `d`.

```
{
  "src":"s",
  "dst":"d",
  "qos":"speed",
  "qos_type":"bandwidth",
  "qos_thr":"throughput",
  "qos_value":90,
  "vnfList":["DPI","VPN","VPN"],
  "dupList":["DPI"],
  "prox_to_src":[1,1,0],
  "prox_to_dst":[0,0,1]
}
```

$\square$

In the next section, we explain how we combine the parameters above are to define the solution to an SFC planning problem.

4.3.2 SFC design problem

In order to formalize the `SFC design problem` we represent a network architecture in abstract terms as a directed graph $G(V, L)$ with a set V of labeled nodes, ranged over by $v_1, v_2 \ldots$, which represent the VNFs and a set $L = \{(u, v) | \forall u, v \in V \wedge u \neq v\}$ of labeled arcs, ranged over by $l_1, l_2, \ldots$, which represent links among different VNFs. The level of a node v denote the type of functionality provided by the specific VNF v in set T, ranged over by $t_1, t_2, \ldots$, and we assume that there exists a total function $Type : V \rightarrow T$ which, for any VNF $v \in V$, returns its label (i.e., its type). We distinguish between a VNF and its type because different VNFs, also in the same domain, can offer the same functionality and have the same type. Nevertheless, when no ambiguity arises, we will identify a VNF with its type. For example, in the service chain request provided by the user, the list of VNF which is provided is, strictly speaking, the list of VNF types which are required (the user is interested in a functionality, not in the specific component implementing it). Label of arcs denote costs of the arcs and we indicate by $c_{u,v}$ the cost of an arc (u, v). Paths are defined as usual. [3]

As we have seen in previous section, conceptually VNFs are organized in domains that is, our graph is divided into several sub-graphs. We represent this structure by introducing a set D of domains, ranged over by $d_1, d_2, \ldots$, and assuming that there exists a total function Domain: $V \rightarrow D$ which for any VNF $v \in V$ provides its domain Domain(v). We assume that each domain in our network has exactly one VNF providing the (domain border) *gateway* functionality. In order to model the domain interconnection described above we assume that the set of arcs in our network consists of: i) the arcs connecting the gateway to all the other VNFs in the

[3] For the notions on graphs not directly defined here please see [37, 42].

same domain, with cost 0 and ii) the arcs connecting a gateway to all the gateways VNF appearing in the other domains, with a positive cost. We are now ready to define the notion of `SFCtree`. Intuitively this represents the chain of functions which, in a given network, satisfy the service request expressed by the user. Note that we consider a tree rather then a simple path because in some cases the chain of functions, beside a source and a target, include some other terminating nodes which provide specific functionalities: for example, a `DPI` VNF has the task of logging messages and does not participate in message routing. Moreover, nodes (VNFs) in the same domain are represented as sons of a gateway.

Definition 4.2 (`SFCtree`) *Given a directed graph $G(V, L)$ representing a network architecture, an* `SFCtree`[4] *is a rooted tree Tr which is a subgraph of $G(V, L)$ and such that the leafs of Tr are (labeled by)* VNFs *types different from* gateway*, while the nodes that are not leafs are (labeled by)* gateway*.*

Example 4.2 *A*n example of `SFCtree` is shown in Fig. 4.4. This `SFCtree` is used to satisfy a service chain of two DPI VNFs which connects domain 1 and domain 4. □

As a first approximation, our configuration problem consists in finding an `SFCtree` which satisfies the service request specified by the user in terms of intents. There are however some additional, domain level, constraints on the VNFs to be used in the SFC which are needed to obtain a correct solution. For example, we may need to know whether a VNF v needs to be "mirrored" , meaning that when v appears in a chain then another, dual, VNF is needed in the same chain (for example an encryption function needs later a decryption). Also, some quantitative information are needed at domain level, such as lower and upper bounds on the number of VNFs of the same type in a given domain. These additional constraints are not expressed by the intents of the users (who might ignore the detailed domain structure of the network) but are introduced in a middle layer before formulating the actual service request. As we have done for SFC user request, we represent these constraints following the JSON Schema.

[4]The definition is parametric w.r.t. the given graph, however we do not represent such a parameter explicitly, to simplify the notation.

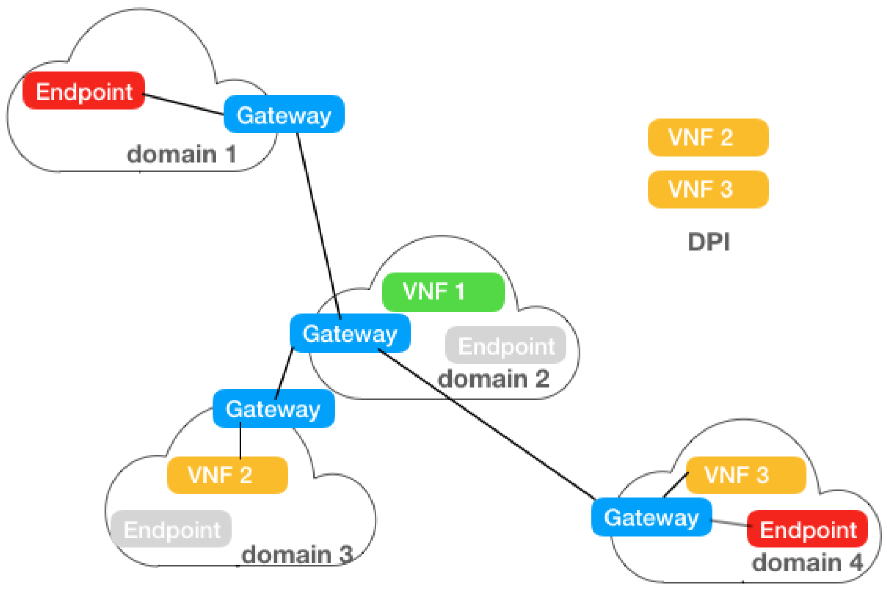


Figure 4.4: `SFCtree` example.

Definition 4.3 (Domain-constraints) *A Domain-constraint is a JSON specification compliant with the following JSON Schema*

```
{
  "type":"array",
  "items":{
        "type":"array",
        "maxItems":4,
        "items":[
          {"type":"string","description":"a domain name"},
          {"type":"string","enum":["DPI","NAT","TS","WANA","VPN"]},
          {"type":"integer","description":"VFN type minimum quantity"},
          {"type":"integer","description":"VFN type maximum quantity"}
        ]
  }
}
```

In the JSON Schema above, we use the `"description"` attribute to hint the content of each element. A Domain-constraint then represents a set of tuples (d, t, m, n) where

d is a domain, t is a VNF type, and m, n are natural numbers, with the meaning that in the domain d there are at least m and at most n VNFs $v \in V$ having the type t.

Example 4.3 *To* complete Definition 4.3, we report an example of a Domain-constraint which could be imposed by domain administrators. Here `s` and `d` are the source and destination domains of Example 4.3.1 and we see that the administrator set to `1` and `2` the minimal a maximal number of `WANA` functions allowed in `s`; the constraint specifies also that a single `DPI` function is required in `s` (i.e., minimal and maximal capacities coincide) and a single `VPN` (and `NAT`) is required in the destination `d`.

```
[
  ["s","WANA",1,2],
  ["s","VPN",5,10],
  ["s","DPI",1,1],... ["other_dom",DPI,1,2],
  ["other_dom","VPN",1,10], ...
  ["d","VPN",1,1],
  ["d","NAT",1,1]
]
```

□

Before defining formally our `SFC design problem` we now need to define when an `SFCtree`—that intuitively represents a solution—satisfies the user request and the Domain constraints. To this aim, we first provide the following definition.

Definition 4.4 *Assume that R is an SFC user request specified as in Definition 4.1 which defines the* `vnfList` $= \{t_1, \ldots, t_n\}$ *and a* `dupList` $= \{e_1, \ldots, e_m\}$. *Then we define* `request-tree(R)` *as the tree $T(V, L)$ where the set of nodes is $V = \{v_1, \ldots, v_n\}$ with $Type(v_i) = t_i$, $\forall i \in [1, n]$ and the set of arcs is $L = \{(v_i, v_j) | v_i, v_j \in V \wedge i < j \wedge Type(v_i) \notin$* `dupList` $\wedge\ (\nexists k, i < k < j, v_k \notin$ `dupList`$)\}$.

Intuitively, given a user request R, `request-tree(R)` is the tree that represents the traversal order of the various VNFs, from the source to the destination domain, to obtain a solution. We have a tree rather then a sequence of VNFs because we take into account also the information provided by `dupList` which, as mentioned before, specifies when the traffic needs to be duplicated before entering in a node (VNF).

Example 4.4 *G*iven a user request which specifies `vnfList` $= \{a, b, c, d\}$ and `dupList` $= \{b\}$, with a in the source domain and d in the destination domain, a `request-tree` $T(V, L)$ consists of $V = \{a, b, c, d\}$, $L = \{(a, b), (a, c), (c, d)\}$. $\square$

Next we define the satisfaction of user request and domain constraints. In the following we use the terminology and notation introduced in Definitions 4.1 and 4.3. We also assume that the last VNF specified in the user `vnfList` is present in the destination domain (if this were not the case we could introduce and additional *Endpoint* VNF but we prefer to avoid this in order to simplify the notation).

Definition 4.5 *We say that an* `SFCtree` $Tr(V_r, L_r)$ *satisfies user request* R *and domain constraints* C *if the following holds, where* `request-tree(R)` $= T(V, L)$ *and* d_{src}, d_{dst} *are the domains values specified in* `dst` *and* `src` *of request R:*

i) the domain of the root of Tr *is* d_{src} *and there exists a leaf in* Tr *whose domain is* d_{dst}*.*

ii) V_r *is the set* V *with some additional gateway nodes and there exists an injective mapping* $m : V \to V_r$ *such that,* $\forall v \in V$, $Type(v) = Type(m(v))$*;*

iii) $\forall(u, v) \in L\ \exists g_u, g_v \in V_r$ *such that* $Type(g_u) = Type(g_v) = gateway \wedge (g_u, m(u)) \in L_r \wedge (g_v, m(v)) \in L_r$ *and there exists a path in* Tr *between* g_u *and* g_v *containing only gateway nodes;*

iv) for each $v \in V$ *if* `prox_to_src`$(v) = 1$ *then* $Domain(m(v)) = d_{src}$ *and if* `prox_to_dst`$(v) = 1$ *then* $Domain(m(v)) = d_{dst}$*;*

v) for each tuple (d, t, m, n) *represented by* C *such that the type* t *appears (as label of a node) in* $T(V, L)$*,* $m \leq$ `Num`$(Tr, d, t) \leq n$ *holds, where* `Num`$(Tr, d, t) = |\{v | v \in Tr,\ Type(v) = t \text{ and } Domain(v) = d\}|$*.*

Note that, as indicated in item *iv*), we assume that the domain constraints refer to the VNF specified in the `vnfList` provided by the user.

We are now ready to state formally our configuration problem.

Definition 4.6 (`SFC design problem`) *Given a graph $G(V, L)$ that represents a network architecture, an SFC user request R and domain constraints C, the* `SFC design problem` *consists in finding an* `SFCtree` *that satisfies the request R and the constraint C. Such an* `SFCtree`*, if it exists, is called an admissible solution. Furthermore, the optimal* `SFC design problem` *consist in finding an admissible solution $G(V', L')$which minimize the following cost function: $\sum_{l \in L'} c_l$. In this case the solution found is called optimal* `SFCtree`*.*

The following result shows that the problem that we are considering here is a difficult one. The proof can be done by the reduction of the k-minimum spanning tree problem which is known to be NP-hard [122].

Theorem 4.1 (NP-hardness) *The optimal* `SFC design problem` *is NP-hard.* [5]

4.4 SFC modeling with Constraint Programming

In order to solve our `SFC design problem` we translate it into a MiniZinc [113] finite domain specification. MiniZinc is a high level, solver independent, constraint modeling language which is widely used and is supported by large variety of constraint solvers. We assume some familiarity with MiniZinc and we invite the reader to consult [113] for further details.

Our translation is a direct encoding of the SFC design problem as defined in Section 4.3 in MiniZinc constraints. More precisely, we first model in terms of the MiniZinc language the network architecture and then we translate in MinZinc the user request and the domain constraints defined in the JSON format. The MiniZinc specification of the network architecture is a straightforward translation of the graph

[5]Theorem proof in Appendix A.1.

described in the previous section and is provided below (comments are indicated by %).

```
int: n_nodes;      % Number of nodes (VNFs).
int: n_domains;    % Number of domains.
int: n_node_links;% Number of arcs (links between nodes).
int: M;            % Upper bound for arc costs.
% Array containing cost of arcs between pairs of gateway nodes.
array[1..n_domains, 1..n_domains] of 0..M:
domain_link_costs; % Array representing the arcs.
array[1..n_node_links, 1..2] of 1..n_nodes:
node_links; % Array describing the properties of the nodes,
% i.e. node id, the type of node, its domain
array[1..n_nodes, 1..3] of int: nodes;
```

Upon a user request expressed in the intent format, we use a script to extract necessary information and by using `dupList` we parses the `vnfList` into `vnf_arcs` that represents the arcs of `request-tree` and finally we create an instance for MiniZinc.

As for the specification of the SFC request and domain constraints, described in definitions 4.1 and 4.3 in terms of JSON specifications, we use a script to extract necessary information and by using `dupList` we parses the `vnfList` into the `vnf_arcs` array below. Analogously we parse the domain constraints to build the `domain_constraint` array and we obtain the following MiniZinc code:

```
int: start_domain;
int: target_domain;
int: n_types;       % Number of VNF types except Gateway
int: vnflist_size; % The length of vnflist
int: n_dcons;       % Number of domain constraint
% The order of VNF in the service request.
array[1..vnflist_size] of 0..n_types: vnflist
% arcs of request-tree derived from vnflist
array[1..vnflist_size-1, 1..2] of 0..vnflist_size: vnf_arcs;
% VNF service in start domain
array[1..vnflist_size] of 0..1: proximity_to_source;
% VNF service in target domain
array[1..vnflist_size] of 0..1: proximity_to_destination;
```

```
% Domain constraints containing: domain id,vnf types, min, max.
array[1..n_dcons, 1..4] of int: domain_constraints;
```

To model our problem we then introduce two groups of MiniZinc variables, the first representing the selection of arcs, links, domains and domain connection, and the second to ensure that the selected nodes corresponding to the VNFs in `vnfList` and their order is feasible.

Next we introduce the constraints which can be classified into three groups: the first one states the relations between variables (a.k.a channel constraints), the second guarantees that the variable values meet the request requirements and the last one ensure the tree properties of the solution. The key variable among all is the variable `link_selection`, it is possible to build a relation with it to any other variables, e.g. to specify if a node or domain is selected it is enough to say whenever a link is selected then the related nodes and their domains are selected. The details of this formalization can be found in [96].

With these constraints we are able to obtain an admissible `SFCtree`. The optimal solution is the obtained by optimizing the sum of domain link costs of among all possible admissible solutions.

4.5 Empirical Validations

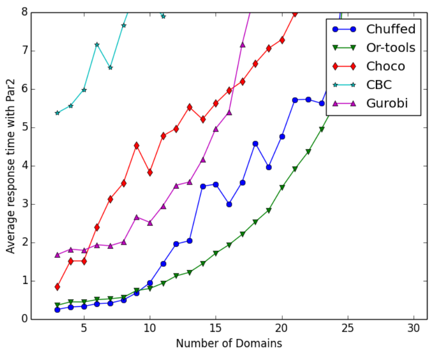
We now describe the validation experiments which we have conducted in order to compare the performance of different state-of-the-art solvers and to assess the efficiency and scalability of our approach.

As for the experiment setup, we have generated the dataset representing the network in a random way. We assume n nodes and m domains with $\frac{n}{m} > 2$. We select m out of the n nodes and consider them as `gateway`, while for the remaining nodes we associate randomly to each of them a VNF type from the set of types assumed in this work (see Section 4.3). Next we defined the arcs according to the definition in 4.3.2 with costs in the range $[1, 100]$. Regarding the SFC user request, we created a dataset of possible requests that may occur in practice, which are compliant with the assumptions we made and with the ETSI specifications [78], from which we randomly choose specific instances. We consider the number of nodes and the number of domains as features that characterize the specific instance dimension. For each instance dimension we generate 10 scenarios and for each scenario we generate 10 requests which will be performed sequentially. We record the response time, that is the time needed to find optimal solution or to discover that the instance is unsatisfiable, with a cutoff time as 5 seconds for each run. The experiments were run on a Debian cluster with machines equipped with Intel Core$i$5 3.30GHz and 8 GB of RAM.

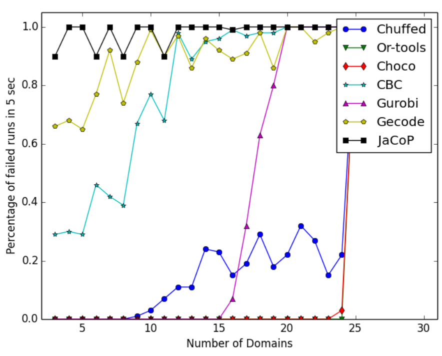
We first compared the performance of five different state-of-the-art CP solvers, namely, Or-Tools v6.7 [63], Choco 4.0.4 [121], JaCoP [88] Gecode [127], Chuffed [34] and two Mixed Integer Programming (MIP) solvers, Gurobi [66] (one of the most performing MILP solvers [67]) and CBC [49][6] on the optimal `SFC design problem`. The solvers were run on scenario with 300 nodes and different number of domains (from 3 to 30), each request was combined with 2 random domain constraints. In the graph 4.5 (a) we show the response time with `Par2` penalty, where when a run was not completed at timeout we consider its runtime as two times of the timeout (10 sec). [7] Under the `Par2` metric, it can be seen that Chuffed and Or-Tools were the most competitive solvers in our case, in particular, Chuffed runs faster with few number of domains (less than 10) while Or-Tools is more robust addressing instance with larger number of domains. The part (b) of Fig 4.5 shows the percentage of runs failed to prove optimality or unsatisfiability within timeout. It can be seen that Choco and Or-Tools were the most competitive where they solved almost all the instances with less than 24 domains. Chuffed started to have unsolved instances when the number of domains goes beyond 9, however, it is still much better that other solvers where they had failed runs even with 3 domains. It worth noticing that the MIP solver Gurobi was less competitive than the CP solver in our case, even

[6]The Or-Tools were downloaded from Google OR official page and other solvers were taken either from SUNNY-CP [7, 9] or from the MiniZinc distribution v2.17.

[7]The performance of Gecode and JaCoP were omitted since their performance were much lower than those of the other solvers.



(a) Response time with 300 nodes



(b) Percentage of Failed Runs

Figure 4.5: Solvers Comparison.

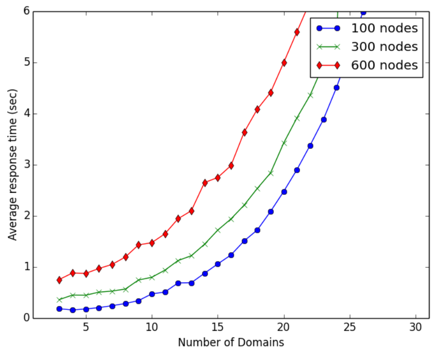
though, the `MIP/ILP` is the most popular approach for NFV/SDN problems today. [8]

In the second set of experiments we considered only the solver Or-Tools and we considered two groups of tests: (i) fixing a number of nodes we vary the number of domains from 3 to 30. (ii) fixing a number of domains we vary the number of nodes from 30 to 800. In this case, the average runtime has excluded failed runs. As one can see from Fig 4.6, our application find the optimal solution for instances having more than 300 nodes and 10 domains in less then a second. [9]

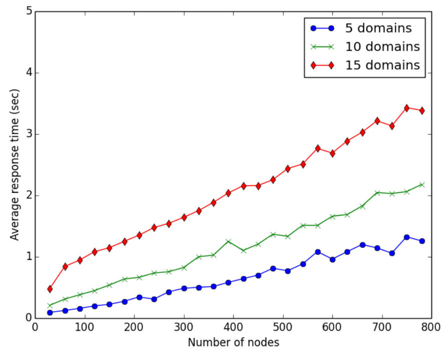
Moreover, for the part (b) of the figure one sees that time grows almost linearly at the growth of node numbers. Since in practical applications one has hardly more than 10 domains and one has hardly a large number of nodes, and also, the links between domains are much less than our fully connected case, the results confirm that our system is relevant to address the `SFC design problem` and can scale up to consider large networks. It is worth mentioning that, for instance, the International Telecommunication Union in its Recommendation [115] sets an upper bound to the time needed to set up of a service at 7.5 seconds, well above the time needed here to solve the SF-Chaining problem.

[8]We note that there are several similar problems [33, 44] which are also solved with Gurobi; also in their cases, the tool's runtime is considerably high.

[9]We also measured the runtime when request instance is unsatisfiable, generally, it takes as much time as computing a satisfiable instance.



(a) Response time in relation with number of domains



(b) Response time in relation with number of nodes

Figure 4.6: System performance varying instance size.

We also conducted other experiments which showed that changing the number of domain constraints does not affect significantly the response time. For more experiment details, we refer the interested readers to Appendix A.3.

4.6 Summary

To the best of our knowledge the only other paper applying CP techniques to programmable communication networks is [91], where the authors consider the specific problem of optimizing the QoS of routing applications. Here we consider a completely different problem, namely the definition of expressive and efficient tools to solve the Service Function Chaining design problem in general. There exists a large body of literature on the problem of mapping an SFC to the (possibly virtualized) substrate network, optimizing some notion of QoS. This problem, also called Service Function Chain Resource Allocation (SFC-RA), has been mainly addressed with (Mixed) Integer Linear Programming (M)ILP techniques. However, since in its full generality SFC-RA is an NP-hard problem, many alternative approaches rely on approximated methods and (meta)-heuristics (cf. [73, 108, 148, 58] for more precise indications). When compared with other exact methods based on (M)ILP, CP provides a more flexible and general approach. Since (M)ILP approaches consider a specific formulation of the problem—customized for a narrow class of applications with a specific function to be optimized—and require a large number of decision variables and (in)equations, it becomes difficult to adapt existing solutions to other cases. Performance-wise, we cannot directly compare our work to other MILP based approaches, since the problem we are solving here is more general than the specific ones treated in the literature. However, our experimental results show that CP solvers are more efficient than MILP solvers on the problem we consider and support our claim that the proposed model can scale efficiently.

As future work, we plan to carry out a more in-depth experimental analysis and evaluation and then to include our tool into a networking tool-chain able to directly apply synthesized SFC plans on target networks. Also, we intend to further investigate the definition of an high level, more abstract, intent-based language for SFC specification. Beside allowing to express quickly and intuitively SFC requests, such an abstract language naturally would allow to use modularization and typing [118] principles with the following benefits: *i*) support for the creation of libraries of standardized SFCs, e.g., configurations that adhere to administrative regulations which can be directly used with little customization effort; *ii*) definition of complex specifications obtained by combining simpler ones; *iii*) possibility of checking (even at writing time, as it happens in standard IDEs) if SFC specifications are well-formed (e.g., if the traffic encrypted by a VPN is decrypted by a complementary function) and if they follow best practices (e.g., by warning the user that, by using a VPN function outside the domain of the source, its traffic is exposed to attackers).

Part II

Constraint Solver Selection with SUNNY

Chapter 5

SUNNY for Algorithm Selection

When applying constraint programming and once a problem model is defined and fine-tuned, the following step is to find out the most suitable solver that offers competitive performance. In Chapter 3, we saw that Chuffed is the best choice for `NightSplit`, while in Chapter 4, we realized that Or-Tools is better than Chuffed for `SFC design problem`. To select the appropriate solver automatically for unseen instances, one can rely on the technique of algorithm selection (AS).

In this chapter, we draw our attention to an AS technique, SUNNY, which is among the few selection methods available designed for constraint solver selection. SUNNY enables us to schedule, from a portfolio of solvers, a subset of solvers to be run on a given CP problem. This approach was proven to be effective in the MiniZinc Challenge, the yearly international competition for CP solvers. In 2015, the COSEAL group released the ASlib benchmarks, enabling the comparison of a wider range of AS systems for problems coming from disparate fields (e.g., ASP, QBF, and SAT). Based on ASlib, the 2015 ICON Challenge on Algorithm Selection was held. SUNNY was adapted to deal with generic AS problems, but unfortunately its performance was not satisfactory. Afterward, more attention was paid to investigating how SUNNY could be configured to suit the ASlib scenarios better. In this chapter, we discuss the advancements we made on SUNNY, which allowed it to obtain promising results in the Open Algorithm Selection Challenge 2017 and in the scenarios of constraint programming.

Structure of this chapter. In Section 5.1 we introduce the problem of algorithm selection. In Section 5.2 we review the literature on algorithm selection before giving background information in Section 5.3. In Section 5.4 we introduce the improved `sunny-as2` tool and in Section 5.5 we show the empirical experiments on which `sunny-as2` was validated. We draw some concluding remarks in Section 5.6.

5.1 Introduction to Algorithm Selection

Solving combinatorial problems is hard, and clearly there does not exist a single, dominant algorithm for each class of problems. A natural way to face the disparate nature of combinatorial problems is to use a *portfolio* of different algorithms (or solvers) to be selected on different problem instances. The task of identifying suitable algorithm(s) for specific instances of a problem is known as per-instance *Algorithm Selection* (AS). By using AS, solvers are able to outperform state-of-the-art solvers in many fields, such as Propositional Satisfiability (SAT), Constraint Programming (CP), Answer Set Programming (ASP) and Quantified Boolean Formula (QBF) [11]. In each of these fields, plenty of domain-specific AS strategies have been studied. However, it is hard to judge which of them is the best strategy in general. To address this problem the *Algorithm Selection library* (ASlib) [23] has been proposed. ASlib consists of scenarios collected from a broad range of domains, aiming to give a cross-the-board performance comparison of different AS techniques. Based on the ASlib benchmarks, rigorous validations and AS competitions have been recently held.

In this work, we focus on the SUNNY portfolio approach [9, 10], originally developed to solve Constraint Satisfaction Problems (CSPs). SUNNY is based on the k-nearest neighbors algorithm. Given an unseen problem instance P, SUNNY generates a schedule of solvers as follows. It first extracts its *feature vector* F_P, i.e., a collection of numerical attributes characterizing P, and then finds the k training instances "more similar" to F_P according to the Euclidean distance. Furthermore, SUNNY selects the best solvers for these k instances; a time slot proportional to the number of solved instances is then assigned to the selected solvers. Finally, these solvers are sorted by average solving time which establishes their order of execution on P. Along with the development of SUNNY, it has been also extended to solve Constraint Optimization Problems (COPs), and to enable the parallel execution of its solvers. The resulting portfolio solver, called `sunny-cp` [10, 9], won the gold medal in the Open Track of the Minizinc Challenge [134]—the yearly international competition for CP solvers—in 2015, 2016, and 2017 [14].

In 2015, SUNNY was extended to deal with general AS problems (for which CP problems are a particular case) [5]. The resulting tool, called `sunny-as` [15], natively handles ASlib scenarios and was therefore submitted to the 2015 ICON Challenge on Algorithm Selection [86] to be compared with other AS systems. Unfortunately, the outcome was not satisfactory: only a few competitive results were achieved by `sunny-as`, that turned out to be particularly weak on SAT scenarios. We therefore tried to improve SUNNY by following two paths: *(i)* feature selection, and *(ii)* neighborhood size configuration.

Feature selection (FS) is a well-know process consisting in removing redundant and potentially harmful features from the feature vectors. A good feature selection can lead to significant performance gains of a prediction system. FS approaches can be distinguished in two main categories: wrappers and filters [69]. Filter methods work as a pre-processing step; they select features by using some scoring function (e.g., statistical tests) independent of the chosen predictor. In contrast, wrapper methods use the prediction system of interest as a black-box to assess the predictive power of selected features. As a result, wrapper methods have a higher computational cost; the features found could be more accurate than those found by filter methods. In the ICON challenge, a version of `sunny-as` used a simple filter method based on information gain that however did not bring significant benefits.

The neighborhood size configuration (shortly, k-configuration) consists in choosing an optimal value k for the k-nearest neighbors algorithm on which SUNNY relies. The work in [94] suggests that the performance of SUNNY can be improved by training and tuning the neighborhood size k on different scenarios.

After performing several studies on different AS scenarios, we developed `sunny-as2`:

an extension of `sunny-as` which combines techniques for the k-configuration, and the feature selection based on the wrapping methods. In 2017, `sunny-as2` was submitted to the *Open Algorithm Selection Challenge* (OASC), a revised edition of the 2015 ICON challenge. Thanks to the new enhancements, `sunny-as2` obtained much better results [93]: it reached the overall third position and, in particular, it was the approach achieving the best runtime minimization (i.e., the goal for which SUNNY was originally designed).

In this work, we detail the technical improvements of `sunny-as2` and we show their impact on the benchmark scenarios of the 2017 OASC competition. The technical improvements include: *(i)* the design of a surrogate function which makes feasible the evaluation of wrapper-based feature selection; *(ii)* the development of a training approach that orthogonally combines the feature selection and the k-configuration. We also empirically discovered that, by selecting a small number of representative instances for training the training speed gets improved without altering too much the prediction performance.

5.2 Related work

Algorithm Selection (AS) aims at identifying on per-instance basis the relevant algorithm, or set of algorithms, to run in order to enhance the problem-solving performance. The study of AS problems has attracted great attention in the SAT community and portfolio-based solutions won SAT competitions for years. For instance, SATZilla won the SAT Challenge from 2007 to 2010 and 2012, 3S and CSHC won gold medals in 2011 and 2013 respectively.

SATzilla [150] relies on runtime prediction models. Its latest version [149] uses a weighted random forest approach provided with a cost-sensitive loss function for punishing misclassifications in direct proportion to their performance impact. 3S [82] conjugates a fixed-time static solver schedule (computed off-line) with the dynamic selection of one long-running solver. This solver is chosen with a k-NN algorithm and is eventually executed after the static schedule. CSHC [102] clusters instead instances aiming to reduce the error of misclassification. Given a test instance to solve it will decide to which cluster it belongs and a best performance solver for that cluster is delegated to solve the test instance. Similarly to 3S, also CSHC has a static schedule in the pre-solving step.

Besides the comparisons in the SAT and CSP settings [12], some of these approaches have been abstracted to work on other scenarios. Following the release of ASlib, for instance, [82] proposed Aspeed as a variant of 3S where the per-instance long-running solver selection has been replaced by a solver schedule. Consequently, in [94] Lindauer et al. released ISA which further improved Aspeed by introducing an optimization objective "timeout-minimal" in their schedule generation.

Apart from `sunny-as2`, the OASC 2017 challenge [18], included three more contestants, each of which coming with two submissions. The system AS-ASL [103] uses a greedy wrapper-based feature selection and their AS selector as evaluator to filter relevant features for their own system. Then, they train their system differently in different submission: AS-ASL uses ensemble learning model while AS-RF uses the random forest. A final schedule is built on the trained model.

Cameron et al. in [29] proposed *ZILLA as an improved version of ZILLA who won the first place in the 2015 ICON challenge. They added functions such as solver sub-sampling, presolving, feature group selection and Hyper-parameter tuning to ZILLA, where ZILLA is built on random forest technique. The winner of the OASC competition is instead ASAP [61]. The ASAP algorithm selector still employs Random Forest but they iterate the optimization of the pre-scheduler and the algorithm selector which yield a more robust and elaborated solution schedule. One thing in common among these three approaches is that all of them attempt to solve an unseen problem instance by fixed solver(s) before AS process. The solver AS-ASL selects a single solver while ASAP and *ZILLA define a static solver schedule.

Among the approaches that did not participate in the OASC challenge we mention [110], which considers the AS Problem as a Recommendation Problem by using the well-known technique of Collaborative Filtering. Its performance is similar to the initial version of `sunny-as`. In [100] an approach is proposed to transform a text-encoded instance into a 2-D image which will then be processed by a Deep Neural Network system. Their model enabled the Deep Neural Network to find out (and also generate) relevant features for Algorithm Selection. Preliminary experiments are quite encouraging even though this approach still lags behind w.r.t. state-of-the-art approaches who are using crafted features.

5.3 Preliminaries

In this section we formalize the Algorithm Selection Problem, and we describe the SUNNY algorithm on which `sunny-as` and `sunny-as2` rely.

5.3.1 Algorithm Selection Problem

We can define an AS scenario as a triplet $(\mathcal{I}, \mathcal{A}, m)$ where: $\mathcal{I}$ is a set of *instances*, $\mathcal{A}$ is a set (or portfolio) of *algorithms* (or solvers), and $m : \mathcal{I} \times \mathcal{A} \to \mathbb{R}$ is a *performance metric*. The algorithm selection problem [123] consists in building a mapping $s : \mathcal{I} \to \mathcal{A}$ such that the overall performance $\sum_{i \in \mathcal{I}} m(i, s(i))$ is minimized. We can see s as an algorithm selector that, for each instance i, aims to predict the best algorithm $A = s(i)$ for instance i.

Since for many scenarios the performance metric m on $\mathcal{I}$ is (partially) known, we can validate the performance of s by partitioning $\mathcal{I}$ into a training set $\mathcal{I}_{tr}$ and a test set $\mathcal{I}_{ts}$: the selector s is trained on the instances of $\mathcal{I}_{tr}$, and evaluated on $\mathcal{I}_{ts}$ by computing $\sum_{i \in \mathcal{I}_{ts}} m(i, s(i))$.

Since the problem instances of $\mathcal{I}$ are typically hard to solve, often a solving timeout τ is set, so that $m(i, A) \leq \tau$ for each $i \in \mathcal{I}, A \in \mathcal{A}$. Some evaluation systems give an additional penalty if $m(i, s(i)) = \tau$; for example, the Penalized Average

Runtime (PAR) score with penalty $\lambda > 1$ is given by $\mathsf{PAR}_\lambda = \dfrac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} m'(i, s(i))$ where:

$$m'(i, A) = \begin{cases} m(i, A) & \text{if } m(i, A) < \tau \\ \lambda \times \tau & \text{otherwise.} \end{cases}$$

Typically, per-instance AS frameworks characterize each instance $i \in \mathcal{I}$ with the corresponding feature vector $\mathcal{F}(i) \in \mathbb{R}^n$, and the selection of the best algorithm A for i is actually performed according to $\mathcal{F}(i)$ (i.e., $A = s(\mathcal{F}(i))$). The feature selection process enables to consider smaller feature vectors $\mathcal{F}'(i) \in \mathbb{R}^m$, derived from $\mathcal{F}(i)$ by projecting a number $m \leq n$ of its features.

Clearly, the AS framework can be arbitrarily extended. For example, we can generalize s in order to select a *schedule* of solvers of $\mathcal{A}$, instead of a single solver $s(i) \in \mathcal{A}$. As we shall see, this is the strategy used by SUNNY.

5.3.2 Feature Selection

The process of deriving a smaller feature vector $\mathcal{F}'(i) \in \mathbb{R}^m$ from a larger one $\mathcal{F}(i) \in \mathbb{R}^n$ with $m \leq n$ is known as feature selection (FS). The purpose of such process is simplifying the prediction model, lowering the training and feature extraction costs, and hopefully improving the prediction accuracy.

FS techniques [68] consists basically of a combination of two components: a search technique for finding good subsets of features, and an evaluation function to score such subsets. Since exploring all the possible subsets of features is computationally intractable for non-trivial feature spaces, heuristics are employed to guide the search of the best subsets. Greedy search strategies usually come in two flavors: forward selection and backward elimination. In forward selection, features are progressively incorporated into larger and larger subsets. Conversely, in backward elimination features are progressively removed starting from all the available features. Combination of these two techniques, genetic algorithms, or local search algorithms such as simulated annealing are also used.

FS approaches can be distinguished in mainly two categories: wrappers and filters. Filter methods select the features on the basis of features' correlation with statistical indicators; [1] Regardless of the model of user's machine learning system, filter methods are particularly efficient and robust to overfitting. In contrast, wrappers evaluate subsets of features based on their correlation with the performance of user's machine learning system. Wrappers methods can be more accurate than filters, but have two main disadvantages in consequence: they are more exposed to the overfitting risk, and they have a much higher computational cost.

In this work we focus on wrapper methods only. We refer the interested readers to [4] to know more about SUNNY with filter methods.

5.3.3 SUNNY and sunny-as

SUNNY is based on the k-nearest neighbors (k-NN) algorithm and embeds built-in heuristics for schedule generation. Despite the original version of SUNNY handled CSPs only, here we describe its generalized version — the one we used to tackle general ASlib scenarios.

Let us fix the set of instances $\mathcal{I} = \mathcal{I}_{tr} \cup \mathcal{I}_{ts}$, the set of algorithms $\mathcal{A}$, the performance metric m, and the runtime timeout τ. Given a test instance $x \in \mathcal{I}_{tr}$, SUNNY produces a sequential schedule $\sigma = [(A_1, t_1), \dots, (A_h, t_h)]$ where algorithm $A_i \in \mathcal{A}$ runs for t_i seconds on x and $\sum_{i=1}^{h} t_i = \tau$. Such a schedule is obtained as follows. First, SUNNY employs k-NN to select from $\mathcal{I}_{tr}$ the subset I_k of the k instances closer to x according to the Euclidean distance computed on the feature vector $\mathcal{F}(x)$. Then, SUNNY uses three heuristics to compute the schedule σ: *(i)* H_{sel}, for *selecting* the most effective algorithms $\{A_1, \dots, A_h\} \subseteq \mathcal{A}$ on the set I_k; *(ii)* H_{all}, for *allocating* to each $A_i \in \mathcal{A}$ a certain runtime $t_i \in [0, \tau]$ for $i = 1, \dots, h$; *(iii)* H_{sch}, for *scheduling* the sequential execution of the algorithms according to their speed in the selected instances I_k.

[1]The statistical indicators for FS include, for instance, Pearson's Correlation, Linear Discriminant Analysis, Chi-Square, etc [69].

Table 5.1: Runtimes (in seconds). τ means the solver timeout.

	x_1	x_2	x_3	x_4	x_5
A_1	τ	τ	**3**	τ	**278**
A_2	τ	**593**	τ	τ	τ
A_3	τ	τ	**36**	**1452**	τ
A_4	τ	τ	τ	**122**	**60**

The heuristics H_{sel}, H_{all}, and H_{sch} are based on performance metric m, but depend on the application domain. For CSPs, H_{sel} selects the smallest set of algorithms $S \subseteq \mathcal{A}$ that solves the most instances in I_k, by using the runtime for breaking ties. H_{all} allocates to each $A_i \in S$ a time t_i proportional to the instances that S can solve in I_k, by using a special *backup solver* for covering the instances of I_k that are not solvable by any solver. Finally, H_{sch} sorts the solvers by increasing solving time in I_k. For Constraint Optimization Problems the approach is similar, but different evaluation metrics are used [13]. For more details about SUNNY we refer the interested reader to [10, 13], below we show Example 1 illustrating how SUNNY works on a given CSP.

Example 1 *Let x be a CSP, $\mathcal{A} = \{A_1, A_2, A_3, A_4\}$ a portfolio, A_3 the backup solver, $\tau = 1800$ seconds the solving timeout, $I_k = \{x_1, ..., x_5\}$ the $k = 5$ neighbours of x, and the runtimes of solver A_i on problem x_j defined as in Table 5.1. In this case, the smallest set of solvers that solve most instances in $N(x, k)$ are $\{A_1, A_2, A_3\}$, $\{A_1, A_2, A_4\}$, and $\{A_2, A_3, A_4\}$. The heuristic H_{sel} selects $S = \{A_1, A_2, A_4\}$ because these solvers are faster in solving the instances in I_k. Since A_1 and A_4 solve 2 instances, A_2 solves 1 instance and x_1 is not solved by any solver, the time window $[0, \tau]$ is partitioned in $2+2+1+1 = 6$ slots: 2 assigned to A_1 and A_4, 1 slot to A_2, and 1 to the backup solver A_3. Finally, H_{sch} sorts the solvers by increasing solving time. The final schedule produced by SUNNY is therefore $\sigma = [(A_4, 600), (A_1, 600), (A_3, 300), (A_2, 300)]$.*

Note that by default SUNNY does not perform any feature selection: it simply removes all the features that are constant over each $\mathcal{F}(x)$, and scales the remaining features into the range $[-1, 1]$ (scaling features is important for algorithms based on k-NN). The default neighborhood size is $k = \sqrt{\mathcal{I}_{tr}}$. The backup solver is the solver $A^* \in \mathcal{A}$ minimising the sum $\sum_{i \in \mathcal{I}_{tr}} m(i, A^*)$.

The `sunny-as` [5] tool implements the SUNNY algorithm to handle generic AS scenarios of the ASlib. In the optional pre-processing phase, performed offline, `sunny-as` can perform a feature selection based on different filtering methods and select a pre-solver to be run for a limited amount of time. At runtime, it produces the schedule of solvers by following the approach explained above.

5.3.4 2017 OASC challenge

In 2017, the COnfiguration and SElection of ALgorithms (COSEAL) group [65] has organized the Open Algorithm Selection Challenge to compare the different algorithm selectors available.

The challenge is built upon the Algorithm Selection library (ASlib) [23] that presents different algorithm selection scenarios. ASlib distinguishes between two types of scenarios: runtime scenarios and quality scenarios. In runtime scenarios the goal is to minimize the runtime of selected solver(s) for solving all instances (e.g., decision problems). The goal in quality scenarios is instead to find the algorithm that obtains the highest score according to some metric (e.g., optimization problems). One of the main difference between the two types of approaches is that runtime scenarios allow easily the computation of the results of a combination of solvers while this is not possible for the quality scenarios. Indeed, ASlib does not contain the partial results of the runs of the algorithms, thus making impossible to reconstruct ex-post the final result of an interleaved execution of them. For this reason, for the OASC it was possible to have selector proposing a schedule of solvers only for runtime scenarios.

The 2017 OASC consists of 11 scenarios: 8 runtime and 3 quality scenarios. Differently from the previous ICON challenge for Algorithm Selection held in 2015, the OASC used scenarios from a broader domain which come from the recent international competitions on CSP, MAXSAT, MIP, QBF, and SAT. In the OASC,

Scenario	Source	Algorithms (m)	Problems (n)	Features (d)	Timeout (τ)
Caren	CSP-MZN-2016	8	100	95	1200 s
Mira	MIP-2016	5	218	143	7200 s
Magnus	MAXSAT-PMS-2016	19	601	37	1800 s
Monty	MAXSAT-WPMS-2016	18	630	37	1800 s
Quill	QBF-2016	24	825	46	1800 s
Bado	BNSL-2016	8	1179	86	2880 s
Svea	SAT12-ALL	31	1614	115	480 s
Sora	SAT03-16 INDU	10	2000	483	5000 s

Table 5.2: OASC Scenarios.

each scenario is evaluated by one pair of training and test set replacing the 10-fold cross validation of the ICON challenge. The participants had access to performance and feature data on training instances (2/3 of the total), and only the instance features for the test instances (1/3 of the total).

In this work, due to the fact that SUNNY produces a schedule of solver not usable for quality scenarios, we focus only on runtime scenarios. An overview of them with their number of instances, algorithm, features, and timeouts is available in Table 5.2.

The OASC results show that `sunny-as2` outperformed the other competitors for the runtime scenarios. For the detailed competition report, we refer the interested readers to [18, 93].

5.4 sunny-as2

`sunny-as2` is the evolution of `sunny-as` and the solver that attended the 2017 OASC competition. The most significant innovations of `sunny-as2` are the introduction of wrapper FS methods, and the automatic k-configuration. Based on training data, `sunny-as2` automatically selects the most relevant features and/or the most performing value of the neighborhood parameter k to be used for online prediction. To improve configuration accuracy and stability, `sunny-as2` relies on cross-validation [84] for off-line training which splits the training data into mutual exclusive folds, then considers each fold in turn as test dataset and the rest as training set to assess the quality of parameter setting [2].

The importance of feature selection and parameters configuration for SUNNY has been shown in the empirical experiments conducted in [94, 4]. In particular, [4] shows the benefits of a proper feature selection, while [94] shows that parameters like the schedule size $|\sigma|$ and the neighborhood size k can have a substantial impact on the performance of SUNNY. In this regard, the authors introduced `TSUNNY`, a version of SUNNY that—by allowing the configuration of both $|\sigma|$ and k parameters—yielded a remarkable improvements over the original SUNNY.

Before introducing the different execution modalities of `sunny-as2`, in the following we will first describe the evaluation function that was used to evaluate the quality of a given parameter setting.

5.4.1 Evaluation function

The Evaluation Function, also known as Induction Function[84], is used to score a setting and guide the search of better parameter values. To evaluate the quality of a given set of settings, usually the tool under evaluation can be run on a relevant benchmark. In our case, however, the execution of SUNNY would have required too much time due to the way SUNNY selects the solvers to execute. Therefore, in order to be able to perform a quicker estimation of the quality of the settings, we have introduced a new simple variant of SUNNY that we called `greedy-SUNNY`, assuming that the quality of the parameters of SUNNY is correlated with the quality of parameters of `greedy-SUNNY`.

`greedy-SUNNY` differs from SUNNY in the way the set of solvers to execute is selected. Given a set $\mathcal{I}$ of the instances of the neighborhood, SUNNY computes the smallest set of solvers in the portfolio that can maximize the resolution of instances in $\mathcal{I}$. In the worst case this can take an exponential amount of time w.r.t. the number of solvers. To overcome this limitation, `greedy-SUNNY`, in a greedy approach,

[2]Different from `sunny-as2`, `sunny-as` had only a limited support for feature selection, and it only allowed the manual configuration of parameters.

starting from an empty set of solvers S adds one solver at the time to S by selecting the solver that is able to solve the largest number of instances in $\mathcal{I}$. The instances solved by the selected solver are then removed from $\mathcal{I}$ and the process repeated until a given number λ of solvers is added to S or no more instances need to be solved (i.e., $\mathcal{I} = \emptyset$). The value of λ is fixed externally by the user but, based on some empirical experiments, its default value was set to a small value (e.g. 3) as also suggested by the offline validations in [94].

With the usage of `greedy-SUNNY`, given a benchmark of training instances $\mathcal{I}_{tr}$ and testing instances $\mathcal{I}_{ts}$ it is possible to assign to the SUNNY settings a score representing its quality. In our case we decided to assign to the set of settings S the PAR_{10} score (c.f. Section 5.3) obtained by executing the schedule produced by `greedy-SUNNY` on the testing instances $\mathcal{I}_{ts}$ by using the training instance $\mathcal{I}_{tr}$. Where cross-validation is applied, the average score was obtained by averaging the score obtained by considering the different training sets folds.

With a little abuse of notation, in the following, we denote with `greedy-SUNNY` both the new evaluation function and the schedule generator on concrete instances.

5.4.2 `sunny-as2` and its execution modalities

`sunny-as2` provides different execution modalities depending on how the configuration of its parameters is conducted. The configuration procedure uses the training instances contained in each scenario. This is done in two phases: data preparation and parameter configuration.

Data preparation. The training instances are selected and split in 10 folds for cross validation by performing the following four steps: 1) each training instance is associated to the solver that solves it in the shortest time; 2) for each solver, the list of its associated instances is ordered from the hardest to the easiest (in terms of time needed to solve them); 3) we select one instance at a time from each set associated to each solver until a global limit on the number of instances is reached; 4) the selected instances are divided into 10 folds for cross validation.

In the first step, if an instance cannot be solved by any of the available solvers it will be discarded as commonly done in the AS community. For the fourth step creating the 10 folds, `sunny-as2` offers two choices: stratified split and random split [84]. The stratified split guarantees that for each label, each fold contains roughly the same percentage of instances associated with that label. The random split instead simply partitions the instances randomly into folds.

Parameter configuration. In order to compare different parameter settings for SUNNY and understand which have more impact on the performance we consider three modalities for `sunny-as2`: k-configuration, wrapper-based FS, and an hybrid system. These are described below.

1. **`sunny-as2-k`**. In this case, we use all the instance features and configure only the neighborhood size value k by considering values in the range $[1, n]$ where n is an external parameter set by the user (default value 80). The best value of k is chosen.

2. **`sunny-as2-f`**. In this case the neighborhood size k is set in the default way of SUNNY (square root of total number of instances) but a wrapper-based feature selection using `greedy-SUNNY` is used to evaluate the quality of a set of features. Iteratively, starting from an empty set of features, `sunny-as2-f` adds to the set of already selected features the tested feature which better decreases the PAR_{10} on the training instances. The iteration stops when the PAR_{10} increases or reaches a given limit of iterations.

3. **`sunny-as2-fk`**. This is a combination of `sunny-as2-f` and `sunny-as2-k` where both the neighborhood size parameter and the set of selected features are configured. More precisely, the procedure **`sunny-as2-f`** is run with different values of k in the range $[1, n]$. The k with the lowest PAR_{10} is then identified. The entire procedure is repeated until the addition of a feature with k varying in $[1, n]$ does not improve the PAR_{10} score or a given limit of iterations is reached. The resulting feature set and k value are chosen for the online prediction.

Before concluding the section, as a summary, the parameters used by `sunny-as2` that have to be decided by the user and are not learnt automatically are the following ones.

1. **split mode:** the mode to create folds for validation which includes random split and stratified split. Default: random.
2. **training instances limit:** the maximum number of instances used in training. Default: 1200.
3. **feature limit:** the limit of features for feature selection, used by `sunny-as2-f` and `sunny-as2-fk`. Default: 5.
4. k **range:** the range of neighborhood size used by `sunny-as2-f` and `sunny-as2-fk`. Default: [1,30].
5. **schedule limit for training (λ):** the limit of schedule size for `greedy-SUNNY`. Default: 3.

By tuning the values, in particular of the first three parameters, the training time can be controlled. A larger number of training instances, a bigger set of features, and a bigger size of the neighborhood increase the running times.

5.5 Empirical Validation

In this section we show the experiments run on the runtime scenarios of OASC benchmark in order to compare various execution modalities and parameter settings of `sunny-as2`. In particular, in the first part, we use `sunny-as2-fk` as baseline to understand the effect of the basic internal parameter values. We examine in sequence: i) the split modes for cross validation, ii) the limit on the numbers of features to select, iii) the limit on the number of training instances, iv) the schedule limit λ, and v) the differences between the SUNNY and `greedy-SUNNY` metrics for training and prediction. Then we compared the different execution modalities of `sunny-as2` as defined in the previous section (`sunny-as2-k`, `sunny-as2-f`, and `sunny-as2-fk`).

To evaluate the quality of an AS system, we used the conventional indicator denoted as *closed gap* [18]. Assuming that m_{VBS} is the performances (in terms of PAR_{10} score) of the Virtual Best Solver (i.e., the oracle solver which always chooses the best solver for each instance, m_{best} is the performance of the best solver across all the test instances, and m_c is the performance of the solver under consideration, the closed gap of the system considered is defined as:

$$\frac{m_c - m_{best}}{m_{VBS} - m_{best}}$$

A good AS system will have a performance m_c close to the virtual best solver m_{VBS}, which leads the closed gap score to be closer to 1. On the contrary, a bad performance consist in having m_c close to the single best solver m_{best}, thus making the close gap close to 0 if not even lower.

In the following, the best closed gap scores will be marked with a bold font. The experiments are conducted on a Linux machines equipped with Intel Core$i$5 3.30GHz processors and 8 GB of RAM.

5.5.1 Stratified vs Random Cross Validation

We start by showing the effects of the different cross validation used to train the data. Table 5.3 compares different cross validation choices for all the 8 scenarios of the OASC challenge involving runtime minimization.

	Caren	Magnus	Monty	Mira	Sora	Quill	Svea	Bado	Average
random*	0.6649	0.5678	0.9081	-0.4423	**0.3163**	0.6799	0.6205	0.7891	0.513
random	**0.9798**	**0.5889**	**0.9721**	**0.0539**	0.2299	0.669	**0.6374**	0.8078	**0.6174**
stratified	0.7889	0.5789	0.4149	-0.0053	0.3139	**0.7297**	0.6211	**0.9026**	0.5431

Table 5.3: Random split Cross Validation vs. Stratified Cross Validation (bold font indicates the best score).

For this experiments we set the internal parameters of `sunny-as2-fk` to the default ones (c.f. 5.4.2) except the split mode one. The three split modes we examined are: *random**, *random* and *stratified*. Both *random** and *random* generate folds in a random way with the only difference that random eliminates all the unsolvable training instances while random* preserves the whole instance set. The *stratified* mode first eliminates the unsolved instances and then generates folds based on class label (fastest algorithm).

The result shows that none of the split modes dominates the others for all the possible scenarios. Overall, after removing the unsolved instances, the random splitting provides a generally better performance: it achieved an average closed gap score of 0.6174 against 0.5431 for the stratified split and 0.513 of the random split without instance elimination.

5.5.2 Number of Training Instances

We studied the impact of the number of training instances on the performances. As before, we use the default parameter values listed in Sec. 5.4.2, just varying the limit of training instances.

Scenario	#inst	50	100	150	200	300	400	500	600	700	800	900	1000	1100	All
Caren	66	0.7879													0.9798
Mira	145	-0.8715	-1.3671												0.0539
Magnus	400	0.4897	0.5054	0.5087	0.5871	0.5849	0.5889								0.5889
Monty	420	0.6343	0.9091	0.9123	0.4803	0.9141	0.9721								0.9721
Quill	550	0.6773	0.7096	0.7436	0.4287	0.6405	0.9043	0.9031							0.669
Bado	786	0.5798	0.4844	0.7872	0.743	0.8258	0.7468	0.7903	0.7677	0.7485					0.8078
Svea	1076	0.5888	0.5522	0.6109	0.568	0.4963	0.4876	0.5434	0.4968	0.5629	0.6017	0.5688	0.6064		0.6374
Sora	1333	0.2924	0.007	0.0765	0.1871	0.1659	0.2961	0.2104	0.1675	0.2349	0.3832	0.1024	0.3809	0.2233	0.2299
Average		0.3973	0.3476	0.5841	0.5035	0.5827	0.6287	0.6302	0.587	0.6013	0.6321	0.5928	0.6324	0.6165	0.6174
Time (h)		0.6502	1.16	1.9	2.91	5.27	9.17	13.24	18.23	23.38	28.61	34.38	41.45	48.12	

Table 5.4: Training results varying number of training instances.

We run `sunny-as2-fk` with different instance limit from 50 to 1100. The results are described in Tab. 5.4 where the first column mentions the scenario name, the second column contains the number of available training instances in each scenario (after eliminating unsolvable ones), and the last column reports the results calculated considering all the training instances. All the other columns contain the results generated considering the fixed number of instances indicated in the first row. The second to last row reports the average closed gap score across all scenarios,[3] and the last row provides the aggregated CPU time that the training procedure took.

Based on the average closed gap score, we can see that by lowering the number of instances the system performance in terms of closed gap score does not alter significantly. With more than 150 instances, the score oscillates around 0.60. The peak score 0.6331 is obtained with 400 instances. In this case, the CPU time used for training is 9.17 hours which is almost 5 times faster than training with the total instances at our disposal (51.87 hours).

After 400 instances, increasing the number of training instances does not improve significantly the `sunny-as2-fk` performance. We conjecture that this is partially due to the procedure for the selection of instances (cf. data preparation in Section 5.4.2) that picks the instances after they have been stratified in classes, thus reducing their skewness. The number of instances is large enough to form an homogeneous set that reflects the instance class distribution of the entire scenario even after a random or stratified split.

Table 5.5 shows the neighborhood size value k selected by `sunny-as2-fk` during the training by varying the number of training instances. Interestingly enough, we can see that all the scenarios use a reasonably small value for k and that the larger value of k is reached with 400 or less training instances. This means that the small number of good quality instances are enough to maintained the prediction accuracy for the considered scenarios.

5.5.3 Limit on the Number of Features

Since we know that a small number of features are enough to provide a competitive performance of an AS system [4, 23], we now try to pinpoint a good value for the limit on the number of features on which our system should rely.

We first run the experiments with the whole set of training instances as a baseline and then we reduced the number of training instances in order to understand if there

[3]In case the scenario has less feature than needed, we considered for computing the average score the result obtained using all the features as stated in the final column of Table 5.4.

Scenario	50	100	150	200	300	400	500	600	700	800	900	1000	1100	All
Caren	7													14
Mira	4	4												17
Magnus	8	3	9	9	10	8								8
Monty	3	3	4	4	8	10								10
Quill	17	15	22	29	24	29	28							22
Bado	3	4	4	4	15	17	6	9	14					9
Svea	3	3	3	3	3	3	3	3	7	9	3	7		6
Sora	3	3	3	5	3	7	3	6	9	7	11	15	15	15

Table 5.5: Neighborhood size k by varying number of training instances.

was some pattern between the number of features and the number of the training instances.

Scenario	1	2	3	4	5	6	7	8	9	10
Caren	0.5869	0.5981	0.9727	0.9798	0.9798	0.9798	0.9798	0.9798	0.9798	0.9798
Magnus	0.3432	0.585	0.589	0.5889						
Monty	0.4711	0.764	0.9794	0.9814	0.9721					
Mira	-0.8743	-0.8622	0.0569	0.0569	0.0539	0.0539	0.0539	0.0539	0.0539	0.0539
Sora	0.2307	0.2657	0.3343	0.3287	0.2299	0.2103	0.2546	0.2754	0.2768	0.2739
Quill	0.4581	0.6358	0.58	0.6397	0.669	0.728	0.7296	0.7078		
Svea	0.4132	0.5174	0.5649	0.6377	0.6374	0.6368	0.662	0.6538	0.6377	0.637
Bado	0.6453	0.7617	0.7984	0.8078						
Average	0.2843	0.4082	0.6095	0.6276	0.6174	0.6222	**0.6311**	0.6299	0.6281	0.6277

Table 5.6: Performance change varying feature limit with entire set of training instances.

With the default parameter values specified in Sec. 5.4.2 except the feature limit, we run `sunny-as2-fk` on all the training instances with feature limits from one to ten and list the results in Table 5.6. The first row shows the different limits set to the feature cardinality while the last row shows the average closed gap score aggregated for each scenario. The scenario names are listed in the first column. Note that some of the values in the columns are left blank to indicate that the feature limit for that specific scenario has not been reached. This is due to the greedy procedure of sunny-as2-f that adds a feature to the set of considered features only if the addition decrease the PAR_{10} score.

As expected, for several scenarios, despite the training cost decreases by adding a new feature, its highest performance was reached with less features. For instance Sora has a good performance with three selected features and adding a new feature does not increase the closing gap. This confirms that in some cases setting a small feature limit could improve the performance.

In general, the results show that the sunny-as2-fk performance improves with more than three features and the improvement stops when the number of feature is bigger than eight. The highest score considering the OASC challenge scenarios is achieved when the limit of features is set to seven.

We then tried to find out if there was a correlation pattern between the limit on the number of features and the limit on the training instances. As done for the results shown in Table 5.6, we considered the limit on training features from 50 to 1200 and the feature limit between 4 and 8. We report the average closed gap score of all the scenarios in each cell of Table 5.7.

As can be seen, when the number of training instances is bigger than 150, the score obtained with different feature limit does not change significantly. The best result is obtained considering a limit of 1000 training instances and 7 features (score 0.6401) that is however very close to the score obtained when the instance limit is set to 400 and the feature limit is set to 4 (average score 0.6398).

Since the difference between these best and second best score is very small but in terms of training time the the peak performance require more than 4 times the time taken by the runner up, in the following experiment we decide to adopt 400 for training instance limit and 4 for feature limit. This decision is done following the spirit of the previous ICON challenge that limited the training time to only 12 CPU hours and by the consideration that the training time is one of the obstacles that hinder the adoption of portfolio based solver in the real world.

Feat \ Inst	50	100	150	200	300	400	500	600	700	800	900	1000	1100	Average
4	0.4333	0.3872	0.5824	0.507	0.5784	**0.6398**	0.595	0.5923	0.6135	0.6233	0.5896	0.6185	0.6269	0.6276
5	0.3973	0.3476	0.5841	0.5035	0.5827	0.6331	0.6039	0.587	0.6013	0.6321	0.5928	0.6324	0.6165	0.6174
6	0.3785	0.3487	0.5988	0.5022	0.6127	0.6345	0.6029	0.5962	0.6315	0.6356	0.5924	0.6388	0.6238	0.6222
7	0.4467	0.4202	0.6086	0.5452	0.6109	0.6365	0.6221	0.5957	0.6291	0.6245	0.6039	**0.6401**	0.6364	0.6311
8	0.4467	0.4203	0.6086	0.5461	0.6104	0.637	0.6221	0.593	0.6324	0.6218	0.6011	0.6365	0.624	0.6299

Table 5.7: Average closed gap score: feature limit vs training instances limit.

Scenario, λ	size1	size2	size3	size4	size5	size6
Caren	0.9855	0.7907	0.9798	0.9798	0.9798	0.9798
Magnus	0.5041	0.5889	0.5889	0.5889	0.5889	0.5889
Monty	0.9101	0.9814	0.9814	0.9814	0.9814	0.9814
Mira	0.0264	0.0569	0.0569	0.0569	0.0569	0.0569
Sora	0.2765	0.3596	0.3596	0.3596	0.3596	0.3596
Quill	0.6521	0.6122	0.9042	0.6408	0.6408	0.6408
Svea	0.5688	0.4807	0.4807	0.4807	0.4807	0.4807
Bado	0.8111	0.7714	0.7669	0.7717	0.7717	0.7717
All	0.5918	0.5802	**0.6398**	0.6075	0.6075	0.6075

Table 5.8: Closed gap by varying the schedule size of `greedy-SUNNY`.

5.5.4 Schedule size λ for `greedy-SUNNY`

In the training procedure, `greedy-SUNNY` uses the parameter λ to limit the size of generated schedule and be faster than the SUNNY approach when computing the schedule of solvers. We have investigated what is a suitable λ value to use, when `greedy-SUNNY` is used for training.

Tab. 5.8 lists the closed gap score of **`sunny-as2-fk`** with different λ values. We set 400 as training instance limit, 4 as feature size limit, $k \in [1, 30]$ and we varied the schedule limit for training λ from one to six. By observing the average results for each λ value, the global peak performance was reached when λ is set to three. When λ is less than three, for most scenarios, the results are worse and when λ is bigger than three the performances are the same if not slightly worse except for one scenario only. As expected, this means that for the considered scenarios, the three best performing solvers are often sufficient to solve the most instances. As such, we set λ to three as default value for `greedy-SUNNY`.

5.5.5 `greedy-SUNNY` vs SUNNY

As previously described, `greedy-SUNNY` was introduced to speed up the training process, hoping that there was at least a correlation with its performance and the original version of SUNNY. Here, we empirically show that `greedy-SUNNY` can be used as a substitute for SUNNY for the training without big degradation of performance.

In the following experiments we use the default parameters changing the approaches used for generating the schedule of solvers in training and in testing using a time limit of a week. Results are reported in Tab. 5.9, where the column names denote the pairs of the function used for the training and testing respectively. For instance, the second column "sunny-gsunny" means that SUNNY has been used for training, and `greedy-SUNNY` for testing. Note that for the Svea scenario SUNNY takes more time than our time cup. This is reported in the table with the "Timeout" string.

The first thing that we can conclude by looking at the column "gsunny-sunny" and "gsunny-gsunny" is that when `greedy-SUNNY` is used for training, using SUNNY for testing is slightly better than using `greedy-SUNNY`. The difference is however very small. We believe that this is due to the fact that for the OASC scenarios only few solvers are enough to solve the majority of instances in the neighborhood. The fact that SUNNY considers all the solvers available therefore does not bring a big advantage. We conjecture that this is a property that good algorithm scenarios should have, providing to have also a good distance metric to evaluate the similarity of the different instances. If not, this would mean that the concept of similarity can not be used to relate the performance of solver over similar instances, thus hinder the possibility to create good selectors.

Surprisingly, by comparing the column "sunny-sunny" and "gsunny-sunny", we find that the results of "gsunny-sunny" are generally higher than "sunny-sunny" which means that `greedy-SUNNY` is better for training than SUNNY. This is a counter intuitive results since we were expecting that SUNNY was better also in training.

Apparently, the possibility of SUNNY to select more solver than what `greedy-SUNNY` has a negative effect of the training. We conjecture that this is probably due to the fact that `greedy-SUNNY` prioritizes for the selection the first solver, which is the most robust one solving more instances in the neighborhood. This may lead to the learning of more robust parameters later.

	sunny - sunny	sunny - gsunny	gsunny - sunny	gsunny - gsunny
Caren	0.9749	0.9717	0.9798	0.9682
Magnus	0.5821	0.5799	0.5889	0.5889
Monty	0.3757	0.3876	0.9814	0.9836
Mira	-0.351	-0.3336	0.0569	0.0564
Sora	0.2767	0.316	0.3596	0.3815
Quill	0.6991	0.7086	0.9042	0.8896
Svea	Timeout	Timeout	0.4807	0.4873
Bado	0.7892	0.768	0.7669	0.7609
All	0.4781	0.407	**0.6398**	0.6396

Table 5.9: Closed gap for different combinations of SUNNY and `greedy-SUNNY` for training and testing.

The performance of `greedy-SUNNY` is useful due to the fact that SUNNY is particularly slow to train scenarios with a large number of solvers. This can be seen in Table 5.10 that describes the hours spent for training using the different approaches. We run the training experiments with a time limit of a week and for this reason we omitted the result for the Svea scenario that based on our estimation would have taken 17000 hours to be completed. The average close gap is computed considering the available results. It is evident that `greedy-SUNNY` is quicker than original SUNNY for any scenarios.

Combining Tab. 5.9 and Tab. 5.10 we conclude that, for training, `greedy-SUNNY` works better than SUNNY in terms of both speed and the quality of configured parameters. For the testing, the two approaches are similar, with a non statistically significant advantage for SUNNY.

	Caren	Magnus	Monty	Mira	Sora	Quill	Svea	Bado	Average*
gsunny	0.05	0.28	0.3	0.18	50.98	0.92	16.37	3.16	7.9814
sunny	2.35	1.35	3.93	0.22	65.97	71.18	-	3.91	21.2729
# solvers	20	19	18	5	10	24	31	8	
# insts	66	400	420	145	1333	550	1076	786	

Table 5.10: Hours spent for training by various evaluation functions.

5.5.6 Comparison of execution modalities

We conclude the experiment section by comparing the different execution modalities of `sunny-as2-f`, `sunny-as2-k`, and `sunny-as2-fk` with the original version of SUNNY that did not exploit any parameter configuration.

	Caren	Magnus	Monty	Mira	Sora	Quill	Svea	Bado	All
sunny	0.3942	0.5857	0.3992	-0.8996	0.1674	0.7697	0.4866	0.7687	0.334
sunny-as2-f	0.7919	**0.6598**	0.3028	-0.4644	0.2076	0.6481	**0.5575**	**0.848**	0.4439
sunny-as2-k	0.7788	0.506	0.5548	0.0103	0.1735	0.8508	0.4866	**0.848**	0.5261
sunny-as2-fk	**0.9798**	0.5889	**0.9814**	**0.0569**	**0.3596**	**0.9042**	0.4807	0.7669	**0.6398**

Table 5.11: Comparisons of `sunny-as2` basic modalities.

Based on the previous results, we used `greedy-SUNNY` for training and SUNNY for testing, 400 as the instance limit for training, and set all other parameters to their default values. The results are listed in Tab. 5.11.

`sunny-as2-fk` yields the best results for 5 scenarios and has the best average closed gap. The original SUNNY is evidently worse than any other execution modality. In addition, we also tried `greedy-SUNNY` for testing. The result's order does not change, and similar to Tab. 5.9, `greedy-SUNNY` is still slightly less competitive than SUNNY.

We would like to conclude by pointing out that we did plenty of experiments by enlarging the interval of search for the hyper-parameter k and the limits of training instances. However, the results show that there was not any significant improvements on the average closed gap score.

5.6 Chapter Summary

Algorithm Selection or Portfolio Approach has attracted a lot attention since it is found to be a powerful method to tackle NP-hard problems. Thanks to the ASlib, different Algorithm Selection techniques tailored in different domains are able to be compared fairly. In this work we presented `sunny-as2` that, by applying the wrapper-based feature selection with the configuration of the neighborhood size, was able to compete in the recent OASC challenge reaching the first position in the runtime minimization category.

As a future work, we are planning to improve `sunny-as2` targeting the solution quality scenarios of the OASC competition where, due to the fact that using schedulers of solver is not allowed, `sunny-as2` is strongly penalized. Another direction for future work is the investigation of the problem of overfitting that may happen due to the usage of the wrapper-based feature selection.

Chapter 6

Conclusions and future extensions

The user states the problem, the computer solves it.

Eugene Freuder

Constraint programming (CP) has been designed to help people express their needs better by isolating the solution procedure from the problem model. With this in mind, we have investigated the application of CP.

The study in this dissertation is split into two parts. In the first part, we applied CP to two specific problems: `NightSplit` for group activity optimization (Chapter 3) and the `SFC design problem` for network service function chain (Chapter 4). We formalized these problems and showed their complexity. Then, we compared different approaches to address them, highlighting the advantages and limitations of CP. In the second part, we discussed an AS technique called SUNNY. We described the advancements of SUNNY, which led it to become an award winner in the 2017 OASC challenge.

The original contributions of this dissertation are as follows.

- We proposed a model called `NightSplitter` for the (sub)group activity optimization.

- We implemented the solution for `NightSplit` using CP and an approximative approach (simulated annealing), as well as investigating the scalability issue of CP.
- We developed a web application for `NightSplit` to conduct practical studies.
- We proposed a framework and a formalization of the `SFC design problem`.
- We demonstrated that the CP solving technique is more efficient than that of MILP for the `SFC design problem`.
- We generalized SUNNY, a constraint solver selection technique, for general AS study using the ASlib benchmarks.
- We proposed improvements to SUNNY, thanks to which it won a prize in the Open Algorithm Selection Challenge.

It is worth mentioning that we have followed up on the `NightSplit` project. By conducting a number of market surveys and interviews with users and business experts, we have found that it is not easy to market the system to the public. Some users are extremely dynamic: they may change their schedules in different situations, and they are against a fixed schedule generated by a computer. There are also users who want to interact with the system in a more flexible way. Gathering friends' preferences and adjusting the system parameters is too complicated for them. They would prefer the system to have voice commands, like Siri or Google Assistant, so that they can reduce the time needed to learn about the tool and feel comfortable feeding their preferences and constraints into the system directly. Indeed, the feedback collected reflects some challenging questions regarding constraint programming, such as how to design a model with a higher degree of flexibility and how to use as little parameters as possible to lower the burden of user input. We suppose that these problems could be addressed partially with the help of a machine learning system that can observe the users' habits and identify the necessary parameter values needed by the CP system, such as the activity area, the group size, and the activity time window.

Comparatively, a constraint system for industrial problems (like the `SFC design problem`) seems to be less difficult for users to understand. The system users are familiar with the routines that they need to follow; they have a clear idea of what they should feed into the system and what they can expect to receive. The users mostly care about the quality of the solution and the response time of the system.

Regarding the performance of the two problems, `NightSplit` and the `SFC design problem`, we have witnessed that state-of-the-art constraint solvers indeed have different strengths when tackling problems in different categories. Chuffed is currently the fastest solver available for `NightSplit`, while or-tools is faster than Chuffed for the `SFC design problem`. These results could be useful in introducing new benchmark scenarios for ASlib. Furthermore, we would like to understand SUNNY's performance using our own case studies.

An interesting direction of SUNNY is to explore its cross-domain applications. The work [110] has successfully moved the collaborative filtering technique from the recommender system domain to the AS problem. Likewise, we can also do it for SUNNY, and we will understand its impact on more prediction and recommendation problems.

In recent years, numerous successes have been seen in AI in different domains. Although CP is one of the fields within AI that has a long history, a large number of AI enthusiasts, who are skilled in imperative languages and machine learning, are still unfamiliar with CP. The application of CP still has strong potential for many challenging problems that involve constraints. We believe that this dissertation will be useful for people who want to explore the strength of CP for solving their problems, improve the solving efficiency of CP with AS, and appreciate the versatility and effectiveness of CP.

Appendix A

Appendix

A.1 Approximation algorithm for `NightSplit`

The approximation approach of `NightSplit` relies on simulated annealing (SA). The pseudo-code 1 shows how SA has been applied to optimize an activity schedule. And the pseudo-code 2 explains how a schedule is modified at each step of SA.

Algorithm 1: Simulated Annealing

```
1  Function SA($steps$):
2      Initialize an activity $schedule$ to be optimized;
3      Initialize temperature $T$;
4      Initialize counter $step \leftarrow 0$;
5      while $step < steps$ do
6          $T \leftarrow \texttt{Tmax} \times exp(\texttt{Tfactor} * \frac{step}{steps})$;
7          $schedule' \leftarrow$ move();
8          if $score(schedule') > score(schedule)$ then
9              $schedule \leftarrow schedule'$;
10         else
11             $schedule \leftarrow schedule'$ with probability
                 $p(T, score(schedule'), score(schedule))$;
12         end
13         $step \leftarrow step + 1$;
14     end
15 return schedule
```

In the SA implementation, `Tmax` and `Tfactor` are internal parameters which regulate the rate of change of the temperature; the probability function $p()$ provides lower chance with the increase of the temperature T.

Algorithm 2: Move method of SA

```
1  Function move():
       Data: Activities,Users,Preferences,schedule
2      Initialize improvement indicator $gain \leftarrow 0$;
3      Initialize $activity \leftarrow []$;
4      Initialize $users \leftarrow []$;
5      while $gain \leq 0$ do
6          Randomly select an $actvity \in$ Activities;
7          Randomly select a subset of $users \in$ Users;
8          $gain \leftarrow$ Preferences($users, activity$);
9      end
10     $schedule \leftarrow$AssignActivityToUser($users, activity, schedule$);
11     $schedule \leftarrow$EliminateOldIncompatibleActivities($users, activity, schedule$);

12     for $u \in users \wedge schedule(u) = \emptyset$ do
13         $activity \leftarrow schedule(u')$ where $u' \in users \setminus u \wedge activity$ is compatible
            for $u$;
14         $schedule \leftarrow$AssignActivityToUser($u, activity, schedule$);
15     end
16 return schedule
```

The move function is used to generate a neighborhood solution based on a given one. In its content, the function eliminateOldIncompatibleActivities has been used to eliminate old activities that violate any of the three constraints: group size, number of groups and activity time. For more technical details, e.g., SA support functions and data structures, we refer the interested readers to [95].

A.2 Hardness of `SFC design problem`

Theorem A.1 (NP-hardness) *The optimal* `SFC design problem` *is NP-hard.*

Proof:

To prove hardness, we reduce the NP-complete problem k-MST [122] to the decision version of `SFC design problem`, i.e., finding whether there exists an admissible `SFCtree` $G(V', L')$ in which the cost function $\sum_{l \in L'} c_l$ is less than or equal to a given value.

An instance of the decision version of k-MST consists of a weighted graph $G(V, L)$, a number k and a number h. The problem is determining whether there exists a subgraph $G^*(V^*, L^*) \subseteq G$, such that $|V^*| = k$ and $\sum_{l \in L^*} c_l \leq h$. Given an instance of k-MST, nodes of V are mapped to the gateway VNFs of the SFC problem, with each gateway representing the presence of a unique domain. Furthermore, arcs of L are mapped to links that connect each pair of domain gateways. [1] Two more domains are introduced as source and target domains and one gateway is in each of them which links to all other gateways with 0 cost. One DPI VNF is created in each domain except for the source and target domains, and each DPI has a link to its own domain gateway. The user request is then introduced, in which the source and target domains are specified, prox_to_src and prox_to_dst are all set to 0 and the requested vnflist contains k DPI. It is assumed that the domain constraints are empty. The problem is finding an admissible `SFCtree` in which the total cost is less than or equal to h.

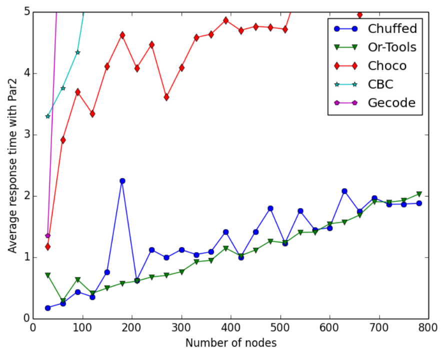
As each domain (except the source and target domains) has exactly one DPI, to satisfy the constraint of vnflist which demands k DPI, the k domain gateways with the total link cost $\leq h$ should be identified. It can be seen that if the instance of k-MST has a solution, the `SFC design problem` instance must have a solution; it is sufficient to link the source and target domain gateways to any nodes that belong to the solution of the k-MST instance. Conversely, given a solution to the `SFC design problem`, it is enough to extract both the selected gateways between the source and target domains and the gateway links in order to build a solution for the k-MST instance.

[1] For pairs of gateways where there is not a corresponding link in L, a link with highest cost is introduced.

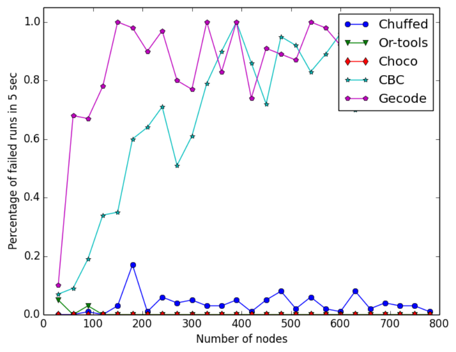
□

A.3 Extended experiments for `SFC design problem`

The solvers comparison with different number of nodes (Fig. A.1) gives similar results as the experiment with different number of domains (Fig. 4.5): The solver Or-Tools is faster and more stable than other constraint solvers.



(a) Response time with 10 domains

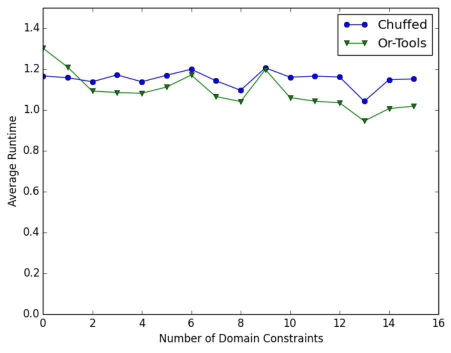


(b) Percentage of Failed Runs

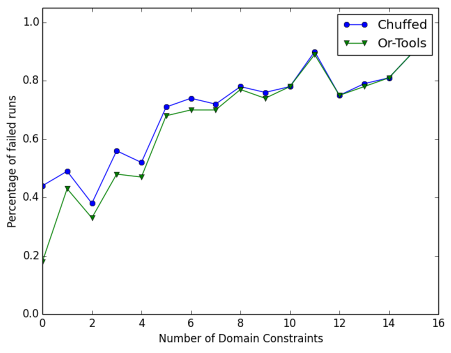
Figure A.1: Solvers Comparison varying number of nodes.

Fig. A.2 shows that increasing the number of domain constraints does not increase the runtime of Or-Tools and Chuffed. However, more domain constraints lead to have higher probability that no satisfiable solution exists; in consequence, the number of failed runs grows, and the runtime of solvers decreases slightly (Or-Tools is more sensible).

To complete the results described in Fig. 4.6, the Fig. A.3 shows the percentage of failed runs; as expected, the number of domains is the key parameter that influences solving efficiency.

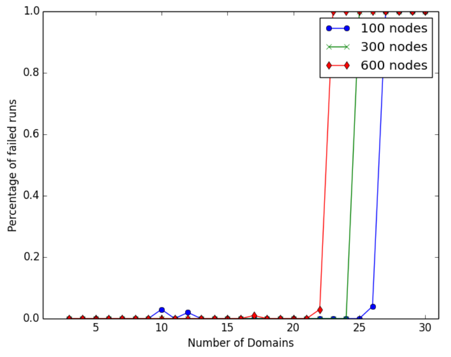


(a) Response time

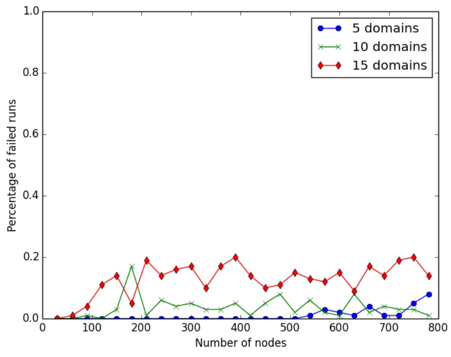


(b) Percentage of Failed Runs

Figure A.2: Solvers performance with 150 nodes and 15 domains varying the number of domain constraints.



(a) Percentage of failed runs varying the number of domains.



(b) Percentage of failed runs varying the number of nodes.

Figure A.3: Solvers performance with 150 nodes and 15 domains varying the number of domain constraints.

References

[1] Emile HL Aarts and Jan HM Korst. Simulated annealing. *ISSUES*, 1:16, 1988.

[2] AlloCiné. Allociné website, 2016. Available at `http://www.allocine.fr`.

[3] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[4] Roberto Amadini, Fabio Biselli, Maurizio Gabbrielli, Tong Liu, and Jacopo Mauro. Feature selection for SUNNY: A study on the algorithm selection library. In *ICTAI*, pages 25–32. IEEE Computer Society, 2015.

[5] Roberto Amadini, Fabio Biselli, Maurizio Gabbrielli, Tong Liu, and Jacopo Mauro. SUNNY for algorithm selection: a preliminary study. In *Proceedings of the 30th Italian Conference on Computational Logic, Genova, Italy, July 1-3, 2015.*, pages 202–206, 2015.

[6] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving CSPs. *CoRR*, abs/1212.0692, 2012.

[7] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Portfolio approaches for constraint optimization problems. In *Lion 8*, pages 21–35, 2014.

[8] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming*, 14(4-5):509–524, 2014.

[9] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. A Multicore Tool for Constraint Solving. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 232–238, 2015.

[10] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY-CP: a sequential CP portfolio solver. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1861–1867. ACM, 2015.

[11] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Why CP Portfolio Solvers Are (under)Utilized? Issues and Challenges. In *LOPSTR 2015*, volume 9527 of *LNCS*, pages 349–364. Springer, 2015.

[12] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An extensive evaluation of portfolio approaches for constraint satisfaction problems. *International Journal of Interactive Multimedia and Artificial Intelligence*, 2016.

[13] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Portfolio approaches for constraint optimization problems. *Annals of Mathematics and Artificial Intelligence*, 76(1-2):229–246, 2016.

[14] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY-CP and the MiniZinc challenge. *TPLP*, 18(1):81–96, 2018.

[15] Roberto Amadini and Jacopo Mauro. SUNNY-AS, 2015. Available at `https://github.com/CP-Unibo/sunny-as`.

[16] Sihem Amer-Yahia, Senjuti Basu Roy, Ashish Chawlat, Gautam Das, and Cong Yu. Group recommendation: Semantics and efficiency. *Proceedings of the VLDB Endowment*, 2(1):754–765, 2009.

[17] Rumen Andonov, Nicola Yanev, and Noël Malod-Dognin. An efficient lagrangian relaxation for the contact map overlap problem. In *International Workshop on Algorithms in Bioinformatics*, pages 162–173. Springer, 2008.

[18] Marius Lindauer at all. Open algorithm selection challenge 2017: Setup and scenarios. In Marius Lindauer et all., editor, *Proceedings of the Open Algorithm Selection Challenge*, volume 79 of *Proceedings of Machine Learning Research*, pages 1–7, Brussels, Belgium, 11–12 Sep 2017. PMLR.

[19] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford university press, 1996.

[20] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS99)*, pages 555–564. MatFyzPress Prague, 1999.

[21] Senjuti Basu Roy, Laks VS Lakshmanan, and Rui Liu. From group recommendations to group formation. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1603–1616. ACM, 2015.

[22] Daniel Le Berre, Pierre Marquis, and Stéphanie Roussel. Planning personalised museum visits. In *Proceedings of the Twenty-Third International Conference on International Conference on Automated Planning and Scheduling*, pages 380–388. AAAI Press, 2013.

[23] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, et al. ASlib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016.

[24] Ludovico Boratto and Salvatore Carta. State-of-the-art in group recommendation and new approaches for automatic identification of groups. In *Information retrieval and mining in distributed environments*, pages 1–20. Springer, 2010.

[25] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):353–387, 1981.

[26] Stephen Boyd and Lieven Vandenberghe. *Convex optimization.* Cambridge university press, 2004.

[27] Sally C Brailsford, Chris N Potts, and Barbara M Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

[28] Franco Callegati, Walter Cerroni, Chiara Contoli, Rossella Cardone, Matteo Nocentini, and Antonio Manzalini. SDN for dynamic NFV deployment. *IEEE Communications Magazine*, 54(10):89–95, 2016.

[29] Chris Cameron, Holger H. Hoos, Kevin Leyton-Brown, and Frank Hutter. Oasc-2017: *Zilla submission. In Marius Lindauer et all., editor, *Proceedings of the Open Algorithm Selection Challenge*, volume 79 of *Proceedings of Machine Learning Research*, pages 15–18, Brussels, Belgium, 11–12 Sep 2017. PMLR.

[30] Yves Caseau, François Laburthe, and Glenn Silverstein. A meta-heuristic factory for vehicle routing problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 144–158. Springer, 1999.

[31] Amedeo Cesta, Angelo Oddi, and Stephen F Smith. A constraint-based method for project scheduling with time windows. *Journal of Heuristics*, 8(1):109–136, 2002.

[32] Peter Chan, Kamel Heus, and Georges Weil. Nurse scheduling with global constraints in CHIP: Gymnaste. In *Proc of Practical Application of Constraint Technology (PACT98), London, UK*, 1998.

[33] Yuh-Rong Chen, Sridhar Radhakrishnan, Sudarshan Dhall, and Suleyman Karabuk. The service overlay network design problem for interactive internet applications. *Computers & Operations Research*, 57:73–82, 2015.

[34] Geoffrey Chu, Maria Garcia de la Banda, Chris Mears, and Peter J Stuckey. Symmetries and lazy clause generation. In *Proceedings of the 16th CP Doctoral programme*, pages 43–48, 2010.

[35] R. Cohen, K. Barabash, B. Rochwerger, L. Schour, D. Crisan, R. Birke, C. Minkenberg, M. Gusat, R. Recio, and V. Jain. An intent-based approach for network virtualization. In *Proc. IM'13*, pages 42–50. IEEE, 2013.

[36] Rackspace Cloud Computing. Openstack platform, 2018. `https://www.openstack.org/`.

[37] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[38] Douglas Crockford. *The application/json media type for JavaScript object notation (json)*, July 2006. DOI:10.17487/RFC4627.

[39] CSIRO Data61. MiniZinc challenge 2017, 2017. Available at `https://www.minizinc.org/challenge2017/results2017.html`.

[40] Bruno De Backer, Vincent Furnon, Paul Shaw, Philip Kilby, and Patrick Prosser. Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of Heuristics*, 6(4):501–523, 2000.

[41] Ayhan Demiriz, Kristin P Bennett, and John Shawe-Taylor. Linear programming boosting via column generation. *Machine Learning*, 46(1-3):225–254, 2002.

[42] Narsingh Deo. *Graph theory with applications to engineering and computer science*. Courier Dover Publications, 2017.

[43] Pierpaolo Di Bitonto, Francesco Di Tria, Maria Laterza, Teresa Roselli, Veronica Rossano, and Filippo Tangorra. A model for generating tourist itineraries. In *2010 10th International Conference on Intelligent Systems Design and Applications*, pages 971–976. IEEE, 2010.

[44] Sevil Dräxler, Holger Karl, and Zoltán Adám Mann. Joint optimization of scaling and placement of virtual network services. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 365–370. IEEE Press, 2017.

[45] Cynthia Dwork. Differential privacy: A survey of results. In *TAMC*, volume 4978 of *LNCS*, pages 1–19. Springer, 2008.

[46] Richard E Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1(1-2):27–120, 1970.

[47] Filippo Focacci, François Laburthe, and Andrea Lodi. Local search and constraint programming. In *Handbook of metaheuristics*, pages 369–403. Springer, 2003.

[48] Internet Engineering Task Force. Service function chaining (SFC) architecture, 2015. https://tools.ietf.org/html/rfc7665.

[49] COIN-OR Foundation. Coin OR, 2016. Available at https://www.coin-or.org.

[50] Eugene C. Freuder. Synthesizing constraint expressions. *Commun. ACM*, 21(11):958–966, 1978.

[51] Thom W. Frühwirth and Pascal Brisset. Optimal placement of base stations in wireless indoor telecommunication. In *CP*, pages 476–480, 1998.

[52] Francis Galiegue, Kris Zyp, et al. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, page 32, 2013.

[53] Hervé Gallaire. Logic programming: Further developments. In *SLP*, volume 85, pages 88–96, 1985.

[54] Inma Garcia, Laura Sebastia, and Eva Onaindia. On the design of individual and group recommender systems for tourism. *Expert systems with applications*, 38(6):7683–7692, 2011.

[55] John Gaschig. Performance measurement and analysis of certain search algorithms. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1979.

[56] Etienne Gauvin. Allocine helper, 2016. Available at `https://github.com/etienne-gauvin/api-allocine-helper`.

[57] Peter J. Stuckey Geoffrey Chu. Chuffed solver description, 2014. Available at `http://www.minizinc.org/challenge2014/description_chuffed.txt`.

[58] J. Gil Herrera and J.F Botero. Resource allocation in NFV: A comprehensive survey. *IEEE Transactions on Network and Service Management*, 13(3):518–532, 2016.

[59] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[60] Carla P Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

[61] Francois Gonard, Marc Schoenauer, and Michele Sebag. ASAP.V2 and ASAP.V3: Sequential optimization of an algorithm selector and a scheduler. In Marius Lindauer et all., editor, *Proceedings of the Open Algorithm Selection Challenge*, volume 79 of *Proceedings of Machine Learning Research*, pages 8–11, Brussels, Belgium, 11–12 Sep 2017. PMLR.

[62] Google. Google or-tools, 2016. Available at `https://developers.google.com/optimization/`.

[63] Google. Google Or-tools, 2018. Available at `https://developers.google.com/optimization/`.

[64] Ronald L Graham. *Handbook of combinatorics.* Elsevier, 1995.

[65] Coseal Group. Coseal group website. Available at `https://www.coseal.net`.

[66] Zonghao Gu, Edward Rothberg, and Robert Bixby. *Gurobi Optimizer Reference Manual, Version 5.0.* Gurobi Optimization, Inc, Gurobi Optimization Inc., Houston, USA, 5.0 edition.

[67] Inc Gurobi Optimization. Gurobi 7.5 performance benchmarks. Technical report, Gurobi Inc, 2017. Available at http://www.gurobi.com/pdfs/benchmarks.pdf.

[68] Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[69] Mark Andrew Hall. *Correlation-based feature selection for machine learning.* PhD thesis, University of Waikato Hamilton, 1999.

[70] Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artif. Intell.*, 14(3):263–313, 1980.

[71] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

[72] Frederick S Hillier. *Introduction to operations research.* Tata McGraw-Hill Education, 2012.

[73] Fei Hu, Qi Hao, and Ke Bao. A survey on software-defined network and openflow: From concept to implementation. *IEEE Communications Surveys & Tutorials*, 16(4):2181–2206, 2014.

[74] IBM. IBM Decision Optimization CPLEX Modeling documentation, 2016. Available at http://ibmdecisionoptimization.github.io.

[75] Google Inc. Comparing MIP and CP, 2018. Available at https://developers.google.com/optimization/assignment/compare_mip_cp#guidelines-for-choosing-between-mip-and-cp.

[76] The European Telecommunications Standards Institute. Network function virtualization in ETSI, 2012. http://www.etsi.org/technologies-clusters/technologies/nfv.

[77] The European Telecommunications Standards Institute. Network functions virtualization (NFV); management and orchestration, 2014. http://www.etsi.org/technologies-clusters/technologies/nfv.

[78] EN ISG. Gs nfv-eve 005 v1. 1.1 network function virtualisation (NFV); ecosystem; report on SDN usage in nfv architectural framework. Technical report, ETSI, T.R. Available: http://www. etsi. org/deliver/etsi gs/NFV-EVE/001 099/005/01.01. 01 60/gs NFV-EVE005v010101p, 2015.

[79] Alexander Ivrii, Vadim Ryvchin, and Ofer Strichman. Mining backbone literals in incremental SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 88–103. Springer, 2015.

[80] Tong Liu Jacopo Mauro. MiniZinc model, 2017. Available at http://cs.unibo.it/t.liu/nightsplitter/mzn.html.

[81] Joxan Jaffar and J-L Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM, 1987.

[82] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *International Conference on Principles and Practice of Constraint Programming*, pages 454–469. Springer, 2011.

[83] Reza Khoshkangini, Maria Silvia Pini, and Francesca Rossi. A self-adaptive context-aware group recommender system. In *AI* IA 2016 Advances in Artificial Intelligence*, pages 250–265. Springer, 2016.

[84] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[85] Michal Kompan and Maria Bielikova. Group recommendations: Survey and perspectives. *Computing and Informatics*, 33(2):446–476, 2014.

[86] Lars Kotthoff. Icon challenge on algorithm selection. *arXiv preprint arXiv:1511.04326*, 2015.

[87] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data Mining and Constraint Programming*, pages 149–190. Springer, 2016.

[88] Krzysztof Kuchcinski and Radoslaw Szymanek. Jacop-java constraint programming solver. In *CP Solvers: Modeling, Applications, Integration, and Standardization, co-located with 19th CP*, 2013.

[89] Vipin Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[90] Jena-Louis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial intelligence*, 10(1):29–127, 1978.

[91] Siamak Layeghy, Farzaneh Pakzad, and Marius Portmann. SCOR: Constraint programming-based northbound interface for SDN. In *Telecommunication Networks and Applications Conference (ITNAC), 2016 26th International*, pages 83–88. IEEE, 2016.

[92] Kwan Hui Lim, Jeffrey Chan, Christopher Leckie, and Shanika Karunasekera. Towards next generation touring: Personalized group tours. In *ICAPS*, pages 412–420, 2016.

[93] M. Lindauer, J. N. van Rijn, and L. Kotthoff. The Algorithm Selection Competition Series 2015-17. *ArXiv e-prints*, May 2018.

[94] Marius Lindauer, Rolf-David Bergdoll, and Frank Hutter. An Empirical Study of Per-instance Algorithm Scheduling. In *LION*, volume 10079 of *LNCS*, pages 253–259. Springer, 2016.

[95] Tong Liu. Groupme: city data and multi-preference activity allocation. Master's thesis, Alma Mater Studiorum - Università di Bologna, Via Zamboni 33, Bologna, Italy, 3 2015.

[96] Tong Liu. SFC implementation, 2018. Available at `http://cs.unibo.it/~t.liu/sfc`.

[97] Tong Liu, Roberto Amadini, and Jacopo Mauro. SUNNY with algorithm configuration. In Marius Lindauer et all., editor, *Proceedings of the Open Algorithm Selection Challenge*, volume 79 of *Proceedings of Machine Learning Research*, pages 12–14, Brussels, Belgium, 11–12 Sep 2017. PMLR.

[98] Tong Liu, Franco Callegati, Walter Cerroni, Chiara Contoli, Maurizio Gabbrielli, and Saverio Giallorenzo. Constraint programming for flexible Service Function Chaining deployment. In *Hawaii International Conference on System Sciences (HICSS-52)*. IEEE Computer Society, 2019.

[99] Tong Liu, Roberto Di Cosmo, Maurizio Gabbrielli, and Jacopo Mauro. Night-splitter: a scheduling tool to optimize (sub) group activities. In *International Conference on Principles and Practice of Constraint Programming*, pages 370–386. Springer, 2017.

[100] Andrea Loreggia, Yuri Malitsky, Horst Samulowitz, and Vijay A Saraswat. Deep learning for algorithm portfolios. In *AAAI*, pages 1280–1286, 2016.

[101] Alan K. Mackworth. Consistency in Networks of Relations. *Artif. Intell.*, 8(1):99–118, 1977.

[102] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm portfolios based on cost-sensitive hierarchical clustering. In *IJCAI*, volume 13, pages 608–614, 2013.

[103] Brandon Malone, Kustaa Kangas, Matti Jarvisalo, Mikko Koivisto, and Petri Myllymaki. AS-ASL: Algorithm selection with auto-sklearn. In Marius Lindauer et all., editor, *Proceedings of the Open Algorithm Selection Challenge*, volume 79 of *Proceedings of Machine Learning Research*, pages 19–22, Brussels, Belgium, 11–12 Sep 2017. PMLR.

[104] A. Manzalini, R. Minerva, F. Callegati, W. Cerroni, and A. Campi. Clouds of virtual machines in edge networks. *IEEE Communications Magazine*, 51(7):63–70, July 2013.

[105] Kim Marriott, Peter J Stuckey, and Peter J Stuckey. *Programming with constraints: an introduction*. MIT press, 1998.

[106] Judith Masthoff. Group recommender systems: Combining individual models. In *Recommender systems handbook*, pages 677–702. Springer, 2011.

[107] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[108] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1):236–262, 2016.

[109] Michela Milano and Mark Wallace. Integrating operations research in constraint programming. *Annals of Operations Research*, 175(1):37–76, 2010.

[110] Mustafa Mısır and Michèle Sebag. ALORS: An algorithm recommender system. *Artificial Intelligence*, 244:291–314, 2017.

[111] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.

[112] John C Nash. The (Dantzig) simplex method for linear programming. *Computing in Science & Engineering*, 2(1):29–31, 2000.

[113] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling

language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[114] Wim Nuijten and Claude Le Pape. Constraint-Based Job Shop Scheduling with ILOG Scheduler. *Journal of Heuristics*, 3(4):271–286, 1998.

[115] Telecommunication Standardization Sector of ITU. *Call processing performance for voice service in hybrid IP networks.* International Telecommunication Union, a body of United Nations, November 2007. Recommendation. Y.1530.

[116] Matthew Perry. Python module for simulated annealing, 2017. Available at `https://github.com/perrygeo/simanneal`.

[117] K. Phemius, M. Bouet, and J. Leguay. DISCO: Distributed multi-domain SDN controllers. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–4, May 2014.

[118] Benjamin C Pierce. *Types and programming languages.* MIT press, 2002.

[119] Florian A Potra and Stephen J Wright. Interior-point methods. *Journal of Computational and Applied Mathematics*, 124(1-2):281–302, 2000.

[120] Carolyn C Preston and Andrew M Colman. Optimal number of response categories in rating scales: reliability, validity, discriminating power, and respondent preferences. *Acta psychologica*, 104(1):1–15, 2000.

[121] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation.* TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[122] Ramamurthy Ravi, Ravi Sundaram, Madhav V Marathe, Daniel J Rosenkrantz, and Sekharipuram S Ravi. Spanning trees—short or small. *SIAM Journal on Discrete Mathematics*, 9(2):178–200, 1996.

[123] John R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

[124] R. V. Rosa, M. A. S. Santos, and C. E. Rothenberg. MD2-NFV: The case for multi-domain distributed network functions virtualization. In *2015 International Conference and Workshops on Networked Systems (NetSys)*, pages 1–5, March 2015.

[125] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[126] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[127] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. *Software download and online material at the website: http://www. gecode. org*, pages 11–13, 2006.

[128] Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Solving RCPSP/max by lazy clause generation. *Journal of scheduling*, 16(3):273–289, 2013.

[129] Laura Sebastia, Inma Garcia, Eva Onaindia, and Cesar Guzman. *E-Tourism*: a tourist recommendation and planning application. *International Journal on Artificial Intelligence Tools*, 18(5):717–738, 2009.

[130] Bart Selman and Carla P Gomes. Hill-climbing search. *Encyclopedia of Cognitive Science*, 81:82, 2006.

[131] Helmut Simonis. Building industrial applications with constraint programming. In *International Summer School on Constraints in Computational Logics*, pages 271–309. Springer, 1999.

[132] Barbara M Smith. Modelling for constraint programming. *Lecture notes for the first international summer school on constraint programming*, 2005.

[133] Balázs Sonkoly, János Czentye, Robert Szabo, Dávid Jocha, János Elek, Sahel Sahhaf, Wouter Tavernier, and Fulvio Risso. Multi-domain service orchestration

over networks and clouds: A unified approach. In *2015 ACM SIGCOMM Conference*, pages 377–378, August 2015.

[134] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc Challenge 2008-2013. *AI Magazine*, 35(2):55–60, 2014.

[135] Gerald Jay Sussman and Guy Lewis Steel Jr. Constraints: A language for expressing almost-hierarchical descriptions. *Artificial intelligence*, 14:1–39, 1980.

[136] Ivan E Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*, pages 6–329. ACM, 1964.

[137] MiniZinc Team. MiniZinc challenge 2016, 2016. `http://www.minizinc.org/challenge2016/challenge.html`.

[138] MiniZinc Team. MiniZinc challenge 2017, 2017. `http://www.minizinc.org/challenge2017/challenge.html`.

[139] Jacopo Mauro Tong Liu. NightSplitter, 2017. Available at `http://cs.unibo.it/t.liu/nightsplitter`.

[140] CA Trauth Jr and RE Woolsey. Integer linear programming: a study in computational efficiency. *Management Science*, 15(9):481–493, 1969.

[141] TripAdvisor. Tripadvisor website, 2016. Available at `https://www.tripadvisor.com`.

[142] Pascal Van Hentenryck and Vijay Saraswat. Strategic directions in constraint programming. *ACM Computing Surveys (CSUR)*, 28(4):701–726, 1996.

[143] Pascal Van Hentenryck, Helmut Simonis, and Mehmet Dincbas. Constraint satisfaction using constraint logic programming. *Artificial intelligence*, 58(1-3):113–159, 1992.

[144] Peter JM Van Laarhoven and Emile HL Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[145] Gérard Verfaillie and Narendra Jussien. Constraint Solving in Uncertain and Dynamic Environments: A Survey. *Constraints*, 10(3):253–281, 2005.

[146] Richard J Wallace. Why ac-3 is almost always better than ac-4 for establishing arc consistency in csps. In *IJCAI*, volume 93, pages 239–245, 1993.

[147] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[148] Yanghao Xie, Zhixiang Liu, Sheng Wang, and Yuxiu Wang. Service function chaining resource allocation: A survey. *CoRR*, abs/1608.00095, 2016.

[149] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 228–241. Springer, 2012.

[150] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research*, 32:565–606, 2008.

[151] Yelp. Yelp dataset challenge, 2016. Available at `http://yelp.com/dataset_challenge/`.