UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2022

From Declarative Models to Local Search

GUSTAV BJÖRDAL

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2021

ISSN 1651-6214
ISBN 978-91-513-1152-4
urn:nbn:se:uu:diva-436139

Dissertation presented at Uppsala University to be publicly examined in Room ITC 2446, Polacksbacken, Lägerhyddsvägen 2, Uppsala, Friday, 23 April 2021 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Laurent Michel (University of Connecticut, USA).

Abstract
Björdal, G. 2021. From Declarative Models to Local Search. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 2022. 47 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-1152-4.

A solver is a general-purpose software for solving optimisation problems. It takes as input a description of a problem, called a model, and uses a collection of algorithms, called its solving technology, to ideally produce an optimal solution as output. Most solvers have a modelling language that cannot be processed by other solvers. This means that there is a risk of making an early commitment to a solver and its technology when writing a model. To address this risk, and to increase the accessibility of solvers, there has been a push for technology-independent modelling languages, a notable one being MiniZinc.

A model written in MiniZinc is transformed by the MiniZinc toolchain in order to suit a targeted solver and its technology. However, for a solver to process a MiniZinc model, it also requires what is called a backend for MiniZinc. A backend translates the transformed MiniZinc model into the solver's own modelling language and synthesises any components not in a MiniZinc model that the solver (or its technology) requires.

The solving technology called constraint-based local search (CBLS) is based on the popular algorithm design methodology called local search, which often quickly produces near-optimal solutions, even to large problems. So, with the advent of CBLS solvers, there is a need for CBLS backends to modelling languages like MiniZinc.

This thesis contributes to three research topics. First, it shows for the first time how to create a CBLS backend for a technology-independent modelling language, namely MiniZinc, and it shows that CBLS via MiniZinc can be competitive for solving optimisation problems. Second, it extends MiniZinc with concepts from local search, and shows that these concepts can be used even by other technologies towards designing new types of solvers. Third, it extends the utilisation of another technology, namely constraint programming, inside local-search solvers and backends.

These contributions make local search accessible to all users of modelling languages like MiniZinc, and allow some optimisation problems to be solved more efficiently via such languages.

Keywords: discrete optimisation, combinatorial optimisation, local search, large-neighbourhood search, MiniZinc, constraint programming, declarative modelling, declarative neighbourhoods

Gustav Björdal, Department of Information Technology, Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-1152-4
urn:nbn:se:uu:diva-436139 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-436139)

Dedicated to coffee and tea: without you
none of this would have been possible.

Acknowledgements

First and foremost, I would like to thank my supervisors Pierre Flener and Justin Pearson. Pierre, it is mind-boggling how much I have grown over these past years and how much of it is thanks to you. Thank you for being constantly supportive while also pushing and challenging me towards perfection. It has been a pleasure to have you as a mentor, both in teaching and in research. Justin, thank you for being a fantastic co-supervisor and balancing out the office with our music and math discussions; indeed sometimes math just cleanses the soul.

Thanks also to Mats, Di, Andreina, Frej, Jean-Nöel, Joe, Ghafour, Lei, and the rest for lighting up the office over the years.

Teaching turned out to be one of the things I most enjoyed over these years and this is undoubtedly thanks to all the wonderful students I have had. I wish you all the best and hope that we meet again soon.

I have had the pleasure of having Kalle Törnqvist as a mentor during the Mentor4Research programme 2019. Thank you, Kalle and the people at UU Innovation, for teaching me so many things about the world outside of academia.

During my PhD studies, I have been fortunate enough to visit the MiniZinc team at Monash University not once but twice! I am very thankful to both Peter Stuckey and Guido Tack for hosting me: it really was some of the best time of my studies. Of course, my time at Monash was made wonderful by many other people as well. Kevin, Dora, Maxim, David, Maria, Graeme, Henk, and Jip thanks for taking care of me around the office and overall just being great people. These visits were made possible by travel grants from the Ericsson Research Foundation and rektors resebidrag från Wallenbergstiftelsen.

Speaking of having a great time, there are many people at the department that I am very grateful to. Kim and Stephan, thanks for always cheering me up and sharing my interest in weird fermented food. Aleksandar thank you for so many life-saving coffee breaks and for introducing me to so many great people around the department. Thanks to Calle, Anna, Fredrik, Kalle, and the rest of the gang for all our lunch breaks together. Thanks also to Johan, Fredrik (again), and Per for helping to organise the infrequent Friday pub!

I would probably not have stayed in Uppsala for as long as I have had it not been for all my friends from outside the Department. Dennis, Javier, and Joachim, it has been great to share this journey with you and to get a glimpse at the PhD student experience at other departments. I look forward to many more discussions and burgers! Lowe, Adam, Sanna, Hussam, and Gusten; Malte, Viking, Per, and Falk; Amanda, Astri, Björn, Viktor, Karin, Olof, Linnea, and Viktoria: thank you all for making these years so much more than just studying.

I must also express my eternal gratitude to Östgöta nation, which has been my anchor point in Uppsala for over 10 years now. A house of memories where I have spent so much time with all of these wonderful people. Truly my second home.

Finally, to the people that I hold dearest: Mamma, Pappa, Andreas, Hanna, Jacob,[1] Anna, and especially Astrid. This would not have been possible without you. You know how much you mean to me; I don't need to say another word.

[1] Du är fortfarande skyldig mig 500kr.

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I **G. Björdal**, J.-N. Monette, P. Flener, and J. Pearson:
A constraint-based local search backend for MiniZinc.
Constraints, journal fast track of CP-AI-OR 2015, 20(3):325–345, July 2015.

II **G. Björdal**, P. Flener, and J. Pearson:
Generating compound moves in local search by hybridisation with complete search.
In: L.-M. Rousseau and K. Stergiou (editors), CP-AI-OR 2019. Lecture Notes in Computer Science, volume 11494, pages 95–111. Springer, 2019.

III **G. Björdal**, P. Flener, J. Pearson, P. J. Stuckey, and G. Tack:
Declarative local-search neighbourhoods in MiniZinc.
In: M. Alamaniotis, J.-M. Lagniez, and A. Lallouet (editors), ICTAI 2018, pages 98–105. IEEE Computer Society, 2018.

IV **G. Björdal**, P. Flener, J. Pearson, and P. J. Stuckey:
Exploring declarative local-search neighbourhoods with constraint programming.
In: T. Schiex and S. de Givry (editors), CP 2019. Lecture Notes in Computer Science, volume 11802, pages 37–52. Springer, 2019.

V **G. Björdal**, P. Flener, J. Pearson, P. J. Stuckey, and G. Tack:
Solving satisfaction problems using large-neighbourhood search.
In: H. Simonis (editor), CP 2020. Lecture Notes in Computer Science, volume 12333, pages 55–71. Springer, 2020.

Comments on Paper Contributions

Paper I

I am the main author. I made the key contributions to the research, made most of the implementation, and made contributions to the writing of some sections.

Paper II

I am the main author. I conceived the idea, led the research, made the implementation, conducted all the experiments, and was the lead writer of most of the sections.

Paper III

I am the main author. I conceived the idea, led the research, made the implementation with some help from the co-authors, conducted all the experiments, and was the lead writer of most of the sections.

Paper IV

I am the main author. I conceived the idea, led the research, made the implementation, conducted all the experiments, and was the lead writer of the entire paper.

Paper V

I am the main author. I conceived the idea, led the research, made the implementation with some help from the co-authors, conducted all the experiments, and was the lead writer of the entire paper.

Other Publication

J. J. Dekker, **G. Björdal**, M. Carlsson, P. Flener, and J.-N. Monette.
Auto-tabling for subproblem presolving in MiniZinc.
Constraints, journal fast track of CP-AI-OR 2017, 22(4):512–529,
October 2017.

Contents

1. Introduction

Optimisation problems are crucial to both industry and society, and are problems where we make decisions that optimise some objective while also satisfying some constraints.

For example, consider a problem of delivering packages in a city, something that takes place every day worldwide. There is a centralised depot where all packages are stored, and from which a fleet of vehicles is dispatched to deliver the packages. Each package has a volume and a destination, and each vehicle has capacity constraints on for example the maximum volume of the packages it can carry. In essence, the problem is to decide, for each vehicle, which packages to deliver and which route to take. Furthermore, we are usually also interested in being as efficient as possible: we also want to optimise some objective, such as minimising the total delivery time, the total fuel consumption, the number of used vehicles, or some combination thereof. That is, to solve this problem, we make decisions that optimise some objective while at the same time satisfying some capacity constraints.

Another example is the problem of personnel scheduling, say of nurses at a hospital. There is a set of nurses, each with some expertise, and days that are split into shifts. The problem is to decide which nurses are working each shift of each day, subject to requirements (or: constraints) such as the number and expertise of nurses on duty, the time of rest between two shifts, the number of free days that each nurse has over a month, etc. Again, we are usually also interested in optimising some objective, such as minimising the total overtime, any imbalance of nurse workloads in the schedule, or some combination thereof. Once more, this is a problem that is solved by making decisions that optimise some objective while at the same time satisfying some constraints.

These are two examples, among countless others, that illustrate what makes an optimisation problem, namely decisions, constraints, and an objective to optimise. While these problems are in general of high importance, they are also notoriously difficult to solve optimally.

There are, to generalise greatly, three approaches that we can take when attempting to solve an optimisation problem:

1. we can write a special-purpose algorithm for the problem by using a design methodology, such as local search or dynamic programming, that one can find in any advanced algorithms textbook;
2. we can use a general-purpose software, called a *solver*, where the input is a description of the problem, called a *model*, and the output is ideally an optimal solution; or

3. we can give up and instead approximate the problem as something that is easier to solve but then without satisfying all the constraints.

As optimisation problems can be very difficult to solve, the third approach, of approximating the problem, is reasonable, but will not be considered further in this thesis. Instead, we focus on the other two approaches where the aim is to solve the problem optimally under all of its constraints. It should be mentioned that there is actually a fourth approach that is very promising, namely quantum computing. In theory, quantum computers are groundbreaking for solving the type of optimisation problems that we consider in this thesis. However, it remains to be seen if this is also the case in practice: hence we will not discuss this approach further here.

The first approach, which we refer to as *writing an ad-hoc algorithm*, is in theory always preferable as the best algorithm for an optimisation problem can always be written ad-hoc. However, in practice, programming from scratch is both time-consuming and error-prone, and there is no guarantee that we even come up with a good algorithm for the problem at all.

The second approach, which we refer to as *model-and-solve*, offers several advantages over the first one and is often preferable in practice. Compared to an algorithm, a model just describes *what* the problem is, rather than *how* it can be solved. For this reason, a model can be easier to maintain (in case the problem changes slightly, which is often the case in practice), and a model offers more reassurance in terms of the correctness of the solutions, due to the descriptive nature of a model. Furthermore, as a solver improves over time, we benefit from these improvements without having to modify any models. Indeed, solvers usually have decades of research behind them, and are actively researched and developed. Solvers can be based on fundamentally different algorithms. We call the algorithms that a solver is based on its *solving technology*. A disadvantage of many solvers is that they have their own language for expressing models, and a model written in one solver's language can often not be passed to another solver.

The main inspiration for this thesis can be attributed to two advancements in the field of optimisation.

First, a solving technology called *constraint-based local search* (CBLS) was introduced in [31] and extended in [48]. There has since been a host of solvers that use some flavour of local search, e.g., [3, 6, 14, 30, 35, 48]. This has been a promising development, as local search is one of the more popular methodologies for writing ad-hoc algorithms for optimisation problems, and in particular because local search is known for producing near-optimal solutions to large problems very quickly, at least when used competently.

Second, there has been a push for modelling languages that are independent of solvers and solving technologies, so that we can write only one model and then run it on many solvers in order to find the best solver. Such modelling languages, and in particular the one called MiniZinc [34] and its toolchain, drastically increase the accessibility of solvers and allow the rapid prototyping

of a model across various solving technologies, in order to select a best combination of a model and solver. However, initially, no such modelling language supported CBLS solvers.

In this thesis, I bridge the gap between technology-independent modelling languages and local search, implementing it for MiniZinc in particular. This work not only allows us to deploy local search via MiniZinc, but also allows more problems to be successfully tackled via MiniZinc in the first place as some problems are in fact best suited for local search. These contributions may eventually further reduce the need for ad-hoc algorithms.

1.1 Terminology

We provide a high-level overview of the concepts required for summarising the contributions of this thesis in Section 1.2, and that are shared among chapters.

A *constraint satisfaction problem* (CSP) is a problem where we want to find an assignment of values for a set of unknowns, called *variables*, such that the assignment satisfies a set of constraints on the variables. We say that such an assignment is *feasible* and that it is a *solution*; otherwise, it is *infeasible* and is not a solution. A *constrained optimisation problem* (COP) is a CSP extended with an *objective function* on the variables, where the sought assignment is a *minimal solution* (or maximal solution). In this thesis, we only consider *discrete problems*, where each variable must take a value from a finite set, called its *domain*.

When the objective function is to be minimised, then the COP is a *minimisation problem*. We introduce everything in terms of minimisation problems as maximisation problems can be turned into minimisation problems by negating the objective function, and as a CSP is a COP with a constant objective function.

As an example of a COP let us consider the travelling salesperson problem with time windows, which is a variation of the vehicle routing problem above of package delivery, but for a single vehicle:

Example 1.1. Given are a set of n locations, with a designated starting location d in $\{1,\ldots,n\}$ called the *depot*; a travel-time matrix T where $T_{i,j}$ is the travel time from location i to location j; and for each location i a time window consisting of an earliest arrival time E_i and a latest arrival time L_i. The *travelling salesperson problem with time windows* (TSPTW) is to find a route of $n+1$ locations that starts at location d at time 0 and ends at location d, that visits each other location exactly once and within its time window, and that has a minimal total travel time.

A route that would arrive at a location before its earliest arrival time is still a valid route, and the arrival time is then instead at least the earliest arrival time at that location, which corresponds to the salesperson waiting for the location

to become available. There is no time window for the return to the depot. Note that the time the salesperson spends serving a location can be added into the travel-time matrix without loss of generality, and we thereby do not consider it.

The unknowns are the route and the arrival times at the locations, the constraints are that each location (except the depot) is visited exactly once and that each location is visited within its time window, and the objective is to minimise the total travel time.

This is a variation (with time windows) of the more commonly known travelling salesperson problem (TSP) [28]. While the TSP is formulated in terms of a salesperson mainly for historic reasons, it remains a very important problem to solve. Indeed, the TSP is at the core of problems in many areas of vehicle routing, shipping, automated warehouses, and even in biology, where for example the TSP is a sub-problem in DNA sequencing. □

A model is a declarative description of a problem and is expressed in a modelling language. Some notable solver-independent languages are AMPL [16], Essence [17], and MiniZinc [34]. A model is given as the input to a solver that is based on some solving technology. Some notable solving technologies are constraint programming [42], integer programming [51], constraint-based local search (CBLS) [48], propositional satisfiability [7], and satisfiability modulo theories [26]. Each solving technology comes with its strengths and weaknesses, and there is no solving technology that dominates all others, in the sense that no solving technology is the overall best one for solving all problems.

In this thesis, we think of a solver as a stand-alone program that takes a model as input. However, it is worth noting that having a modelling language is not necessary for being a solver, because many solvers are also (or exclusively) programming libraries that are used via an interface, through which something equivalent to a model is passed.

During solving, a solver will in some way explore the space of all possible assignments of values to the variables, called the *search space*, in order hopefully to find an optimal solution.

One aspect that can distinguish solving technologies is whether they use systematic search or local search when looking for solutions. *Systematic search* looks for solutions by systematically exploring the entire search space (and usually not by enumerating it), and is therefore guaranteed to eventually find an optimal solution or determine that no solution exists. *Local search* looks for solutions by starting from some assignment and then repeatedly moving to a new assignment, which is drawn from what is called a *neighbourhood* of assignments similar to the current one. Local search is usually a randomised process, and in general offers none of the guarantees of systematic search, but can in practice often find near-optimal solutions very quickly. A key difference between systematic search and local search is that as the size of a problem grows, say for larger values of n in Example 1.1, systematic search quickly becomes impractical, while local search tends to scale better to larger sizes.

Systematic search is sometimes also referred to as *complete search* as it will eventually perform something akin to a complete exploration of the search space. In contrast, local search is sometimes referred to as *incomplete search* as it generally does not explore the entire search space. However, there is some ambiguity in these two terms: if systematic search is stopped early, say due to exceeding a time limit, then the systematic search was in fact an incomplete search. The terms *systematic search* and *local search* are therefore here deemed preferable.

1.2 Contributions

This thesis contributes to three research topics: it shows for the first time how to connect a CBLS solver to a technology-independent modelling language (**Paper I** and **Paper II**); it adds technology-independent and declarative neighbourhood syntax to MiniZinc and shows how they can be used successfully across solvers of different solving technologies (**Paper III** and **Paper IV**); and it extends the utilisation of constraint-programming technology, which performs systematic search, during local search in order to solve problems more efficiently (**Paper II**, **Paper IV**, and **Paper V**).

1.3 Outline

This thesis is organised as follows. Chapter 2 gives a brief overview of MiniZinc and how discrete optimisation problems can be modelled in it. Chapter 3 introduces the solving technologies that are used in this thesis, namely constraint programming (Section 3.1), constraint-based local search (Section 3.2.2), and large-neighbourhood search (Section 3.2.3). Chapter 4 summarises the motivations, contributions, and results of this thesis. Finally, we conclude in Chapter 5.

2. Modelling of Discrete Optimisation Problems

We give an overview of models of discrete optimisation problems (Section 2.1) and an overview of a language for expressing models, namely MiniZinc [34] (Section 2.2).

2.1 Models

A *model* is a formal description of an optimisation problem that specifies the variables of the problem, the values that each variable can take, the constraints on the variables, and the objective function on the variables. A model is a *declarative* description of a problem as it only defines *what* the problem is, as opposed to an algorithm, which describes *how* the problem can be solved. A model can be expressed in many ways, for example by mathematical notation as shown in Example 2.1 and its Model 2.1:

Example 2.1 (continued from Example 1.1)**.** A mathematical model of the travelling salesperson problem with time windows (TSPTW) is shown in Model 2.1. Each of the $n+1$ variables x_i represents the ith location that is visited in the route, and is constrained to take a value in the set $\{1,\ldots,n\}$ (line 2.7). The first and last location visited is the depot (line 2.2), and each location is visited exactly once except for the depot, which is visited twice (line 2.6). The objective (line 2.1) is to minimise the sum of the travel times between consecutively visited locations. The arrival time at the ith visited location (not to be confused with location i) is denoted by the variable a_i (line 2.8), which is a natural number, except for the last visited location, for which we need no such variable as there is no time window for the return to the depot. Each a_i must take a value within the time window of the ith visited location (line 2.5), and must be at least the arrival time at the previous location plus the travel time from there (line 2.4). Recall that if the salesperson is to arrive at the ith visited location before its earliest arrival time, i.e., $a_{i-1} + T_{x_{i-1},x_i} < E_{x_i}$, then a_i will still take some value within the time window due to the constraint in line 2.5, which corresponds to the salesperson waiting at least until that location becomes available. The arrival-time variable a_1 for the first location is constrained to be 0 (line 2.3), which corresponds to the starting time of the route being 0. □

Model 2.1 A mathematical model of TSPTW; note that it is not linear.

$$\text{minimise} \quad \sum_{i=1}^{n} T_{x_i,x_{i+1}} \tag{2.1}$$

subject to

$$x_1 = d = x_{n+1} \tag{2.2}$$

$$a_1 = 0 \tag{2.3}$$

$$a_{i-1} + T_{x_{i-1},x_i} \leq a_i \qquad \forall i \in \{2,\ldots,n\} \tag{2.4}$$

$$E_{x_i} \leq a_i \leq L_{x_i} \qquad \forall i \in \{1,\ldots,n\} \tag{2.5}$$

$$x_i \neq x_j \qquad \forall i,j \in \{1,\ldots,n\},\ i < j \tag{2.6}$$

$$x_i \in \{1,\ldots,n\} \qquad \forall i \in \{1,\ldots,n+1\} \tag{2.7}$$

$$a_i \in \mathbb{N} \qquad \forall i \in \{1,\ldots,n\} \tag{2.8}$$

Model 2.1 is a *parametric model* as it is defined for any values of its parameters n, d, E, L, and T. We call a concrete set of parameters *data* and when a model is paired with data, they form an *instance*.

Models can also be expressed in declarative languages, which we call *modelling languages*. By using a modelling language to describe a problem, we can analyse and transform the model, and solve instances of the problem, by using off-the-shelf problem-independent software that we call a *solver*. This model-and-solve paradigm provides a convenient approach for problem solving, as writing a model can be significantly easier than writing a problem-specific algorithm for solving the problem. For example, Model 2.1 is significantly simpler than any algorithm for solving TSPTW. Furthermore, contrary to what one might believe, the model-and-solve approach usually does not sacrifice performance for convenience. Solvers are discussed in Chapter 3.

2.2 MiniZinc

MiniZinc [34] is an open-source toolkit for a modelling language that has been developed as a push towards a standard modelling language that is independent of solvers, and even independent of solving technologies. By using a solver-independent language, we do not have to make an early commitment to a specific solver and solving technology when writing a model.

The MiniZinc language supports both parameters and variables of Boolean, integer, integer-set, and float types, and arrays thereof. MiniZinc was extended in [2] to support string variables, but this extension is not yet part of the official language.

Constraints are declared using constraint predicates (as defined in Section 2.2.1 below) and expressions. Expressions are built using typical programming operators and syntax like `+`, `-`, `*`, `div`, `X[i]`, `not`, $\wedge$, etc. We do not give the full grammar and semantics of expressions in MiniZinc as this amount of detail is not important for this thesis, but note that $\wedge$ denotes the logical 'and' operator.

Example 2.2. Model 2.2 shows a parametric MiniZinc model that corresponds to Model 2.1. Lines 2 and 3 declare the two integer parameters `n` and `d`, line 4 declares the 2d parameter array `TravelTime` where `TravelTime[i,j]` is the travel time from location `i` to location `j`, and lines 5 and 6 declare two arrays of parameters, where `EarliestArrival[i]` and `LatestArrival[i]` form the time window of location `i`. The main variables are declared on line 8 as elements of an array `Route`, which corresponds to the x_i variables in Model 2.1 and is indexed by what we call the *index set* `1..n+1`, where `a..b` is the set of integers between `a` and `b` inclusive. The syntax `var 1..n` on line 8 specifies that each element of `Route` is a variable that has the domain `1..n`, i.e., they can take a value in the integer set `1..n`. Furthermore, the array `ArrivalTime`, which corresponds to the a_i variables in Model 2.1, is declared on line 9. Here the syntax `var int` means that these variables can take any integer value.

The constraints on lines 11 and 12 require the depot to be the first and last location visited. The constraint from line 13 to line 15 requires the variables of `Route` (except for `Route[n+1]`) to all take different values. On line 16 the arrival time at the depot (which corresponds to the starting time of the route) is constrained to be 0. Lines 17 until 19 constrain the arrival time at the `i`th location in the route to be at least the arrival time at the `(i-1)`th location plus the travel time between these two locations, while lines 20 until 23 constrain the arrival time to be within the time window of each location. Finally, the `solve` statement on line 25 specifies that this is a minimisation problem and gives the objective function to minimise, namely the sum of the travel times between consecutively visited locations. □

2.2.1 Constraints

A *constraint predicate* is the name of a commonly required relationship between variables and is either user-defined or part of the MiniZinc language, such as the infix $=$ and $\leq$ predicates. A predicate is either natively supported by a solver or requires a *decomposition*, which is a definition of the predicate expressed in MiniZinc in terms of others: decompositions can be solver-specific, but default ones are shipped with MiniZinc.

Model 2.2 A MiniZinc model for TSPTW that corresponds to Model 2.1.

```
% Parameters
int: n;
int: d;
array[1..n, 1..n] of int: TravelTime;
array[1..n] of int: EarliestArrival;
array[1..n] of int: LatestArrival;
% Variables
array[1..n+1] of var 1..n: Route;
array[1..n] of var int: ArrivalTime;
% Constraints
constraint Route[1] = d;
constraint Route[n+1] = d;
constraint forall(i, j in 1..n where i < j)(
  Route[i] ≠ Route[j]
);
constraint ArrivalTime[1] = 0;
constraint forall(i in 2..n)(
  ArrivalTime[i-1] + TravelTime[Route[i-1],
    Route[i]] ≤ ArrivalTime[i]
);
constraint forall(i in 1..n)(
  EarliestArrival[Route[i]] ≤ ArrivalTime[i] ∧
  ArrivalTime[i] ≤ LatestArrival[Route[i]]
);
% Objective function
solve minimize sum(i in 1..n)(TravelTime[Route[i],
  Route[i+1]]);
```

Example 2.3. The constraint `even(x)`, which constrains an integer variable `x` to take an even value, has the user-defined predicate `even`, which can have the following decomposition:

```
predicate even(var int: x) = (x mod 2 = 0);
```

Indeed, `x` is even if and only if `x` is congruent with 0 modulo 2. □

A special class of constraint predicates is the class of *global-constraint predicates*. Although there is no agreed-upon definition of what is required to be a global constraint, the name is often used for complex constraints that are parametrised in the number of arguments, have a complex decomposition, or some combination thereof [4].

Example 2.4. The `alldifferent(X)` constraint in MiniZinc has a commonly used global-constraint predicate that constrains the variables in array `X` to all take different values, and can have the following decomposition:

```
predicate alldifferent(array[int] of var int: X) =
  forall(i, j in index_set(X) where i < j)(
    X[i] ≠ X[j]
  );
```

This declares that `alldifferent` can be applied to an array with any index set and variables of any integer domain, and constrains each pair of elements at distinct indices in `X` to take different values. Note that lines 13 to 15 in Model 2.2 can (and should, as argued below) be replaced by

```
constraint alldifferent(Route[1..n]);
```

where `Route[1..n]` has the first `n` elements of `Route`. □

Using a global-constraint predicate in a model, as opposed to using a decomposition, offers several advantages. First, the model becomes more concise, which improves the readability and maintainability of the model. Second, if a better decomposition is discovered for some predicate, then any model using that predicate will benefit from this discovery, without one having to change the model. Finally, a global-constraint predicate captures a complex combinatorial substructure of a problem that can otherwise be difficult to automatically identify. This allows solvers to then easily reason about this complex substructure. For example, by instead using `alldifferent(Route[1..n])` in Model 2.2 for lines 13 to 15, where there are `n` variables with the domain `1..n`, it is trivial to automatically discover that `Route[1..n]` has to be a permutation of the set `1..n`, and thereby easy for solvers to exploit this information for faster solving. Discovering this fact in Model 2.2 would require that essentially the `alldifferent(Route[1..n])` constraint is identified. In general, automatically identifying that some constraints are equivalent to a global constraint is a non-trivial task [27].

The Global Constraint Catalogue [4] discusses most of the global-constraint predicates identified in the literature. The MiniZinc language includes an extensive list of constraint predicates, including many global ones.

2.2.2 FlatZinc and Backends

Solvers do not directly solve the problem described by a MiniZinc model, but instead have their own solver-specific modelling language and can only handle models expressed in that language. Therefore, in order to use a solver for a MiniZinc model, the model must be translated into the solver's language and any components not in the MiniZinc model that the solver requires must be

synthesised. We call such a translate-and-synthesise unit the solver's *backend* for MiniZinc. Note that we distinguish between a *solver* and a *backend*, and that the words are not synonyms: a solver need not have a backend for MiniZinc, and a backend can be independent of any particular solver and can even provide input suitable for many solvers. Some notable solvers that have a backend for MiniZinc are Choco [40], Gecode [18], and SICStus Prolog [9] of constraint-programming technology (see Section 3.1); Gurobi [21], CPLEX [23], and Cbc [24] of integer-programming technology via the common backend of [5]; OscaR.cbls [14] and Yuck [30] of CBLS technology (see Section 3.2.2); Picat-SAT [52] of propositional-satisfiability technology; OptiMathSAT [11] of optimisation-modulo-theories technology; and Chuffed [10] and CP-SAT [20] of hybrid technologies. Note that some solvers offer several solving technologies via MiniZinc: for example Gecode can also perform large-neighbourhood search (a flavour of local search, see Section 3.2.3).

Designing a backend can be a significant challenge as there is usually *not* a one-to-one correspondence between the components of a MiniZinc model and the components that a solver requires. Indeed, this challenge is the main focus of **Paper I** and for example of [5, 11, 52]. In order to simplify the job of a backend designer, MiniZinc offers an intermediate language called FlatZinc that each MiniZinc model, together with data, is compiled into. FlatZinc is a subset of the MiniZinc language and the compilation process is called *flattening*. During flattening, a MiniZinc model is transformed by introducing new variables and constraints such that the resulting FlatZinc model consists only of variable (and parameter) declarations, constraints that each have a single constraint predicate, and a `solve` statement with a single variable, which is constrained by the constraints to take the value of the objective function.

Example 2.5. During flattening, the constraint $\texttt{x} \star \texttt{y} \leq \texttt{z}$ is replaced by adding a new variable `a` and stating the two constraints $\texttt{x} \star \texttt{y} = \texttt{a}$ and $\texttt{a} \leq \texttt{z}$. For the statement `solve minimize x*z` a new variable `b` is constrained by $\texttt{x} \star \texttt{z} = \texttt{b}$ to take the value of the objective function, and the statement becomes `solve minimize b`. □

Note that as part of the flattening process, all parameters in a MiniZinc model must be given values: data must be supplied. Therefore, a FlatZinc model is always an instance.

Another important aspect of flattening is that a model is always flattened for use by a given solver via a backend: the flattening process can also make solver-specific changes to the model. Specifically, each solver or backend provides to MiniZinc the list of (global-)constraint predicates that the solver natively supports, so that during flattening the constraints with supported predicates are kept intact while the constraints with unsupported ones are replaced by using either the default decomposition provided by MiniZinc or the solver-specific decomposition.

3. Solving Technologies for Discrete Optimisation

We briefly describe the solving technologies that are used in this thesis, namely constraint programming (Section 3.1), constraint-based local search (Section 3.2.2), and large-neighbourhood search (Section 3.2.3).

Recall from Chapter 2 that all concepts are introduced for minimisation problems, omitting satisfaction and maximisation problems without loss of generality. For the sake of brevity, we will in the context of solvers abuse the terminology of Chapter 2 slightly by saying that a solver solves a model rather than an instance.

3.1 Constraint Programming (CP)

A constraint programming (CP) solver uses systematic search to solve a CP model, defined as follows:

Definition 3.1. A *constraint-programming model* for a constrained minimisation problem is a tuple $\langle V, D, C, o \rangle$ consisting of:

- a set V of variables;
- a function D that maps each variable v in V to its *initial domain*, which is the set $D(v)$ of values it can take;
- a set C of constraints on V that must be satisfied in any solution; and
- a variable o in V, called the *objective variable*, that has constraints in C that require it to take the value of the objective function, which is to be minimised.

A CP solver associates each variable v with what is called its *current domain*, denoted by $\mathrm{dom}(v)$. The current domains are collectively called a *store*. Initially, each $\mathrm{dom}(v)$ is $D(v)$: the initial store is $\{v \mapsto D(v) \mid v \in V\}$. During solving, values are removed from the current domains in the store. When the current domain of a variable is a singleton, then the variable is said to be *fixed* to the value in that set. When all variables are fixed by a CP solver, then they form a solution.

A CP solver interleaves propagation (Section 3.1.1) and systematic search (Section 3.1.2).

3.1.1 Propagation

Consider a CP model $\langle V, D, C, o \rangle$. For each constraint $c(v_1, \ldots, v_n)$ in C over variables v_i in V a CP solver instantiates a corresponding propagator for predicate c provided with the CP solver:

Definition 3.2. A *propagator* for a constraint $c(v_1, \ldots, v_n)$ is an algorithm that performs *propagation*: it infers some values in the current domains of $v_1, \ldots, v_n$ that are impossible under the constraint and removes them. When a propagator cannot remove any further values, it is said to be at a *fixpoint*.

Example 3.1. Consider the variables x, y, and z with the current domains $\{1,2,3,4\}$, $\{5,6,7,8\}$, and $\{10,11,12\}$ respectively, and the linear equality constraint $x+y=z$. A propagator for $x+y=z$ can remove value 1 from $\mathrm{dom}(x)$, as for each value w in $\mathrm{dom}(y)$ and each value u in $\mathrm{dom}(z)$, we have $1+w \neq u$. Likewise, the value 5 can be removed from $\mathrm{dom}(y)$, and no other values can be removed based on the current domains. □

Note that a propagator might empty the current domain of a variable, and that it need not remove all values that are impossible under the constraint and the current domains. The degree to which a propagator removes values is referred to as its *consistency*. There usually are multiple propagators of different consistencies for each constraint predicate in a CP solver, as a propagator that is able to remove more values might take significantly more time to execute. There is a default propagator for each constraint predicate, and the modeller can override the default one when there are multiple propagators to choose from. See [44] for more details on propagators.

3.1.2 Search

In order to solve a CP model, a CP solver builds a *search tree* by interleaving propagation and search, as sketched in Algorithm 3.1. Each node in the tree corresponds to a store. We first describe a search for all solutions, both optimal and sub-optimal ones, and then mention some refinements for more efficiently searching for an optimal solution.

For a CP model $\langle V, D, C, o \rangle$, the solving starts by the call CP-SOLVE($\{v \mapsto D(v) \mid v \in V\}, C, o, B$) to the recursive procedure CP-SOLVE, where the first argument is the *root node* of the tree and B is a branching strategy. Upon each call to CP-SOLVE, the store is first updated by running all propagators until they are each at a fixpoint (line 2). If the current domain of some variable is empty (line 3), then the node is said to be *failed* and the procedure returns to the caller (line 4), which corresponds to backtracking to the parent node. When all variables are fixed (line 5), then a solution has been found and is output (line 6), and the search backtracks to the parent node in order to

Algorithm 3.1 Overview of depth-first all-solutions search by a CP solver for a CP model $\langle V, D, C, o \rangle$, current store S, and branching strategy B.

```
1:  procedure CP-SOLVE(S, C, o, B)
2:      S ← PROPAGATE(S, C)                          ▷ reduce the current domains
3:      if HASEMPTYCURRENTDOMAIN(S) then
4:          return                                    ▷ current node is failed
5:      if ALLFIXED(S) then
6:          OUTPUTSOLUTION(S)                         ▷ solution is found
7:          return
8:      v ← SELECTVARIABLE(S, B)                      ▷ select variable to branch on
9:      {P_1, ..., P_p} ← PARTITION(v, S, B)          ▷ partition dom(v)
10:     for i ∈ {1, ..., p} do
11:         CP-SOLVE(S, C ∪ {v ∈ P_i}, o, B)          ▷ explore child with v ∈ P_i
```

search for additional solutions (line 7). If backtracking does not happen, then the branching strategy B selects a variable v with at least two values in its current domain (line 8) and partitions $\mathrm{dom}(v)$ into $p \geq 2$ non-empty disjoint subsets (line 9). For each part P_i of the partition, a *child node* is constructed by restricting v to take a value in that part, and the new node is recursively explored (lines 10 and 11). This depth-first exploration of the search tree is a systematic search that is guaranteed to eventually output all solutions, both optimal and sub-optimal ones. We discuss at the end of this section how only to search for a solution that is better than the one previously output, if any.

Example 3.2. Consider again Example 3.1. After propagation in the root node, the variables x, y, and z have current domains $\{2,3,4\}$, $\{6,7,8\}$, and $\{10,11,12\}$ respectively. Consider a branching strategy that selects a variable with at least two values in its current domain, using the priority order x, y, z, and partitions that variable's current domain into two parts, where the first part has the smallest value of the current domain, and the second part has the remaining values.

From the root node, the search creates two child nodes by partitioning the current domain of x into $\{2\}$ and $\{3,4\}$. The search then visits the node where $x \in \{2\}$ and, upon propagation, the current domains in this node are $\{2\}$, $\{8\}$, and $\{10\}$ respectively, and a first solution has been found.

The search then backtracks and visits the other child of the root node, where upon propagation the current domains become $\{3,4\}$, $\{6,7,8\}$, and $\{10,11,12\}$. The search then continues like this, first visiting the node where $\mathrm{dom}(x)$ is $\{3\}$, until all solutions have been found, namely $[x,y,z] \in \{[2,8,10], [3,7,10], [3,8,11], [4,6,10], [4,7,11], [4,8,12]\}$. □

In order to accelerate the solving, a CP solver usually employs many refinements to both the propagation and the search [43]. Nevertheless, such a systematic search can still take a significant amount of time. Therefore, one can limit the search by imposing some budget on the CP solver, such as a maximum runtime, a maximum number of nodes to explore, or a maximum number of failed nodes to be encountered. When the CP solver has exhausted its budget, then it stops early and returns its currently best solution.

In order to efficiently reach an optimal solution to an optimisation problem, a CP solver usually employs branch-and-bound search as a refinement of the usual depth-first search. In *branch-and-bound* search, whenever a new solution is found, the CP model is extended with the new constraint $o < o'$, where o' is the value that the objective variable o has in that solution. Branch-and-bound search ensures that each new solution is better than the previous one, and allows propagation to further reduce the size of the search tree.

3.2 Local Search

Local search, as described for instance in [22], is a family of algorithm design methodologies for solving optimisation problems and usually does not process a model. That being said, some local-search frameworks do classify as solving technologies: they solve a model using local search.

In this thesis, we are concerned with two of these solving technologies, namely constraint-based local search [48] and large-neighbourhood search [45], which we discuss in Sections 3.2.2 and 3.2.3 respectively. But first we discuss local search in general in Section 3.2.1.

3.2.1 A Prototypical Local-Search Algorithm

Consider a constraint-free discrete minimisation problem that has a set of variables V, where each v in V takes a value from some finite domain $D(v)$; and an objective function f that maps V to an integer value that is to be minimised. A prototypical local-search algorithm for solving such a problem is Algorithm 3.2, which iteratively changes an *assignment*, where each variable v in V is associated with a value in its domain $D(v)$. We here focus on problems without constraints for the sake of simplicity, but local search can deal with constrained problems using for example the approaches we show in Sections 3.2.2 and 3.2.3.

Overview

LS-SOLVE starts by creating some assignment, which is called the *initial assignment* and initialises the *current assignment*, of the variables in V that is then iteratively improved by, at each iteration, first probing a set of changes

Algorithm 3.2 A prototypical local-search algorithm for a set of variables V with domains given by D and an objective function f.

```
1: procedure LS-SOLVE(V,D,f)
2:     A ← INITIALISE(V,D)                          ▷ create the initial assignment
3:     while not DONE(A,f) do    ▷ loop while stopping criteria are not met
4:         N ← GENERATENEIGHBOURS(A)    ▷ generate a neighbourhood
5:         m ← SELECTMOVE(A,N,f)          ▷ probe moves and select one
6:         A ← COMMIT(A,m)                  ▷ commit to selected move
7:     return A
```

and then performing one of them, until some stopping criteria are met, in the hope of finding an optimal solution or at least a near-optimal one.

Next, we define the key aspects of local search, namely its initialisation strategy, its neighbourhood structure, its heuristic, and its meta-heuristic. We call a particular design of those aspects a *local-search strategy*, and we call an algorithm that implements a local-search strategy a *local-search algorithm*.

Initialisation Strategy

On line 2 of Algorithm 3.2 an initial assignment A is created and becomes the current assignment. The initial assignment is generated by an *initialisation strategy*, which usually has some amount of randomisation and takes some problem-specific knowledge into account. For example, a problem-independent initialisation strategy is to assign each variable a random value within its domain. Another initialisation strategy is to use a problem-specific algorithm to generate an assignment that has some desirable property. For example, if some variables must all take different values, then those variables can be initialised to random but different values.

Neighbourhood Structure

At each iteration (line 3 will be explained as part of the heuristic below), over lines 4 and 6, the current assignment A is changed by performing a move:

Definition 3.3. A *move* is a change to the values of some variables in the current assignment. If a move changes assignment A into assignment A', then A' is said to be a *neighbour* of A. A set of moves (or, equivalently, the set of neighbours they reach) is called a *neighbourhood*. A neighbourhood where the moves have a shared structure is said to have a *neighbourhood structure*.

Example 3.3. A common neighbourhood structure is the *swap neighbourhood structure*, where each move only changes two variables, namely by swapping their values. We use the notation $x :=: y$ to mean "swap the current values of the variables x and y". Consider three variables $[x,y,z]$ with a current assignment of $[1,1,2]$. The neighbourhood of all possible swap moves of two variables

is $\{x :=: y,\ x :=: z,\ y :=: z\}$, or equivalently the set of neighbours $\{[1,1,2], [2,1,1], [1,2,1]\}$; note that the current assignment here happens to be in its own neighbourhood, which is not disallowed in general. □

On line 4 the neighbourhood of the current assignment A is generated via the call GENERATENEIGHBOURS(A). Note that in practice the neighbourhood generation is usually lazy as SELECTMOVE(A,N,f) on line 5 might not consider all neighbours, as discussed next.

Heuristic

A *heuristic*, which probes each move of the neighbourhood and selects one, is performed on line 5. A move can be *probed* by tentatively performing the move on the current assignment, recording the value of the objective function at the obtained neighbour, and undoing the move. In practice moves can be probed more efficiently by utilising problem-specific features. Moves are probed in order to determine the quality of neighbours, such as whether they are improving or not:

Definition 3.4. A neighbour A' to assignment A is *improving* if $f(A') < f(A)$, and likewise the move from A to A' is said to be improving. If an assignment has no neighbour that is improving, then it is a *local minimum* of the objective function f. If there are no assignments that are improving on a local minimum, then the latter is also a *global minimum* of f.

Note that since local search does not systematically explore all assignments, it can only determine that it has reached a global minimum in case $f(A)$ reaches an a priori known lower bound, such as the sum of the n shortest travel times in Example 1.1 (note that this lower bound is very unlikely to be feasible).

Heuristics, in general, are *greedy* in that they only select an improving move (if one exists), without considering the impact of the move over several subsequent iterations. We call them *improving heuristics*. Two improving heuristics are the *first-improving* heuristic, which selects the first probed move that is improving, and the *best-improving* heuristic, which probes all moves and selects a most improving move.

Heuristics usually employ some amount of randomisation when selecting a move, such as randomised tie-breaking in case of two or more equally improving neighbours. There are many other heuristics: see [22] for an overview.

Finally, the selected move is *committed* on line 6: the move is performed and the resulting neighbour replaces the current assignment.

This is repeated until some stopping criteria are met on line 3. Usually, the stopping criteria are the exhaustion of some budget like for example the number of iterations, the running time, or some combination thereof, but can also be based on the current assignment A or the objective function f.

Meta-Heuristic

When using only an improving heuristic, Algorithm 3.2 will eventually reach a local minimum and be unable to make any further moves as none are improving (by definition). Since local search has no way of determining if a local minimum is also a global minimum, the search must continue, and therefore it needs some mechanism for escaping a local minimum. In order to accomplish this, a meta-heuristic is commonly used. A *meta-heuristic* forces the heuristic to sometimes deviate from its behaviour so that the local search can move into parts of the search space that it might otherwise not visit. There are many meta-heuristics: some notable ones are Simulated Annealing [25] and Tabu Search [19]; for other examples see [22]. In this thesis, we are only considering Tabu Search:

Definition 3.5. *Tabu Search* extends a heuristic by introducing a *short-term memory* M that contains assignments. The size of M is called the *tabu tenure*, or just *tenure*, and determines how many assignments M can hold before it is full. At each iteration, the new current assignment is recorded in M after committing a move. If M was full, then the oldest assignment in M is first deleted. Tabu Search also extends the heuristic so that it can perform non-improving moves, if this was not already the case, and so that the heuristic cannot select a move that reaches an assignment currently in M.

Tabu Search ensures that if the search reaches a local minimum, then the heuristic can make a non-improving move, and the short-term memory ensures that the heuristic does not revisit any of the last t assignments, including local minima, where t is the tenure. This prevents the heuristic from getting stuck in a cycle of repeating the same (up to t) moves.

3.2.2 Constraint-Based Local Search (CBLS)

Usually local-search strategies are problem-specific and have an ad-hoc implementation that is tailored specifically for a problem; recall that we call the implementation a local-search algorithm. As a result the effort for designing a local-search algorithm can be significant, and yet code reusability for other problems is limited. This effort should not be underestimated as the success of a local-search algorithm is highly dependent on an efficient implementation.

In order to increase code reusability and significantly reduce the implementation effort for efficient local-search algorithms, *constraint-based local search* (CBLS) [32, 48] is proposed as a solving technology, rather than an algorithm design methodology. A CBLS solver provides abstractions for expressing a model, similar to those of constraint programming, and provides programming abstractions for implementing local-search strategies that utilise the model in order to efficiently generate and probe moves:

Definition 3.6. A *CBLS model* is a tuple $\langle V, D, C, F, o \rangle$ consisting of:

- a set V of variables;
- a function D that maps each variable to its domain;
- a set C of constraints on V;
- a set F of *functional constraints* on V, each functionally defining some variables in terms of others and written as $[o_1, \ldots, o_n] = f(i_1, \ldots, i_m)$, where f is a function symbol, with $C \cap F = \varnothing$; and
- a variable o in V that has constraints in $C \cup F$ that require it to take the value of the objective function, which is to be minimised.

For example, the constraint $x + y = z$ of Example 3.1 is functional. Note that a constraint in C can also be functional: it is the modeller who decides which functional constraints of the problem are in C rather than F, as there are restrictions on the constraints in F (see Definition 3.10 below).

Definition 3.6 does not necessarily correspond exactly to how models are structured across all CBLS solvers. Indeed, CBLS solvers represent and implement a model in slightly different ways in practice. Our intention here is instead to just define the relevant concepts for this thesis and to highlight the aspects in which a CBLS model is conceptually more complex than a MiniZinc model.

Furthermore, note that the term "constraint-based local search" is sometimes used liberally to mean any local-search algorithm that somehow involves constraints, but such a definition applies to essentially all local-search algorithms, as constraints are a very general concept, whether or not they are stated in a model. Therefore, we use CBLS to exclusively mean a solving technology in the spirit of what is outlined in [48] and next.

Overview

Below we describe the parts of CBLS that are essential for this thesis, namely invariants, constraints with violation, implicit constraints, and local-search strategies (now in a CBLS context). Finally, we briefly discuss some CBLS solvers. Compared to [48], we sometimes introduce additional terminology: this is not necessary when working directly with a CBLS solver, but the distinctions are at the core of **Paper I**, which concerns the translation of any MiniZinc model into a CBLS model and the synthesis of a local-search strategy from the MiniZinc model.

Invariants

For a CBLS model $\langle V, D, C, F, o \rangle$, a *CBLS solver* holds a current assignment for all variables in V. Some of these variables are occasionally changed by moves during search and the remaining variables are each automatically changed by an invariant that implements a functional constraint in F:

Definition 3.7. For a functional constraint $[o_1,\ldots,o_n] = f(i_1,\ldots,i_m)$, an *invariant* is an algorithm that is said to *compute* new values for the variables $o_1,\ldots,o_n$, and thereby change them, in response to any change of at least one of the variables $i_1,\ldots,i_m$: the invariant ensures that the functional constraint is always satisfied. We call the variables $o_1,\ldots,o_n$ the invariant's *output variables*, and the variables $i_1,\ldots,i_m$ its *input variables*. Each o_j is said to *depend on* each i_j.

Definition 3.8. Each output variable of an invariant is called a *defined variable* of the CBLS model. Each variable that is not a defined variable is called a *search variable* of the CBLS model, and is subject to changes by moves during search.

We denote an invariant using the arrow symbol, $[o_1,\ldots,o_n] \leftarrow f(i_1,\ldots,i_m)$, in order to distinguish it from its functional constraint. Note that each input variable of an invariant for a functional constraint in a model can be either a search variable or a defined variable. Note also that a variable can be changed by either a move or an invariant, depending on whether it is a search variable or a defined variable.

Example 3.4. The functional constraint $[y_1,y_2,y_3] = \text{Sort}([x_1,x_2,x_3])$, sometimes written as $\text{Sort}([x_1,x_2,x_3],[y_1,y_2,y_3])$, requires that the array $[y_1,y_2,y_3]$ is a permutation of $[x_1,x_2,x_3]$ sorted in non-decreasing order, and functionally defines the output variables y_j in terms of the input variables x_i. Consider a current assignment where $[x_1,x_2,x_3]$ is $[2,1,3]$. The invariant $[y_1,y_2,y_3] \leftarrow \text{Sort}([x_1,x_2,x_3])$ ensures that $[y_1,y_2,y_3]$ is $[1,2,3]$ in that assignment. If x_2 is changed to the value 3, then the invariant changes y_1 and y_2 to be 2 and 3 respectively. □

Example 3.5. The functional constraint $z = \text{Element}(x,[y_1,y_2,y_3])$, sometimes written as $\text{Element}(x,[y_1,y_2,y_3],z)$, requires that z is y_x, and functionally defines the output variable z in terms of the input variables x and the y_i. Consider a current assignment where $[x,y_1,y_2,y_3]$ is $[2,4,5,6]$. The invariant $z \leftarrow \text{Element}(x,[y_1,y_2,y_3])$ ensures that z is 5 in that assignment. If x is changed to the value 3, then the invariant changes z to be 6. If instead y_2 is changed to be 7, then the invariant changes z to be 7. □

When the current assignment of a search variable is changed by a move, the CBLS solver must notify each invariant whose output variables transitively depend on the changed variable, so that the invariant can change its output variables. Algorithms for efficiently determining the invariants that are transitively affected by a move as well as the algorithm that each invariant uses to change its output variables are at the core of a CBLS solver. These algorithms, which are called *propagation* (not to be confused with CP-style propagation: recall Section 3.1.1) and *incremental recomputation*, are out of scope for this thesis,

where we just use CBLS solvers off-the-shelf. For more information the reader is referred to [32, 35, 48] as a starting point.

In the context of **Paper I**, there are some further details regarding invariants and functional constraints that we must consider. First, what happens when a defined variable is used for computing its own value, and second, what happens with a functional constraint that can be reformulated into another functional constraint that functionally defines a different set of output variables. We discuss these cases next.

Definition 3.9. For an invariant $[o_1, \ldots, o_n] \leftarrow f(i_1, \ldots, i_m)$, we say that some o_j *statically depends* on some i_k when the value of o_j depends on the value of i_k for all values of $i_1, \ldots, i_m$; otherwise, we say that o_j *dynamically depends* on i_k.

Example 3.6. In Example 3.4, each y_j statically depends on each x_i because the value of each y_j is always computed by sorting all the x_i. In Example 3.5, the output variable z statically depends on x and dynamically depends on each y_i because the value of z depends always on x but only on y_x. □

If x statically depends on y and y statically depends on z, then x statically depends on z. If either x dynamically depends on y, or y dynamically depends on z, or both, then x dynamically depends on z.

CBLS solvers normally require that a CBLS model is valid in order for the propagation algorithm to behave properly:

Definition 3.10. A CBLS model is *valid* if each variable is an output variable of at most one invariant and does not statically depend (transitively) on itself.

If a model is valid and a variable dynamically depends on itself, then the local-search strategy is responsible for ensuring that the search never visits an assignment where the value of the variable is actually computed in terms of itself. For example, for $z \leftarrow \text{Element}(x, [z, y])$ the local-search strategy must ensure that x never becomes the index of z, as z would then be undefined.

Some functional constraints can be rewritten, for example by our MiniZinc backend presented in **Paper I**, into other functional constraints, but only one of them can be implemented as an invariant at a time:

Example 3.7. Consider the functional constraint $z = x + y$. It can be rewritten into $x = z - y$ and $y = z - x$, which are also functional constraints and also have an output variable that is statically defined by the input variables. However, due to Definition 3.10, at most one of these can be implemented as an invariant, as the output variables would otherwise statically depend on themselves. □

Constraints with Violation

For a current assignment to be a solution, it must satisfy all constraints. It may seem beneficial to somehow only allow solutions to be considered during search, so that the local search can focus on improving the objective value. However, only visiting solutions during local search can in general make the search space *disconnected*, meaning that from some initial assignment, some solutions cannot be reached via the used neighbourhood structure, as shown next:

Example 3.8. Consider the constraint $x < 50 \Leftrightarrow x > y$, an initial assignment where x is 60 and y is 100, which satisfies the constraint, and a neighbourhood structure of moves that change the current value of a single variable to any other value. If moves that yield an assignment that violates this constraint are disallowed, then there is no sequence of moves from this initial assignment that yields an assignment where x is less than 50: in order for x to take a value lower than 50, say 10, variable y must first take a value lower than 10, but y cannot take such a value lower than 50 unless x first takes a value lower than 50. This deadlock means that the search space is disconnected. This could be detrimental if for example we are minimising x, but becomes a non-issue if we allow moves to yield infeasible assignments. □

Therefore, in order to allow a transit through infeasible assignments, a CBLS solver implements the constraints in the set C of a CBLS model $\langle V, D, C, F, o \rangle$ as constraints with violation:

Definition 3.11. For a constraint $c(x_1, \ldots, x_n)$, a *constraint with violation* is obtained by first introducing a solver-internal defined variable, called its *violation variable* and denoted by $v(c(x_1, \ldots, x_n))$, and then using invariants for defining its value, called the *violation*. The violation must be 0 if the current assignment of $x_1, \ldots, x_n$ satisfies $c(x_1, \ldots, x_n)$, and must otherwise be a positive integer corresponding to how far away the current assignment is from satisfying the constraint.

Example 3.9. Consider the constraint $x \leq y$ and a current assignment where x is 0 and y is 10. The violation variable can be defined using the invariant $v(x \leq y) \leftarrow \max(0, x - y)$, as in [48]. Since the constraint is satisfied by the current assignment, the violation variable $v(x \leq y)$ has value 0. Upon a change of x to value 30, the constraint is no longer satisfied and $v(x \leq y)$ is changed by the invariant to take value 20, that is the amount by which x exceeds y. □

Note that for a CBLS model $\langle V, D, C, F, o \rangle$, the constraints of both C and F are implemented as invariants, the difference being that the invariants for C define solver-internal variables, which are not in V.

Example 3.10. Consider the constraint AllDifferent($[x_1, x_2, x_3]$), where the union of the domains of the variables x_i is some set S, and a current assignment where $[x_1, x_2, x_3]$ is $[1, 2, 3]$. As in [48], the violation variable can be defined using the invariant

$$v(\text{AllDifferent}(X)) \leftarrow \sum_{d \in S} \max(0, \text{count}(X, d) - 1)$$

where $\text{count}(X, d)$ denotes the number of occurrences of d in the array X. Note that this invariant can in turn be expressed by using invariants for $\sum$, max, and count. Since the constraint is satisfied by the current assignment, the violation is 0. Upon a change of both x_1 and x_2 to the current value 3 of x_3, the constraint is no longer satisfied and $v(\text{AllDifferent}([x_1, x_2, x_3]))$ is changed by those invariants to take value 2, that is the number of variables x_i that must take a new value in order to satisfy the constraint. □

There is no consensus on what the violation of a constraint should measure, or alternatively what its unit is. In Example 3.9, the violation is defined as the minimum total numeric increase (for x) or decrease (for y) needed for at least one of the variables in order to satisfy the constraint: the violation is not proportional to the number of variables that need to be changed. But in Example 3.10, the violation is defined as the minimum Hamming distance between the current assignment and an assignment that satisfies the constraint: the violation is the number of variables that need to be changed. Indeed, this is a design choice in the CBLS solver that is individual for each constraint predicate. See [48, pp. 101–102] for a further discussion of this topic.

Since the violation of all constraints in a CBLS model must be 0 in every solution, we also have the notion of global-violation variable:

Definition 3.12. The *global-violation variable*, denoted by $g(C)$, of the constraints in a set C is a solver-internal variable that represents an aggregation of the violation variables for C. The variable $g(C)$ must take value 0 when the violations of all constraints in C are 0, and otherwise must take a positive integer value corresponding to how far away the current assignment is from satisfying all the constraints in C.

In practice the global-violation variable is usually defined as a weighted sum of the individual violations, with each violation weight being either given as an annotation to its constraint, or 1 by default. Note that other aggregations are possible: for example, the global-violation variable can be defined as the maximum among all violation variables.

By having constraints with violation and thereby allowing infeasible assignments to be visited, the function to minimise by the local search for a CBLS model $\langle V, D, C, F, o \rangle$ is therefore not just the value of o, but also the value of $g(C)$. This can in principle be achieved by performing a lexicographic

minimisation of the objective $\langle g(C), o \rangle$, but in practice the two components of this pair are usually aggregated by the modeller in a problem-specific way, usually in the local-search strategy.

Implicit Constraints
A constraint of a problem that is part of neither the set C nor the set F in a CBLS model $\langle V, D, C, F, o \rangle$ must be implemented as part of the local-search strategy:

Definition 3.13. A constraint of a problem is an *implicit constraint* when it is initially satisfied by the initialisation strategy and always maintained satisfied by the neighbourhood structure.

Example 3.11. In Example 3.10, we showed how an AllDifferent($[x_1, \ldots, x_n]$) constraint can in the general case be implemented as a constraint with violation. We now show how to implement it as an implicit constraint in the particular case where each x_i has the *same* domain S of size n (that is the number of x_i variables), meaning that $[x_1, \ldots, x_n]$ must be a permutation of S. By initialising $[x_1, \ldots, x_n]$ to be a random permutation of S and using a swap neighbourhood structure (see Example 3.3), the constraint is satisfied by the initial assignment and cannot be violated by any moves, as swapping the position of two values in a permutation can only create another permutation. Note that even in the general case, regardless of the domain of each x_i, an AllDifferent($[x_1, \ldots, x_n]$) constraint can be implemented as an implicit constraint (see, e.g., **Paper I**). □

The use of implicit constraints means that a CBLS model might not be a complete description of a problem, and that the model can only be used together with a local-search strategy that implements the implicit constraints, if any. One can of course include the implicit constraints as constraints explicitly in the model, in order to make the model portable to arbitrary solving technologies, but since a local-search strategy that implements implicit constraints is usually desirable (as the search then ensures that those constraints are always satisfied), and since having those constraints explicitly in the model comes with an overhead within the invariant propagation, the implicit constraints are usually omitted from the CBLS model in practice.

Local-Search Strategies in CBLS
Recall from the overview of Section 3.2.1 that a local-search strategy is a particular design of an initialisation strategy, a neighbourhood structure, a heuristic, and a meta-heuristic. A local-search strategy for a CBLS solver is conceptually not different from what is described in that section, except that it now utilises the components of a CBLS model $\langle V, D, C, F, o \rangle$. Specifically, it evaluates the objective function by reading the current value of the objective

variable o, it considers the global-violation variable $g(C)$, and it considers the violation variables of the individual constraints in C.

Some CBLS solvers provide features that simplify some aspects of expressing a local-search strategy, or even allow the entire local-search strategy to be synthesised from the model, as discussed in the following mini-survey.

CBLS Solvers

There are around a dozen frameworks, such as [3, 6, 14, 15, 30, 32, 35, 39, 48, 50], that can be classified as CBLS solvers in that they provide generic features, such as a modelling language or a programming library, for expressing models that can be utilised by a local-search strategy. For a CBLS solver, a local-search strategy is either expressed by the modeller declaratively, or expressed by the modeller procedurally as a local-search algorithm, or synthesised by the solver, or obtained by a combination thereof. We here briefly discuss some notable solvers, with a focus on how the local-search strategy is obtained. Note that some modelling languages, including MiniZinc, have no syntax for expressing a local-search strategy, and therefore solvers (or backends) that use such a language must synthesise it.

Localizer

Historically, the first CBLS solver was Localizer [32], which has a declarative modelling language and introduced the idea of using invariants for efficiently computing the values of some variables and implementing functional constraints. Localizer introduced, and requires the use of, syntax for declaratively expressing parts of the local-search strategy. This syntax allows multiple neighbourhood structures to be combined declaratively as part of the local-search strategy. Localizer is currently not publicly available.

Comet

Comet [48] succeeded Localizer, by extending most of its features. Among other things, Comet introduced differentiable objects [33] as an alternative and more efficient approach for probing the changes of the global-violation variable or objective variable that are caused by moves. In terms of expressing a local-search strategy, Comet departed from the declarative approach of Localizer in favour of expressing neighbourhoods in a more procedural fashion, by generating and using closures for expressing moves [47]. This allows more complex local-search strategies to be expressed, including ones that include systematic-search components. As an alternative to having a modeller-expressed local-search strategy, Comet can also synthesise one by treating some constraints in the model as what we call implicit constraints [49]. These constraints have to be annotated in the model with the keyword '**hard**', and this requires that a pre-defined initialisation strategy and neighbourhood structure exist for the predicate of such a constraint. A predefined (meta-)heuristic then has to be selected for the model, which gives some further control over the

synthesised local-search strategy. Comet is currently available upon request to its authors.

Kangaroo
Kangaroo [35] is mainly an effort towards improving the algorithms on invariants by using what they call *lazy propagation* (not to be confused with propagation in constraint programming). Kangaroo is implemented in C++ and provides a library for expressing CBLS models, as opposed to the dedicated modelling languages of Localizer and Comet. The local-search strategy has to be expressed by the modeller procedurally. To the best of my knowledge, Kangaroo was never publicly released.

LocalSolver
LocalSolver [6] is a commercial CBLS solver that provides both a library (for multiple programming languages) and a dedicated modelling language for expressing CBLS models. Notably, LocalSolver always synthesises the search strategy from the model and it has no syntax for expressing one.

OscaR.cbls
OscaR.cbls [14], which is part of the OscaR framework [36], is a CBLS solver written in the Scala programming language and is based on the design of Comet as described in [48]. OscaR.cbls is a library for expressing CBLS models and search strategies. Since Scala allows most of its syntax to be redefined on-the-fly, OscaR.cbls is able to provide a syntax that is fairly similar to the modelling language of Comet while still being a library. Local-search strategies are primarily expressed procedurally, but OscaR.cbls was extended with a library of *neighbourhood combinators* [13], which allow neighbourhood structures to be declaratively combined in order to form more complex ones. Note that the neighbourhood structures to be combined still have to be expressed procedurally. OscaR.cbls is open-source under the GNU lesser general public license. In **Paper I**, we were the first to show how to create a CBLS backend for MiniZinc: it uses OscaR.cbls as the underlying solver. Since MiniZinc does not have syntax for expressing a local-search strategy, we show how it can be synthesised from a MiniZinc model (see the summary in Chapter 4).

Yuck
Yuck [30] is a CBLS solver in the style of Comet and has a backend for MiniZinc. The backend of Yuck is comparable to what we introduced in **Paper I** (see Chapter 4) and synthesises a local-search strategy since MiniZinc has no syntax for expressing one. The local-search strategy is based on Simulated Annealing and uses a neighbourhood structure that is a combination of generic neighbourhood structures and implicit constraints. Yuck is open-source under the Mozilla Public License 2.0.

Athanor

The Athanor solver [3] uses Essence [17] as modelling language and implements a CBLS model by a parse tree of the Essence model. The parse tree also serves as an evaluation tree. It is not clear if these trees can represent variables that dynamically depend on themselves, as a tree by definition cannot contain cycles and thereby would be unable to represent such cyclic dependencies. A novelty of Athanor is that it dynamically unrolls quantifications. For example, for the constraint $\forall i \in S : r_i \Rightarrow c_i$, which states that the constraint c_i must be satisfied whenever the constraint r_i is satisfied, Athanor dynamically introduces the invariants for c_i when r_i is satisfied, and removes the introduced invariants when r_i is not satisfied. Athanor uses an approach similar to [1] in order to synthesise a local-search strategy from an Essence model (as there is no syntax for expressing one) by making use of variables of arbitrarily nested types, such as a sequence of sets of sequences of integer variables. In short, multiple neighbourhood structures are synthesised by combining pre-defined parametric neighbourhood structures of each type (integer, sequence, set, etc.). Then the (meta-)heuristic uses a regret-minimising multi-armed bandit algorithm for selecting one of the neighbourhood structures to use at each iteration. Athanor is open-source under the BSD-3-Clause license.

3.2.3 Large-Neighbourhood Search (LNS)

Large-neighbourhood search (LNS) [45] is a popular local-search algorithm for solving optimisation problems. LNS can be implemented on top of a solver of any solving technology with systematic search and thereby makes use of a model provided to that solver, which by extension makes LNS a solving technology.

Overview

We describe LNS in the context of constraint programming (CP) technology (see Section 3.1). A prototypical algorithm for LNS is Algorithm 3.3 and takes as argument a CP model. We highlight some of its similarities to Algorithm 3.2, but also invite the reader to the compare them further. We first describe the operators of Algorithm 3.3, namely initialising, freezing, and improving, and then discuss in more detail some variations and improvements of these operators. We do not discuss lines 3 and 7 as they are essentially like in Algorithm 3.2.

Initialisation

In order to solve an optimisation problem for which we have a CP model $\langle V, D, C, o \rangle$, the LNS-SOLVE algorithm requires an initial solution, which becomes what we call the *current solution* A of V (line 2). This initial solution is usually obtained by calling CP-SOLVE$(\{v \mapsto D(v) \mid v \in V\}, C, o, B)$ of Algorithm 3.1 with some branching strategy B, but limiting the CP solver to stop

Algorithm 3.3 A prototypical LNS algorithm.

```
1: procedure LNS-SOLVE(⟨V,D,C,o⟩)
2:     A ← INITIALISE(⟨V,D,C,o⟩)                    ▷ create an initial solution
3:     while not DONE(A) do      ▷ loop while stopping criteria are not met
4:         F ← FREEZE(A)                              ▷ create a fragment F ⊂ A
5:         A′ ← IMPROVE(⟨V,D,C,o⟩,F)     ▷ improve A based on fragment
6:         A ← COMMIT(A,A′)        ▷ commit to A′ if it is improving over A
7:     return A
```

after finding a first (and possibly sub-optimal) solution. Note that in principle there are no requirements on how the initial solution is actually obtained.

Freeze

In each iteration of LNS-SOLVE, the FREEZE operator selects a non-empty strict subset of A, called the *fragment* and denoted by F (line 4). The variables in the fragment are then *frozen*, that is they are forced to keep their value in the current solution during execution of the subsequent IMPROVE operator. Some variables, such as the objective variable o, are never in the fragment, as discussed when presenting LNS variations below.

Note that the FREEZE operator defines the set of similar assignments that will be probed and thereby corresponds to the GENERATENEIGHBOURS operator in Algorithm 3.2. Note that an assignment need not be a solution, but that LNS only visits solutions, due to the IMPROVE operator described next.

Improve

Next, the IMPROVE operator uses the CP solver in order to look for an improving solution A' (line 5) where each variable in F has the same value as in A, but with an improved objective value, i.e., $A'(o) < A(o)$. That is, IMPROVE calls CP-SOLVE$(\{v \mapsto D(v) \mid v \in V\}, C \cup \{v = A(v) \mid v \in F\} \cup \{o < A(o)\}, o, B)$ with some branching strategy B.

The CP solver called by the IMPROVE operator is usually limited to perform an incomplete systematic search, meaning that the search is given a budget such as a time limit or a limit on the number of failed nodes. This budget is usually small so that many fast iterations are performed per time unit, rather than a few slow ones. This, however, means that the IMPROVE operator can fail to find a solution, i.e., that A' does not exist when the budget is exhausted. If a new solution A' is found, then it is committed and replaces the current solution A, otherwise A remains unchanged (line 6).

The IMPROVE operator essentially probes a large neighbourhood of assignments that share the fragment F. We say that these assignments *overlap*, as each variable in F is unchanged across these assignments. Note that the IMPROVE operator in Algorithm 3.3 corresponds to the SELECTMOVE operator

in Algorithm 3.2 as it selects a neighbour from the neighbourhood defined by the FREEZE operator.

LNS Variations

Selecting the fragment is arguably the most crucial design aspect of LNS. Usually, some variables should never be in the fragment as they become fixed via CP-style propagation when the other variables are fixed, or because they are expected to change in each iteration, such as the objective variable o. Specifically, a variable that is functionally defined by a constraint should never be in the fragment. For the sake of simplicity, we do not refer to these variables when we discuss which variables the FREEZE operator considers for freezing. Furthermore, the FREEZE operator should freeze sufficiently few variables so that an improving solution can be found, but not so few that the search space of the CP solver becomes so large that the solver is unable to find any new solution before its budget is exhausted.

A crude but problem-independent and surprisingly effective FREEZE operator is to randomly select some percentage of the variables of V to put in the fragment, with percentages between 80% and 90% often being good choices in practice. LNS that uses this FREEZE operator is often referred to as *random LNS*, and is what we use in **Paper V** quite successfully.

However, for most problems it is believed to be beneficial to use a problem-specific FREEZE operator. For example, in [45] where LNS is used for solving vehicle-routing problems, the FREEZE operator is problem-specific. It iteratively constructs the fragment from the empty set by adding a variable (that represents a location) at random, but with a bias based on its relatedness (such as short travel distance) to variables (i.e., locations) already in the fragment, until the fragment is sufficiently large.

There have also been some successful approaches for defining the relatedness of variables in order to select a fragment in a problem-independent way. Notably, relatedness is defined in [37] by how many values each variable causes CP propagators to remove from other variables' current domains; in [29] by the impact on the objective that the current value of each variable has; and in [41] by explanations of why, for example, an improving solution could not be found.

It can be beneficial for the IMPROVE operator to allow a non-improving solution, or at least a non-worsening one, as LNS can otherwise end up in a local minimum that it is unable to escape. For example, as an extreme case, it could be that the current solution does not overlap with any improving solution, but that there are (non-)worsening solutions that overlap with both the current solution and an improving one.

Finally, we again emphasise that using CP as the underlying solving technology of LNS is also a design choice and that the IMPROVE operator can indeed be implemented by using any solver of any solving technology with systematic search, or even by problem-specific algorithms that repair partial assignments.

4. Summaries of Papers

I A CBLS Backend for MiniZinc

In **Paper I** we were the first to show how to create a CBLS backend for a technology-independent modelling language like MiniZinc. We built our backend, which we call fzn-oscar-cbls, by targeting the OscaR.cbls solver [14]. Its implementation is open-source under the GNU lesser general public license.[1] Recall from Section 2.2.2 that a backend for MiniZinc must *translate* any FlatZinc model, which is a CP model $\langle V, D, C, o \rangle$, into a model for the targeted solver. Furthermore, a backend must *synthesise* any remaining components required by the solver, here an entire local-search strategy (and its algorithm), that is an initialisation strategy, a neighbourhood structure, a heuristic, and a meta-heuristic. Both of these aspects are non-trivial when targeting a CBLS solver, as discussed next.

The first challenge is the translation step, as there are discrepancies between a FlatZinc model $\langle V, D, C, o \rangle$ and a CBLS model $\langle V', D', C', F', o' \rangle$, the trivial translation being $V' = V$, $D' = D$, $C' = C$, $F' = \varnothing$, $o' = o$. Recall from Section 3.2.2 that a constraint of a problem can be implemented in one of three ways for a CBLS solver: as an invariant in case the constraint is functional (called a *one-way constraint* in **Paper I**) and in F', as a constraint with violation if it is in C', or as a constraint that is implicit and part of the local-search strategy; the latter two ways are also applicable for functional constraints. However, at the moment MiniZinc provides no syntax for expressing that a constraint should belong to any of these categories.

We address this first challenge by proposing what we in our subsequent papers and here call a *structure identification scheme*. The scheme automatically partitions the constraint set C into the sets C', F', and N', where N' is the set of implicit constraints that are to be implemented by the neighbourhood structure. Creating this partition is non-trivial as there can be many partitions of C and there is no well-defined way of determining which partition is best (or even what it means to be best). Furthermore, any partition must result in a valid CBLS model (see Definition 3.10). Our structure identification scheme prioritises identifying as many constraints as possible to be among the functional constraints F' by considering constraint reformulations like in Example 3.7. Note that variables can in principle dynamically depend on themselves, namely

[1] `https://www.it.uu.se/research/group/optimisation/software#fznoscarcbls` – Accessed on March 5, 2021. Note that fzn is an abbreviation of FlatZinc.

when the local-search strategy ensures that the value of each variable is never actually computed in terms of itself. However, determining when this is possible and synthesising an appropriate local-search strategy requires a significantly deeper analysis of the model. Therefore, our approach always avoids partitions where variables in any way depend on themselves. This is a limitation of the current approach. We believe it is beneficial to prioritise having constraints in F', as this means that there are more defined variables in the CBLS model, which in turn gives fewer search variables for the local-search strategy to consider. Only some constraint predicates, namely those for which we have a predefined neighbourhood structure such as AllDifferent (see Example 3.11), are considered for implicit constraints. See Section 4.2 of **Paper I** for further details on our structure identification scheme.

The second challenge is the synthesis of a local-search strategy (and algorithm). Since a FlatZinc model does not contain any local-search strategy we design one that can run on any CBLS model $\langle V', D', C', F', o' \rangle$ and set N' identified by the structure identification scheme.

We address this second challenge by proposing a local-search strategy that is both problem-specific, namely when using implicit constraints, and generic, namely by using a variation of Tabu Search (as explained below). In particular, the neighbourhood structure of the local-search strategy is the union of several neighbourhood structures that are each either specific to an implicit constraint or generic. We ensure that all search variables belong to at least one neighbourhood structure. The initialisation strategy follows from the neighbourhood structure. The heuristic and meta-heuristic go through three consecutive phases:

1. a greedy phase tries to quickly improve the global violation $g(C')$ of the initial assignment by, for a few iterations, focusing on changing the variables that do not belong to the neighbourhood structure of an implicit constraint;
2. a satisfaction phase uses a Tabu Search in order to find a first solution, by only minimising $g(C')$ until it reaches zero (which means that a solution was found); and
3. if a solution is found, then an optimisation phase uses a Tabu Search in order to minimise the objective $\alpha \cdot g(C') + \beta \cdot o'$, where α and β are positive coefficients that are dynamically tuned during search in order to balance between improving the global violation, which may increase again during phase 3, and improving the value of the objective variable o'.

The Tabu Searches used in the last two phases are adaptive in the sense that they monitor the change in the global violation (and objective value) and change the tabu tenure whenever the search appears to have stagnated. See Section 5 of **Paper I** for further details on our local-search strategy.

We experimentally evaluate the performance of fzn-oscar-cbls by running it on all models and instances that appeared in the MiniZinc Challenge [46] between the years 2010 and 2014; this is an ongoing annual competition between solvers with a backend for MiniZinc. Our evaluation shows the

importance of partitioning C and in particular that having constraints in F' has (perhaps not unsurprisingly) a significant positive impact on solving time and solution quality. Furthermore, our evaluation shows that fzn-oscar-cbls is complementary to other backends as it can easily solve, sometimes even to optimality, some instances that the other backends had difficulty solving during a MiniZinc Challenge.

Update

Our backend fzn-oscar-cbls has evolved since publishing **Paper I**. One notable addition is the use of CP propagators from the OscaR.cp solver (of the same OscaR framework as OscaR.cbls). We use the CP propagators in order to reduce the domains of variables in the CBLS model before and during the local search. Indeed, many published MiniZinc models have domains that are not declared as tight as they could be, which means that performing propagation before starting the local search can be crucial for good performance, whereas solvers that perform systematic search (such as CP solvers) are usually not hindered by non-tight declared domains. Furthermore, whenever a new solution is found, we use CP propagators for the new best value of the objective variable in order to possibly remove values in the domains of some search variables. If values are removed, then this decreases the size of the search space and again sometimes drastically improves the solving time.

Between the years 2015 and 2020 (inclusive) we entered fzn-oscar-cbls into the annual MiniZinc Challenge. Each year fzn-oscar-cbls has overall ranked low, but has been one of the top backends for at least one of the 20 problems. The overall low ranking is not surprising: many of the MiniZinc models used for the challenge are not formulated in a local-search friendly way, in addition to having non-tight domains. Specifically, there often are models where what could be expressed as functional constraints is instead expressed by non-functional ones (see **Paper II**). Compared to the functional formulations, such non-functional formulations can have an insignificant effect on backends of solving technologies with systematic search, but they sometimes prevent fzn-oscar-cbls from finding any solutions at all, which yields an overall low ranking. But, on a positive note, the fact that fzn-oscar-cbls each year excels on a few problems is very promising and highlights its complementary nature: often these are problems where all other solving technologies struggle to find a high-quality solution, if even any at all. Overall, one might surmise that when an appropriate model is used, then fzn-oscar-cbls, and by extension CBLS via MiniZinc, can be highly competitive. This warrants further research on dealing with unfavourable model formulations.

II Generating Compound Moves in Local Search by Hybridisation with Complete Search

The structure identification scheme of **Paper I** implicitly partitions the variables of a FlatZinc model into two sets V_s and V_d of respectively the search variables and the defined variables. The synthesised local-search strategy defines moves on all variables in V_s in its neighbourhood structure. However, following the publication of **Paper I** we noticed empirically that for some models, including the TSPTW model shown in Model 2.2, the synthesised local-search strategy performs significantly worse than expected. It turns out that when a model has variables that represent auxiliary information, which we call *auxiliary variables*, such as the `ArrivalTime[i]` variables in Model 2.2 (as discussed below), and these variables end up in the set V_s, then the local-search strategy is unable to perform meaningful moves on these variables.

The intuition behind why it is challenging to make a meaningful local-search move on an auxiliary variable is that it should ideally only be changed in response to a move on one or more non-auxiliary variables. Furthermore, an auxiliary variable should ideally not even be changed via local-search moves, as a value can often be easily determined given an assignment of the non-auxiliary variables. For example, in Model 2.2 it is only meaningful to change the values of the `ArrivalTime[i]` variables in response to a change of some `Route[i]` variable: for a given assignment of the `Route[i]` variables it is trivial to compute an assignment of the `ArrivalTime[i]` variables without requiring any local search on them.

In **Paper II** we address this issue of auxiliary variables, and by extension improve the performance of fzn-oscar-cbls, by proposing what we call *compound-move generation* (CMG). First, we offer two approaches for identifying auxiliary variables in a model: allowing a modeller to annotate variables in a MiniZinc model in order to explicitly designate auxiliary ones; and a crude automatic scheme where variables that do not belong to a neighbourhood structure of an implicit constraint are assumed to be auxiliary. Then, the local-search strategy is modified to only consider moves on non-auxiliary variables and, upon each move, use a solver with systematic search (called *complete search* in the paper) to determine values of all auxiliary variables. This means that we augment each move on non-auxiliary variables into a *compound move* that also changes all auxiliary variables.

A similar idea was proposed in [12] for vehicle routing problems, where propagation (but not search) by a CP solver is used after each move in order to check if there is a feasible assignment of the auxiliary variables. CMG can be seen as a generalisation of this, as [12] does not consider what to do when feasibility cannot be determined via only propagation (which is why we use also systematic search) and as we present CMG in a problem-independent context.

We implement CMG in fzn-oscar-cbls by using the CP solver OscaR.cp as the systematic-search solver and discuss several refinements and variations of CMG. In our experimental evaluation we see that by using CMG we significantly improve the performance of fzn-oscar-cbls on some problems.

Finally, it should be noted that CMG is not just beneficial for backends that target CBLS solvers, but also for local-search solvers in general, as CMG is a generic way of creating meaningful moves for a model with auxiliary variables. Indeed, a similar approach was afterwards proposed in [8] for the local-search solver LocalSolver [6].

III Declarative Local-Search Neighbourhoods in MiniZinc

A drawback of **Paper I** is that *all* parts of the local-search strategy, that is an initialisation strategy, a neighbourhood structure, a heuristic, and a meta-heuristic, have to be synthesised from a MiniZinc model. This is by no means a perfect process, and can at times make it very challenging for the backend to solve some problems. This can be of particular frustration to a modeller attempting to model a problem where a good local-search strategy is well known and might even be easy to formulate directly in the underlying CBLS solver of the MiniZinc backend. Indeed, for the same reasons, MiniZinc offers *search annotations*, which are hints on which branching strategy a CP backend should use.

In **Paper III** we address this limitation by extending the MiniZinc language and toolchain with syntax and support for declaratively defining a neighbourhood structure and an initialisation strategy at the MiniZinc level, which we jointly call a *declarative neighbourhood*, which is then passed via FlatZinc to any backend that cares to use it. This means that local-search backends then only need to synthesise a heuristic and a meta-heuristic based on the now prescribed neighbourhood structure and initialisation strategy, which gives the modeller more control over the backends and more predictable performance. We show how to use a declarative neighbourhood within fzn-oscar-cbls and how its moves can be efficiently evaluated by using key features of CBLS solvers, namely invariants and constraints with violation.

The main challenge is that the declarative neighbourhood defined at the MiniZinc level should (or arguably must) be independent of any backend and its solving technology, in the very spirit of MiniZinc. Thereby, the main contribution is our description of how to compile a declarative neighbourhood into an extended version of FlatZinc that supports function declarations, is still a subset of MiniZinc, and does not rely on any CBLS-specific details.

We experimentally evaluate the expressiveness of declarative neighbourhoods and their impact on performance by stating in our new syntax some well-known neighbourhood structures and initialisation strategies, including

some of the ones that fzn-oscar-cbls hard-codes for implicit constraints. The evaluation shows that declarative neighbourhoods only have a small overhead compared to the hard-coded implementations of neighbourhood structures and initialisation strategies for implicit constraints, and that declared problem-specific neighbourhood structures and initialisation strategies (unsurprisingly) outperform non-specific ones.

IV Exploring Declarative Local-Search Neighbourhoods with CP

In **Paper IV**, we show that the declarative neighbourhoods of **Paper III** can not only be used in a CBLS backend for MiniZinc, but can also be used to perform local search with any systematic-search solver. This shows that our declarative neighbourhoods are actually independent of solving technologies.

We accomplish this by showing how the flattened version of any declarative neighbourhood can be transformed into a CSP that is parameterised by the current assignment and encodes the neighbourhood, i.e., the solutions to the CSP are the neighbours of the current assignment. This CSP can then be solved by any systematic-search solving technology; in particular we use CP for our implementation.[2] This approach, which is conceptually similar to yet different from LNS, is introduced as a methodology in [38] and is contemporary with both LNS and CBLS. However, this methodology did not gain much traction at the time, possibly because the encoding has to be handcrafted for each problem and each neighbourhood structure, and thus has limited reusability as opposed to what is enabled by LNS and CBLS. However, starting from our declarative neighbourhoods, we show how to generate this encoding automatically.

In order for our encoding to be efficient, we also introduce a new global-constraint predicate, called Writes, which is now part of MiniZinc. As part of our contribution, we define an efficient decomposition for this predicate.

Finally, we show that by using our encoding based on the Writes predicate we are able to implement a local-search solver that is a backend for MiniZinc by only using a CP solver. We thereby provide an approach that is orthogonal to **Paper I** for creating a local-search backend for MiniZinc.

Our experimental evaluation of our new backend, which we call LS(CP) (read "local search using CP") and implement on top of the OscaR.cp solver, shows that LS(CP) can be competitive with backends using either CBLS or LNS. Interestingly, LS(CP) does not suffer from the issue with auxiliary variables that we addressed in **Paper II**, as it is a side-effect for LS(CP) to find values for these variables for each move by using systematic search rather than local search.

[2]Note that in **Paper IV** we use the term *valuation* in place of *assignment* when in a local-search context, and the term *solution* to refer to the assignment that the CP solver obtains when exploring a neighbourhood.

V Solving Satisfaction Problems using LNS

Recall from Section 3.2.3 that large-neighbourhood search (LNS) is only defined for optimisation problems and that it requires an initial solution for the search to start. Therefore, in order to use LNS to solve a satisfaction problem, we must first soften some constraints in a manner analogous to the constraints with violation of CBLS (see Definition 3.11). For example, a soft constraint for $x+y=z$ can be expressed by introducing a variable v, called a *violation variable*, and a variable s, and using the constraints $s=(x+y)-z$ and $v=|s|$, which constrain the variable v to take value 0 if the original constraint holds and otherwise to be the distance s between $x+y$ and z. We call $x+y=z$ the *softened constraint*, and $s=(x+y)-z \wedge v=|s|$ its *soft constraint*. LNS can then be used to try and reduce the sum of all introduced violation variables to zero, thus satisfying the original satisfaction problem.[3] That is, we now have an optimisation problem where an initial solution can easily be found (because even large values of violation variables are fine), thus fulfilling the requirements of LNS.

In fact, it is often necessary to soften constraints when using CP in a local-search context in general (even one that is not LNS), as local-search strategies often need to be able to explore infeasible parts of the search space, as shown in Example 3.8. Indeed, in both **Paper II** and **Paper IV**, some constraints were automatically softened in the CP models (which are solved as subtasks during local-search solving) in order for the local search to progress, as mentioned in the respective papers.

However, softening constraints in this manner for CP can give poor performance in general as the soft constraints rarely cause propagation, which can result in a significantly longer local search and the softened constraints not being satisfied until very late in the local search. This is not only an issue when solving satisfaction problems with LNS, but also something we observed in the experiments of both **Paper II** and **Paper IV**.

In **Paper V**, we address this issue by defining a new type of propagator, which we call a non-failing propagator. Our key observation is that softening for a CP solver entirely replaces a constraint, and thereby its propagator. However, this is often overkill, as the propagator for the softened constraint could have performed some (but not all) propagation while still allowing infeasible parts of the original search space to be explored during the CP search at each LNS iteration. A *non-failing propagator* removes values right up until it would empty the current domain of some variable, and at that point the propagator is disabled instead of forcing a backtrack. This means that a non-failing propagator can never itself force a backtrack.

Our approach is then to use a non-failing propagator for each softened constraint *in addition to* a normal propagator for its introduced soft constraint. We show that, by only making a few tiny changes to the PROPAGATE procedure

[3] Note that in **Paper V** we use *penalty* as a synonym for what we here call *violation*.

called at line 2 of Algorithm 3.1 in an existing CP solver, *any* propagator can be made non-failing without modifying its code. Our experimental evaluation shows that non-failing propagators for the softened constraints, when used in conjunction with normal propagators for soft constraints, can greatly improve LNS performance on satisfaction problems, compared to just using soft constraints.

Furthermore, some optimisation problems are hard to satisfy, which means that it is difficult to use LNS for solving these problems as the local search cannot start until an initial solution has been found. Thankfully, finding an initial solution is a satisfaction problem, so we can use softening and an initial round of LNS for finding an initial solution. We also evaluate the use of non-failing propagators and softening for solving this initial satisfaction problem, and show not only that the use of non-failing propagators again greatly improves LNS performance, but also that (perhaps a bit surprisingly) once an initial solution has been found, the LNS iterations can sometimes easily improve this solution without the need for the soft constraints.

While **Paper V** focuses on LNS, its contributions should also significantly benefit the local-search backends of **Paper II** and **Paper IV**, but testing this is future work.

Clarifications

In **Paper IV**, we claim that "[since] the max-clique problem reduces to achieving domain consistency on a Writes constraint, domain-consistent propagation is NP-hard". However, it would have been more intuitive to instead say that the graph-colouring problem reduces to achieving domain consistency. A proof can be sketched as follows, but note that it uses terminology not covered by this introduction of this thesis. The $\text{Writes}(O,I,P,V)$ constraint requires among other things that $\forall i \neq j : V_i \neq V_j \Rightarrow P_i \neq P_j$, which means that we can define a dynamically evolving graph where each node corresponds to a P_i variable and there is an edge between each pair P_i and P_j when $\text{dom}(P_i) \cap \text{dom}(P_j) = \varnothing$ or $\text{dom}(V_i) \cap \text{dom}(V_j) = \varnothing$. That is, there is an edge between each pair that must take different values. The chromatic number of this graph is a lower bound on $|\cup_i \text{dom}(P_i)|$ and unsatisfiability can be determined early when $|\cup_i \text{dom}(P_i)|$ is less than the lower bound. Note that *any* graph can appear upon having enough variables P_i and tailoring each $\text{dom}(V_i)$ to force edges, namely by associating a unique integer with each edge (i, j) not in the graph and having that integer only in both $\text{dom}(V_i)$ and $\text{dom}(V_j)$: the graph can thereby be independent of each $\text{dom}(P_i)$.

The Yuck solver runs in a deterministic mode by default, which was not known to us at the time of writing **Paper III** and **Paper IV**. This means that Yuck was essentially evaluated over a single run rather than 10 in both papers.[4]

[4]Personal communication by Michael Marte on July 17, 2020.

Algorithm 1, which is used to exemplify propagation, of **Paper V** is only correct for *positive* integer coefficients in the array A. However, this does not in principle affect any of the examples of the paper, where negative coefficients are in fact used, as any negative coefficients can be dealt with by using so-called variable views. In any case, the solver used in the experimental evaluation has a correct implementation and Algorithm 1 is not a contribution of that paper.

5. Conclusion

With the increasing focus on high-level and solver-independent modelling languages like MiniZinc, the model-and-solve approach is becoming more accessible to a wider audience.

This thesis can be summarised as a push towards making the power of local search more accessible via MiniZinc. We accomplish this by contributing to three research topics over five papers:

- In **Paper I**, we were the first to show how to create a constraint-based local search (CBLS) backend for a language like MiniZinc, and that such backends can be competitive with those of solving technologies that are based on systematic search. Furthermore, in **Paper II** we identified a weakness with CBLS backends and showed how to overcome this weakness.
- In **Paper III** and **Paper IV**, we introduced syntax to MiniZinc for declaratively defining a neighbourhood structure and an initialisation strategy, which allows a local-search concept to be part of MiniZinc, similarly to how constraint programming (CP) branching strategies are already part of the MiniZinc language. We showed how these neighbourhood structures can be flattened in a solver-independent way, and how they can be further encoded and used by any solving technology with systematic search.
- In **Paper V**, we focused on another local-search technology, namely large-neighbourhood search (LNS), and showed how LNS can be improved upon for solving satisfaction problems by improving how CP solvers deal with soft constraints. This also benefits other local-search technologies that rely on CP, like those shown in **Paper II** and **Paper IV**.

These contributions allow more problems to be successfully tackled via languages such as MiniZinc, as some problems are in fact best solved via local search. Furthermore, this may eventually reduce the need for ad-hoc local-search algorithms.

Of course, there is still much work to be done, and this thesis should be seen as a first step. Here we created a backend for MiniZinc based on CBLS, and focused on improving its neighbourhood structures. Future work therefore includes improving the remaining components of a synthesised local-search strategy: the heuristic and the meta-heuristic. In addition, other aspects of a CBLS backend, such as its structure identification scheme, also deserve further research.

Sammanfattning på svenska

Optimeringsproblem är något som ständigt dyker upp i industrin och i samhället. För att lösa dem väl behöver man fatta optimala beslut som uppfyller vissa villkor och begränsningar.

Låt oss till exempel titta på problemet med att leverera paket i en stad, vilket är en viktig del av dagens e-handel. Vi har en depå där alla paket lagras för att sedan skickas ut med olika fordon för leverans. Varje paket har en volym och en destination, och varje fordon är begränsat i den totala volymen av paket som det kan bära, samt i hur långt det kan köra. För att lösa problemet måste vi bestämma vilka paket varje fordon ska leverera, samt vilken rutt som ska tas. För att vara så effektiva som möjligt, måste vi också ta optimala beslut gällande aspekter såsom den totala leveranstiden, den totala bränsleförbrukningen, och antalet fordon som används. Alltså, för att lösa detta problem måste vi fatta beslut som optimerar vissa aspekter, samtidigt som vi uppfyller villkor och tar hänsyn till kapacitetsbegränsningar.

Ett annat exempel där optimering behövs, är problemet med personalplanering på ett sjukhus. Låt oss anta att vi har sjuksköterskor med olika expertis, och arbetsdagar som är uppdelade i skift. I varje skift finns det alltid ett krav på antalet sjuksköterskor som arbetar samt att viss expertis finns tillgänglig. För att lösa det här problemet måste vi bestämma vilka sjuksköterskor som ska arbeta vid varje skift, men också uppfylla vissa ytterligare krav (eller: begränsningar); såsom vilotiden mellan skift, antalet lediga dagar som varje sjuksköterska har under en månad, och så vidare. Vi är återigen intresserade av att optimera vissa aspekter, såsom att minimera den totala övertiden, minimera eventuell obalans mellan sjuksköterskans arbetsbelastningar i schemat eller liknande. Alltså, det här är återigen ett problem som löses genom att fatta beslut som optimerar vissa aspekter men samtidigt tar hänsyn till begränsningar.

Det ovanstående är exempel på den typ av optimeringsproblem som vi fokuserar på i den här avhandlingen. De kallas *diskreta optimeringsproblem* då man har ett givet antal diskreta alternativ för varje beslut. Problemen är mycket viktiga att lösa, men också notoriskt svåra att lösa optimalt. För enkelhetens skull kan man säga att det finns tre tillvägagångssätt för att lösa ett diskret optimeringsproblem:

1. vi kan skriva en algoritm specifikt för problemet genom att till exempel använda en designmetodik, såsom lokal sökning eller dynamisk programmering;
2. vi kan använda generell programvara, som kallas för en lösare (eng. solver), som tar som input en beskrivning av ett problem, vilket kallas för en modell, och ger i bästa fall en optimal lösning som output; eller

3. vi kan ge upp och istället lösa en förenklad version av problemet som kan vara lättare att lösa men då inte längre uppfyller alla villkor som ställs i problemet.

Eftersom diskreta optimeringsproblem kan vara mycket svåra att lösa i praktiken så är det tredje tillvägagångssättet fullt rimligt och används ofta. Dock fokuserar vi i den här avhandlingen på de andra två tillvägagångssätten där vi försöker lösa problemet optimalt och samtidigt uppfylla *alla* villkor. Det bör även nämnas att det faktiskt finns ett fjärde alternativ som är en mycket lovande lösningsmetod, nämligen kvantalgoritmer som körs i kvantdatorer. I teorin så är kvantdatorer banbrytande specifikt när det gäller att lösa diskreta optimeringsproblem, men det återstår att se ifall de även är det i praktiken: därför kommer vi inte att diskutera dem vidare här.

Det första tillvägagångssättet, som vi kallar för att skriva en ad-hoc-algoritm, är i teorin alltid att föredra eftersom den bästa algoritmen för ett problem alltid kan skrivas ad-hoc. I praktiken är det dock mycket tidskrävande att skriva en sådan algoritm från grunden, och det finns ingen garanti för att vi ens hittar en bra algoritm för problemet oavsett hur mycket tid vi lägger på det.

Det andra tillvägagångssättet, att använda en lösare, har flera fördelar jämfört med det första, och är ofta att föredra i praktiken. Till skillnad från en algoritm, beskriver en modell vad problemet är snarare än hur det kan lösas. Därmed kan en modell vara betydligt enklare att skriva och förstå än en algoritm för att lösa problemet. Samtidigt är en modell lättare att underhålla och utöka utifall att problemet förändras något, vilket ofta är fallet i praktiken. Dessutom kan vi dra nytta av när en lösare uppdateras utan att behöva modifiera några modeller. Lösare kan baseras på fundamentalt olika algoritmer. Vi kallar den typ av algoritmer som en lösare bygger på för dess lösningsteknik. En nackdel med vissa lösare är att de har sitt eget språk för att uttrycka modeller, och att en modell skriven på en lösares språk normalt sett inte kan användas i en annan lösare.

Huvudinspirationen för denna avhandling är två viktiga framsteg kring lösare och deras modelleringsspråk.

Det ena framsteget är framtagningen av en lösningsteknik som kallas villkorsbaserad lokal sökning (eng. CBLS). Efter dess introduktion har det kommit en våg av lösare som använder lokal sökning som deras lösningsteknik. Detta är en lovande utveckling, eftersom lokal sökning också är en av de mer populära metoderna för att skriva ad-hoc-algoritmer, då den snabbt kan producera nästan optimala lösningar för väldigt stora och komplicerade problem.

Det andra framsteget är att det har tagits fram flera modelleringsspråk som är oberoende av både lösare och lösningsteknik, så att vi endast behöver skriva en modell som vi sedan kan köra på många lösare. Sådana modelleringsspråk, och särskilt ett som kallas MiniZinc, ökar drastiskt tillgängligheten för lösare och möjliggör att man snabbt kan ta fram prototyper av modeller för olika lösningstekniker. Ursprungligen hade dock inga av dessa teknikoberoende modelleringsspråk stöd för CBLS-lösare.

I denna avhandling kopplar jag ihop teknikoberoende modelleringsspråk (specifikt MiniZinc) och lokal sökning. Detta arbete gör det möjligt för oss att använda lokal sökning via språk som MiniZinc, och möjliggör också att fler problem kan lösas effektivt via MiniZinc, då vissa problem faktiskt är bäst lämpade för lokal sökning. Detta arbete kan med tiden ytterligare minska behovet av att någonsin behöva skriva ad-hoc-algoritmer.

References

[1] Özgür Akgün, Saad Attieh, Ian P. Gent, Christopher Jefferson, Ian Miguel, Peter Nightingale, András Z. Salamon, Patrick Spracklen, and James Wetter. "A framework for constraint based local search using Essence". In: *IJCAI 2018*. Ed. by Jérôme Lang. IJCAI Organization, 2018, pp. 1242–1248 (referenced on page 27).

[2] Roberto Amadini, Pierre Flener, Justin Pearson, Joseph D. Scott, Peter J. Stuckey, and Guido Tack. "MiniZinc with strings". In: *LOPSTR 2016: Revised Selected Papers*. Ed. by Manuel Hermenegildo and Pedro López-García. Vol. 10184. LNCS. Springer, 2017, pp. 59–75 (referenced on page 7).

[3] Saad Attieh, Nguyen Dang, Christopher Jefferson, Ian Miguel, and Peter Nightingale. "Athanor: High-level local search over abstract constraint specifications in Essence". In: *IJCAI 2019*. Ed. by Sarit Kraus. IJCAI Organization, 2019, pp. 1056–1063 (referenced on pages 2, 25, 27).

[4] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. "Global constraint catalogue: Past, present, and future". In: *Constraints* 12(1) (Mar. 2007), pp. 21–62. The catalogue and the current working version are available at `http://sofdem.github.io/gccat` (referenced on pages 9, 10).

[5] Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark Wallace. "Improved linearization of constraint programming models". In: *CP 2016*. Ed. by Michel Rueher. Vol. 9892. LNCS. Springer, 2016, pp. 49–65 (referenced on page 11).

[6] Thierry Benoist, Bertrand Estellon, Frédéric Gardi, Romain Megel, and Karim Nouioua. "LocalSolver 1.x: a black-box local-search solver for 0-1 programming". In: *4OR – A Quarterly Journal of Operations Research* 9(3) (Sept. 2011), pp. 299–316. LocalSolver is available at `https://www.localsolver.com` (referenced on pages 2, 25, 26, 34).

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009 (referenced on page 4).

[8] Léa Blaise, Christian Artigues, and Thierry Benoist. "Solution Repair by Inequality Network Propagation in LocalSolver". In: *PPSN XVI*. Ed. by Thomas Bäck, Mike Preuss, André Deutz, Hao Wang, Carola Doerr, Michael Emmerich, and Heike Trautmann. Vol. 12269. LNCS. Springer, 2020, pp. 332–345 (referenced on page 34).

[9] Mats Carlsson, Greger Ottosson, and Björn Carlson. “An open-ended finite domain constraint solver”. In: *PLILP 1997*. Ed. by H. Glaser, P. Hartel, and H. Kuchen. Vol. 1292. LNCS. Springer, 1997, pp. 191–206. SICStus Prolog is available at `https://sicstus.sics.se` (referenced on page 11).

[10] Geoffrey Chu. “Improving Combinatorial Optimization”. PhD thesis. Department of Computing and Information Systems, University of Melbourne, Australia, 2011. Available at `http://hdl.handle.net/11343/36679`; the Chuffed solver and MiniZinc backend are available at `https://github.com/chuffed/chuffed` (referenced on page 11).

[11] Francesco Contaldo, Patrick Trentin, and Roberto Sebastiani. “From MiniZinc to optimization modulo theories, and back”. In: *CP-AI-OR 2020*. Ed. by Emmanuel Hebrard and Nysret Musliu. Vol. 12296. LNCS. Springer, 2020, pp. 148–166 (referenced on page 11).

[12] Bruno De Backer, Vincent Furnon, Paul Shaw, Philip Kilby, and Patrick Prosser. “Solving vehicle routing problems using constraint programming and metaheuristics”. In: *Journal of Heuristics* 6(4) (Sept. 2000), pp. 501–523 (referenced on page 33).

[13] Renaud De Landtsheer, Yoann Guyot, Gustavo Ospina, and Christophe Ponsard. “Combining neighborhoods into local search strategies”. In: *Recent Developments in Metaheuristics*. Ed. by Lionel Amodeo, El-Ghazali Talbi, and Farouk Yalaoui. Vol. 62. ORCS. Springer, 2018, pp. 43–57 (referenced on page 26).

[14] Renaud De Landtsheer and Christophe Ponsard. “OscaR.cbls: An open source framework for constraint-based local search”. In: *ORBEL-27, the 27th annual conference of the Belgian Operational Research Society*. 2013. Available as `https://www.orbel.be/orbel27/pdf/abstract293.pdf`; the OscaR.cbls solver is available at `https://bitbucket.org/oscarlib/oscar/branch/CBLS` (referenced on pages 2, 11, 25, 26, 30).

[15] Luca Di Gaspero and Andrea Schaerf. “EasyLocal++: An object-oriented framework for the flexible design of local-search algorithms”. In: *Software: Practice and Experience* 33(8) (June 2003), pp. 733–765 (referenced on page 25).

[16] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. 2nd ed. Cengage Learning, 2002. Available at `https://ampl.com/resources/the-ampl-book` (referenced on page 4).

[17] Alan M. Frisch, Matthew Grum, Chris Jefferson, Bernadette Martinez Hernandez, and Ian Miguel. “The design of Essence: A constraint language for specifying combinatorial problems”. In: *IJCAI 2007*. Morgan Kaufmann, 2007, pp. 80–87 (referenced on pages 4, 27).

[18] Gecode Team. *Gecode: A Generic Constraint Development Environment*. 2021. The Gecode solver and its MiniZinc backend are available at `https://www.gecode.org` (referenced on page 11).

[19] Fred Glover and Manuel Laguna. “Tabu search”. In: *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, 1993, pp. 70–150 (referenced on page 18).

[20] Google Optimization Team. *OR-Tools: Google’s Software Suite for Combinatorial Optimization*. 2021. Available at `https://developers.google.com/optimization` (referenced on page 11).

[21] Gurobi Optimization, Inc. *Gurobi Optimizer Reference Manual*. 2017. Available at `https://www.gurobi.com` (referenced on page 11).

[22] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004 (referenced on pages 15, 17, 18).

[23] IBM. *CPLEX*. `https://www.ibm.com/analytics/cplex-optimizer`. 2021 (referenced on page 11).

[24] johnjforrest, Stefan Vigerske, Haroldo Gambini Santos, Ted Ralphs, Lou Hafer, Bjarni Kristjansson, jpfasano, EdwinStraver, Miles Lubin, rlougee, jpgoncal1, h-i-gassmann, and Matthew Saltzman. *coin-or/Cbc: Version 2.10.5*. Mar. 2020. URL: `https://doi.org/10.5281/zenodo.3700700` (referenced on page 11).

[25] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. “Optimization by simulated annealing”. In: *Science* 220(4598) (May 1983), pp. 671–680 (referenced on page 18).

[26] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. 2nd ed. Texts in Theoretical Computer Science, an EATCS Series. Springer, 2016 (referenced on page 4).

[27] Kevin Leo, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. “Globalizing constraint models”. In: *CP 2013*. Ed. by Christian Schulte. Vol. 8124. LNCS. Springer, 2013, pp. 432–447 (referenced on page 10).

[28] Shen Lin and Brian W. Kernighan. “An effective heuristic algorithm for the traveling-salesman problem”. In: *Operations Research* 21(2) (1973), pp. 498–516 (referenced on page 4).

[29] Michele Lombardi and Pierre Schaus. “Cost impact guided LNS”. In: *CP-AI-OR 2014*. Ed. by Helmut Simonis. Vol. 8451. LNCS. Springer, 2014, pp. 293–300 (referenced on page 29).

[30] Michael Marte. *Yuck: A local-search constraint solver with FlatZinc interface*. 2021. Available at `https://github.com/informarte/yuck/` (referenced on pages 2, 11, 25, 26).

[31] Laurent Michel and Pascal Van Hentenryck. "Localizer: A modeling language for local search". In: *CP 1997*. Ed. by Gert Smolka. Vol. 1330. LNCS. Springer, 1997, pp. 237–251 (referenced on page 2).

[32] Laurent Michel and Pascal Van Hentenryck. "Localizer". In: *Constraints* 5(1–2) (2000), pp. 43–84 (referenced on pages 18, 21, 25).

[33] Laurent Michel and Pascal Van Hentenryck. "A constraint-based architecture for local search". In: *ACM SIGPLAN Notices* 37(11) (2002), pp. 101–110. *OOPSLA 2002* (referenced on page 25).

[34] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. "MiniZinc: Towards a standard CP modelling language". In: *CP 2007*. Ed. by Christian Bessière. Vol. 4741. LNCS. Springer, 2007, pp. 529–543. The MiniZinc toolchain is available at `https://www.minizinc.org` (referenced on pages 2, 4, 6, 7).

[35] M.A. Hakim Newton, Duc Nghia Pham, Abdul Sattar, and Michael Maher. "Kangaroo: An efficient constraint-based local search system using lazy propagation". In: *CP 2011*. Ed. by Jimmy Lee. Vol. 6876. LNCS. Springer, 2011, pp. 645–659 (referenced on pages 2, 21, 25, 26).

[36] OscaR Team. *OscaR: Scala in OR*. 2012. Available at `https://bitbucket.org/oscarlib/oscar/wiki/` (referenced on page 26).

[37] Laurent Perron, Paul Shaw, and Vincent Furnon. "Propagation guided large neighborhood search". In: *CP 2004*. Ed. by Mark Wallace. Vol. 3258. LNCS. Springer, 2004, pp. 468–481 (referenced on page 29).

[38] Gilles Pesant and Michel Gendreau. "A constraint programming framework for local search methods". In: *Journal of Heuristics* 5(3) (Oct. 1999), pp. 255–279. Extends a preliminary version at CP 1996, LNCS, vol. 1118, pp. 353–366, Springer (1996) (referenced on page 35).

[39] Cédric Pralet and Gérard Verfaillie. "Dynamic online planning and scheduling using a static invariant-based evaluation model". In: *ICAPS 2013*. Ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini. AAAI Press, 2013, pp. 171–179 (referenced on page 25).

[40] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco: A Free and Open-Source Java Library for Constraint Programming*. 2014. Available at `https://www.choco-solver.org` (referenced on page 11).

[41] Charles Prud'homme, Xavier Lorca, and Narendra Jussien. "Explanation-based large neighborhood search". In: *Constraints* 19(4) (Oct. 2014), pp. 339–379 (referenced on page 29).

[42] Francesca Rossi, Peter van Beek, and Toby Walsh, eds. *Handbook of Constraint Programming*. Elsevier, 2006 (referenced on page 4).

[43] Christian Schulte and Mats Carlsson. "Finite domain constraint programming systems". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Elsevier, 2006. Chap. 14, pp. 495–526 (referenced on page 15).

[44] Christian Schulte and Peter J. Stuckey. "Efficient constraint propagation engines". In: *ACM Transactions on Programming Languages and Systems* 31(1) (Dec. 2008), pp. 1–43 (referenced on page 13).

[45] Paul Shaw. "Using constraint programming and local search methods to solve vehicle routing problems". In: *CP 1998*. Ed. by Michael Maher and Jean-François Puget. Vol. 1520. LNCS. Springer, 1998, pp. 417–431 (referenced on pages 15, 27, 29).

[46] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. "The MiniZinc Challenge 2008–2013". In: *AI Magazine* 35(2) (summer 2014), pp. 55–60. See `https://www.minizinc.org/challenge.html` (referenced on page 31).

[47] Pascal Van Hentenryck and Laurent Michel. "Control abstractions for local search". In: *CP 2003*. Ed. by Francesca Rossi. Vol. 2833. LNCS. Springer, 2003, pp. 65–80 (referenced on page 25).

[48] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005 (referenced on pages 2, 4, 15, 18, 19, 21–23, 25, 26).

[49] Pascal Van Hentenryck and Laurent Michel. "Synthesis of constraint-based local search algorithms from high-level models". In: *AAAI 2007*. Ed. by Adele Howe and Robert C. Holte. AAAI Press, 2007, pp. 273–278 (referenced on page 25).

[50] Christos Voudouris, Raphael Dorne, David Lesaint, and Anne Liret. "iOpt: A software toolkit for heuristic search methods". In: *CP 2001*. Ed. by Toby Walsh. Vol. 2239. LNCS. Springer, 2001, pp. 716–729 (referenced on page 25).

[51] Laurence A. Wolsey. *Integer Programming*. Wiley, 1998 (referenced on page 4).

[52] Neng-Fa Zhou and Håkan Kjellerstrand. "The Picat-SAT compiler". In: *PADL 2016*. Ed. by Marco Gavanelli and John Reppy. Vol. 9585. LNCS. Springer, 2016, pp. 48–62 (referenced on page 11).

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2022

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-436139

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2021